A framework for distributed computing over the internet

Ong, Te-Min

2005

Ong, T.-M. (2005). A framework for distributed computing over the internet. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/2417>

<https://doi.org/10.32657/10356/2417>

Nanyang Technological University

Downloaded on 26 Apr 2023 18:24:17 SGT

A Framework for Distributed Computing over the Internet

Ong Te-Min

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of
Master of Engineering

2003

ACKNOWLEDGEMENTS

The author would like to express his gratitude to these people who made all this possible. Special thanks go to:

- Prof. Lee Bu Sung, for his indefatigable supervision over the progress of the project and his generous assistance in times of distress.
- Prof Yeo Chai Kiat, for her invaluable ideas and suggestions as well as frank advice during critical junctures.

Thanks also to my fellow student Lim Teck Meng and the staff of the Automation Technology Laboratory for their assistance with the provisioning of equipment and software for the tests required.

TABLE OF CONTENTS

CHAPTER	PAGE
TABLE OF CONTENTS.....	I
LIST OF FIGURES	IV
LIST OF TABLES	VI
LIST OF EQUATIONS.....	VII
LIST OF ACRONYMS.....	VIII
ABSTRACT	X
CHAPTER 1 INTRODUCTION.....	1
1.1 BACKGROUND.....	1
1.2 PURPOSE AND SCOPE.....	2
1.3 REPORT ORGANIZATION.....	3
CHAPTER 2 LITERATURE SURVEY	4
2.1 METACOMPUTING	4
2.2 GRID COMPUTING.....	5
2.2.1 <i>The Globus Project</i>	6
<i>Communication</i>	6
<i>Resource Allocation</i>	6
<i>Authentication Methods</i>	6
<i>Information Service</i>	7
<i>Data Access</i>	7
2.2.2 <i>The Sun Grid Engine</i>	9
2.3 VOLUNTEER COMPUTING.....	11
2.3.1 <i>Java Applet Based</i>	12
<i>Javelin</i>	12
<i>The Knitting Factory</i>	12
<i>SuperWeb</i>	13
<i>Locust</i>	14

2.3.2	<i>Java Application Based</i>	14
	<i>Ninplet</i>	15
2.3.3	<i>Hybrid Systems</i>	15
	<i>Javelin++</i>	15
	<i>Bayanihan</i>	16
2.3.4	<i>Miscellaneous Volunteer Computing Systems</i>	16
	<i>Distributed.Net</i>	16
	<i>SETI@Home</i>	16
2.4	COMMERCIAL SYSTEMS	17
2.5	SUMMARY	18
CHAPTER 3 THE UNISON SYSTEM		21
3.1	WEB-BASED AND JAVA-BASED DISTRIBUTED COMPUTING	21
3.1.1	<i>Deployment Issues</i>	21
3.1.2	<i>Limitations of Application-based Systems</i>	22
3.1.3	<i>Web-based and Java-based Systems</i>	23
3.2	SYSTEM DESIGN PRINCIPLES	26
3.3	UNISON SYSTEM ARCHITECTURE	21
3.3.1	<i>Broker</i>	29
3.3.2	<i>Host</i>	32
3.3.3	<i>Client</i>	36
3.4	TASK MODEL	40
3.5	THE SCHEDULER	42
3.6	UNISON PROGRAMMING LIBRARY	48
CHAPTER 4 EXPERIMENT RESULTS		50
4.1	BENCHMARK IMPLEMENTATION	50
4.1.1	<i>Local Alignment Benchmark</i>	50
4.1.2	<i>Mersenne Prime Search Benchmark</i>	58
4.1.3	<i>Raytracing Benchmark</i>	59
4.2	RESULT ANALYSIS	61
4.2.1	<i>Mersenne Primality Test Results</i>	62
4.2.2	<i>Local Alignment Benchmark Results</i>	64
4.2.3	<i>Raytracing Benchmark Results</i>	65
4.3	MISCELLANEOUS BENCHMARKS	67
4.3.1	<i>Effect of JVM Version on Local Alignment Application</i>	68
4.3.2	<i>Effect of JVM Version on Mersenne Primality Application</i>	69

4.3.3	<i>Effect of Data Granularity on Mersenne Primality Application</i>	70
CHAPTER 5	CONCLUSION	72
5.1	SUMMARY	72
5.2	FUTURE WORK	73
BIBLIOGRAPHY		75
APPENDIX A	SCHEDULING TERMS	I

LIST OF FIGURES

Figure	Page
FIGURE 1 GLOBUS GRAM JOB FLOW	8
FIGURE 2 SOFTWARE JOB FLOW IN SUN GRID ENGINE	11
FIGURE 3 UNISON FRAMEWORK	28
FIGURE 4 BROKER COMPONENTS	30
FIGURE 5 IDL DEFINITION FOR BROKER	31
FIGURE 6 IDL DEFINITION FOR TASK INFORMATION OBJECTS	32
FIGURE 7 STATE TRANSITION DIAGRAM FOR UNISON HOSTS	33
FIGURE 8 HOST COMPONENTS	35
FIGURE 9 IDL DEFINITION FOR HOSTS	36
FIGURE 10 HOST FMWCHART	37
FIGURE 11 CLIENT FLOWCHART	38
FIGURE 12 MAIN MENU	39
FIGURE 13 CLIENT REGISTRATION MENU	39
FIGURE 14 CLIENT JOB SUBMISSION MENU	40
FIGURE 15 CLIENT TASK SUBMISSION MENU	40
FIGURE 16 JOB-TASK RELATIONSHIP	41
FIGURE 17 IDL DEFINITION FOR SCHEDULER	42
FIGURE 18 TASK QUEUE	46
FIGURE 19 EXAMPLE OF SEQUENCE IN FASTA FORMAT	53
FIGURE 20 BLOSUM62 MATRIX	54
FIGURE 21 INITIALIZATION OF H MATRIX	55
FIGURE 22 CALCULATION OF H VALUES	56
FIGURE 23 TRACEBACK TO GET LOCAL ALIGNMENT	56
FIGURE 24 THE LOCAL ALIGNMENT	56
FIGURE 25 COMPUTATIONAL MODEL FOR MCALALIGNMENT BENCHMARK	57
FIGURE 26 THE LUCAS-LEHMER PRIMALITY TEST	58
FIGURE 27 PSEUDO CODE FOR LUCAS-LEHMER TEST	59
FIGURE 28 COMPUTATIONAL MODEL FOR MERSENNE PRIME SEARCH BENCHMARK	59
FIGURE 29 TEST SCENE FOR RAYTRACING BENCHMARK	60
FIGURE 30 MERSENNE PRIMALITY BENCHMARK EXECUTION TIME	60
FIGURE 31 SPEEDUP OBTAINED FOR MERSENNE PRIMALITY BENCHMARK	60
FIGURE 32 LOCAL ALIGNMENT EXECUTION TIME	60
FIGURE 33 ACHIEVABLE SPEEDUP FOR LOCAL ALIGNMENT BENCHMARK	60
FIGURE 34 EXECUTION TIME FOR RAYTRACING BENCHMARK	66
FIGURE 35 SPEEDUP OBTAINED WHEN RUNNING RAYTRACING BENCHMARK	66

FIGURE 36 EXECUTION TIME FOR LOCAL ALIGNMENT BENCHMARK ON DIFFERENT JVMs	67
FIGURE 37 EXECUTION TIME OF MERSENNE PRIMALITY BENCHMARK ON DIFFERENT JVMs.....	69
FIGURE 38 EFFECT OF DATA GRANULARITY ON EXECUTION TIME OF MERSENNE PRIMALITY BENCHMARK....	70

LIST OF TABLES

Table	Page
TABLE 1 COMPARISON BETWEEN 3 NETWORKED COMPUTING MODELS	2
TABLE 2 SUMMARY OF RELATED SYSTEMS	20
TABLE 3 TASK TABLE.....	48
TABLE 4 JOB TABLE.....	48
TABLE 5 UNISON LIBRARY API.....	49

LIST OF EQUATIONS

Equation	Page
EQUATION 1 HOST CHARACTERIZATION EQUATION	44
EQUATION 2 RECURRENCE EQUATIONS FOR FINDING $H(I, j)$	53

LIST OF ACRONYMS

BLOSUM	Blocks SubstitutionMatrix
CORBA	Common Object Request Broker Architecture
DAMPP	Distributed Applet-based Massively Parallel Processing
DNA	Deoxyribonucleic Acid
DRM	Distributed Resource Management
FCFS	First Come First Serve
GASS	Global Access to Secondary Storage
GRAM	Grid Resource Allocation Management
GSI	Globus Security Infrastructure
GSS	Generic Security System
HPC	High Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IIOP	Internet Inter-ORB Protocol
ORB	Object Request Broker
JAXB	Java Architecture for XML Binding
JAXM	Java API for XML Messaging
JAXP	Java API for XML Parsing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-based RPC
JNDI	Java Naming and Directory Interface
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MDS	Metacomputing Directory Service
MPI	Message Passing Interface
MPMD	Multi Program Multiple Data
NOW	Network of Workstations
POA	Portable Object Adaptor
PVM	Parallel Virtual Machine
RDF	Resource Description Framework
RMI	Remote Method Invocation
RPC	Remote Procedure Call

RSL	Resource Specification Language
SMP	Symmetric Multiprocessor
SPMD	Single Program Multiple Data
SSL	Secure Sockets Layer
URL	Uniform Resource Locator
XML	Extensible Markup Language

ABSTRACT

The advent of sophisticated desktop and mobile devices has brought vast computing power into the hands of millions of consumers. By providing such a vast resource pool it has paved the way for the evolution of distributed computing into new forms. We present Unison, a framework for distributed computing in Java, to harness this new resource over the Internet.

Unison employs the code-shipping paradigm to broker this processing power. Clients only have to insert Java code for whatever computation they desire to perform into our host applet and then upload it together with the associated data to the server. These tasks are distributed in parallel to idle hosts managed by a Unison broker. The framework utilizes CORBA as the underlying communication layer and handles all the communication needs of the broker and hosts. Clients are decoupled from the system and only have to concentrate on the functionality of their applications with the added advantage of not having to remain online throughout the computation.

This thesis describes the design behind the system and how it overcomes inherent difficulties present in Java. It also includes a description of an implementation of the system and a series of adaptations of real world applications such as local sequence alignment. Using these applications a set of experiments were conducted to compare the performance of Unison with another Java applet based distributed computing system. The results show that Unison is ideally suited for the running of such applications and that the system is able to reach a level of performance comparable to the reference system.

CHAPTER 1 INTRODUCTION

1.1 Background

The phenomenal growth of the Internet has brought about great interest in network technology research and the appearance of high-speed broadband connections to end-user's doorsteps. At the same time, numerous technological advances have driven prices of personal computers down and facilitated their presence on the desktops of the general populace. These vast pools of computing resources which are left idling after their active hours present a huge computational power to be tapped.

Many efforts at harnessing this collection of resources have been underway for quite some time now. Take for example United Devices and its partnership with Intel; University of Oxford and the National Foundation for Cancer Research in a project using THINK to analyze three-dimensional molecular models in the search for a cure for cancer; Genome@Home¹ which is a project devoted to the study of human genomes and design of new proteins; and Pioneer by Parabon Computation. All of these projects bear a common trait: they were conceived with the intention of showing how the masses can be encouraged to participate in a common cause to tackle problems which were purely in the domain of proprietary and high-end systems in the past. These serve as real-life evidence of the tantalizing potential of volunteer computing.

From a historical point of view, present volunteer computing systems have their roots in parallel processing systems and from whence distributed systems came into the picture. The common thread between the three lies in where processors within the system are located and the nature of inter-processor links. We have seen systems move from tightly-coupled multi-processor parallel computers to networks of workstations connected by LANs to loose federations of computers interconnected by WANs. This evolution has been fueled by the need for ever greater amounts of computing power. In this day and age when personal computers and broadband connections are commonplace we now have the requisites in place for harnessing CPU cycles on a large scale and worldwide. At this juncture new questions arise from issues that have not been considered problems in controlled environments such as that found in parallel computers. We have to take

¹ <http://genomeathome.stanford.edu> (Online, as of April 2003)

into consideration rapidly changing environmental factors such as link bandwidth and latency; host connectivity and reliability; and security and administrative domains. There is a multitude of areas which still require study but at the same time we should take heed of the lessons learnt during the process of designing systems in the past.

1.2 Purpose and Scope

This report introduces Unison, a framework for distributed computing over the Internet. Its purpose is to harness and broker idling computing resources. Distributed computing is a form of parallel computing that makes it easy for people to share their idle machines' computing power to solve large computational problems. However, unlike parallel computing, distributed computing does not require specialized software to be installed on volunteer computers, or for dedicated user accounts on those machines.

Unison utilizes a code-shipping paradigm to distribute the job execution over the network of computers. There are three main paradigms used in networked computing models: *proxy computing*, *remote computing* and *code shipping*. They differ in the way in which programs and data, which these programs are to utilize, are handled. User machine refers to any end-user's personal computer while the term server refers to any computer whose main role is to link resources to users. A comparison of these models can be found in Table 1. In *proxy computing*, programs and data are stored on the user machine. They are both uploaded to the server which computes the data and returns the result. However this approach is hindered by the presence of heterogeneous computer architectures on the network. A framework that can accommodate different machine types will require considerable effort to realize. In *remote computing*, the program resides on a server and user data is uploaded to the server for computation. Numerical libraries process this data on the server to return a result to the user machine. In *code shipping*, the program codes reside on a server and are downloaded to the user machine together with either a portion or a complete set of data. The computations are performed on the user machine. The generated result is then uploaded to the server.

Table 1 Comparison between 3 networked computing models

	<i>Proxy Computing</i>	<i>Remote Computing</i>	<i>Code Shipping</i>
Program Location	User Machine	Server	Server
Data Location	User Machine	User Machine	User Machine
Executed at	Server	Server	User Machine

Unison explicitly avoids OS or Java Virtual Machine (JVM) modifications and native code. Instead we opt to take a pure Java approach that builds on Java's write-once-run-anywhere strength to enable cross-platform portability. Any future optimizations will continue this policy.

We intend to include within the scope of our study these issues:

1. How the bulk of existing projects have concentrated on application-specific problems and thus we would like to focus on how to allow general-purpose applications to optimally run on our system.
2. We will study how systems handling Single Program Multiple Data (SPMD) programs can be extended to apply to Multiple Program Multiple Data (MPMD) programs.
3. What kind of incentives can entice users to proffer their computers' CPU cycles? How should these incentives be monetary or in some other form?
4. How are clients of the system expected to participate? Are they to be solely in charge of code and data distribution, or is a third party going to do that on their behalf?
5. How security and privacy of participating computers is ensured?

1.3 Report Organization

This report is organized in five chapters. This is the first chapter in which we give an introduction to distributed computing, outline the objectives and lay out the scope of the project.

