A FRAMEWORK FOR OPPORTUNISTIC CLUSTER COMPUTING USING JAVASPACES

by

JYOTI BATHEJA

A thesis submitted to the

Graduate School-New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

written under the direction of

Professor Manish Parashar

and approved by

____

____

____

____

New Brunswick, New Jersey

May, 2001

ABSTRACT OF THE THESIS

A Framework for Opportunistic Cluster Computing using JavaSpaces

by JYOTI BATHEJA

Thesis Director:

Professor Manish Parashar

Heterogeneous networked clusters are being increasingly used as platforms for resource-intensive parallel and distributed applications. The fundamental idea is to provide large amounts of processing capacity over extended periods of time by harnessing the idle and available resources on the network in an "opportunistic" manner. In this thesis we present the design, implementation and evaluation of a framework that uses JavaSpaces to support this type of opportunistic (adaptive) parallel/distributed computing in a non-intrusive manner over networked clusters. The framework targets applications exhibiting coarse grained parallelism and has three key features: 1) portability across heterogeneous platforms, 2) reduced overheads for configuration of participating nodes, and 3) automated monitoring of system state (using SNMP) to ensure non-intrusive behavior. Experimental results presented in this thesis demonstrate that for applications that can be broken into manageable components, such an opportunistic adaptive parallel computing framework can provide performance gains. Furthermore, the results indicate that monitoring and reacting to System State enables us to minimize intrusiveness to the machines within the cluster.

ACKNOWLEDGEMENTS

I thank Manish Parashar for being a patient thesis advisor and for providing an environment to conduct excellent research. He has built a great research lab, TASSL, and I am grateful that he has allowed me to be a part of it.

I thank my brother Deepak Batheja for encouragement and moral support during the entire span of my thesis. This thesis would never have been a reality without his continual support and guidance. He also taught me a great deal about doing and presenting research.

I also thank everyone in the TASSL lab and ECE Department for making the past two years a wonderful experience. I really appreciate all the support and encouragement especially from Pravin Bhandarkar.

I thank IEEE for providing me with the avenues and flexibility to work on this thesis and balance the demands of a full time job.

Finally, I thank my family for their love and constant encouragement throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background and Motivation

Traditional High Performance Computing is based on massively parallel processor supercomputers or high-end workstation clusters connected over high-speed networks. Most of these resources are expensive. The processing involved is usually resource intensive and voluminous in nature. However most high performance computing problems are iterative; applying the same function to varied data sets. Parallel processing achieved using dedicated processors is static in nature. That is, the hosts participating in the parallel processing remain fixed throughout the entire computation. It has been observed that resources available at an enterprise such as desktop PCs are not fully exploited. Consider a scenario, wherein most of the workstations in a given enterprise become idle at a given time. Say for example; 80% of the employees do not work beyond official work hours rendering their PCs idle. Optimal use of such PC resources while they remain idle can be effectively used to the advantage of an enterprise.

To summarize, traditional high performance computing brings out two key problem concerns: static dedicated supercomputers are an expensive proposition for parallel processing and available networked resources are untapped and have not been exploited to their full capacity. In such a case, it would be an intelligent proposition to utilize those idle networked resources into doing something useful. This identifies a need to devise a mechanism to facilitate parallel processing of high performance computations on networked resources.

Recent advances in cluster computing follows two trends by which parallel processing is accomplished. *Job level parallelism*: The framework for distributed computing is responsible for identifying and monitoring available resources. If during the course of the computation a resource renders itself unavailable, the framework is responsible for checkpointing the state of the computation and migrating the job from the unavailable machine onto idle ones, finally resuming the job from where it was last left off. Checkpointing involves creating an entire memory map of the system state at the time of migration such that the next processor that will continue execution of the job has complete contextual state information to taken on job computation from the point it was left off. This approach has been very successful in targeting applications that cannot be decomposed further. This leads to a more reactive style of exploiting available resources since the framework needs to be aware of the state of other machines and provide clean check pointing mechanisms before the idle hosts begin the computation. "*Adaptive parallelism*" – a term coined by the Piranha project group [1], focuses on dynamically utilizing processor resources based upon the availability. This model works for a certain class of problems that can be easily decomposed into smaller independent tasks. In this case the available processors are treated as part of a resource pool. Each processor aggressively competes for tasks as and when they are made available to the resource pool. Under "adaptive parallelism," the pool of processors executing a parallel program may grow or shrink during the program execution based upon the processor availability. Large computationally intensive problems can be decomposed into smaller independent tasks and deployed over a network of such machines in order to utilize their processing capabilities.

Utilizing available idle resources on a networked cluster provides a more cost effective option. The main objective of this thesis is to design and implement a framework that uses JavaSpaces [2] to aggregate networked computing resources and non-intrusively exploit idle resources, both within and among institutions, for parallel/distributed computing. However there are challenges that must be addressed to make it a viable option. These challenges are listed below and will be revisited at the end of the thesis to measure the success of our framework. These include:

- *Adaptability to cluster dynamics:* Changing system and network parameters on participating nodes must be accounted for. We observe that with the advent of Web technologies, availability of system resources and network bandwidth have become sporadic in nature and difficult to predict due to variation in network traffic and resulting latencies thereof. In such a situation, the parallel process must be able to handle the computation within the given resource constraints, or off load its work to other systems if the constraints become intolerable.

- *System configuration and management overhead:* Dynamically incorporating resources into the "cluster" may require system configuration and software installation. Provisions in system configuration such as modification of existing access control mechanisms or download and installation of additional software components required for participation in parallel processing are some examples of the overhead involved. These modifications and overheads must be kept to a minimum and made transparent to the computation process.
- *Heterogeneity:* Clusters are typically heterogeneous in the computing resources and so are the associated configurations, available services and tools. This heterogeneity must be address in a seamless manner. For example, tools for measurement of constraining factors may not be standardized across heterogeneous nodes. The task manager utilities for performance measurements provided with Windows NT are not present in Windows 95 and are widely different across heterogeneous operating systems. Differences across heterogeneous platforms, in terms of instruction sets, process representation and structure, memory organization, system/kernel interfaces tend to limit the scope of efficient task migration.
- *Security and Privacy concerns:* Secure access across networked resources must be ensured to provide a level of assurance to users making their machines available for the computation. Policy decisions must be enforced so as to ensure that the external parallel processes adhere to the limits set for data access and resource utilization. This may also require using secure and standardized set of protocols to ensure the confidentiality of data over the interconnection network.

- *Non intrusiveness:* Introducing a framework to facilitate parallel computing should not require changing any existing legacy code or standards employed by the participating nodes. Also execution of the problem application must not hog the system resources, thus paralyzing the user from doing any useful work. Instead the parallel processing must run as a secondary background process that does not affect the normal functioning of user programs.

To summarize, loosely coupled cycle stealing clusters warrants an adaptive parallel computing model to account for the heterogeneity and dynamism of the cluster environments.

1.2 Overview

The aim of this thesis is to exploit networked resources by utilizing JavaSpaces as the backbone to aggregate the available resource pool as a powerful computing engine, and to develop an infrastructure that exploits idle resources within institutions. This thesis presents the design and implementation of a framework to enable adaptive parallelism using JavaSpaces.

Our approach to enable opportunistic cluster computing uses standardized technologies such as Java that provides a sandbox model. The sandbox model of Java assures secure code execution within the boundaries as set in policy configuration files. Java also addresses portability of code across heterogeneous environments. JavaSpace technology is used as a coordination tool and provides remote communication mechanisms. This facilitates a model for deployment and execution of parallel code to remote hosts that addresses the issues listed above. More importantly this option leverages existing investments in computational resources, thereby utilizing these resources to the advantage of an enterprise.

1.2.1 Key Features

The key features of this thesis are:

- *Java as the core programming language:* The framework is implemented mainly in Java with few simple native methods to optimize operations for System State Abstraction that are difficult to achieve in Java. We refer to System State Abstraction as a means of extracting System State data in terms of CPU Usage, Page Faults, etc. and abstracting this data to components that transform this data into meaningful interpretable information. Differences in architecture across heterogeneous hosts are handled by utilizing Java as the core programming language. Java programs produce byte code that can be executed or interpreted on any host that implements JVM (Java Virtual Machine). In addition standardized virtual machine facilitates easy object migration between heterogeneous remote hosts.

- *System State Abstraction:* In our model for opportunistic cluster based computing the decision to participate in a problem solution is driven based on the System State parameters. The framework provides sustained monitoring of System State using SNMP (Simple Network Management Protocol) [3][4] to enable the parallel process to react to changing environment variables that may render the system unavailable for computation. This minimizes the intrusiveness on the individual machines. This module is made up of two components, namely: Java and WinSNMP API [5].
- *Remote Node Configuration Techniques:* Our framework builds on JavaSpaces technology and facilitates global deployment for parallel execution of code across all nodes participating in the computation. The deployment task is generic and designed to minimize the costs and overheads associated with configuration and management. Besides, neither RMI nor Jini provides mechanisms of starting up a process out of nothing on a remote host. Hence we envisage that the user will download a jar-file of the remote node configuration classes onto the machines that will host the parallel jobs. This initialization process is a one-time process that establishes connection with the Network Monitoring module and instantiates worker processes depending upon the initialization signals received. Our framework provides runtime access to code that is loaded on the remote hosts. This is critical to transfer protocol signals to the remotely executing worker code.

1.2.2 Work Progression

The focus of our work is to identify the various software components that can be aggregated to assist in opportunistic parallel processing, implementing these software components as classes within an object oriented framework, finally validating this framework with a library of diverse parallel applications. We explain the progression of our work using Figures 1.1 and 1.2 shown below.

Phase 1 represents the problem domain identification and analysis phase. Bearing the requirements in mind we identified problems from three different categories, a web page pre-fetching scheme based on a web page rank, a scientific ray tracing problem, and a real work financial application problem. Next we analyzed the sequential execution of the problem to see the level of parallelism that could be exploited within the given problem.

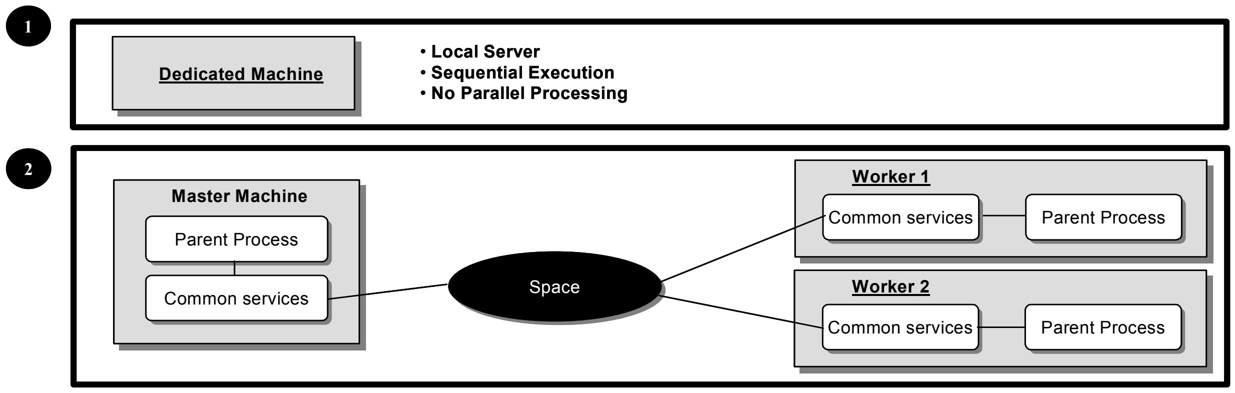


Figure 1.1 Phases 1 and 2 in Progression of Work

Phase 2 involved porting the problem domain to function within the JavaSpaces infrastructure. The master and workers entities interact with each other via the space. Each of these entities comprises of two main components, namely, the parent process dealing with application level details, and common services such as Jini's infrastructure services, for example directory lookup, provided to facilitate communication.

Phase 3 involved the design and implementation of the architecture and the individual modules. An additional component for remote node configuration as a part of the common services is included in this phase for worker processes. The common services are available to all workers. Our framework also implements a Load Balancing Engine that spreads the load across any other processes that may be spawned, executed and terminated on the worker nodes. Its implementation is based on the SNMP Manager processes that communicate with SNMP Agent processes on the worker nodes and an Inference Engine that computes the signals to be passed to the worker processes based on configurable threshold ranges and system parameters that are continually monitored on the worker machines.

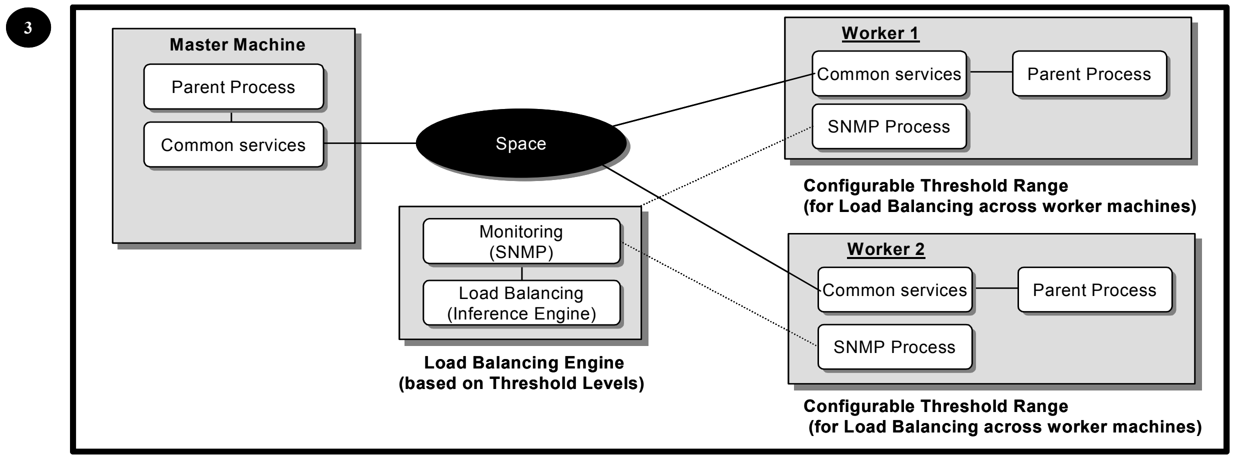


Figure 1.2 Phase 3 in Progression of Work

Experimental results demonstrate that for applications that can be broken into smaller, independent, manageable components, such an opportunistic adaptive parallel computing framework can provide performance gains. Furthermore, the results show that monitoring and reacting to system state enables us to minimize intrusiveness to the machines in the cluster.

1.3 Thesis Outline

Chapter 2 covers related work in which we survey some of the prevalent architectures employing adaptive parallelism developed by others. It also includes a comparison of the various adaptively parallel architectures. In Chapter 3, we present the JavaSpace abstraction model as a tool that facilitates distribution of parallel tasks. In Chapter 4, we present our framework architecture highlighting the goals we have achieved. Here we describe the mechanism developed for remote node configuration and System State Abstraction in greater detail. Chapter 5 is dedicated to the experimental evaluation of the three applications implemented under our framework. Our framework has been tested for adaptation to dynamically changing conditions. Chapter 6 draws conclusions on the effectiveness of the research, and suggests some possible future directions.

Chapter 2

Related Work

This chapter provides a brief overview of the existing work on adaptive computing. It covers both loosely coupled cluster and web based architectural models. A comparison of these architectures is also included in this chapter.

2.1 Related Adaptive Computing Architectures

A broad classification of adaptive computing architectures is presented in Figure 2.1. Cluster based adaptive computing architectures exploit available resources within a networked cluster, such as a LAN, for parallel/distributed computing. Recent work in adaptive parallelism has extended this model to exploit resources connected via the Internet. This has lead to the web based adaptive computing model.

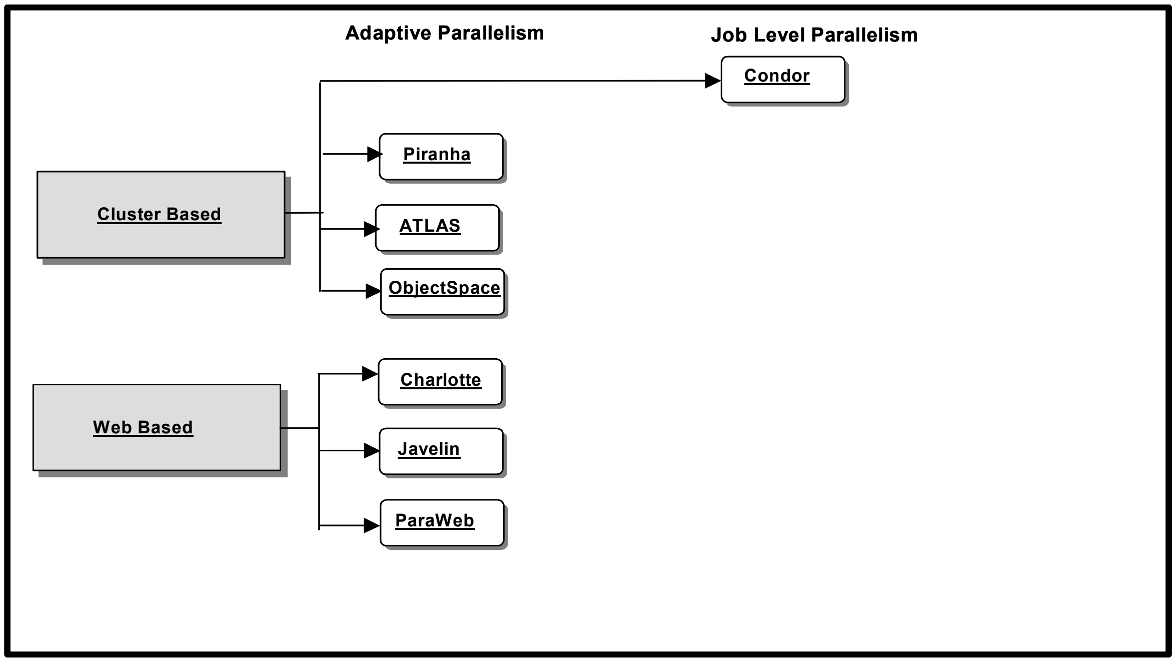


Figure 2.1 Classification of related parallel computing architectures

Job Level Parallelism involves allocating entire jobs to idle resources, and migrating them between resources when the resources become unavailable. Systems supporting Job Level

Parallelism must be able to save and restore System and Computation State when the jobs are migrated. The Condor system support cluster-based Job Level Parallelism. In Adaptive Parallelism, the job is broken into smaller tasks, and job scheduling and load balancing schemes are used to distribute these tasks to exploit idle resources. Systems supporting Adaptive Parallelism include cluster-based systems such Piranha, Atlas, and Anaconda, and web-based systems such as Charlotte, Javelin, and ParaWeb. The different systems are discussed below.