Chapter 2 contains an overview of existing distributed computing technologies: metacomputing, grid computing and volunteer computing. Descriptions of Java based volunteer computing systems are given along with a comparison.

The next chapter gives a description of Unison architecture and explains the role of each of the three major entities in the system. It also relates example applications we have developed for system testing.

Chapter 4 outlines the three applications we have developed to test the system. Each test case is presented and the results from these tests are analyzed. In chapter 5 we conclude this report by summarizing the work done so far and highlighting the contributions made by this project.

CHAPTER 2 LITERATURE SURVEY

Broadly speaking, work done in present day distributed computing has its roots in metacomputing. In this chapter we present a brief overview of the history of metacomputing and that of its progeny: grid computing and volunteer computing. We also give a description of the projects and some commercial examples. In our context we will use the terms distributed computing and volunteer computing interchangeably.

2.1 Metacomputing

A metacomputer can be thought of as a networked virtual supercomputer dynamically constructed of local or geographically distributed resources. The primary motivation for the existence of metacomputers is the need for accessing resources that are not located within a single computer system. Replicating costly resources such as supercomputers is economically unfeasible. Therefore a way has to be devised to promote remote access to the resource, a result of which is to maximize its utilization.

The primary form taken by metacomputers is that of Networks Of Workstations (NOWs). These allow the pooling and use of otherwise idle workstations in local or campus wide networks in parallel processing. To this end, general purpose and cross platform parallel processing systems such as Message Passing Interface (MPI) [GLS94] and Parallel Virtual Machine (PVM) [GBD94] were conceived. Although these systems improved on the ease of writing parallel applications by abstracting details with a high level API, they still required considerable effort on the part of administrators in setting up the runtime environment on each machine individually.

In order to support control by the owner over the running application, remote shell access has to be provided (for PVM). This can be undesirable if access to the machine is intended to be limited only to the running of parallel applications. Furthermore, administrators have to establish contact with each other before commencing the Network of Workstations (NOW) setup. Accounts have to be manually created to enable the parallel runtime environment to function. This could take days or even weeks to setup the NOW.

2.2 Grid Computing

Grid computing is poised to develop in a way that is analogous to how basic infrastructure such as railroads, expressways, power and gas pipeline grids progressed. Just as towns and cities grew in step with the provision of transport links and electricity, computational grids are expected to become as commonplace as these ubiquitous amenities. The key concept here is the scenario where computers and other specialized resources are available anytime and anywhere at the flick of a switch. Grid computing is a natural extension of the concept of metacomputing [CS92].

The Grid alludes to an infrastructure to enable the integration and sharing of resources such as high-end computer systems, networks, scientific instruments and databases. These resources are likely to be administered by disparate organizations and the Grid promises to permit easy access to them while at the same time ensuring that a certain degree of security is in place.

What are the possible applications for grid computing? These four possibilities were identified in [DFP96]:

1. Desktop supercomputing

These applications couple high-end graphics capabilities with remote supercomputers and/or databases. This coupling connects users more tightly with computing capabilities, while at the same time achieving distance independence between resources, developers, and users.

2. Smart instruments

These applications connect users to instruments such as microscopes, telescopes, or satellite downlinks that are themselves coupled with remote supercomputers. This computational enhancement can enable both quasirealtime processing of instrument output and interactive steering.

3. Collaborative environments

A third set of applications couple multiple virtual environments so that users at different locations can interact with each other and with supercomputer simulations.

4. Distributed supercomputing.

These applications couple multiple computers to tackle problems that are too large for a single computer or those that can benefit from executing different problem components on different computer architectures.

The following sections outline some typical grid computing systems. Two real-world examples: The Globus Project and The Sun Grid Engine are described.

2.2. 1 *The Globus Project*

At the heart of the Globus project lies the metacomputing infrastructure toolkit. It is a collection of modules that encompass services provided by a typical metacomputing abstract machine such as: communications, resource allocation, authentication, information service and data access. This abstract machine is to be the basis for the construction of a range of services, applications and infrastructures.

Communication

The Globus communication model was based on the Nexus communication library [FKT96]. It provided unicast and multicast services but is in the process of being deprecated.

Resource Allocation

The Grid Resource Allocation Management (GRAM) [CFK98] protocol and client API allows the management of computational resources and monitoring them to control the computation on these remote resources. In this architecture, an extensible *resource specification language* (RSL) is used to communicate requests for resources between components. These components include applications, resource brokers, resource co-allocators and resource managers. At each stage in this process, information about resource requirements is coded into an RSL expression by the application or refined by one or more resource brokers and co-allocators; information about resource availability and characteristics can be obtained from an information service like the Metacomputing Directory Service (MDS) described below.

Authentication Methods

The Generic Security System (GSS) [LINN93] increases the degree of abstraction at the toolkit interface. GSS defines a standard procedure and API for obtaining credentials (passwords or certificates), for mutual authentication (client and server), and for message-oriented encryption

and decryption. GSS is independent of any particular security mechanism and can be layered on top of different security methods, such as Kerberos and Secure Sockets Layer (SSL).

Globus implements a grid security architecture called the Globus Security Infrastructure (GSI) [FKT98]. This extends from the GSS-API as well as standard public key protocols to provide metacomputing features not specific to GSS.

Information Service

Globus includes a single unified access mechanism for information about the system including: configuration details about resources such as the amount of memory, CPU speed, number of nodes in a parallel computer, or the number and type of network interfaces available; instantaneous performance information, such as point-to-point network latency, available network bandwidth, and CPU load and application-specific information, such as memory requirements or program structures found effective on previous runs.

This mechanism is known as the MDS [FFK97] and is built upon the data representation and application programming interface defined by the Lightweight Directory Access Protocol (LDAP). [FL] describes how the MDS utilizes LDAP while [CFF01] describes an updated version of the service, MDS-2.

Data Access

To permit applications to access data not present locally but instead at a remote site, Globus features a Global Access to Secondary Storage (GASS) [BFK99] mechanism that facilitates data movement and access. GASS itself uses the other Globus modules for security and communication whilst those modules use GASS to perform remote executable staging.

Software job flow for Globus is shown in Figure 1. The steps involved when a job is submitted are:

1. Request submission

The GRAM client library is called by an application. It interacts with a gatekeeper at a remote site to authenticate itself and to submit a request. This request consists of a resource specification and a callback (which will be used by the job manager).

Figure 1 Globus GRAM job flow

2. Job Manager Creation

The gatekeeper responds to a client request by: (1) Performing mutual authentication of user and resource; (2) Mapping a local user name to the remote user and (3) Starts a job manager that executes as if it were run by that local user. Actions (1) and (2) are handled by calls to the GSI.

3. Send Request to Local Scheduler

The job manager created by a gatekeeper is responsible for creating processes requested by the user. It does this by submitting a resource allocation request to the underlying local scheduler (resource management system). Upon successful creation of these processes, the job manager monitors their state, informs the client of any state transitions via the callback and handles control operations such as process termination. The job manager only exists for as long as the job continues to run.

4. Process Creation

GRAM implementations have been constructed to run with several local schedulers such as Condor [LLM88] and LSF [ZHO92]. The actual processes are allocated and created by these local schedulers.

2.2.2 The Sun Grid Engine

The Sun Grid Engine is a product introduced by Sun Microsystems after their acquisition of Gridware. The source code is available along with binaries for Solaris and Linux operating systems. Sun categorizes grids into three types (their product is targeted towards realizing compute grids):

1. Compute Grids

Compute grids consist of distributed compute resources such as desktops, servers, and High Performance Computing (HPC) systems

2. Access Grids

Access grids consist of distributed audio-visual equipment, such as cameras, microphones, speakers, and video screens, set up to provide a virtual collective presentation room.

3. Data Grids

This is formed from distributed storage devices including disk and tape devices along with the necessary software to migrate data as needed.

The Sun Grid Engine acts as a Distributed Resource Management (DRM) software. It optimizes utilization of software and hardware resources in heterogeneous networked environments. Sun Grid Engine software aggregates the computing power available in cluster grids and presents a unified and simple access point to users needing compute cycles by distributing computational workload. It provides job accounting information and statistics that are used to monitor resource utilization and determine how to adjust this allocation. The interface enables administrators to specify job options for priority, hardware and license requirements, dependencies, and time windows, and to define and control user access to computer resources.

DRM software is based on batch queuing. If the required resource is unavailable to execute a job, then the job is queued until resources are available. In addition to this, DRM software also monitors host computers in the cluster for balanced load conditions. Servers in a Sun Grid Engine cluster are referred to as hosts. These are of five types:

1. *Master host*

A single host is selected to be the Sun Grid Engine master host. This host handles all requests from users, makes job scheduling decisions, and dispatches jobs to execution hosts.

2. *Execution hosts*

Systems in the cluster that are available to execute jobs are called execution hosts.

3. *Submit hosts*

Submit hosts are machines configured to submit, monitor, and administer jobs, and to manage the entire cluster.

4. *Administration hosts*

Sun Grid Engine administrators use administration hosts to make changes to the cluster configuration, such as changing DRM parameters, adding new nodes, or adding or changing users.

5. *Shadow master host*

While there is only one master host, other machines in the cluster can be designated as shadow master hosts to provide greater availability. A shadow master host continually monitors the master host, and automatically and transparently assumes control in the event that the master host fails. Jobs already in the cluster are not affected by a master host failure.

Jobs are submitted to the master host and are held in a spooling area until the scheduler determines that the job is ready to run. The Sun Grid Engine software matches available resources to job requirements, such as available memory, CPU speed, and available software licenses. The requirements of the jobs may be very different and only certain hosts may be able to provide the corresponding service. As soon as a resource becomes available for execution of a new job, the Sun Grid Engine software dispatches the job with the highest priority and matching requirements. The Sun Grid Engine software scheduler takes into account the order the job was submitted, what machines are available, and the priority of the job.

Software job flow for the Sun Grid Engine is shown in Figure 2. The details are as follows:

1. Job submission

When a user submits a job from a submit host, the job submission request is sent to the master host.

2. Job scheduling

The master host determines the host to which the job will be assigned. It assesses the load, checks for licenses, and evaluates any other job requirements.

3. Job dispatch & execution

After obtaining scheduling information, the master host then sends the job to the selected execution host. The execution host saves the job in a job information database and starts a shepherd process which starts the job, and waits for completion.

4. Accounting information

When the job is complete, the shepherd process returns the job information, and the execution host then reports the job completion to the master host and removes the job from the job information database. The master host updates the job accounting database to reflect job completion.

Figure 2 Software job flow in Sun Grid Engine

2.3 Volunteer Computing

Volunteer computing is a form of metacomputing that focuses on maximizing the ease in which people may participate by making their machines part of the metacomputer. Compared to this, grid computing systems, along with MPI and PVM, have complex setup requirements for each

compute server. Whereas volunteer computing aims to eliminate these setup requirements to enable ordinary users to join the computing network. By not having to install daemons on every volunteer machine, the time taken to get a machine to join the network is shortened. It seeks to extend the reach of such networks to encompass thousands of machines and thus tap this enormous reservoir of computing power.

Volunteer computing systems can be broadly classified into two categories: application based and web based. Each has its own advantages and tradeoffs such as: relatively greater control over the runtime environment for application based systems vs. ease of deployment for web based systems (along with a more restricted environment). In either case, the program will automatically communicate with a server to transfer results and data then run the computation without any user intervention. The next three sections will describe several web and application based systems along with some hybrid web/application based systems.

2.3.1 Java Applet Based

Javelin

Javelin [CCI97] is a research project that models a distributed computing system as brokers, clients and hosts. Clients need to program applets and a HTML page containing the program applet before registering the Uniform Resource Locator (URL) of that HTML page with the broker. Hosts register with the broker with an intention to share resources. The broker assigns tasks to the host. The host runs a 2-frame HTML page. One of the frames runs a daemon applet for controlling the other frame to load a task URL. The Javelin system, however, has the limitation that the applet loaded could kill the daemon running on the host, i.e. the program applet call the browser to load a new page thus stopping the daemon applet. This would disable all communication between the host and the server. If the daemon was to run as an application, the security issues would then pose a problem. Unison tries to avoid this problem by making sure that daemon applets are not necessary and that communication is established between the applet and the server directly.

The Knitting Factory

The Knitting Factory [BKK98] is another web based computing infrastructure that uses Java applets as clients and Java applications as servers. The main focus of this project is to decentralise the system by avoiding the need for tying computational tasks to HTTP servers. To this end they

have written a lightweight HTTP server, to be embedded in *initiators* (Charlotte clients), whose sole function is to serve Java classes.

The Knitting Factory lacks accounting mechanisms to determine the recompense for computational power offered by hosts. Furthermore, Java Remote Method Invocation (RMI) is also used for inter-applet communication (assuming that signed applets were used) thereby incurring the same penalty as experienced by Javelin++. As an example of an application of the Knitting Factory, an extension of Charlotte [BKK96] was implemented by providing a means with which to run Charlotte *Managers* on machines without an HTTP server.

Distriblets (2001)

Distriblets [FWCOI] is an extension of [FWB99] to allow the application programmer to develop a Java applet for distribution to multiple machines. The computation model used in [FWB99] is still present in this system. The main differences now are:

- i) Helper computers run applets
- ii) The distribution server is only responsible for coordinating task execution by directing helper computers (servers) to computation servers with work to be done (clients)
- iii) The user's program in the distriblet has to contain requests for permissions to perform actions violating the sandbox.

The possibility of running offline computations is mentioned but without any details on how the helper applet can detect whether the computer is connected to the Internet. Furthermore, clients of this system now need to run a customized web server in order to allow helper computers to fetch applets from it. In our system, we do not require clients to be present when hosts are running their jobs. Hence all that a client has to do is to submit a job and upload the required files all via HTML forms. There is no need to run anything except a web browser on the client.

Super Web

SuperWeb [AIS97] is also structured around a system of brokers, hosts and clients. However, in their case brokers only perform bookkeeping and matching hosts with clients. Hosts have to retrieve tasks directly from the client. This implies the presence of a web server on each client for hosts to fetch tasks from. Furthermore, it implies that the client must stay online throughout the time spent executing and serving all components of whatever job it wants to complete.

Locust

Locust (LOW cost Computing Utilizing Skimmed idle Time) [MAY99] is a brokerage system for idle CPU resources on the Internet. It is largely based on the Distributed Applet-based Massively Parallel Processing (DAMPP) system [VAN97] and utilizes DAMPP's Jobmaster class as a prototypical cornerstone. This system will be used as a basis for comparison of the performance of Unison.

2.3.2 Java Application Based

Distriblets (1999)

In comparison, Distriblets [FWB99] addresses the problem of portability by using Java. Each computer needs to run a Java application with a security model built-in to talk to a centralized distribution server and computation server. It involves computation of custom written application programs that are registered on the distributed server. A helper application connects to a computation server whose function is to distribute work, downloads the necessary Java classes to execute, executes them, and returns the result to the computation server. However, this system is restrictive compared to Unison because the computation server will have to be online to wait for computation results to be returned by the various helper applications. In contrast, Unison caches whatever results returned so that it can be retrieved at any time.