2.1.1 Condor (University of Wisconsin – Madison)

Condor [6] achieves *Job level parallelism* on cluster based systems. In this model, users submit their computational jobs to Condor, which queues these jobs, executes them, and then notifies the user of the result. It provides a queuing mechanism, scheduling policy, prioritization scheme, and resource classifications to assist in parallel processing.

Condor provides several unique capabilities geared towards effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds for matchmaking. These terms are defined below. Condor actively monitors the state of participating resources and manages resources by offloading the remaining job from loaded machines onto available idle resources.

- *Check pointing and Migration:* Condor's Periodic Checkpoint feature can periodically checkpoint the job in order to safeguard the accumulated computation time on job from being lost in the event of a system failure.
- *Remote System Calls:* In Condor's Standard Universe execution mode, the local execution environment is preserved for remotely executing processes via Remote System Calls. Within the Condor model the program behaves as if it was running as the user who submitted the job, and on the workstation where it was originally submitted, independent of the machine that executes it.

- *Matchmaking:* The ClassAd mechanism in Condor provides an extremely flexible and semantic-free, expressive framework for matching Resource Requests with Resource Offers. Job requirements/preferences and resource availability constraints can be described in terms of powerful, arbitrary expressions, resulting in Condor being flexible enough to adapt to nearly any desired policy.

Condor facilitates process migration by means of transferring checkpoint files among participating nodes. These checkpoint files contains the memory image of the application and could include cached input and intermediate file data. Condor assumes that this means of saving the state is sufficient for most real world applications. However, there are a lot of details in the process's state, which are implicit or known only to the kernel. It is difficult to reconstruct the entire state by means of only the checkpoint file. Another major limitation is that the condor check pointing code should be linked with the user's code. This is difficult in cases of third party software usage [7].

In order to better support parallel applications under Condor, CARMI (Condor Application Resource Management Interface) was implemented. CARMI [8] provides a framework for interfacing PVM with Condor [9] and is intended to work in a generic message-passing environment. Resource requests and replies inherent to the functioning of Condor is limited by the synchronous nature of PVM since it only provides a blocking, procedure call interface. However the robustness of the message passing API provided by PVM poses itself as a lucrative option. CARMI presents an asynchronous API to this effect. It employs the master-worker paradigm called the WoDi (Work Distributor) framework and relies on the job management API provided by PVM [10].

2.1.2 Piranha (Yale University)

The Piranha model [1] achieves *Adaptive parallelism* on cluster based systems. It uses the Linda [11] model to support tuple spaces for scheduling work among processors. The Linda model introduces the concept of tuple spaces that concurrent processes can access in order to insert, delete and update data called *tuples*. All these operations on tuples are atomic in nature. The tuple space may be considered as a repository of objects that is utilized by hosts to share data. Synchronization constructs for supporting message passing and shared memory abstractions such as barriers; semaphores are inherently handled by the tuple space.

Piranha focuses on recycling idle nodes that are routinely wasted in LAN systems. It employs the master worker paradigm wherein the master process decomposes the problem domain into small, manageable tasks and puts them into the space. Worker processes compete with each other to execute the parallel program on these task producing appropriate results. These results are later aggregated by the master to provide a meaningful solution to the problem. Based on processor availability the system transitions from the available state to the unavailable state, resulting in execution of the retreat functionality to provide smooth synchronization. Decisions for availability are made based on user-defined criteria with system-supplied defaults. This system is however limited by the fact that it does not span multiple networks incorporating heterogeneous hosts. Also it is noted [12] that it takes a couple of seconds to vacate a node if it is rendered unavailable. Even then the node may still play a minor role in the on going computation as a tuple server. A major drawback of the Piranha model is that it requires modifications to the Linda system and only supports Linda programs that have been modified to suit it.

2.1.3 ATLAS (Joint Effort: The University of California at Berkley and The University of Texas at Austin)

ATLAS [13] achieves *Adaptive parallelism* on cluster based systems. It employs Java and Cilk technology to implement a hierarchical work stealing approach. Such as hierarchical model offers three main advantages:

- It keeps the computations localized by forcing applications to use sibling resources first. Local clusters offer more bandwidth and lower latencies as compared to randomly distributed efforts.

- Sub trees naturally map to existing administrative domains. Thus privacy and security concerns are addressed via administrative control.
- Fault tolerance is handled by the inherent tree structure. For any given stolen thread, the sub tree rooted at that thread provides the reference point. A time out is associated with every sub computation and checkpointing mechanism is employed to restart computation upon timeout.

Use of Java as the programming language assists in porting the ATLAS programs across heterogeneous environments. Cilk is a C based programming language that employs a thread scheduling algorithm based on work stealing techniques. The ATLAS programs comprises of a number of procedures that consist of a sequence of non-blocking, continuation-passing threads. This hierarchical model of program execution can be viewed as a tree of procedures wherein the parent thread awaits the return of all its children procedures. The architecture comprises of three key entities: clients that request the computation, managers that oversee all results and file accesses and compute servers that perform the actual computations. Each compute server is provided with a runtime library for work stealing, thread management, and marshalling of objects for communication to other compute servers. The threads on these compute servers are executed in a depth-first, stack-based manner.

ATLAS provides a global file sharing system that could lead to conflicts in access control. It is noted that porting the runtime library to a new platform requires a platform specific Java Interpreter and conversion mechanisms for the native Cilk libraries.

2.1.4 ObjectSpace (University of Alabama, Huntsville)

ObjectSpace [14] is a cluster based parallel computing system developed at the University of Alabama. ObjectSpace uses several implementation features described in the JavaSpace specification and is based on the Linda programming model. It relies on features provided by RMI for exchange of objects among the participating processes. It employs the master worker paradigm for distributed computing. The master task transfers the worker tasks into the ObjectSpace. On the worker nodes, by pointing the browser to the known ObjectSpace server URL, users retrieve a small applet that loads the worker tasks, executes it, and returns results to the ObjectSpace. This preliminary work on adaptive parallelism involves building a framework analogous to JavaSpaces. Ongoing research efforts at the University of Alabama extend upon the ObjectSpace model to build *Anaconda*. Anaconda provides an overall solution framework by including the core capabilities of ObjectSpaces and addressing some of the shortcomings of ObjectSpace. Some of the key features being addressed within the Anaconda framework include adaptive parallelism and fault tolerance. The framework comprises of three entities: the client that submits the job, the hosts that perform the actual computations and the brokers that act as managers for load balancing and scheduling jobs between clients and hosts. It implements a discovery protocol for registering brokers and their services. Fault tolerance is restricted to the hosts and is not supported among brokers. Hence, it applies Job level parallelism among the various brokers to checkpoint the jobs onto secondary data stores in the event of partial failures at the broker level. The brokers ensure fault tolerance among the various hosts by periodically polling the health of the hosts in its domain. If a response is not received, the broker assumes a partial failure and returns the jobs back into the pool to be re-executed by other hosts. In an effort to minimize the total computation times, the framework employs eager scheduling techniques. This involved scheduling of the same job to a number of hosts. Results obtained from the hosts that first completes are sent to the client discarding subsequent ones. In order to limit the number of copies of the same job floating among various hosts, Anaconda applies two replication techniques: *automated replication* and *manual replication.* Automated replication attempts to classify host machines based on the computation times. Jobs returned to the pool are tagged indicating that subsequent jobs should not be scheduled on the tagged host. Manual replication involves specifying the exact number of instances of a particular job that can be created on a given broker. It is the broker's responsibility to ensure that no two instances are allocated to the same host. Digital signatures are used to address security and privacy concerns.