Atlas

Atlas [BBB96] is an adaptation of Cilk [BE95] to the Java language. Cilk is a language for multithreaded parallel programming based on ANSI C. It is designed for general purpose parallel computing and is of an algorithmic nature. The runtime system provides a scheduler for estimating the performance of programs based on abstract complexity measures.

Atlas modifies this work stealing scheduler to include some form of hierarchical stealing. The system consists of clients, managers and compute servers. In this case Atlas compute servers correspond to hosts in Unison and their managers to Unison brokers.

Managers are arranged in a hierarchy where a manager may be under only one manager. Work stealing may only be done between compute servers under the same manager. Managers keep

track of idle compute servers under them and steal work on their behalf from its own manager and siblings.

Atlas suffers from the same drawback of requiring clients to be online throughout their job execution. Furthermore, compute servers run a mixture of Java code and native libraries thus making portability an issue.

Ninflet

Ninflet [TMN98] is an extension of Ninf [SNS97] which stresses on avoiding Java language extensions or JVM modifications. Ninflet consists of three components: Ninflet Server, Ninflet Dispatcher and Client Program. These three roughly correspond to Unison's host, broker and client respectively. Ninflet clients have to provide a web server from which other servers can download Ninflet class files and therefore suffer from the same lack of ease of use on part of clients.

Dispatchers typically match servers with clients within the same network. This is done to avoid servers fetching Ninflets from a client that is in a geographically distant network and thereby incurring high communication costs.

2.3.3 Hybrid Systems

Javelin++

Javelin++ [BKN99] is an extension of Javelin that features improvements over the old system in areas such as: (1) scalability, by supporting a multiple broker network; and (2) fault tolerance, by introducing a new distributed deterministic eager scheduler. Another major change is the use of Java RMI instead of TCP sockets. However this change causes their system to lose one of its most appealing aspects, that of ease of use in any browser. Whereas Javelin supported hosts as Java applets, Javelin++ allows hosts to be programmed as both Java applets and applications. However with the caveat that the applets be run only on browsers with a JVM that supports Java RMI and be provided with an appropriate security policy. It is a known fact that Microsoft's Internet Explorer (IE) includes a JVM that lacks support for RMI thereby preventing Javelin++ host applets from being run within. In contrast, Unison hosts run as applets on any browser without the need for native code (browser plugins) or elaborate security policies.

Bayanihan

Bayanihan [SHW98] uses HORB [HIR97], a distributed object package similar in functionality to Java RMI which runs in any JVM not necessarily only Java 1.1 compliant JVMs, to realise a system of Servers, Workers and Watchers. It too supports clients in both Java applet and application forms. Using HORB as an alternative to Java RMI allows them to circumvent the lack of support on IE. However this is not without a penalty: the Java archive (jar) file containing HORB classes has to be downloaded together with Bayanihan client applets thereby prolonging the start up delay experienced by users of their system. Volunteer servers are used to exploit the principle of locality in order to minimize communication latencies and the authors suggest the use of server pools to handle large numbers of clients.

Bayanihan does not separate the functions of job submission and distribution. It assumes that jobs are already present on a Bayanihan server and clients connect to it to obtain part of the computation. It remains to be seen how authors of volunteer computing programs submit their programs to the system.

2.3.4 Miscellaneous Volunteer Computing Systems

Distributed.Net

Distributed.net [BEB] is an organization dedicated to the development, deployment and advocacy of distributed computing. It has been and is involved in several projects such as RSA Laboratory's RC5 [KY98] secret-key challenge and data encryption standard (DES) decryption contest. Their client software for the current RC5-64 competition is available for numerous operating systems. The source code is in C and porting to other operating systems requires a significant amount of effort. In contrast, our Unison system is written in Java and benefits from the write once, run anywhere property inherent in every Java applet or application. Furthermore, Trojan horse programs posing as RC5 client software have been encountered. This is not a problem with Unison as only Java applets are used. The browser's sandbox restricts the ability of applets to run malicious code and hence this problem becomes less of a concern in Unison.

SETI@Home

SETI@Home [AND99] is another project of a similar nature to Distributed.net except for the way in which their client software interleaves its operation with other programs on the client machine.

On Windows and Mac OS, this client software runs whenever the user's screensaver is activated and displays a graphical depiction of the data being processed. The key point here is that even though the rate of data processing is hastened by work distribution across the Internet, the data itself is not of a real time nature. A considerable amount of time passes between the reception of signals by the radio telescope and their arrival at the distribution servers. There is no incentive for people to actually install the SETI@Home client other than for the novelty of being part of this global search for life from outer space. As in the Distributed.net project, the issue of providing for a generic client application will become a major problem in the future. Unison is not beset by such problems, as it is reliant on the presence of a Java virtual machine. It is the onus of operating system vendors to provide a standards compliant Java virtual machine. SETI@Home, like Distributed.net, lacks a security framework and relies on the running of native code, usually C or C++ on client machines thereby exposing them to the risk of Trojan software (which are able to make socket connections arbitrarily to any third party) masquerading as the official client program. Moreover, it lacks any form of recompense, monetary or otherwise, for donors of CPU time.

2.4 Commercial Systems

There has been considerable interest in the area of volunteer computing and this has extended to include commercial attempts. We present a short list of companies whose main product or service is related to our area of interest:

- Avaki² (formerly Applied Metacomputing)

Avaki is an attempt to tap into the commercial potential of the Legion [GWF94] project. Legion is an object-based metasystem developed at the University of Virginia. Users see the illusion of a single computer connected to various data and physical resources. Legion works on top of the user's operating system. It features a scheduler with policy framework as well as a user-controlled context space where objects can be created and controlled. Avaki was involved in the University of Virginia's NPACI-net spanning four continents. Their technology was recently applied to a large scale protein folding simulation study.

² <http://www.avaki.com/> (Online as of April 2003)

- Centrata³

Centrata is an application infrastructure provider creating product solutions for next-generation datacenters. It is currently in the process of setting up its business and does not have any projects yet.

- Datasynapse⁴

DataSynapse offers a platform, LiveCluster, that has been explicitly designed for mission-critical deployments in an enterprise environment where turnaround performance is critical. It provides powerful features, including centralized, browser-based administration, robust security measures, and redundant fault-tolerance.

- Entropia⁵

Entropia provides a platform for distributed computing based on PrimeNet. It has been involved in two commercial research initiatives: FightAIDSatHome and SaferMarkets.

- Parabod⁶

Parabon Computation offers Pioneer, a compute engine that enables people to join their network as a provider. They have been involved in the Compute Against Cancer philanthropic computing effort and have a novel idea of rewarding participants with sweepstake prizes.

- United Devices Inc.⁷

This company is involved in the Intel-United Devices Cancer Research Project that has generated significant interest locally.

2.5 Summary

A summary of the volunteer computing systems described earlier is given in Table 2. In contrast to these systems:

³ <http://www.centrata.com> (Online as of April 2003)

⁴ <http://www.datasynapse.com/homec/index.jsp> (Online as of April 2003)

⁵ <http://www.entropia.com> (Online as of April 2003)

⁶ <http://Nmv.parabon.cod> (Online as of April 2003)

⁷ <http://www.ud.camkome.kit> (Online as of April 2003)

- Unison does not require complex setup procedures for either clients or brokers. All that is needed is a web browser on the client and an appropriate Java runtime and web server on the broker.
- Unison Hosts are easily deployable. The only software required is a Java-enabled web browser. There is no need to install third-party applications as the Host Applet is entirely self-contained.
- The Host Applet runs without user intervention. Task execution and data transfer are automatically handled by the broker without having the user to choose the task to be run.
- Daemon applets are no longer necessary. We do not separate the functionality of task execution and control on the host into distinct applets. Thus there is no danger of the daemon applet being stopped by host applet opening a new browser window.
- Unison features an accounting mechanism. This is the basis for providing an incentive for people to utilize the system by sending jobs and for others to contribute their CPU cycles.
- Unison clients can be offline while the job executes. Since brokers monitor job execution and result collection on behalf of clients. An added advantage is clients do not need to run web servers.
- Unison guarantees that security and privacy of host computers are not compromised. The applet sandbox is part of this arrangement.
- There is no need to run native libraries and thus we avoid the security and portability issues that arise with the use of such libraries.

In short, Unison is designed to provide a heterogeneous platform to support distributed computing with no prior installation of software and with minimum human intervention. Moreover, it provides real-time handling of jobs as well as a measure of security for the client systems. Incentives are given to users to avail their systems for use.

Table 2 Summary of Related Systems

Project Name	Applet Based	Application Based	Online Computation	Client Software	Host Software	Comm. Layer
Javelin	√		No	Browser	Browser	Applets
The Knitting Factory	√		Yes	Embedded HTTP Server (Lightweight)	Browser	Java RMI
Distriblets (2001)	√		Client must be online. Host does not.	HTTP Server (Lightweight)	Browser	Applets
SuperWeb	√		Yes	HTTP Server	Browser	HORB
Atlas		√	Yes	JVM + Native Libraries	JVM + Native Libraries	Sockets
Distriblets (1999)		√	Yes	JVM	JVM	Sockets
Ninflet		√	Yes	JVM + HTTP Server	JVM	Java RMI
Javelin++	√	√	Yes	JVM	JVM/Browser	Java RMI
Bayanihan	√	√	Yes	JVM + HTTP Server	JVM/Browser	HORB

CHAPTER 3 THE UNISON SYSTEM

In this chapter we present the idea of web-based distributed computing and a description of a Java-based implementation called Unison. The reasons behind the design decisions made for this system are also discussed. The objective of Unison is to utilize spare computing resources of idle networked workstations to execute programs written with our framework. The following sections contain detailed descriptions and comments on the concepts and design philosophies for the Unison system and its components. Section 3.1 lays out the case for pursuing web-based distributed computing over existing application-based systems such as distributed.net and SETI@home. Section 3.2 presents the principles for Unison system design. Section 3.3 gives an overview of our system. Section 3.4 describes the Job/Task Model we adopt to define a structure for work submitted to the system. Section 3.5 shows how the scheduler works while 3.6 contains a description of the Unison programming library.

3.1 Web-based and Java-based Distributed Computing

In order for our system to achieve widespread acceptance we need to be able to easily deploy Unison hosts and brokers over the Internet. We have identified areas that have come under study recently albeit under different research aims. Firstly, the extensive use of firewalls has fragmented the Internet changing it from its early form of an open network; secondly, the problem of security over different administrative domains is an issue. We need to keep in mind how the Java sandbox environment affects our architecture.

3.1.1 *Deployment Issues*

Firewalls around corporate intranets drastically affect the ability of both Java applets and applications from establishing connections to machines across the Internet. HTTP tunneling has become the recent choice for networks within firewalls that allow traffic through port 80, but this is dependent on individual firewall configurations. Some networks may even have port 80 blocked.

The presence of firewalls acts as a deterrent to the deployment of distributed computing systems. Many current systems have a way of routing their traffic through port 80. Even firewalls that analyze traffic passing through it to see if it is a valid HTTP stream can be fooled into allowing data through by disguising traffic from the application as valid HTTP data. Firewalls are here to stay and we have no choice but to structure our system on the basis that the Internet is divided into many small networks. Within each network unhindered communication may take place but traffic between networks is restricted to certain ports. We need to investigate how effective HTTP tunneling is as for the case of Visibroker gatekeeper and perhaps adapt it for use in our brokers.

3.1.2 Limitations of Application-based Systems

The great popularity and success of distributed computing systems such as distributed.net and SETI@home have their basis in their great accessibility. They allow the general populace to volunteer by downloading and installing a program. These programs proceed to run unobtrusively as screensavers; some minimize themselves to a tiny system tray icon while others can even run without any trace on the “desktop” or process list. The success of these systems is evident in the huge number of volunteers they have managed to attract. Owing to these vast numbers they are able to achieve far higher levels of performance than traditional NOW.

These systems are not without their limitations. For instance, volunteers still require a certain level of technical competence to locate and download the appropriate binary files for their machine architecture and then install them. Furthermore, they still have to figure out how to run the program.

Even if this hurdle is overcome, users still may not want to install the program due to problems with security. The primary concern in the past was of allowing remote shell access to project administrators of traditional NOW systems like PVM. Present day distributed computing systems do not require such access, and indeed the bulk of these programs run on operating systems for which remote shell access is not available by default, e.g. Windows 95/98. However these programs are essentially untrusted codes that are able to perform arbitrary actions such as establishing network connections to any machine on the Internet, read or write to the local filesystem and basically any actions that may be performed by the user himself. They are only limited to the permissions granted to the user. The potential for major security compromises is real especially when these programs are run by the root user/administrator. In fact the presence of Trojan horse programs is acknowledged by both distributed.net and SETI@home and they have taken the step of warning users not to download their client programs from other websites. They

have also provided checksums of their client programs to allow users to verify their integrity but this is not an effective measure as most users probably do not know how to verify these checksums and the vast majority use an operating system that does not include any utility programs to compute the checksum.

There are situations where users may not have the permissions required for installing software on their systems. Places such as these include Internet cafes and in general places where operating systems are configured to disallow new software installation by unauthorized persons. The lack of a client program for a particular operating system is also another hindrance (e.g. United Devices client program only runs on Windows).

The bulk of these distributed computing systems are *ad hoc* and geared towards running specific applications. They do not make use of standardized cross-platform libraries, such as PVM for communication. In addition, non web-based systems face the problem of providing client programs for multiple platforms. This limits their applicability and programmability as the effort required to develop different versions of the same client program for multiple operating systems and computer architectures is non-trivial.

3.1.3 *Web-based and Java-based Systems*

Sun Microsystem's Java technology is widely popular and rapidly becoming ubiquitous. It allows us to address many problems by writing volunteer programs as platform-independent applets that are automatically downloaded and executed on users' machines whenever they visit a webpage with a Java-capable browser. Java emancipates distributed computing systems from the constraints of traditional NOWs by featuring:

1. Platform Independence. This is the primary reason for Java's initial successes. The authors of Java have long extolled the benefits of having to only write one version of code and have it run anywhere (Write Once Run Anywhere).
2. Security. Java applets execute within the context of a sandbox [GJS96, MF97]. This ensures a secure environment where untrusted code may be run. Applets are prevented from accessing user's data and users can participate without fear of security problems caused by viruses or Trojan horses.
3. Ease-of-use. For a Java applet based system, participation is possible by simply visiting the URL of the server with a Java-capable browser. No downloading or installation of software is required. Thus even virtually computer-illiterate people are able to participate.

Java's rich libraries with support for graphical user interfaces, network communications, high level distributed computing services and miscellaneous utility classes allow programmers to concentrate on program logic rather than low level details. Like C++, Java is also object-oriented, but it allows the programmer to take advantage of object-oriented design techniques without the complexity associated with C++. Developing extensible and general-purpose frameworks is easy with Java. This should be the preferred approach rather than of the majority of *ad hoc* systems today.

However some of these features can act to curb the usefulness of Java. The first limitation is due mainly to the way in which Common Object Request Broker Architecture (CORBA) applets usually do not allow a Portable Object Adaptor (POA) or servant objects to be instantiated. There are workarounds to this but most of them involve proprietary implementations of Internet Inter-ORB Protocol (IIOP) proxies such as Visibroker's *Gatekeeper* or JacORB's *Appligator*. An IIOP proxy is an application running on the same machine as the web server from whence the applet was downloaded. The applet itself cannot establish network connections to any machine other than the web server and hence what the IIOP proxy does is to accept incoming connections on behalf of the applet. It includes the logic required to determine which applet to route the request to. This fundamental requirement prohibits inter-host communication and thus limits the scalability of the system.

A long download time is required for the relatively large *vbjorb.jar* file containing Visibroker classes. For Visibroker 4.5.1 this file is almost 3 MB. A user on a 56 Kbps dialup connection to the Internet would typically need around 5 minutes to complete this download before the applet is started. This is a significant delay that could deter them from running the applet again.

Jar file caching by the browser is one way to alleviate this long delay. However each browser has its own caching implementation and we have no means with which to specify that the *vbjorb.jar* file persists in the cache. Moreover cache space is usually shared with other files (HTML and other image files) and browsers may choose to flush the jar files in its cache when it runs out of cache space. A solution to this problem could be to use the Java Plugin. It keeps its own class/jar cache separate from the browser's and we are allowed to specify that a particular jar file is "sticky", meaning that it should remain in the cache. This may be enabled by defining the PARAM *cache-option* with the value *Plugin* within the OBJECT tag in the applet HTML file.

Netscape Communicator includes an outdated version of Visibroker that cannot interoperate with the version that we are using and because of this, running host applets from our system is not

possible without user intervention. This necessitates the removal of the *iiop10.jar* file from Communicator's directory. It is a severe hindrance to the ease of use of the system and we cannot expect users of the system to carry out this extra step. Previous versions of Visibroker (e.g. 3.4 and earlier) could run with this file still present since it allowed us to specify which singleton Object Request Broker (ORB) to use in a parameter to the Visibroker runtime.

While the applet sandbox provides a secure environment in which to run untrusted code, recent improvements to Sun's JVM allows for Java applications to be run within the same sandbox. Furthermore there has been a policy-based fine grained access control mechanism in place since Java 1.2. Java classes within a jar file may be signed using jarsigner provided with the JDK and the user can use *policytool* to specify permissions to be granted to Java classes from a particular codebase for his computer. These permissions include access rights such as read or write ability to the local filesystem (or even to certain files or directories of the filesystem) or the ability to open network connections to particular machines. This is a far cry from the browser specific implementation of signed applets in Java 1.1, wherein developers had to contend with platform specific tools, different archive file formats and signature encodings. There were even specific APIs to be used: Netscape's Capability API⁸ (`netscape.security` classes) and Microsoft's `com.ms.security` classes⁹. Both of these are remnants of the time when browsers implemented their own security framework. Java, from version 1.2 onwards, has such a rich security framework that this API and browser JVMs may be replaced by the Java Plugin, we see no need for these APIs any longer. Furthermore, signing the applet can also help to promote accountability of the system. Any malicious action on part of the applet can be traced back to the organization that used a signing certificate from a certificate authority to sign the applet, thereby enabling a certain measure of trust on the part of the host.

Performance of host applets can be improved by prefetching data for the next task without reloading the current applet (assuming that the next task uses the same applet code). This avoids the overhead associated with applet downloading and initialization. How much data an applet can handle before disk swapping occurs is browser dependent. We have no way of setting memory usage by the JVM unlike for Java applications where non-standard command line options (`-Xms` and `-Xmx`) can be used to specify the initial and maximum memory to be allocated to the JVM respectively.

⁸ http://developer.netscape.com/docs/manuals/signedobj/javadoc/Package-netscape_security.html (Online, as of April 2003)

⁹ <http://www.microsoft.com/java/sdk/default.htm> (Online, as of April 2003)

3.2 System Design Principles

To demonstrate the benefits that are realizable from web-based distributed computing we have developed the Unison web-based system using Java. In designing this system we followed these principles:

- *Dynamic system*
The system will consist of participants who can join or leave on their own volition. Connection time to the system can also be of arbitrary duration. Volunteer systems are able to leave abruptly without disturbing the execution of Unison programs. Allocation of compute resources is handled by brokers and is transparent to users of the system.
- *Independent computation and communication*
Computation and communication on each host is handled by several threads and is in effect independent of each other, The host is not required to wait for results to be uploaded before requesting for another task.
- *Thread-based computation*
The unit of parallelism in Unison is a Java thread. Any program submitted to the system can take advantage of Java's multi-threaded nature. By overlapping communication with computation we are able to write more efficient programs.
- *Job, task and data mapping*
A computing job submitted by a client is mapped to tasks while the data file to be processed is mapped to data objects according to the nature of the job and the current composition of the system.
- *Good accessibility*
We strive to maximize accessibility and ease-of-use for users to volunteer their systems by simply directing their browser to a URL.
- *Programmer-friendly Unison programming interface*
Unison provides a programming interface that makes it easy for people to write Unison applications. All that is required is for the programmer to override a method in the library by providing his own program logic.

3.3 Unison System Architecture

Unison extends the old concept of NOW by allowing system deployment on an Internet-wide basis. It provides a framework for a user to program tasks in the Java programming language and to submit them to the system. These tasks are then packaged by Unison into applets so that they can be downloaded and executed on multiple computers across the Internet.

Distributed systems have traditionally been based on three paradigms: shared memory, distributed memory and message passing. However, the recent trend has been to build distributed systems on components, Remote Procedural Calls (RPC) or RMI. Unison is based loosely on the latter except that it utilizes the rich set of high level services provided by CORBA packages.

Unison consists of three main entities:

1. A Broker

The broker is the main entity in Unison. It is responsible for assigning work to participating machines as well as collecting the results from completed tasks. Brokers coordinate allocation of computing resources to clients.

2. AHost

A Unison host is a member machine in a network of workstations. Basically, hosts run Java applets that spawn several threads to handle computation of the given task, communication with the broker and a user interface to keep the user of the host machine aware of the status of the ongoing application execution.

3. A Client

A Unison client is someone who has a compute intensive application and wants to take advantage of the accumulated resources afforded by Unison. The client is provided with means to submit this application along with the data to be processed to the system.

With reference to Figure 3, our framework can be divided into two main interfaces: Computing and Administrative. Within the context of the former, brokers run as CORBA applications and hosts run CORBA applets both of which communicate with each other via IIOP through their respective ORBs. While under the administrative interface, clients are represented by users interacting with the servlet engine through a web browser.

In our current implementation, this servlet engine resides on the same machine as a broker hut it may also he run on its own. The role of any of the participants in Unison is not fixed. A machine running as a host at a point in time can also act as a client and submitjobs to the system later on. There can be any number of brokers, hosts and clients, hut at least one broker must be present in order to distribute tasks.

An early version of the framework implementation is described in [OLY01]. Further findings and refinement of the framework can be found in [OLL02].

Figure 3 Unison Framework

Unison is designed to be versatile enough to be able to run in a worst-case environment of computers connected by links possessing these characteristics: high latency and low bandwidth, as well as Symmetric Multiprocessor (SMP) machines. Coarse-grain parallel computations are particularly suitable for execution in such an environment as these applications exhibit the characteristic of high computation to communication ratio to take advantage of the resources of a distributed system.

The system works in the following manner:

1. Clients and hosts join Unison by authenticating themselves with a servlet engine.
2. A client may choose to submit a job or view the status of jobs previously submitted to Unison. The process of job submission and status reporting is handling by our web application running on the servlet engine. Further details can be found in section 3.3.3.
3. A host's ID and password are checked before it is redirected to a broker.
4. This broker then assigns a task when the host requests for a task.

Each broker will handle a certain number of hosts before it refuses to accept more. The administrator sets the number of hosts each broker can handle. A servlet engine running the client management web interface allows clients to interact with the system.

3.3.1 *Broker*

Broker Architecture

At the center of Unison is a group of brokers that collects, manages and services requests for computational resources. Brokers control how tasks and data are handled in the system. They handle task collection for clients, task distribution to hosts and the retrieval of results from hosts that have finished processing the data. The description here holds true for every broker in the Unison framework. Clients interact indirectly with brokers through a collection of servlets that allow them to upload tasks and download result files. A central database is updated by these servlets whenever new jobs are submitted. Brokers do not have direct contact with clients as they only check the database for the presence of new jobs. They compete for those jobs with other brokers on a first-come-first-served basis. Possession of a job also means that all the task components of the job belong to that broker.

The broker will then have to acquire all applet class files and data files belonging to whatever job that they have taken from the database. A HTML file is generated for each class file received and this contains the name of the data file as one of its parameters. These HTML, class and data files are placed in the same directory. This directory is present on the broker's local filesystem and is meant to hold files related to the jobs owned by the broker. Therefore all that the broker needs to do in order to get hosts to run that task is to provide the URL of the task's HTML file. The use of applets means that every broker will have to run its own web server. This requirement is

mandated by the presence of the Java sandbox and its restrictions on where applets are able to make connections.

Broker Implementation

The broker is implemented entirely as a CORBA/Java application. The present prototype is based on Java 1.3.1 running on RedHat Linux 6.2. The CORBA package used was Visibroker 4.5 and the web server Apache. Figure 4 shows the 2 major components of a broker: *Host Manager* and *Scheduler*. These components are implemented as Java classes within the *Unison* package.

Figure 4 Broker Components

With reference to Figure 4, the Host Manager acts as an interface between the broker and its hosts. It is responsible for instructing hosts to run tasks as well as for collecting results of completed tasks from hosts.

The Scheduler consists of three main components: a Data Manager, a Task Manager and a Dispatcher with queue. The Task Manager looks for jobs from the database/job pool and when it chooses a job it invokes the Data Manager to fetch the required data. Following this, the Host Manager is queried to find an idle host and these three pieces of information are placed on the queue. The queue follows a FIFO scheme where the individual component tasks of the job are sent to the Host Manager which then assigns it to a host for execution.

The host manager is defined as a CORBA module as shown in Figure 5. The first two methods allow hosts to register and deregister themselves with the system. Of note would be the *HostCallbacks* object which is actually a remote reference to a CORBA object running on the host. This is used by the system to enable hosts to perform asynchronous communication with the broker. We will cover this aspect in greater detail later in this chapter. The host's IP address is kept by the broker to enable it to find out if the host starts more than one instance of the host applet on the same machine. We allow the same host account to login to the system from different machines and tracking the IP address is just a rudimentary way of enforcing that policy.

```
module HostManager {
    interface HostLogin (
        boolean register_host(in string id, in string password,
            in string host_ipaddr, in Host::HostCallbacks thehost);
        boolean unregister_host(in string id, in string password,
            in string host_ipaddr);
        taskinfo get_task(in string id);
        void task_done(in long long process_id, in data processed_data,
            in string hostid);
    };
};
```

Figure 5 IDL Definition for Broker

The last two methods are used by hosts to request for a task and to return results to the broker. The current implementation uses a taskinfo data structure is shown in Figure 6. Section 3.4 contains more details on the task model employed in Unison.

This data structure just contains two members: the process ID of the task to be run, and the actual URL that the host's browser is supposed to load. Details about how the host keeps information about itself between applet switches will be given in a later section. The *data* type is just an array of bytes containing results produced by the host. The scheduler will be described in detail in section 3.5.

3.3.2 Host

Hosts constitute a source of computing power for tasks submitted to Unison. They are able to share CPU time between Unison tasks and other applications running on the same machine without causing undue interference in the operation of these other tasks. By interference we refer to i) The consumption of CPU cycles to the extent of causing other user applications to have a sluggish response and ii) System memory use is constrained by the limits imposed on the browser JVM. The Host Applet has to be explicitly started; we do not require users to run this applet for a minimum duration. If the running of the Host Applet affects the user's machine's performance then he can choose freely to discontinue running of the applet.

Host States

Hosts exist in four states namely **Disjoint**, **Busy**, **Ready** and **Running**. These correspond to situations when they are: not connected to Unison; loading a new applet; ready to accept a task and running a Unison task respectively. These states are shown in Figure 7.

Disjoint hosts that encounter the system initially have to login via a HTML form linked to a servlet. The transition from **disjoint** to **busy** happens when authentication with the database is successful. The very first action the host performs after this is to load a host applet which has an empty `run()` method. This applet knows that it does not possess a valid task by checking that its process ID encoded as a `PARAM` in the applet HTML file is zero. It then proceeds to request for a task from the broker by invoking the host manager's `get_task()` method with its host ID as the only parameter. The host enters into a **ready** state once the applet (without a valid task) has been loaded. The second applet causes its `run()` method to be invoked and this implies a transition to the **running** state. When this method has run to completion, the host returns back to the **ready** state and stays there until the broker instructs it to invoke `run_task()`. The host may re-enter into a **disjoint** state if its connection to the broker is severed or interrupted, in which case the broker will have to handle this by timing out and reissuing the task.

```
struct taskinfo {
    long long process-id;
    string url;
};
```

Figure 6 IDL Definition for Task Information Objects

Figure 7 State Transition Diagram for Unison Hosts

Host Architecture

Unison hosts run an applet that is launched from a HTML page generated by a servlet. We use this servlet to generate a HTML page containing startup parameters needed by the applet. The applet itself contains a CORBA callback object.

Applets are chosen for two reasons, namely, the ease of deployment and the ability to run untrusted programs in a trusted environment. Most popular browsers typically include a Java virtual machine hence eliminating the need for additional software to be installed on the host machine. It is our intention to build on the security provided by an applet's sandbox to run our applications. This is the way in which we ensure that Unison applications are able to access the host computing resources and not other resources such as its local storage media.

The callback reference is passed to the broker during the login process. It provides methods to access information about the host such as its state, the type of task currently being executed, the task ID, and host ID. Furthermore, it enables hosts to listen for commands from brokers that may instruct it to start execution of a new Unison task or to stop execution and switch to another task. Future versions of Unison will allow hosts to save the state of the running task so that they can resume execution of this task after preemption.

The initial prototype used valuetype objects to represent tasks. A valuetype object contains programming logic and data; it is another way to get code transferred and run on the host. Unlike traditional CORBA variables which are remote references, valuetypes enable the invoker to

receive a construct similar to a data structure but also containing program code in addition to data. However at this point in time few CORBA vendors support this CORBA data type in their products. Furthermore we have faced difficulty in using the correct factory object to instantiate the valuetype on the host side. This factory object is implemented and specified within the code using the `register-value-factory()` method. There are two ways of getting a new task's code to be run:

1. Dynamically edit and compile the applet source code to insert the class name of the new task whenever the host switches tasks.
2. Overwrite the class file with new task's class file but renaming it so that there is no need to edit the applet source.