2.1.5 Charlotte (New York University)

Charlotte [15] is a web based computing system that implements distributed shared memory over the Java Virtual Machine and provides support for fine-grained distribution of the computing task. The key feature of the shared memory architecture is that it requires no support from the compiler or the operating system, as is the case with most shared memory architectures. This distributed-shared memory is implemented at the data level; i.e. every basic data type in Java has a corresponding charlotte data type. Data consistency is maintained using the atomic update protocol allowing concurrent reads but exclusive writes/updates. The computing task is downloaded and run as an applet on the host machine in this system. Charlotte has been ported to work within the Knitting Factory Infrastructure [16] that provides a Jini like framework for discovering idle resources and facilitates inter-applet communication. It brings together three types of entities: clients, brokers and hosts. Clients seeking computing resources, register their request with a broker and submit their work in the form of an applet. Hosts willing to participate in the computation contact the broker and download the applets. Charlotte achieves load balancing and fault masking by implementing two concepts, *eager scheduling and two-phase idempotent execution* (TIES). *Eager scheduling* involves repeated assignment of a job until it is executed to completion by at least one worker. This leads to excessive data traffic, multiple copies of the same job and inconsistent memory views across jobs. The memory management technique employed is called *two-phase idempotent execution* (TIES). TIES guarantees correct execution of shared memory by reading data from the client and writing it locally in the workers memory space. Upon completion the dirty data is written back to the client who invalidates all successive writes for that particular job thus maintaining only 1 copy of the resulting data. The infrastructure relies on the applets security model that enables browsers to execute untrusted applets in a trusted environment. All applet-to-applet communication is routed through the broker. It primarily functions as a work distributor among various browsers and a forwarding agent to facilitate inter applet communication. This can be a potential bottleneck.

2.1.6 Javelin (University of California, Santa Barbara)

Javelin [17] implements a global computing infrastructure catered to achieve coarse-grained parallelism over the Internet. The participating nodes are required to have Java capable browsers in order to participate in the computation. It involves three entities: a broker that matches resource requirements of the clients to availability of hosts, clients that request computing resources by registering with brokers; and hosts that perform the computation via downloadable Java applets. The applet first spawns a daemon process that actively listens for available tasks from the broker. Key advantages of this architecture are:

- Due to a URL based computational model, the worker hosts can perform the necessary computation offline and later reconnect to the Internet to return the results from the computation.
- It leverages the existing security model provided by Java to execute untrusted code within a trusted environment.
- Ubiquity of web browsers is leveraged to minimize the administrative overheads involved in preparing worker nodes.
- The infrastructure provides for applet balancing schemes wherein applets can seamlessly register with another broker terminating its registry from the existing broker.
- The infrastructure provides a set of library functions to assist portability to include Linda and SPMD programming models on top of the Javelin substrate.

Use of applets prevents communication via datagrams. Additionally, untrusted applets are restricted from accessing local file resources. Hence all forms of communication needs to be routed through the broker from which the processing was initiated. This may lead to a bottleneck.

2.1.7 ParaWeb (York University)

ParaWeb [18] is a web based computing system centered towards solving coarse-grained problems and provides two separate models. One model provides a runtime system (Java Parallel

Runtime System) that requires modifying the Java Interpreter to provide global shared memory and transparent thread management mechanism across remote machines. This model makes use of Java's built-in synchronization functions and maps it into either acquire or release operations. Calling a synchronized method in Java is equivalent to the acquire operation in this model. Wait methods executed within threads are mapped to the release operation thus providing means for exiting in this model. The other model provides sets of libraries (Java Parallel Class Library) that facilitate message passing and thread management across remote machines. In this model each compute server runs a daemon process that registers with the local scheduling server. The scheduling servers may be viewed as intermediate managers that are responsible for administering the compute servers and defining authentication and authorization policies. Instantiation of a new object that extends the RemoteThread class on the client side triggers the RemoteThread class to contact the scheduling server. The scheduling server returns the address of the compute server that will participate in the computation. Next the client sends the compiled byte code to the remote compute server for code execution. Upon completion results are returned to the client, which then notifies the scheduling server about the job completion. ParaWeb allows clients to download worker applications and execute them locally. It also allows for a client to upload and execute programs on remote compute servers. Such uploading of execution code requires the remote compute servers to run byte-code interpreters.

2.2 Discussion

Table 2.1 shows a comparison of the existing adaptive computing architectures described in section 2.1. The list of contributions and applications is not exhaustive. Principally, there are two ways in which the adaptive computing can support parallel processing. *Cluster based* approach revolves around deploying the computational task across loosely coupled networked resources. *Web based* approach extends the idea to run the parallel computation over the Internet. The collection of machines connected over the Internet; if aggregated presents us with unparalleled processing capacity. Additionally, it provides us with a system that is not restricted to any shared file system or user account and presents us with a much wider scope of a heterogeneous clientele. However, special care must be taken to safeguard the system against malicious users and buggy code. Note that most of the systems (including the framework presented in this thesis) described above implement the master worker paradigm in one form or the other. Our framework is Java-based and builds on the JavaSpaces infrastructure. It supports heterogeneous cluster computing.

Most of the systems that exploit adaptive parallelism require manual management – for example they present the user with a graphical interface for stopping background tasks at a worker when the machine is no longer available. Such event driven interfaces at the application layer are easy to implement on the worker machines, but require knowledge and explicit effort on the users part. In the framework presented in this thesis, we attempt to automate this decision-making by monitoring and using system parameters. A dedicated network management module identifies and monitors system state parameters (using SNMP) to track resource availability. Given the heterogeneous clients involved, we felt the need to isolate this System State Abstraction into a network management module and utilize SNMP services for system monitoring. We observed that standardized services such as SNMP are mature enough and make more information available in terms of statistics, monitoring and reliable messaging.

Furthermore, in the presented framework, we actively address configuration management issues. In our system, the worker nodes need not be involved with the intricacies of the worker code. In order to facilitate this we provide remote node configuration mechanisms. This involves remotely loading the worker classes at runtime. Integrating this with the network management module, and providing runtime signals to enable the worker to react to changing system parameters, has been one of our most challenging tasks.

Architecture	Category	Approach	Key Contributions	Applications
Condor	Cluster based	Job level parallelism, Asynchronous	Queuing mechanism, Check	High Throughput Monte Carlo

		offloading of work to idle resources on a request basis	pointing and process migration, Remote procedure call and matchmaking	Simulations
Piranha	Cluster based	Adaptive parallelism, Centralized work distribution using the master worker paradigm	Implementation of Linda Model using tuple spaces, survives partial failure, efficient cycle stealing	Monte Carlo simulations, LU Decomposition, Ray Shade
ATLAS	Cluster based	Adaptive Parallelism, Hierarchical work stealing	Fault Tolerance, improved scalability due to hierarchical model, safe heterogeneous execution, no administration effort	Double recursion to compute Fibonacci numbers, (POV-Ray) Ray Tracing
ObjectSpace /Anaconda	Cluster based	Job level parallelism at brokers	Eager scheduling, automated/manual job replication	Traveling Salesman Problem
Charlotte	Web based	Adaptive parallelism, Centralized work distribution using downloadable applets	Abstraction of Distributed Shared Memory, directory look up service for idle resource identification, embedded lightweight class server on local hosts, direct inter-applet communication, fault tolerance	Matrix multiplication, statistical physics application for computing the 3D Ising model
Javelin	Web based	Adaptive parallelism, Centralized work	Heterogeneous, secure execution	The Mersenne Prime

		distribution using downloadable applets	environment, portability to include Linda and SPMD programming models	Application: a primality test.
ParaWeb	Web based	Adaptive parallelism, Centralized work distribution using scheduling servers	Implementation of Java Parallel Runtime System and Java Parallel Class Library to facilitate upload and download of execution code across the web	Parallel matrix multiplication

Table 2.1 Comparison of adaptive computing architectures

Chapter 3

Jini/JavaSpaces Architecture

The Web presents us with a platform to distribute information content. This architecture inherently offloads from the user mundane tasks such as finding and displaying content. However, no general mechanism exists for distributing executable code. The Jini infrastructure was evolved out of this need to transparently distribute functionality. It provides additional functionality in the form of services that can be availed for distribution of executable content and object sharing. This makes it a promising technology suited for distributed cluster computing. This chapter provides a prelude discussion of the Jini architecture [19] and the JavaSpaces service that is the backbone for the presented framework.

3.1 Design Overview of Jini

The Jini technology is a runtime infrastructure that resides on the network and provides mechanisms to enable addition, removal, discovery and access of services. It provides a set of APIs and network protocols that assist in building and deploying truly distributed systems that are organized as a *federation of services* [20]. A federation of services presents a view of the network that doesn't involve a central governing authority, instead it can be thought of as a group composed of equal peers. Thus a Jini service joins a federation to share its services with clients and Jini clients join a federation to gain access to services. This presents a Service-based model. A service is any entity capable of performing some function. Services advertise their capabilities via a look up server. The primary function of lookup servers is to assist Jini enabled clients to discover and access services. In the following sections, we discuss two key services of interest to this thesis: the Look up services, and the JavaSpace service.

3.1.1 Lookup Service

The lookup service primarily maintains a mapping between each Jini service and its attributes. Whenever a Jini enabled device advertises its service the lookup server adds that information to the map. A typical Jini client requests the lookup for a list of Jini servers that match the requested attributes. It is then up to the Jini client to select the specific Jini server from the returned list. Figure 3.1 illustrates the process by which a Jini client discovers and accesses services.

Upon connecting to the network, the first step involves locating the lookup service. The Jini client achieves this by using the *discovery* protocol. The discovery protocol involves broadcasting a presence announcement by dropping a multicast packet on a well-known port. This packet contains the client's IP address and port number so that the lookup server can contact it. When the lookup server receives this broadcast request, it returns its address to the Jini client, which stores the address and its link to the lookup server. The client sends further requests for the lookup server directly to this address. Once a Jini device that provides a service identifies a lookup service, it uses the *join* protocol to become a part of the federation. This is achieved by sending a copy of its service item to the lookup service where it is stored.