The first method was undesirable since we want to avoid having to recompile the applet code every time a new task is required while the second is tedious because of file manipulation and storage. In either case the advantage of cleaner code is negated by the need for the broker to manage task class files.

We found the mixture of pass by value and pass by reference semantics in Interface Definition Language (IDL) [OMG02a, OMG02b] undesirable. IDL is technology-independent syntax for describing object encapsulations. IDL is only a definition language; mappings to several programming languages (e.g. Java, C++, Ada, COBOL, etc) are provided. The interfaces have attributes and operation signatures. Each interface specified in the IDL, specifies the operation signatures through which requests are made. The IDL supports inheritance to promote reuse. Its specifications are compiled into header files and stub and skeleton programs. Stubs are used as proxy objects (of the server) on clients while skeletons are part of the build process for the server application.

We decided not to adopt this approach using valuetypes but instead to provide a template Java class for the client to fill in his program logic. We also generate a HTML file so that hosts may run the applet.

Host Implementation

The host is implemented as a CORBA/Java applet. We have limited the applet code such that it does not use features introduced after Java 1.1. It has been tested on the Java 1.1 JVMs present in browsers such as Netscape Communicator and Internet Explorer, as well as Java 1.3.1 via the Java

plugin. Java applets cannot rely on the classpath set on the machine but on one of the following instead:

- All required class files are placed in the same directory.
- All required Java packages are based on the same directory as the applet's class file.
- All class files are stored in a Java archive (jar) file referenced by the ARCHIVE tag in the applet's HTML file.

As we require both our Unison package as well as the CORBA classes provided by Visibroker to be present and due to the limitation of only being able to specify a single jar file in the ARCHIVE tag, we had to use the third method. The components of the host applet are shown in Figure 8. Interaction with the broker is mainly through the ORB except for the socket connection that is used by the broker. This connection is torn down after the broker ascertains the host's IP address but may be kept throughout the host's lifetime to enable us to test for host's presence.

Figure 8 Host Components

The host applet itself is defined as a CORBA module in IDL as shown in Figure 9:


```

module Host (
    interface HostCallbacks (
        boolean run-task 0;
    );
};

```

Figure 9 IDL Definition for Hosts

The only method *run-task0* is used by the broker as a callback to instruct the host to request for a task by calling the broker's *get_task()* method. The sequence of events that occur when a host joins the system is shown in Figure 10.

3.3.3 Client

Unison clients come in the form of Java servlets that present the user with a web interface to a broker. The actions that a client may take are shown in Figure 11. A client is first presented with a main menu shown in Figure 12. It then has to register itself first as shown in Figure 13. It can then perform these actions: naming and submitting jobs (see Figure 14); uploading class files corresponding to user-defined computational tasks (see Figure 15); uploading one or more data files; viewing the status of submitted tasks and retrieving processed data.

Submission of task requests and required data files is done via a Java upload servlet. The broker checks the database periodically for the presence of new tasks. This is a requirement of our current system as we sought to defer implementation of an application directly linked to the database to a later stage.

Results are returned by hosts to the broker's *HostManager* which then proceeds to write the results out to a file and transfers it to the database so that it can be retrieved later on by the client. The Host Manager and hence broker are decoupled from the client-side job submission/result collection process. Therefore clients do not receive notification of job completion and have to login to the system to check whether their jobs have been completed.

Figure 10 Host Flowchart

Figure 11 ClientFlowchart

Figure 12 Main Menu

A screenshot of a web browser window displaying the 'Client Registration' form. The browser's address bar shows the URL 'http://155.69.146.85:8080/clientServlet/'. The form contains several input fields: 'Username', 'Password', 'Confirm Password', 'Name', 'Address', and 'Phone'. The 'Gender' field has two radio buttons labeled 'Male' and 'Female'. At the bottom of the form are 'Submit' and 'Reset' buttons. The browser interface includes a menu bar (File, Edit, View, Go, Communicator, Help) and a toolbar with various navigation icons.

Figure 13 Client Registration Menu

Figure 14 Client Job Submission Menu

Figure 15 Client Task Submission Menu

3.4 Task Model

In the context of Unison, a job represents an entire Unison application. It encompasses a number of tasks. Unlike other parallel computing systems such as PVM, tasks are Java threads and not processes. Unison tasks follow the Single Program Multiple Data (SPMD) programming style.

Unison supports a generic framework for jobs submitted by clients. It is designed to support a variety of applications that may perform a series of computations on the same set of data or just transform a single piece of data. Examples of such applications include Mersenne primality testing and ray tracing. Any work submitted by a client to Unison is termed as a *job*. A job may

consist of one or more *tasks*. A task is an independent sequential thread of execution that does not rely on data from other tasks. The system allows the client to specify how many *stages* are present for a particular job. A stage is a logical grouping of tasks that share a common data input from the broker. It is a means by which the system coordinates the collection of partially computed results from hosts. It helps a broker to synchronize data generation for the next stage by ensuring that all the inputs for the next stage are ready before they are issued to hosts for processing. The 4 different kinds of jobs that are handled by Unison are shown in Figure 16.

Figure 16 Job-Task Relationship

The parallel situation in Figure 16 shows the tasks that may be parallelized by the broker. The granularity of data input for these tasks can either be determined by the broker, in which case the broker treats the task as a piecewise computation and divides the data into roughly equal sized chunks, or the client may specify the number of pieces the data may be divided into. Tasks that share common data input follow a SPMD programming model in which the same code is used to run different portions of data. The broker is responsible for concatenating results from hosts into a single unit before passing on to tasks in the next stage.

In Unison all tasks are Java threads. We limit the execution of tasks on each host to only a single task. This is essential because we cannot rely on the priority specification for threads in Java. It is well known that thread behaviour is dependent on the underlying operating system (for native threads). Having many Unison applications run as multiple threads on each host is undesirable because of the problem of degraded system performance due to concurrent thread execution.

By employing this model, Unison is able to handle multistage jobs easily. A single job can thus consist of many operations for which the underlying complexity is handled by brokers. The whole process is transparent to hosts. And in doing so it affords a great degree of flexibility to users of the system.

3.5 The Scheduler

This section describes how the scheduler works in Unison brokers. It examines issues that affect job scheduling such as host classification and data granularity determination. The scheduler is a logical grouping of entities such as the task and data managers. These entities are not CORBA objects as the scheduler is designed to run on a single machine. A list of some scheduling terms can be found in Appendix A.

The scheduler handles the fetching of jobs, tasks and data from the database. The part of the scheduler that interacts with the host manager is defined as a CORBA module as shown in Figure 17.

```
module Scheduler {
    interface SchedulerControl {
        taskinfo get_task(in string id);
        void ret_data(in long long process-id,
                     in data processed-data, in string hostid);
    };
};
```

Figure 17 IDL definition for scheduler

The scheduler's two methods mirror the methods of the host manager because it is the only component in the broker that is allowed direct access to the job and data pools. These pools are implemented as circular queues. Instead of dequeuing each item from the job queue whenever it is issued, we choose to either set or unset a flag variable. The item is only removed from the queue

when results have been returned from the host. Extensions of the system to incorporate eager scheduling and to handle host failures are possible. This can be done by reissuing the same queue items. Jobs on the repository itself are issued on a First Come First Serve (FCFS) basis to brokers by the database. In addition to that, jobs with a smaller job ID are issued first thus ensuring that older jobs are given priority over newer ones.

Host Classification

We have determined some useful benchmark metrics by which hosts may be classified. Examples include these criteria:

- **CPU speed (F_{CPU})**

This can be determined using simple timed loops or Linpack [DJJ] results. The benchmark will be run multiple times in order to average out periods of intense CPU activity on part of other applications on the system. The results can then be compared with the established database of Linpack results to give a rough estimate of CPU speed.

- **Link Latency to Broker (T_{Link})**

The COMMS1, or pingpong, communication benchmark from the Parkbench suite [HB94] can be used to test link latencies between brokers and hosts.

- **Link Bandwidth (BW_{Link})**

The COMMS3 benchmark, also from the Parkbench suite, is used to test the saturation bandwidth of the network. This is done by getting every node in the network to send messages to every other node and finding the time taken for all the nodes to finish receiving all messages sent to it.

This benchmark is not currently feasible for implementation in our system since it requires the ability for applets to send data directly to each other. However we could still include a modified version of the benchmark in the future that uses the broker as a proxy for data sent between hosts.

- **Reliability (R)**

We are interested in recording how often a host disconnects from the network. This is more of an indication of the reliability of a host's connection to the Internet or the inclination of dialup users to join the system. We intend to implement a means of quickly

testing results to see if hosts try to falsify them. This test must out of necessity be of a short duration if the broker is to be able to test results from many hosts.

Other historical data that can be kept on the host may include how often the host accepts a task but fails to provide results within a certain timeframe and whether a client intends to cheat the system by acting as a host from the same machine.

- **Geographical Locality (L)**

Geographical closeness of hosts to the broker is strongly tied with host link latency. Hosts and brokers within the same region usually enjoy fast links with high bandwidth compared to hosts connected via WAN to remotely located brokers. We should exploit this locality to avoid sending jobs with large datasets over slow WAN links and thus improve job throughput of our system. We could determine the location of a host by:

1. Finding the time zone (TZ) of the computer by using the *Calendar* class. However this is dependent on the user setting the correct time zone on his computer.
2. Looking at the IP address of the host, but this is not an exact indicator.

Hosts can then be classified according to this equation shown in (1).

$$Class_{host} = k_1 F_{CPU} + k_2 \overline{T_{Link}} + k_3 BW_{Link} + k_4 R + k_5 L'$$

where

$$L' = |TZ_{Broker} - TZ_{Host}|$$

Equation 1 Host Characterization Equation

A $Class_{Host}$ value is unique to each broker. It is an indication of how a host relates to a broker since it contains broker-host specific values such as those mentioned above (except F_{CPU}). The values of k_i , i in $[1..5]$ will have to be determined either by experimentation or simulation.

In our current implementation, brokers only use F_{CPU} to sort hosts so that “better” hosts can be allocated tasks earlier than slower ones. The difficulty lies in determining how well a rating given to a host reflects its computer capability and user behavior.

Host Sorting Algorithm

To aid in host classification we define a host sorting algorithm. The host queue object provides the host sorting method. It calls each hosts' `get_bmresult()` to obtain the latest averaged Linpack result. This algorithm is run at periodic intervals that can be set by the broker administrator. What it does is to reorder hosts in the queue so that the fastest hosts are placed near the head of the queue.

```

1.   For each host
    a. Call callback object's get_bmresult()
2.   End for
3.   Create a stack host-stack
4.   For the first host
    a. tmp_min = host.bmresult
5.   While host queue not empty
    a. For each host
        i. If host.bmresult < tmp_min
            1. tmp_min = host.bmresult
        ii. Go to next host
    b. End for
    c. For each host
        i. If host.bmresult == tmp_min
            1. Push host onto host-stack
            2. Remove host from queue
            3. break
        ii. End if
    d. End for
6.   End while
7.   While host stack not empty
    a. Pop host from stack
    b. Enqueue host on host queue
8.   End while

```

Data Granularity Determination

The client is allowed to enter the number of pieces into which the submitted data file may be split before distribution to hosts as shown in Figure 15. This means that brokers do not have to analyze the contents of these files and just simply rely on the client's input to partition data into roughly equal sized pieces. This is a form of domain decomposition. The broker can override the data

granularity value entered by the client, as it is only a suggestion. Data objects are placed in a simple linked list which the broker accesses by following the associated job and task IDs.

Task Representation

Tasks are represented in the system as Java *Task* objects in a double-linked list. The scheduler keeps a reference to the current task being processed as well as to the current stage to be processed for each job. As shown in Figure 18, the first *Task* objects of each job are linked in a circular queue. As each task is dispatched to the host manager, the next task to be processed is found by advancing the reference to the current task reference of the next job in the queue. The current task reference is only moved when all the results for that stage have been received.

Figure 18 Task Queue

Task Updater Algorithm

This algorithm is used by the broker to keep its repository filled with jobs. The values MIN_JOBS and MAX_JOBS are set by the broker administrator and these determine when to get new jobs from the database and the limit to the number of jobs that a broker can handle respectively.

```

While broker runs
  1.   If num-jobs < MIN-JOBS
      a. While num-jobs < MAX-JOBS
          i. Get job from database
          ii. num_jobs++
      b. End while
  2.   End if
End while

```

Broker Scheduling Algorithm

We define a scheduling algorithm used by brokers. The client supplies the value *min_granularity*. The broker is able to take advantage of all hosts under its control if the number of hosts is at least equal to the minimum data granularity. In this case it adjusts the granularity so that equal portions of the data set *are* distributed among the hosts. If the broker does not have enough hosts to satisfy this minimum requirement, then the scheduler proceeds to examine the next job from the queue. The job that requires more hosts than what the system *is* able to provide at that point in time will have to wait until enough hosts join the system before it can be run.

```

While broker is running
  1.   While num_of_idle_hosts > 0 and
      num_of_unfinished_jobs > 0
  2.   Get first incomplete job from queue
  3.   Get current stage for this job
  4.   If num_of_idle_hosts >= min_granularity of the stage
      a. Create num-of-idle-hosts copies of the classfile
      b. Tell Data Manager to split the data file into
         num_of_idle-hosts pieces
      c. Call Host Queue to sort hosts according to bmresult
      d. For num_of_idle_hosts
          i. Dequeue host from host queue
          ii. Generate index.html file
          iii. Call host's run-task() passing it the URL in a
              taskinfo object
      e. End for
  5.   End if
  6.   Get next job from queue
  7.   End While
End while

```

The broker keeps track of the tasks that have yet to be completed. This information is stored in task tables such as the one shown in Table 3. Task tables show the current task each host is executing and whether the task has run to completion or not. Job tables shown in Table 4 relate task IDs to jobs. The broker uses these tables to determine which tasks belong to a particular job and whether the job is complete. Completed jobs and their associated tasks are removed from these tables as well as the relevant queue structures.

3.6 Unison Programming Library

The Unison programming interface is shown in Table 5. A Unison application uses these methods to perform all the operations required during execution of its component tasks. The Unison library is for Java programming of SPMD style applications over networked computers. It can also run on other kinds of architectures such as shared memory multiprocessor machines as long as the platform has a JVM.

Host ID	Task ID	Status
undamaris	0	Completed
voceumana	1	Incomplete
spitzflote	2	Incomplete
principale	3	Incomplete

Table 4 Job Table

Job ID	TaskID	Data ID
0	0	0
0		1
1	2	
1		

CLASSIFICATION	OPERATION	DEFINITION
Initialization	<i>init()</i>	Get parameters from HTML page
Command	<i>run-task0</i>	Broker tells host to run a task
Request	<i>get-task()</i>	Host requests for a task
Submission	<i>task-done()</i>	Result submission after task completion

Initialization

Besides passing data and parameters via method invocations, hosts can also retrieve data from the HTML page on which the host applet originates. Java applet loading mechanism also has support for multiple parameters. This greatly simplifies data parameter passing from the broker.

Command

Instead of waiting for a host to request for work, a broker may use this method to tell the host to run a particular task. In our current implementation use of this method is avoided but it can be incorporated in later versions of Unison so that brokers may adopt a “push” model.

Request

A host uses this method to request for tasks from the broker. It passes the host ID to the broker and receives a task information object (see Figure 6) in return.

Submission

Invoking this method signals task completion. The parameters include the host ID, task ID and array containing the results.

CHAPTER 4 EXPERIMENT RESULTS

To evaluate the performance of Unison we have implemented three benchmark applications. These applications were developed and run on both Unison and Locust [MAY99]. They cover fields such as: Local alignment, a procedure in which similarities between sequences can be found. This has a common application in bioinformatics and life sciences; Mersenne prime search, inspired by the famous Internet-based effort of the same name; Raytracing, which can involve huge numbers of complex mathematical operations. These applications are described in detail in section 4.1.

The performance results obtained from running these applications are analysed in section 4.2. Both Locust and Unison are applet based systems, however they use different communication libraries: UDP from the *juvu.net* package for Locust and CORBA method invocations by Unison. Section 4.3 describes a series of benchmarks to test the effect of different JVMs on application performance.

4.1 Benchmark Implementation

4.1.1 Local Alignment Benchmark

An alignment refers to the procedure of comparing two or more sequences by looking for a series of individual characters or character patterns that are in the same order in the sequences. This comparison is carried out on either Deoxyribonucleic Acid (DNA) or protein sequences and either between two sequences (pairwise sequence alignment) or between many sequences (multiple sequence alignment). Our work is primarily concerned with pairwise alignment.

Pairwise sequence comparison involves looking at nucleotides (for DNA sequences) or amino acids (for protein sequences) and trying to find a new arrangement (alignment) of these residues such that the two new sequences show a certain degree of homology. Sequence homology is an indication of evolutionary relatedness among sequences. This alignment is then scored based on the number of identical or similar residues modified by the insertion of gaps.

A higher score is a sign of stronger alignment between sequences and the greater likelihood they are to be related. Very similar sequences which are almost identical along their lengths almost certainly have the same function (physiological activity of an organ or body part). Sequences which are only weakly similar may or may not be related, and no firm conclusions can be drawn about their relationship.

We find that the bulk of organisms share many similar sequences but that some sequences are also quite unique to an organism or group of organisms. The shared sequences can all be compared to each other to try and define the common domains that provide a particular function, and in other cases, to predict which of the organisms are most closely related based upon the degree of homology.

Sequence comparison is a useful technique because it confers the ability to link the biological function to the genome and to propagate known information from one genome to another. By being able to determine the percentage of sequence homology between two sequences, we are able to determine if they are derived from a common ancestral sequence and hence a common ancestor.

There are three main methods for performing alignment, they are:

1) Dot Matrix

In this method of sequence comparison, one sequence (A) is listed across the top of a page and the other sequence (B) is listed down the left side. Starting with the first character in B, one then moves across the page keeping in the first row and placing a dot in any column where the character in A is the same. The second character in B is then compared to the entire A sequence, and a dot is placed in row 2 wherever a match occurs. This process is continued until the page is filled with dots representing all the possible matches of A characters with B characters. Any region of similar sequence is revealed by a diagonal row of dots. Isolated dots not on the diagonal represent random matches which are probably not related to any significant alignment. Other descriptions of the dot matrix method may be found at [STA91, GIBB70].

2) Word (k-tuple) Search

An example of this is the heuristic sequence alignment algorithm FASTA [PL88]. FASTA first searches for short ordered sequences of k residues (k-tuples or ktups) that occur in both the query sequence and the sequence database. The algorithm then scores the 10 ungapped alignments that contain the most identical k-tuples. These ungapped

alignments are then tested for their ability to be merged into a gapped alignment without reducing their score below a certain threshold. For those merged alignments that score over the threshold, an optimal local alignment of that region is then computed and the score reported.

3) Dynamic Programming

Saul Needleman and Christian Wunsch first introduced the concept of dynamic programming to biological sequence comparison [NW70]. Dynamic programming is an algorithm technique that takes advantage of optimal substructure in problems, i.e. decompose a large problem into a number of small problems. Solve the small problems and use these to solve the large problem. It solves optimization problems in which a large number of solutions exist but only one or a few are best. In the context of local sequence alignment, dynamic programming algorithms work by solving subproblems, storing each intermediate result in a table along with the score and finally choosing the sequence of solutions that yields the highest score. The result is a new sequence where the number of high scoring residue pairs are maximized and the number of gaps and low scoring pairs minimized.

Two algorithms exist for pairwise sequence comparison: Needleman-Wunsch and Smith-Waterman [SW81]. The former is used to find global alignments while the latter finds local alignments. Named after its inventors Temple Smith and Michael Waterman, this algorithm differs from the Needleman-Wunsch algorithm in the way traceback is performed. Local alignments are not required to extend over the entire length of both sequences; traceback starts from the highest similarity score in the matrix, not necessarily from the bottom right value.

What we have done is to integrate into our framework a program that not only implements the Smith-Waterman algorithm but also applies it to database searches. This program works in 2 phases:

- a. Finding a similarity score
- b. Traceback to get the alignment

It maps into our framework in this manner: (a) is parallelized by distributing the protein sequences from the database to hosts; while (b) is run on a single arbitrary machine. As a test we used sequences from “genpept.fsa” taken from GenBank¹⁰ [BKL02].

¹⁰ <http://www.ncbi.nlm.nih.gov/Genbank/index.html> Online, as of April 2003)

An example of a protein sequence described in FASTA (Pearson) format is shown in Figure 19. We used a FASTA file as our test database. This is just a flat file of records where the first line prefaced by the ‘>’ character contains human readable information about the origin of the sequence followed by one or more lines containing the protein sequence.

```
>gi|1246879|emb|CAA01779.1| (A25901) XPR2 [Yarrowia lipolytica]
MKLATAFTILTAVLAAPLAAPAPAPDAAPAAVPEGPAAAAYSSILSVVAKQSKKFKHHKRDEPL
EST
```

Figure 19 Example of sequence in FASTA format

The set of similarity scores $H(i, j)$ is found using the following set of recurrence equations in (2) for two sequences $S1$ and $S2$ where $S1$ is the query sequence and $S2$ is a subject sequence from the database :

$$\begin{aligned}
 |S1| &= m \\
 |S2| &= n \\
 H(i, j) &= \max \begin{cases} 0 \\ E(i, j) \\ F(i, j) \\ H(i-1, j-1) + St(S1_i, S2_j) \end{cases}, 1 \leq i \leq m, 1 \leq j \leq n \\
 E(i, j) &= \max \begin{cases} H(i, j-1) - \alpha \\ E(i, j-1) - \beta \end{cases}, 0 \leq i \leq m, 1 \leq j \leq n \\
 F(i, j) &= \max \begin{cases} H(i, j-1) - \alpha \\ F(i-1, j) - \beta \end{cases}, 1 \leq i \leq m, 1 \leq j \leq n
 \end{aligned}$$

Equation 2 Recurrence equations for finding $H(i, j)$

$H(i, j)$ is modified by two main factors: probability of substitutions and by insertion and deletion of residues.

Where $St(S1_i, S2_j)$ is found by looking up the corresponding residue pair from a substitution table, in this case the Blocks Substitution Matrix (BLOSUM62). This matrix is used for scoring protein sequence alignments [HH92].

The values in the BLOSUM62 matrix are based on the observed amino acid substitutions in a large set of approximately 2000 conserved amino acid patterns, called blocks. They are logarithms of ratios of two probabilities: the probability of meaningful occurrence of a pair of residues *in* the alignment, and that of random occurrence of an amino acid in a sequence alignment. A negative value is an indication that the probability of random occurrence is greater than that for meaningful occurrence. This can then be used to modify the similarity score. BLOSUM matrices are derived from the Blocks database which is a set of ungapped alignments of sequence regions from families of related proteins. These sequences are sorted or clustered into closely related groups. The numerical value associated with a BLOSUM matrix represents the cutoff value for the clustering step, e.g., a value of 62 indicates that sequences will be put into the same cluster if they are more than 62% identical. Figure 20 shows the BLOSUM62 matrix.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	5

Sequences change not only by point mutation but also by insertion and deletion of residues. In order to get meaningful alignments, we need to introduce gaps into either or both of the sequences being aligned. Gap penalties are expressed in the algorithm by α and β , which refer to the gap

opening and gap extension penalty respectively. We used $\alpha = 10$ and $\beta = 2$ for our experiment. These penalties are affine, meaning that the cost of opening a gap is different from the cost of extending a gap. A higher gap opening penalty makes it costly to add gaps to the alignment so that gaps only appear where they are essential. The gap opening penalty also tends to be higher than associated extension penalty. This is a reflection of the fact that insertions and deletions tend to occur over several residues at a time.

We work out an example showing how local alignment is performed:

S1 = PAWHEAE

S2 = HEAGAWGHEE

Using BLOSUM62

and $\alpha = 1$, $\beta = 1$

The algorithm runs in three phases:

1. Initialization

$$H(i,0) = E(i,0) = 0, 0 \leq i \leq m$$

$$H(0,j) = F(0,j) = 0, 0 \leq j \leq n$$

The top and leftmost rows of the H , E and F matrices are set to zero, as shown in Figure 21. Position $H(i, 0)$ refers to the alignment of suffix $S1_{1..i}$ to an empty sequence and likewise $H(0, j)$ refers to the alignment of suffix $S2_{1..j}$ to an empty sequence.

	0	H	E	A	G	A	W	G	H	E	E
0	0	0	0	0	0	0	0	0	0	0	0
P	0										
A	0										
W	0										
H	0										
E	0										
A	0										
E	0										

Figure 21 Initialization of H matrix

2. Calculating $H(i, j)$

This is done according to the recurrence equations given in Equation 2. The H , E and F matrices are filled from downwards and towards the right as shown in Figure 22.

3. Traceback

Traceback starts from the position of the largest H score and ends at a cell with a zero score. At every step there are three choices of which pointer to follow. Moving from position (i,j) to (i-1,j) implies the insertion of a gap in sequence S2. Similarly, moving from position (i,j) to (i,j-1) means that a gap is placed in sequence S1. This is shown in Figure 23 where traceback starts from the highest H score of 18. The resulting local alignment is given in Figure 24.

	0	H	E	A	G	A	W	G	H	E	E
0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	4	0	4	0	0	0	0	0
W	0	0	0	0	2	0	15	5	3	1	0
H	0	8	0	0	0	0	5	13	13	3	1
E	0	0	13	3	1	0	3	3	13	18	8
A	0	0	3	17	7	5	3	3	3	12	17
E	0	0	5	7	15	6	3	1	3	8	17

Figure 22 Calculation of H values

	0	H	E	A	G	A	W	G	H	E	E
0	0	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	4	0	4	0	0	0	0	0
W	0	0	0	0	2	0	15	5	3	1	0
H	0	8	0	0	0	0	5	13	13	3	1
E	0	0	13	3	1	0	3	3	13	18	8
A	0	0	3	17	7	5	3	3	3	12	17
E	0	0	5	7	15	6	3	1	3	8	17

Figure 23 Traceback to get local alignment

AWGHE
AW-HE

Figure 24 The local alignment

The test was run on subject sequences from *Yarrowia lipolytica* extracted from *genpept.fsa* (translated proteins from GenBank [BKL02]) using a parser written by us. The query sequence was taken from the same file and belongs to *Saccharomyces cerevisiae* (commonly known as baker's yeast, one of the first unicellular eukaryotic organisms to be completely sequenced). The

computational model employed is shown in Figure 25 where sequence of actions taken is as follows:

1. Unison is started with the query sequence and organism name given from the command line.
2. All sequences belong to this organism are extracted from the database using our parser.
3. All hosts are initialized with the query sequence.
4. Host requests for a subject sequence.
5. Host returns the score for a particular iteration of the local alignment algorithm.

Steps 4 and 5 are repeated until the pool of subject sequences is exhausted. For the sake of simplicity we limit requests for subject sequences to return only a single sequence.

Figure 25 Computational model for local alignment benchmark

There are two kinds of issues that we have to address. The first has to do with iterations of the local alignment algorithm, while the other is related to the database searching nature of our implementation.

Running this algorithm can produce multiple matrix values with the same highest H score. Also, when doing traceback it is possible to have more than one possible path to take, i.e., both diagonal and left cells hold the same maximum H value.

Database searching entails running the algorithm hundreds, if not thousands of times over many sequences. The question is how many high scoring local alignments from the sequence database we should return in answer to a query. This problem is compounded by the possibility of multiple high scoring alignments from each sequence. What we do is to look for the highest score amongst all the alignments of the subject sequences. If there is more than one alignment sharing the same high score we return the first alignment to be computed.

4.1.2 Mersenne Prime Search Benchmark

In 1644, the French monk Marin Mersenne conjectured that numbers of the form $2^p - 1$ were prime for certain prime numbers p . Specifically, he stated in the preface to his *Cogitata Physica-Mathematica* that the numbers $2^p - 1$ were prime for:

$p = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127$ and 257

These numbers are composite for all other positive integers $p < 257$. He was later proved to be wrong for 67 and 257, as well as having missed out 89 and 107. There has been much mathematical research devoted to the discovery of Mersenne primes. A large scale Internet based search for Mersenne primes called the Great Internet Mersenne Prime Search (GIMPS) has been running since 1996 and has led to the discovery of several new Mersenne Primes [WOL].

Mersenne primes can be found using the following theorem in Figure 26 (the proof can be found in [BRU93]).

For p odd, the Mersenne number $2^p - 1$ is prime if and only if $2^p - 1$ divides $S(p-1)$ where $S(n+1) = S(n)^2 - 2$ and $S(1) = 4$.

Figure 26 The Lucas-Lehmer primality test

The theory for this test was initiated by Lucas in the late 1870's and then made into this simple test about 1930 by Lehmer. The sequence $S(n)$ is computed modulo $2^p - 1$ to save time. This test is ideal for binary computers because the division by $2^p - 1$ (in binary) can be done using rotation and addition only. The pseudo code used in our benchmark is shown in Figure 27.

We implemented the Lucas-Lehmer primality test in the Unison framework, with each unit of computation consisting of a single prime p value that was used to create candidate Mersenne

numbers 2^p-1 to test for primality. These values were then distributed to the hosts which ran this test and returned a true or false value depending on the outcome of the test

```

Lucas-Lehmer-Test(p) :
    s := 4;
    for i from 3 to p do s := s^2-2 mod 2^p-1;
    if s == 0 then
        2^p-1 is prime
    else
        2^p-1 is composite;
    
```

Figure 27 Pseudo code for Lucas-Lehmer test

The goal of this application was to test the functionality and performance of Unison rather than to discover new Mersenne primes. Therefore we limit testing to values of p for which the primality of 2^p-1 is already known. This allows the tests to complete within a reasonable timeframe on the order of minutes compared to years for p in the untested range.