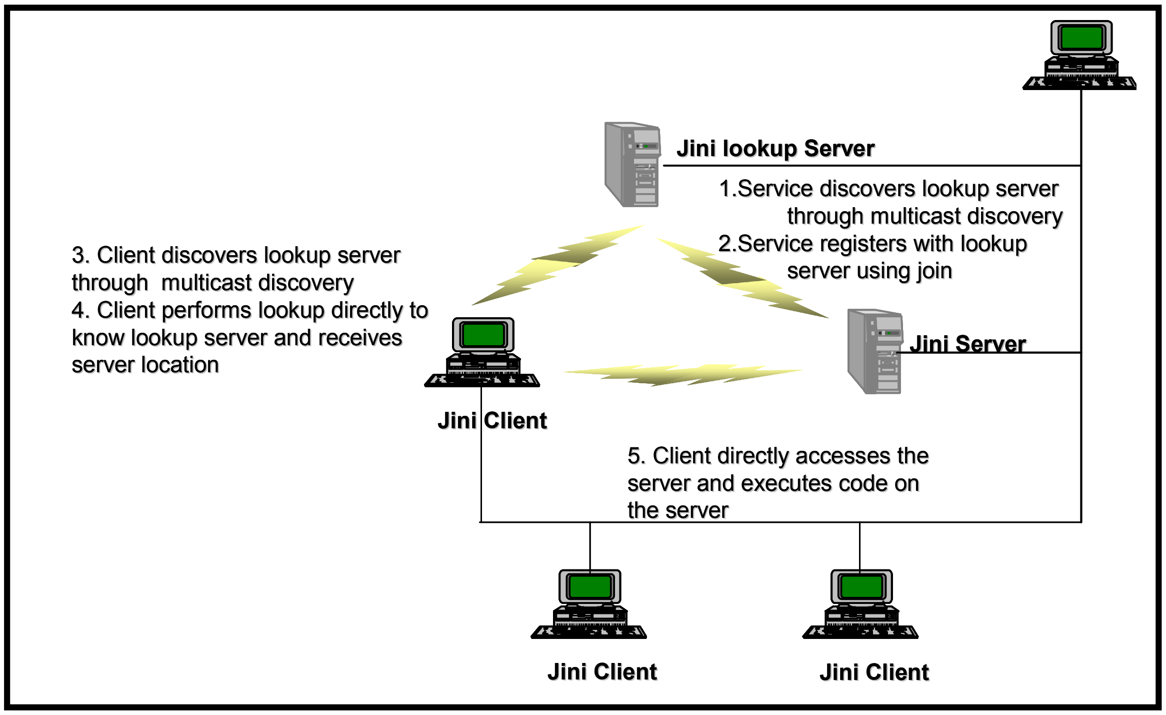

Jini lookup Server
1.Service discovers lookup server through multicast discovery
2.Service registers with lookup server using join
3. Client discovers lookup server through multicast discovery
4. Client performs lookup directly to know lookup server and receives server location
Jini Server
Jini Client
5. Client directly accesses the server and executes code on the server
Jini Client
Jini Client

Figure 3.1 Process of discovery and access of Jini services by Jini clients

Having discussed the lookup service, the next subsection shifts focus to one particular service built on top of the Jini substrate. This service, called JavaSpaces provides a storage facility for Java objects.

3.1.2 JavaSpace service

JavaSpaces is a Java implementation of a tuple-space system, and is provided as a service based on Sun's Jini technology. JavaSpaces technology provides a programming model that views applications as a collection of processes cooperating via the flow of objects into and out of one or more spaces. A space is a shared, network accessible repository for objects [21]. Several aspects of distributed computing are inherently handled by the JavaSpaces technology. JavaSpaces provide associative lookup of persistent objects. It also maintains security through transactions. Using space-based implementation allows transacting executable content across the network. This decouples the semantics of distributed computing from the semantics of the problem domain. A loose coupling like this means that the two elements can be managed and developed independently [22]. This eliminates issues around multithreaded server implementation, low-level synchronization, or network communication protocols, usual requirements of distributed application design. This design offers two key advantages: *Load Balancing:* The workers stay busy and compute tasks in relation to their availability and ability to do the work. *Scalability:* The more number of workers added to the computation improves scalability.

3.1.2.1 Key Features

Just as file systems provides data storage facilities, JavaSpaces provides a storage facility for objects. Unlike files or raw un-typed data stored in file systems, the items stored in JavaSpaces are Java objects. This provides more information since the object represents the actual class code and several behavioral aspects that come with the definition of the object. It also leverages features that come with Java objects such as strong typing, mobile code, and secure execution.

Described below are some of the key features provided by JavaSpaces:

- *Spaces are shared:* Spaces are network accessible shared memories that many remote processes can interact with concurrently. A space itself handles the details of concurrent access, leaving the programmer to focus on client design and the protocols between them. The shared memory model also allows multiple processes to simultaneously build and access distributed data structures, using objects as building blocks.
- *Spaces are persistent:* Spaces provide reliable storage for objects. Once stored in the space an object will remain there until a process explicitly removes it. Processes can also specify a lease time for the object, after which it will be automatically destroyed and removed from the space.
- *Spaces are associative:* Objects in a space are located via associative lookup, rather than by memory location or identifier. Associative lookup using templates provides a simple means of finding the objects of interest, according to its content, without the need to know its name, present owner, creator or location for storage.
- *Spaces are transactionally secure:* The JavaSpaces technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- *Spaces allow exchange of executable content:* While in space, objects are passive data. This data cannot be modified nor can its methods be invoked. However when an object is read or taken from a space, a local copy is created. Like any other local object, its public fields can be modified and its methods can be invoked. This capability presents us with a powerful mechanism for extending the behavior of our applications through a space.

- *Spaces facilitate searching of objects:* Objects residing in the space can be searched based on their class, or super class or the interfaces they implement. Attribute based search can be employed for matching templates to actual entries.

3.1.2.2 Client access to JavaSpaces

As mentioned earlier, space is a Jini service and like all services it has to register with the Jini lookup service to provide the client with a remote reference handle. Alternatively, Sun Microsystems reference implementation of JavaSpaces allows registering with the RMI registery. Figure 3.2 and 3.3 shows a UML interaction diagram illustrating the use of Jini lookup service [23] to access the space proxy.

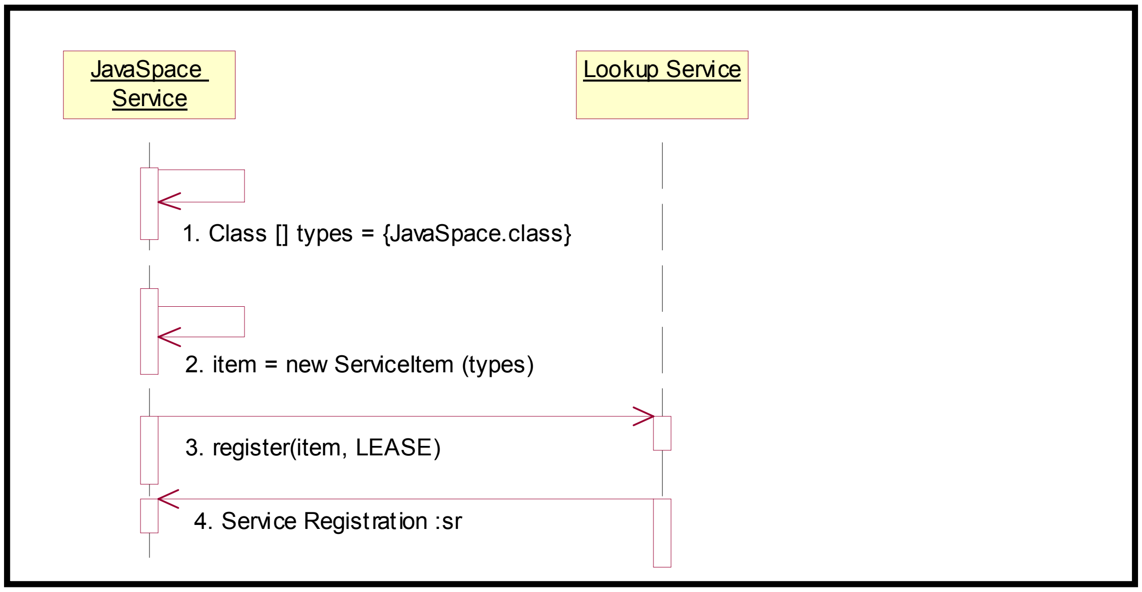

JavaSpace Service
Lookup Service
1. Class [] types = {JavaSpace.class}
2. item = new ServiceItem (types)
3. register(item, LEASE)
4. Service Registration :sr

Figure 3.2 Registration process of JavaSpaces service with the Lookup Service

1. The proxy object of the JavaSpace service implements an interface that is well known by the client. This proxy object is easily downloadable to the clients.
2. The JavaSpace service creates a service item.
3. It registers this service item with the lookup service.
4. The lookup service returns a Service Registration object, which contains its service I.D., and the lease period.