Figure 28 Computational model for Mersenne prime search benchmark

The benchmark is run on Unison following the model shown in Figure 28. The broker receives the start and end exponent for the range of values of p to be tested from the command line. The hosts keep requesting for exponents until the entire range of exponents has been tested.

4.1.3 Raytracing Benchmark

We have ported a sequential raytracer written in Java to Unison as an example of a real world application that can benefit from additional compute resources. This raytracer was written by

Frederico de Moraes from the Universidade Estadual de Campinas which was first parallelized by Laurence Vanhelsuwe as part of his DAMPP system [VAN97].

Raytracing begins from a scene defined in a scene description language or mathematical description. Photorealistic synthetic images can be generated via raytracing. A scene is defined by the camera, light sources and objects built from textured primitives such as spheres, cones and cubes. As the name implies, raytracing software mathematically traces rays of light as they bounce around the scene, reflecting and refracting until they reach the lens of the imaginary camera. This involves millions of floating point operations which can take anywhere from minutes to days to complete. Each imaginary beam of light going through a pixel of the image is traced independently; therefore we can parallelize the process by distributing sub regions of the image to each host.

Figure 29 Test scene for raytracing benchmark

The scene to be rendered measures 600 by 480 pixels and we divide it into scanlines. Each scanline consists of all the pixels of the same row thus creating 600 work units. The scene is shown in Figure 29 and it consists of 98 spheres and a plane.

4.2 Result Analysis

The experimental setup consists of eight Intel-based personal computers connected by a 100 megabit Ethernet network. Each PC has a 500 MHz Pentium III CPU and 128 MB of RAM. The operating system is RedHat Linux 7.2 while the version of the JDK used to run the brokers is 1.3.1 _02 with the HotSpot client virtual machine turned on by default. Host applets are tested on Netscape Communicator 4.77 using the built-in JRE ^{1.1.5}. The *CORBA* package is Visibroker for Java 4.5 and the web server Apache 1.3.12.

We assume these conditions to hold true throughout the execution of the benchmarks:

1. Each PC runs only benchmark programs.
2. No other resource intensive processes run on any PC during benchmarking.
3. Measurements taken include applet startup time for all hosts except for the first host that establishes contact with the broker.
4. Time taken for code distribution, data transfer and result uploading is also included in our measurements.

The performance metric used for these experiments is execution time from which another metric speedup can be found. Speedup is traditionally measured on a dedicated parallel multiprocessor where all processors are homogeneous both in hardware and software configuration. Speedup can be defined as:

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

Where T_1 is the time taken to run the program on a single processor and T_p is the execution time of the same program on p processors. This definition still holds true in our setup where we have kept all processors and software homogeneous in order for the measurements to be meaningful.

We utilized the framework provided for writing applications to be run on Locust to port these benchmarks. In this way we are able to make a comparison of the performance of these two Java applet based volunteer computing systems.

4.2.1 Mersenne Primality Test Results

The Mersenne prime search benchmark works on a range of integers that are used as exponents for floating point operations. There is minimal communication involved as a data request only returns an integer while the transmission of a result only involves up to five bytes. Therefore this benchmark can serve as a gauge of the systems' computational capabilities.

Figure 30 shows execution time for the Mersenne primality benchmark running on Unison and Locust. Figure 31 gives us an idea of how well both systems scale as more hosts are added. It can be seen that the results are as expected for a divide-and-conquer style benchmark like this. Total execution time falls as the number of hosts increases and the amount of computation done by each host falls. The systems scale accordingly with performance nearing ideal levels in the case of Locust. This is consistent with the minimal amount of communication during execution of the benchmark.

Locust has the advantage until the total number of hosts reaches five. This is an indication of the limitation imposed by the rigid architecture of Locust where a fixed number of result receiver threads (three for this test) are created at startup. As the number of hosts in the system increases a bottleneck is created by hosts contending with each other to transmit results back to the server. This results in hosts spending time idling and waiting for their results to be uploaded. A way of alleviating this would be to improve the efficiency of Locust by allowing hosts to overlap computation with communication.

The slower timings from Unison in general are due to the relatively large overhead of repeated CORBA method invocations with small parameters as compared to simple UDP transmissions in Locust during data and result transfer.

Figure 30 Mersenne primality benchmark execution time

Figure 31 Speedup obtained for Mersenne primality benchmark

4.2.2 Local Alignment Benchmark *Results*

The local alignment benchmark works by running the Smith-Waterman algorithm on a database of sequences. It produces a score for each sequence; traceback is then performed on the highest scoring sequence.

Figure 32 shows execution time for this benchmark on Unison and Locust. Every iteration of the benchmark involves the transfer of a sequence from broker to host. This can range anywhere from 100 to over 1000 bytes. We intentionally kept granularity limited to a fine level of one sequence per request to test the impact of higher communication levels on the systems.

Figure 32 Local alignment execution time

As can be seen from Figure 33, the speedup achievable by Locust is again nearer that of the ideal than Unison. Locust also performs better than Unison even with five or more hosts in the system. The poorer performance of Unison can be attributed to the tradeoff between having a higher level programming abstraction with CORBA versus speed using UDP. Using CORBA as a communication mechanism inevitably introduces overhead such as time spent on data marshalling so that it can never be as efficient as the networking library already present in the **JDK**. In this test even the aforementioned bottleneck cannot bring Locust's performance down to Unison's level.

Figure 33 Achievable speedup for local alignment benchmark

4.2.3 Raytracing Benchmark Results

Execution time for the raytracing benchmark is shown in Figure 34 and the speedup obtained in Figure 35. As with the Mersenne primality test, the computation time of each piece of work depends on the complexity of the portion of the scene being traced.

The performance of Unison and Locust are almost identical for one to three hosts. It however degrades progressively for both systems from five hosts onwards. This is due to the nature of this application, the pool of 600 pieces of work is rapidly exhausted when five or more hosts are in the system. With the work pool exhausted, hosts are left idling and the execution timing is constrained by the completion time of the remaining unfinished work units.

As can be seen from Figure 35, Unison's performance is on par with Locust's when six or seven hosts are in the system. The Visibroker package on which Unison runs features a thread-pooling policy that offers flexibility in the way in which brokers handle host requests. A thread is assigned for each host request but only for the duration of that request. When the request is completed the thread is freed and placed back into a pool of available threads so that it can be reassigned to process future requests. This method is efficient because the overhead associated with thread

creation and destruction is reduced by reusing threads. New threads are created if all threads are busy. In this way Unison is not limited by a fixed number of threads as in the case of Locust.

Figure 34 Execution time for raytracing benchmark

Figure 35 Speedup obtained when running raytracing benchmark

4.3 Miscellaneous Benchmarks

We conducted these tests using modified versions of the local alignment and Mersenne primality benchmarks. The programs were adapted to run as Java applications and their performance when running on different Java virtual machines measured. A variety of JVMs from IBM and Sun were employed and the use of the Just-in-time (JIT) compiler was toggled between tests. Beginning from JDK 1.4.0 the option to run the JVM without a JIT has been removed. In both cases the applications were run only on a single machine so communication overhead is no longer a factor in the systems' performance. We focus on the efficiency of computational code, memory allocation and of I/O operations.

A JIT compiler was incorporated into the JDK since version 1.1 and that proved to be a major performance enhancement for Java programs with up to a ten-fold performance improvement compared to programs run on version 1.0 JVMs. The version 1.1.8 JVM is IBM's implementation while the rest are from Sun.

Figure 36 Execution time for local alignment benchmark on different JVMs

4.3.1 Effect of JVM Version on Local Alignment Application

Performance of SSEARCH is included for reference in the local alignment benchmark as shown in Figure 36. The speed of SSEARCH is up to 11.5 times faster than the best timing of our application on the 1.4.1-01 JVM. This result is expected as the FASTA package is written in C and optimized for several platforms including Linux. The Java core library also suffers from performance limitations [HN99]. This is caused mainly by excessive synchronization, excessive allocation and other programming inefficiencies. Many public methods in Java are *synchronized* so that classes can be thread-safe and the likelihood of the occurrence of race conditions is reduced. While this provides ease of use for multi-threaded programming, it does so at the expense of performance. In particular the *java.io* package that handles I/O streams uses a lot of fine-grained locking in its read/write operations. This has a direct performance impact on our local alignment benchmark as the Java program has to read a large file containing the subject sequences. This is an operation in which the C-based SSEARCH has a far superior advantage.

We can see that running without a JIT compiler imposes 7.5-fold and 8.2-fold performance degradation for versions 1.1.8 and 1.3.1-01 respectively. The HotSpot VM was introduced to replace the Exact VM that was included with version 1.2.0 JVM. It is of two types: client and server. These two HotSpot virtual machines are essentially two different JITs that interface with the same runtime system. The HotSpot server VM is especially useful in optimizing CPU-intensive code while the client VM is optimal for applications that need to start up quickly or that require a small memory footprint. In all cases where the HotSpot VMs were used, the server VM proved to be more suitable for our application with up to 1.4-fold speed improvement over the client VM. We also noticed that each new version of the JVM has added performance improvements such that the latest version (1.4.1-01) performs 1.3 times better than the HotSpot engine in version 1.3.1_02.

Figure 37 Execution time of Mersenne primality benchmark on different JVMs

4.3.2 Effect of JVM Version on Mersenne Primality Application

The results from the Mersenne primality benchmark are shown in Figure 37. These results are consistent with the findings of the previous test. The main difference lies in the superior performance of IBM's **JVM** compared to Sun's version 1.3.1-02 JVM even with the JIT compiler disabled for both. In fact the IBM JVM performs better than the 1.3.1-02 JVM, this is not unexpected as IBM's implementation has been known to perform better than the reference implementations.

The benefits of moving to newer JVMs can also apply to Unison if we adapt it to an application-based model. The introduction of the Java Network Launching Protocol (JNLP), WebStart and Java Plug-in pave the way for the rapid adoption of new JVMs. In particular JNLP and WebStart add an extra dimension to application deployment by allowing them to be started remotely and cached with version checking. Whereas the Java Plug-in finally overcomes the performance hurdle caused by the archaic JVM bundled with Netscape and Internet Explorer by replacing it with a newer JVM. Unison can also realize these benefits in future versions.

Figure 38 Effect of data granularity on execution time of Mersenne primality benchmark

4.3.3 Effect of Data Granularity on Mersenne Primality Application

Another test was run using the Mersenne primality benchmark on Unison with one, two and four host applets. Unlike the previous two tests communication time is a factor in performance measurements. We varied the size of each "chunk" (basically a range of p exponents) and observed the impact it had on the system's performance. Figure 38 shows the results obtained. It can be seen that when only one host participates, increasing the size of each chunk reduces benchmark execution time. The test with a chunk size of 1000 exponents runs 2.2% faster than the test with chunk size 10. This is caused by the ratio of computation to communication increasing due to the reduction in the number of method invocations for task requests and result uploading. However, the case of two and four hosts shows an anomaly: performance actually degrades when data granularity is coarse. Increasing chunk size reduces the number of tasks for hosts to work on. Computation time for each chunk isn't the same owing to the fact that larger exponents take a longer time to compute while those at the lower end of the range complete sooner. Therefore a host may start working on a chunk of smaller exponents and find that there is no work left after completing this chunk as other hosts are still processing the other pieces of work. The amount of parallelism is limited by the coarse granularity for this particular application. Even though

communication overhead is reduced, execution tends to become serialized and thus defeats the purpose of running the application on this system.

CHAPTER 5 CONCLUSION

5.1 Summary

This report presented a framework for tapping the latent potential of distributed computing by using common technologies like Java, web browsers and CORBA. It also presents some ways in which to handle new problems that arise.

We have looked into past and current efforts to provide a platform for running such applications. From the early projects involving multi-processor machines to clusters of workstations and from hence to pervasive computing using thousands of computers located at disparate corners of the Internet, we have seen the gradual emergence of the heightened role of the formerly lowly personal computer. As prices for personal computers continue to spiral downwards compared to largely expensive supercomputers, companies will seek to capitalize on the idea of using the Internet as a huge metacomputer.

The Unison framework allows clients located remotely to submit their programs to the system via a web interface. It then matches these programs with computers across the Internet. These computers participate simply by running an applet in a browser and in doing so allow us to harness their spare CPU cycles for our volunteer computing applications.

Unison has successfully harnessed the potential of processing power on the Internet for distributed computing applications while focusing on important issues like host participation and security, generic application execution and portability.

We have aimed to avoid the dual pitfalls suffered by other projects by refraining from introducing modifications to the Java runtime and native code. By maintaining the use of the standard JVM Unison can act as a generic base for Java applications to be processed with data over the web. This framework handles all broker-host communication; therefore clients only need to concentrate on the functional programming of the application. We have given an outline of how our system acts as a transactional broker between clients as the consumers of processing resources and hosts as the providers of processing resources.

Our main contributions include:

1. A framework for general-purpose applications. Instead of forcing application developers to adhere to a strict framework when implementing their program, we just require them to fill in Java code representing their program into our applets.
2. Support for generic applications that can consist of multiple arbitrary operations working on data from previous stages or on raw data. This mechanism to support multistage applications is transparent to hosts.
3. Support for SPMD programs. Chapter 3 outlines how we can handle applications with tasks that do not have communicational independence and those which have data dependencies.
4. How the use of a remote object system (i.e., CORBA) simplifies programming and offers greater flexibility to users of the system.
5. We have shown that real world applications such as local alignment not only work but also scale reasonably on our framework.

5.2 Future Work

More work can be done in some areas, for example in the field of grid computing. The Grid [FKT01, FK98b] refers to the concept of a large-scale execution environment for advanced science and engineering applications where access to compute resources is as pervasive as access to electric power and water. Globus [CFK98, FK97, FK98a] and Legion [GWF94] are projects on metacomputing infrastructures, Globus provides numerous services such as registration, authentication and resource location/discovery, while Legion's approach is based on an object-oriented model. Other Java based distributed computing systems have been successful in incorporating grid services and we intend to look into how our framework can be incorporated into a grid environment.

Our implementation uses a proprietary mechanism for defining data dependencies. Data dependencies exist when a task's input data is fed by results processed by another task. We need a more flexible way in which to represent these dependencies either using references between data structures or some other external representation. A better way would be to use a platform

independent language for data interchange such as Extensible Markup Language (XML)¹¹ or Resource Description Framework (RDF)¹². Since there is such a rich Java XML API (JAXP, JAXB, JAXM, JAX-RPC and JAXR)¹³ we can easily incorporate the use of this feature into our system.