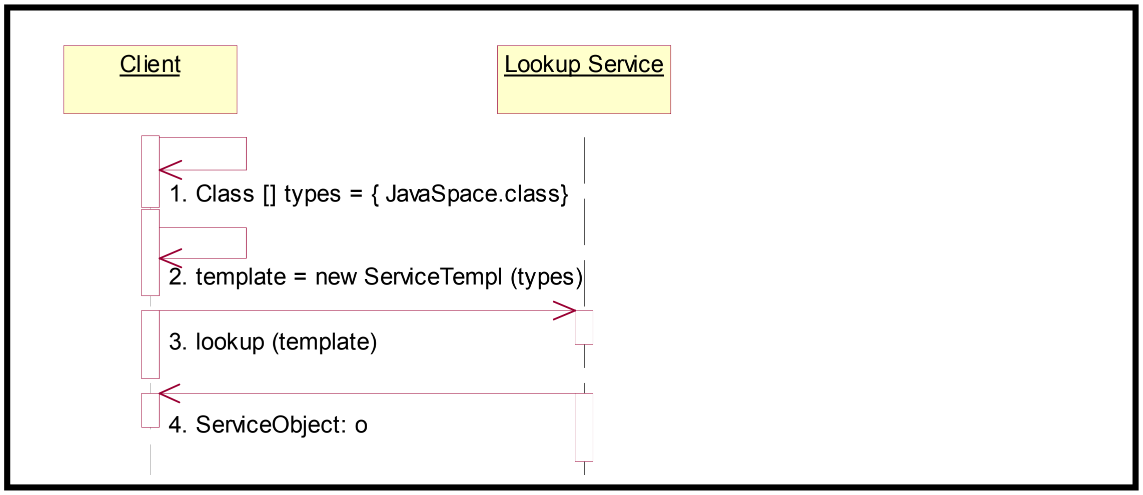

Client
Lookup Service
1. Class [] types = { JavaSpace.class}
2. template = new ServiceTempl (types)
3. lookup (template)
4. ServiceObject: o

Figure 3.3 Client access of JavaSpaces service via the Lookup Service

1. The Client creates a template using the known interface, which the JavaSpace service implements.
2. The Client creates a Service Template.
3. The Client searches the lookup using this service template to check if the JavaSpace service has registered with the lookup service.
4. The lookup service returns a reference of the JavaSpace service, if the JavaSpace service has registered with it.

3.1.2.3 Overview of JavaSpace API

The JavaSpace API leverages the type semantics used in Java and simplifies operations to store, retrieve and lookup objects in the space. The objects are stored in the form of *entries*. An entry is a no-method, tagging interface that an object can implement to participate in space based computations. Given an entry, we can interact with the space using a few basic operations: *write*, *read* and *take*.

- *The write method:* The write method places a copy of an entry into the space. The write method is invoked on the space object obtained after gaining access to the space. Thus it performs create and update functions.

- *The read method:* Once an entry exists in space, any process that gains access to the space can read the entry. Entries are read by associatively matching it with a template. An entry matches a template if it has the same type or subtype as the template, and if, for every specified not null field in the template an exact match is found in the entry. The null fields act as wildcards. The read operation causes the thread of execution to wait until the matching value of the template is obtained or the specified wait period, whichever comes first. Thus it lets programs get a local copy of an object.
- *The take method:* The take operation is similar to read expect it removes the matching entry from the space. It can be thought of as moving the object from the space to the local program. Thus it combines read and delete operations.

These three basic operations are available in different flavors to facilitate access and manipulation of objects in the space. The JavaSpace programming model typically follows the following master worker pattern. In this model, applications or classes use the write operation to place objects into the JavaSpace; they also remove objects from the space for processing with the take operation. When the processing has been completed, the write operation again returns the resulting object to the space.

The reference implementation of JavaSpaces currently runs in a single JVM on a single CPU. There is no support yet for spaces that are distributed over multiple machines. As the federation of clients and services grows to encompass legions of workers, many masters and massive numbers of task and result entries, its easy to see that the use of a single JavaSpace running in a single JVM could pose a potentially serious bottleneck: network traffic could pose a problem, as could storage limitations of the space.

Chapter 4

A Framework for Opportunistic Parallel Computing on Clusters

The framework presented in this chapter employs JavaSpaces technology to facilitate parallel computing on networked clusters. The parallel workload is distributed across the worker nodes using the bag of task model with the master inputting independent problem task into the space and the worker computing results on the tasks taken from the space. The key features of this framework are:

- The framework uses Java as the core programming language to leverage portability across diverse platforms.
- The framework requires minimal configuration management overhead for set up.
- The framework provides a dedicated network management module that uses SNMP to identify idle resources, monitor System State and enable automated system adaptation.
- The framework provides runtime access to the remotely loaded code executed at the worker nodes.

4.1 Targeted Applications

The presented cluster computing framework and the underlying parallel computing model supports applications having the following characteristics:

1. *High Computational Complexity*: The applications must be sufficiently complex so as to require large computational resources.
2. *Massively Partitioned Problems:* The applications must be divisible into relatively coarse-grained subtasks that can be solved individually and independently, and the final solution is built based on the results of these subtasks.
3. *Small Input and Output Sizes:* Each of the subtasks must have relatively small sizes of input and output.

4.2 Framework Architecture

A schematic overview of the framework architecture is shown in Figure 4.1. It comprises of three key components: the Client-side (Master) components, the Server-side (Worker) components and the Network Management Module.

4.2.1 Master Module

The Master component defines the problem domain for a given application. The application domain is broken down into sub tasks that are JavaSpace enabled[1]. The master need not be concerned about who computes its tasks or how, but just throws tasks into the space and waits. All interaction among the master and worker processes occurs in the form of task and result entries exchanged through a single JavaSpace. The JavaSpace registers as a Jini service and relies on Jini for the remote lookup during the discovery phase of the service. It also inherently handles all the low-level communication issues.

4.2.2 Worker Module

The worker component provides the computational content for the application domain. It picks up the next available task from the JavaSpace and performs the required computation repeating the process until there are no more problem tasks left in the space. It also responds to signals resulting from the Inference Engine to start/stop/pause/resume the worker computation. The Master and Worker implementations run as processes within the application layer. The workers need not be Jini aware in order to interact with the Master processes. Interaction with the master process is via a virtual, shared JavaSpace.

4.2.2.1 Remote Node Configuration Engine

The Remote Node Configuration Engine resides on the worker module and handles System configuration and management. Provisions to system configuration such as modification to existing access control mechanisms or download and installation of additional software components required for participation in parallel processing are some examples of the overhead involved. These modifications and overheads must be kept to a minimum. Traditional distributed computing packages such as PVM or MPI require considerable administrative overhead. Additionally, use of native-compiled binaries inhibits inclusion of heterogeneous machines to simultaneously participate in the computations. In an effort to minimize the configuration overhead of deploying worker code, we have implemented a remote node configuration mechanism. This mechanism facilitates remote loading of the worker implementation classes at runtime. The mechanism is described in greater detail in Section 4.3 with the overall operation of the framework.

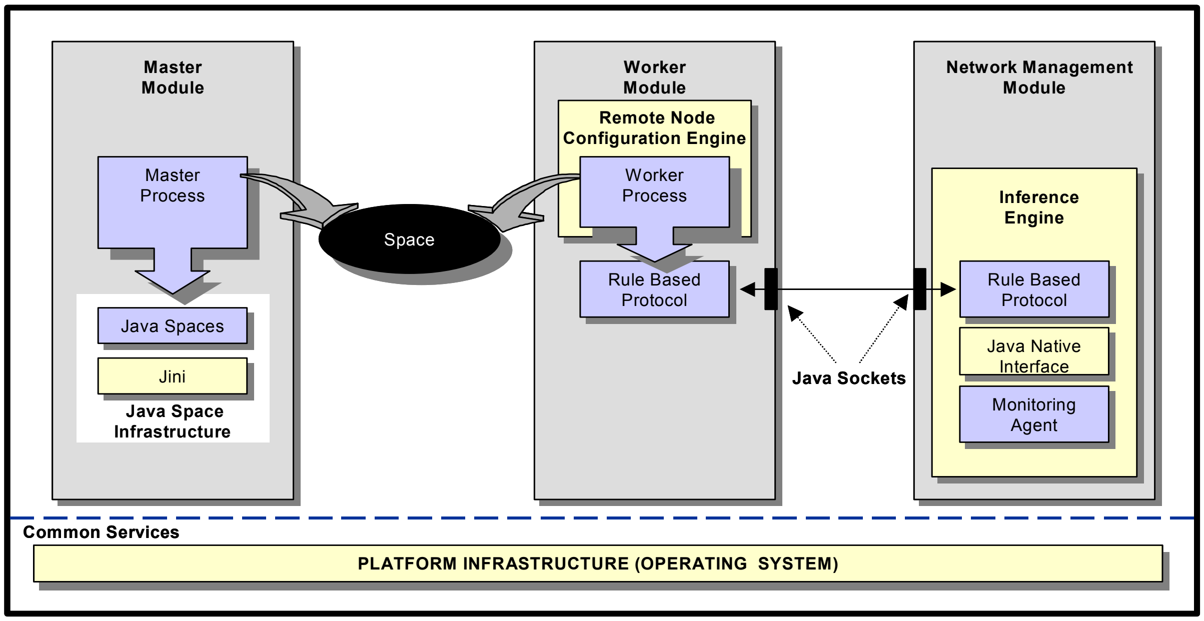


Figure 4.1 Architectural diagram

4.2.3 Network Management Module

In order to support non-intrusiveness while using remote networked resources, we included an idle resource monitor that checks for idle resources and monitors their state. In the presented framework, the network-monitoring agent that runs on each of these machines performs this function. The agent is operational only when the framework is in operation, and is dormant at

[1] JavaSpace required the Objects being passed across the Space to be in a Serializable format. In order to transfer an entry to or from a remote space, the proxy to the remote space implementation first serializes the fields and then transmits it into the space.

all other times. Once the worker host is identified, it is registered with an Inference Engine module. This module queries the System State of the identified worker and dynamically manages the scheduling of tasks to workers using this state information. Task scheduling management is driven by policies defined within the Rule Base. The two entities participating in the Rule base protocol are the worker host and the network manager. The network manager manages all the workers registered with the inference engine. It is responsible for looking up available resource on the registered worker hosts, activating the worker processes, and managing worker thread priorities on the hosts.

4.3 Implementation and Operation

The framework implements the master-worker pattern with JavaSpaces as the backbone. The entire software has been developed in Java to facilitate portability across heterogeneous platforms and to leverage the write once run anywhere feature. The SNMP agent implementation for obtaining system CPU usage values, required for network management was developed in our lab. by Pravin Bhandarkar [24] as a part of his research project. Details on the SNMP agent implementation is presented in Appendix A. Enabling Java access to C system calls to provide CPU utilization information required an additional layer of Java Native Interface (JNI). An overview of the JavaSpaces implementation and its operation is shown in Figure 4.2. The three components are explained below.

4.3.1 Master Module

The Master iterates through all the tasks that need to be computed, creating a task entry for each and writing it into the space. This is the *task-planning* phase. The master then iterates again, this time removing the result for each task processed from the space and combining them into some meaningful result. This is the *result aggregation* phase. Our experience with JavaSpaces shows that these two phases can be relatively time consuming and are dependent upon the size of data that is transferred in and out of the space.

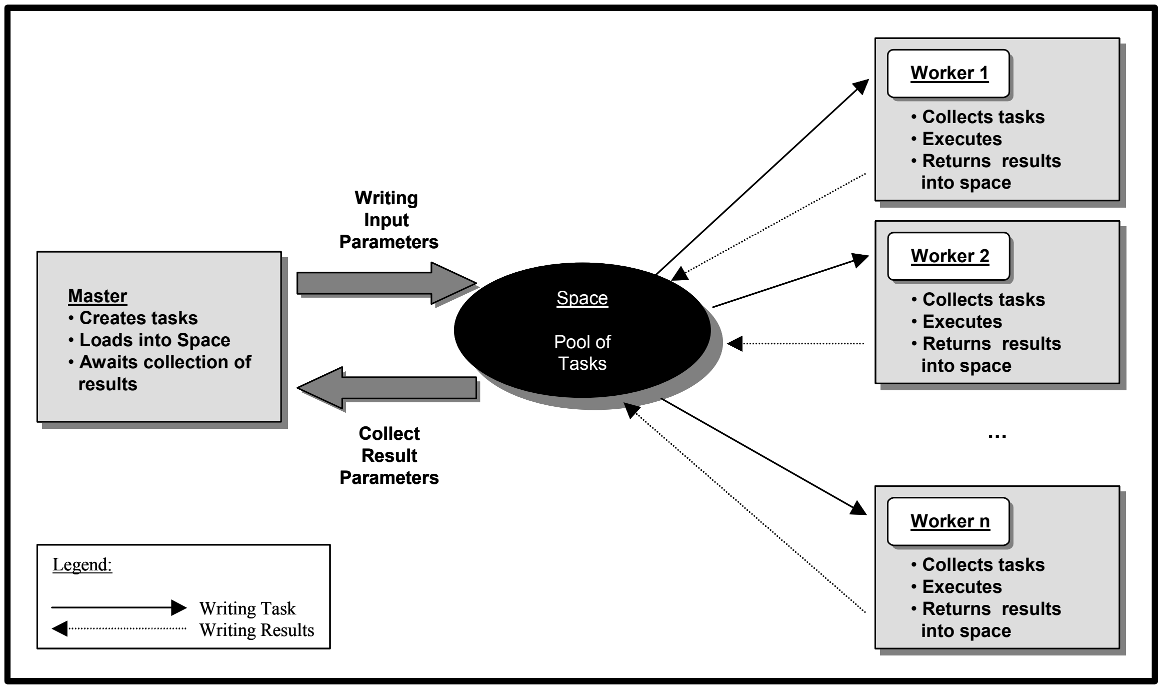


Figure 4.2 Master worker paradigm within our framework

4.3.2 Worker Module

The worker continually removes a task from the space, computes it, and writes the result of the computation back into the space. The result is later read and aggregated by the master process. When taking or reading objects, processes use simple value-matching look up to find the objects of interest in the space. If a matching object isn't found immediately the process waits until one arrives. Matchmaking in JavaSpaces is achieved by referring to each object using a unique Task ID and the space within which it resides.

4.3.2.1 Remote Node Configuration Engine

Our system employs the dynamic class loading mechanism provided by the Java Virtual Machine. Some of the key responsibilities of custom class loaders include locating and fetching the required class files, consulting the security policy, and defining the class object with the appropriate permissions.

Implementation consists of a simple Java application starter program that can load jar and class files from URL at runtime. It is a sub class of the URLClassLoader as supported in JDK1.2.

The required classes for remote configuration of the worker nodes are easily downloadable from the web server residing at the master in the form of executable jar files. The worker implementation classes are loaded at runtime from within the base configuration classes, and the appropriate method to start the worker application thread is invoked. Once the worker class is loaded it is mapped onto a separate name space on the worker machine until the runtime process that invoked it is destroyed. Our modification of the network launcher [25] provides mechanisms to intercept calls from the inference engine and interpret them as signals to the executing worker code. This is important to ensure correct functioning of the Rule Base protocol. The rule base protocol transfers the signal resulting from the Inference Engine to the Worker nodes. This signal must be transmitted to the worker thread at runtime where it will take effect on the worker computation. The remote node configuration engine responds to signals received from the inference engine. However, preemptive execution of the signal by the worker might leave the worker in an unstable state. For example, if the worker thread is in the middle of a computation and a stop signal received by the parent process forces the worker thread to terminate, the task that is taken from the space is not yet returned resulting in a lost task. In order to ensure that no work is lost such a behavior is unacceptable. Thus, understanding the state of the worker process at the time when the signal is received is very critical to the functioning of this module. The child process's standard output is piped into the parent process's standard input and used as a means of communication with the worker thread. The parent process that listens to signals from the Inference Engine also spawns off a thread to listen to the child process. From the workers perspective, a stable state to stop the worker thread is achieved when it has completed its current task execution and before it fetches the next task. This state is communicated to the parent process via the standard output.

4.3.3 Network Management Module

The Network Management Module serves two functions, namely; monitoring the worker machines for system state parameter and devising a decision-making mechanism to facilitate non-intrusive code execution. In our current implementation, we monitor CPU usage and study the load patterns from successive runs of the JavaSpace implementation. The SNMP layer provides an interface to the underlying WinSNMP DLLs that query the worker's SNMP agent to get the relevant parameter. This layer was built on top of the Windows architecture. The tool allows the user to simultaneously monitor various hosts. It consists of two main components: the manager component that runs on the SNMP server and the NT agent component that runs on the host to be monitored. The NT agent is based on the Microsoft Extension API and provides the basic functionality for constructing an extension agent dynamic link library (DLL) capable of communicating with the SNMP service and interacting with network management application using SNMP. Once the SNMP service is activated on a host, the NT agent DLL is loaded as an extension agent DLL by the SNMP service. When a request message from the manager component is received, the querying process is invoked. Each request message contains one or more variable bindings. For processing each variable binding, the matching between the OID (Object ID) of the NT agent MIB (Management Information Base) variable and OID specified in the variable binding is checked first. Next their attributes are compared and finally appropriate system parameter values are returned if all the SNMP security checks have been passed. This information is used to establish triggering thresholds. If the CPU usage exceeds the established threshold, the current task is completed and its results are returned to the space, and no other task is assigned to the loaded worker. This mechanism provides smooth clean-up and synchronization.

4.3.4 Operation of the Rule Base Protocol

The Rule Base Protocol defines the interaction between the Network Management Module and the Worker Module (see Figure 4.3). The messaging protocol is implemented using Java sockets. Its operation is described as follows:

The SNMP Client on the worker module initiates participation into the parallel computation by registering with the SNMP Server. The server invokes the SNMP service as well as communicates with the Inference Engine. The Inference Engine maintains a list of all machines whose system parameters need to be monitored and assigns a unique ID to the new worker. Next it adds the workers IP address to the list. The SNMP server then continues to retrieve the requested system state parameter.