Resource pricing is an interesting area to explore and we would like to incorporate any new findings in the future. Two types of payment exist: fungible and nonfungible [DN95]. The difference between the two lies in the fact that in the former case whatever material was used to make the payment may be reused in other transactions later on. Examples of fungible payment systems include Millicent [GMA], CyberCash¹⁴, Open Market¹⁵, eCash¹⁶, The Digital Silk Road [HT] and NetBill [ST]. Out of these systems, Millicent and NetBill are more appropriate for handling situations in which there are many transactions involving small sums such as in our case. Another way in which we could handle pricing without actual involvement of cash is to offer CPU time of the system in return for host participation. Nonfungible payments have been used to combat abuse of resources by making the party requesting access produce a proof of work. Some examples of this include hash cash [DUS] and client puzzles [JB99].

Resource awareness is important if the system is to scale efficiently. Resource aware applications are programs which are able to dynamically respond to changes in network connectivity and compute resource availability. An example is Sumatra [ARS97], this is a system in which a resource monitor Komodo [ARS97] helps applications using Java extensions to keep track of changes in their operating environment and thus enable scheduling to take these factors into account. A resource monitor could be used to keep track of network bandwidth and latency as well as host loads so that hosts need not directly query each other. It also serves as a repository of information about hosts, Hosts can be marked as having lost connectivity after a timeout by the resource monitor so that other hosts can be informed of this event.

¹¹ <http://www.w3.org/XML/> (Online, as of April 2003)

¹² <http://www.w3.org/RDF/> (Online, as of April 2003)

¹³ <http://java.sun.com/xml/index.html> (Online, as of April 2003)

¹⁴ <http://www.cybercash.cod> (Online, as of April 2003)

¹⁵ <http://www.openmarket.cod> (No longer online, as of April 2003)

¹⁶ <http://www.digicash.cod> (No longer online, as of April 2003)

BIBLIOGRAPHY

- [AIS97] D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Towards a Global Web-Based Parallel Computing Infrastructure. In Proceedings of the 11th IEEE International Parallel Processing Symposium (IPPS), Geneva, April 1997.
- [AND99] D. Anderson et al. (1999). SETI@home: The Search for Extraterrestrial Intelligence. Technical report, Space Sciences Laboratory, University of California at Berkeley.
- [ARS97] A. Acharya, M. Ranganathan, J. Saltz. "Sumatra: a Language for Resource Aware Mobile Programs, Mobile Object Systems, Lecture Notes in Computer Science, No. 1222, Springer Verlag (**D**), pp. 111-130, February 1997.
- [BBB96] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications, 1996.
- [BEB] Adam L. Beberg, et al. Project Bovine.
<http://www.distributed.net/rcS/> (Online, as of April 2003)
- [BFK99] J. Bester, I. Foster, C. Kesselman, J. Tedesco, S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. Sixth Workshop on I/O in Parallel and Distributed Systems, May 5, 1999.
- [BJK95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk An efficient multiheaded Runtime system. In Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'95, Santa Barbara, California, July 1995.
- [BKK96] A. Baratloo, M. Karaul, Z. Kedem, P. Wycko. Charlotte: Metacomputing on the Web. Proc of the 9th International Conference on Parallel and Distributed Computing Systems, Sep 1996.
<http://cs.nyu.edu/milan/charlotte/> (Online, as of April 2003)
- [BKK98] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem. An Infrastructure for Network Computing with Java Applets. Proceedings of ACM Workshop on Java for High-Performance Network Computing, February 1998.
- [BKL02] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, D. L. Wheeler. GenBank, Nucleic Acids Research 2002 Jan 1;30(1):17-20

- [BKN99] S. Brydon, P. Kmiec, M. Neary, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. In ACM 1999 Java Grande Conference, pages 171--180, San Francisco, **CA**, June 1999.
- [BL97] R. D. Blumofe, P. A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. Proc. of the USENIX 1997 Annual Technical Symposium, January 1997.
- [BRU93] J. W. Bruce, "A really trivial proof of the lucas-lehmer test," Amer. Math. Monthly, 100(1993) 370-371.
- [CCI97] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schausser, and D. Wu. 'Javelin: Internet-Based Parallel Computing Using Java', Concurrency: Practice and Experience 9(11): 1139- 1160, November 1997.
- [CFF01] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. Grid Information Services for Distributed Resource Sharing. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-IO), IEEE Press, August 2001.
- [CFG95] N. Carriero, E. Freeman, D. Gelemter, and D. Kaminsky. "Adaptive Parallelism and Piranha," Computer, vol. 28, no. 1, Jan. 1995, pp. 40--49.
- [CFK98] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke. A Resource Management Architecture for Metacomputing Systems. **Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing**, 1998.
- [CS92] C. Catlett and L. Smarr. *Metacomputing*, Communications of the ACM, 35 (1992), pp. 44-52.
- [DFP96] T. DeFanti, I. Foster, M. Papka, R. Stevens, and T. Kuhfuss. *Overview of the I-WAY: Wide area visual supercomputing*, International Journal of Supercomputer Applications, 10 (1996), pp. 123-130.
- [DJJ] J. J. Dongarra. Performance of Various Computers using Standard Linear Equations Software.
<http://www.netlib.org/benchmark/performance.ps> (Online, as of April 2003)
- [DN95] C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail, Weizmann Institute Tech. Report CS95-20, 1995.
- [DUS] A. Dustman. Hash cash A partial hash collision based postage scheme.
<http://www.cypherspace.org/~adam/hashcash/> (Online, as of April 2003)
- [FFK97] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. *A directory service for configuring high-peformance distributed computations*, preprint, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

- [FL] I. Foster, G. von Laszewski. Usage of LDAP in Globus.
[ftp://ftp.globus.org/pub/globus/papers/ldap-in-globus.pdf](http://ftp.globus.org/pub/globus/papers/ldap-in-globus.pdf) (Online, as of April 2003)
- [FK97] I. Foster, C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit, Intl J. Supercomputer Applications, 11(2):115-128, 1997.
- [FK98a] I. Foster, C. Kesselman. The Globus Project: A Status Report, *Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop*, pg. 4-18, 1998.
- [FK98b] Ian Foster and Carl Kesselman. The Grid Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, November 1998.
- [FKT96] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication, *Journal of Parallel and Distributed Computing*, 37 (1996), pp. 70-82.
- [FKT98] I. Foster, C. Kesselman, G. Tsudik, S. Tuecke. A Security Architecture for Computational Grids. *Proc. 5th ACM Conference on Computer and Communications Security Conference*, pp. 83-92, 1998.
- [FKTOI] I. Foster, C. Kesselman, S. Tuecke. The Anatomy of the Grid Enabling Scalable Virtual Organizations, to be published in Intl. J. Supercomputer Applications, 2001.
- [FWB99] D. Finkel, C. E. Wills, B. Brennan, and C. Brennan. Distriblets: Java based distributed computing on the Web. *Internet Research Electronic Networking Applications and Policy*, Vol. 9, No. 1, 1999.
- [FWC01] David Finkel, Craig E. Wills, Michael J. Ciaraldi, Kevin Amorin, Adam Covati and Michael **Lee**. An applet-based anonymous distributed computing system. In *Internet Research Electronic Networking Applications and Policy*, Vol. 11, No. 1, 2001.
- [GBD94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallelism*. MIT Press, 1994.
- [GIBB70] A. J. Gihbs and G. A. McIntyre (1970). The diagram, a method for comparing sequences. Its use with amino acid and nucleotide sequences, *Eur. J. Biochem.* 16:1-11.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
<http://java.sun.com/> (Online, as of April 2003)
- [GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, 1994.
- [GMA] S. Glassman, M. Manasse, M. Ahadi, P. Gauthier and P. Sobalvarro. The Millicent Protocol for Inexpensive Electronic Commerce.

- <http://www.w3.org/Conferences/WWW4/Papers/246/> (Online, as of April 2003)
- [GWF94] A. S. Grimshaw, W. A. Wulf, J. C. French, A.C. Weaver, and Paul Reynolds Jr. "Legion: The Next Logical Step Toward a Nationwide Virtual Computer," Computer Science Technical Report, University of Virginia, CS 94-21, June, 1994.
- [HB94] R. Hockney and M. Berry. Public International benchmarks for parallel computers report-1. Tech. Rep., Parkbench Committee, 1994.
- [HH92] S. Henikoff and J. G. Henikoff (1992). Amino acid substitution matrices from protein blocks, Proc. Natl. Acad. Sci. USA 89:10915-10519.
- [HIR97] S. Hirano. BORB: Distributed Execution of Java Programs, in: Proc. WWCA'97, Lecture Notes in Computer Science, Vol. 1274. Springer, Berlin, 1997.
- [HN99] A. Heydon and M. Najork. "Performance limitations of the Java core libraries", ACM 1999 Java Grande Conference San Francisco, California, June 12-14, 1999,
- [HT] Norman Hardy and Eric Dean Tribble. The Digital Silk Road.
<ftp://ftp.netcom.com/pub/jo/joule/DSR/DSRI.txt.gz> (No longer online, as of April 2003)
- [JB99] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks, In Proceedings of the 1999 Network and Distributed System Security Symposium (NDSS'99). Internet Society, March 1999.
<http://www.isoc.org/ndss99/proceedings/papers/juels.pdf> longer online, as of April 2003)
- [KY98] Burton S. Kaliski Jr. and Yiqun Lisa Yin. On the Security of the RC5 Encryption Algorithm. RSA Laboratories Technical Report TR-602, September 1998.
- [LINN93] J. Linn. Generic security service application program interface, Internet RFC 1508, (1993).
- [LLM88] M. Litzkow, M. Limy, and M. Mutka. Condor - a hunter of idle workstations. In Proc. 8th Intl Conf. on Distributed Computing Systems, pages 104-111, 1988.
- [MAY99] Michael May. Locust - a brokerage system for accessing idle resources for web-computing. In Proceedings of the 25th EUROMICRO Conference (EUROMICRO'99), Milan, Italy, September 1999.
- [MF97] Gary McGraw, Edward W. Felten. Java™ Security Hostile Applets, Holes and Antidotes, Wiley Computer Publishing, 1997.
- [NKB01] Rob V. van Nieuwpoort, Thilo Kielmann and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications, In proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming, 2001.

- [NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search of similarities in the amino acid sequence of two proteins, *Journal of Molecular Biology*, 48:443-453, 1970.
- [OLL02] T.M. Ong, T.M. Lim, B.S. Lee and C.K. Yeo. Unicorn: Voluntary Computing over Internet, *Operating Systems Review, ACM*, 36(2):36-51, April 2002.
- [OLY01] T.M. Ong, B. S. Lee and C. K. Yeo. A Framework for Distributed Computing over the Internet. *Proceedings of International Conference on Information, Communications and Signal Processing (ICICS) 2001*.
- [OMG02a] Object Management Group. CORBA Core Specification, online document, December 2002.
<http://www.omg.org/cgi-bin/doc?formal/02-12-02.pdf> (Online, as of April 2003)
- [OMG02b] Object Management Group. IDL to Java Language Mapping, online document, August 2002.
<http://www.omg.org/cgi-bin/doc?formal/02-08-05> (Online, as of April 2003)
- [PL88] W. R. Pearson and D. J. Lipman (1988). Improved Tools for Biological Sequence Comparison. *Proceedings of the National Academy of Sciences (PNAS)* 85:2444-2448.
- [SHW98] L. F. G. Samenta, S. Hirano, S. A. Ward. Towards Bayesian: Building an Extensible Framework for Volunteer Computing Using Java, *Concurrency: Practice and Experience*, Vol. 10(11-13), 1015-1019(1998)
- [SNS97] M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A Network based Information Library for a Global World-Wide Computing Infrastructure, In *Proceedings of HPCN97 (LNCS-1225)*, pages 491-502, 1997,
- [ST] M. Sirbu, J. D. Tygar. NetBill: An Internet Commerce System Optimized for Network Delivered Services.
<http://www.ini.cmu.edu/netbill/> (Online, as of April 2003)
- [STA91] D. J. States and M. S. Boguski (1991). Similarity and homology, in *Sequence analysis primer*, M. Gribskov and J. Devereux, editors. Stockton Press, New York. pp. 92-124.
- [SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences, *J. Mol. Biol.* 147(1981) pp. 195-197.
- [TMN98] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. Ninflet: A migratable parallel objects framework using Java, In *ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.
- [VAN97] L. Vanhelsuwe. "Create your own supercomputer with Java", *Java World*, 2(1), 1997.

- [WEI98] J.B. Weissman. "Gallop: The Benefits of Wide-Area Computing for Parallel Processing," Journal of Parallel and Distributed Computing, Vol. 54(2), November 1998.
- [WOL] G. Woltman. "The great Internet Merseine prime search", <http://www.mersenne.org> (Online, as of April 2003)
- [ZHO92] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In Proc. Workshop on Cluster Computing, 1992.

APPENDIX A SCHEDULING TERMS

Some terms which occur frequently throughout the study on scheduling are described here.

Adaptive Parallelization

A scheduler that handles adaptively parallel computations is able to react to changes in its environment whereas a static scheduler works on the assumption that characteristics of its execution environment would remain the same throughout its lifetime, e.g. the number of processors or hosts available is constant.

Schedulers in a distributed computing system operating across a WAN have to handle such vagaries by monitoring changes in their environment and responding appropriately. Gallop [WEI98] tackled such issues. Projects such as Piranha [CFG95] and Cilk-NOW [BL97] have undertaken studies on adaptive parallelism and we can learn from their experience.

Piecework / Divide and Conquer

A piecework computation may be decomposed into a set of autonomous sub-computations that only communicate for scheduling and returning of results. Examples of such computations include raytracing and matrix multiplication. These applications have been adapted to run on volunteer computing systems like Javelin and Bayanihan. The main reasons for their popularity are:

1. They impose minimal communication demands on the underlying network.
2. The data granularity of the sub-computations can be varied according to the number of hosts in the system.
3. Scheduling of these sub-computations is straight-forward since the hosts need not directly communicate with each other.

Computations based on a divide-and-conquer model differ from piecework computations in the manner in which sub-computations are divided. In the divide-and-conquer model a computation is recursively subdivided thus forming a hierarchical structure linking these sub-computations.

Work Stealing

Work stealing was popularized by Cilk [BJK95], it is a technique in which processors or hosts that run out of work take work units from double-ended queues on hosts that have work. The people working on Satin [NKB01,45] conducted a study and comparison of variations on how the victim host from which work is to be stolen is chosen, they are:

1. Random Stealing
2. Cluster-aware Hierarchical Stealing
3. Cluster-aware Load-based Stealing
4. Cluster-aware Random Stealing

Essentially the second, third and fourth variants take into account factors such as low bandwidth and high latency links present in a WAN when they decide on where to steal work from. Our present system does not allow hosts to have direct contact with each other so it remains to be seen if work stealing is a technique that can be employed here.

Eager Scheduling

Eager schedulers assign the same piece of work to multiple hosts in an attempt to retrieve the result from the fastest host. This also has the effect of masking host failures and slow hosts. Charlotte [BKK96] and Javelin employ eager schedulers.