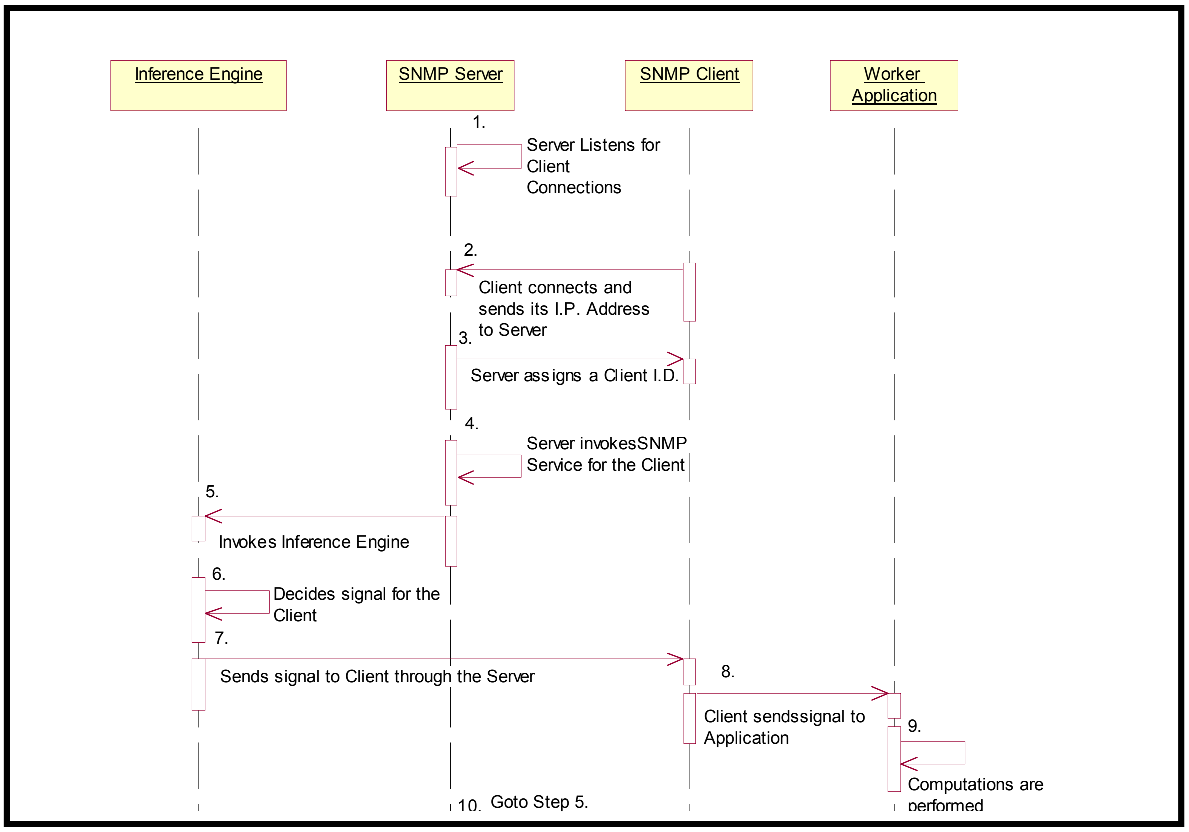

Inference Engine
SNMP Server
SNMP Client
Worker Application
1.
Server Listens for Client Connections
2.
Client connects and sends its I.P. Address to Server
3.
Server assigns a Client I.D.
4.
Server invokesSNMP Service for the Client
5.
Invokes Inference Engine
6.
Decides signal for the Client
7.
Sends signal to Client through the Server
8.
Client sendssignal to Application
9.
Computations are performed
10. Goto Step 5.

Figure 4.3 Sequence diagram for the rule base protocol

The SNMP value measured is the averaged value of the worker's CPU utilization. As these values are returned they are added to the respective entry in the list. Based on this return value and the established threshold ranges, the Inference Engine makes a decision and passes an appropriate signal back to the worker. Threshold values for the signals are based on heuristics. The Rule Base currently encoded allows for four types of signals in response to the varying load conditions; namely: *start, stop, pause* and *resume.* In response to these signals the worker process can be in one of four states, viz. *running, stop, pause* (see figure 4.4).

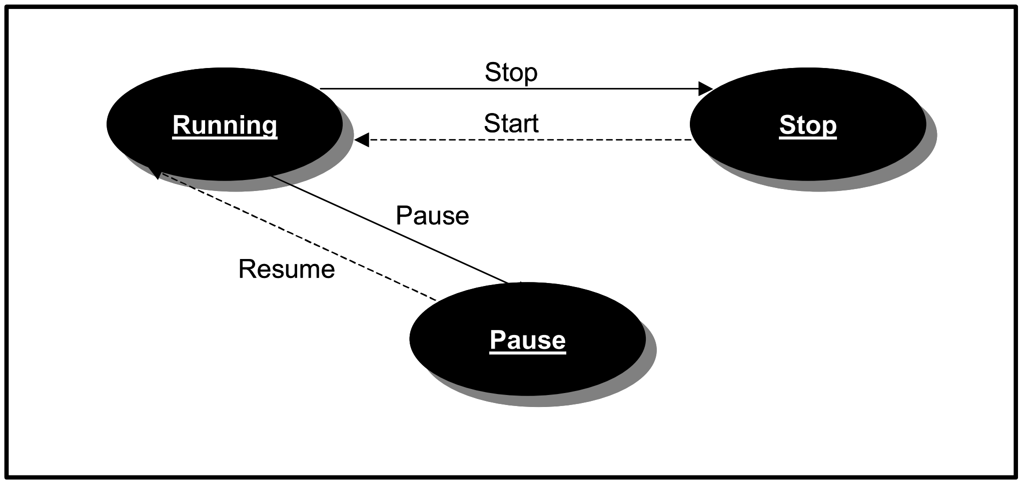


Figure 4.4 Worker state transition diagram

Running: This state indicates that the worker node is can now be considered as idle, and can start participating in the parallel application. A Start or Resume signal is sent when the CPU load at the worker is in the range of 0% - 25%. On receiving the start signal, the protocol implementation on the worker initiates a new runtime process for actual work execution. The new thread first goes through the remote class loading phase and then starts off a worker thread for task execution. However, if the worker receives a resume signal after it is in the pause state, it does not require loading of the worker classes since they are already loaded into the workers memory. It simply removes the lock on the interrupted execution thread and resumes computations.

Stop: This state indicates that the worker node can no longer be used for computations. This may be due to a sustained increase in CPU load caused by a higher priority (possibly interactive) job being executed. The cutoff threshold value for the stop state is in the range of 50% to 100%. Upon receiving the stop signal, the worker gets a handle into the worker thread and sends it the stop signal. At this point the worker backs off. It first interrupts the executing worker thread and then executes the shutdown/cleanup mechanism. The shutdown mechanism ensures that the currently executing task completes and its results are written into the space. After cleanup, the worker thread is killed and control returns to the parent process. Hence the next startup will require reloading of the worker classes at runtime.

Pause: This state indicates that the worker node is experiencing increased CPU loads and is not completely idle and hence it should temporarily not be used for computation. However, the load increase might be transient and node could be reused for computation in the near future. Threshold values for the pause state are in the range of 25% - 50%. Upon receiving this signal the worker gets a handle into the worker process and sends it the pause signal. At this point the worker is supposed to back off, but unlike the stop state the back off is temporary – until it gets the resume signal. This minimizes overheads for transient load fluctuations at the worker. As in the stop state, the pause goes into effect only after the worker writes the results of the currently executing task into the space. However the worker process is not destroyed in this state in this state but only interrupts the execution until the resume signal is received.

Chapter 5

Experimental Evaluation of the Framework

In this chapter we evaluate the JavaSpaces based opportunistic cluster-computing framework with three applications: 1) a real world financial application that uses Monte Carlo (MC) simulation for Option Pricing, 2) a scientific Ray Tracing application and 3) a pre-fetching scheme for optimal web page accesses.

5.1 Application Description

The three applications presented are broadly compared in table 5.1 with reference to metrics such as scalability, CPU memory requirements, and inter-dependency within the application should it exist. The following sub sections are organized by first describing the applicability of the applications in the real world and then presents the implementation approach.

Metrics	Option Pricing Scheme	Ray Tracing Scheme	Pre-fetching Scheme
Scalability	Medium	High	Low
CPU Memory Requirements	Adaptable depending on number of simulations	High	Low
Dependency	No	No	Yes

Table 5.1 Classification of the Evaluated Applications

5.1.1 Parallel Monte Carlo Simulation for Stock Option Pricing

A stock option is a derivative, that is, its pricing value is derived from something else. Parameters such as varying interest rates and complex contingencies can prohibit analytical computation of options and other derivative prices. Monte Carlo simulation using statistical properties of assumed random sequences is an established tool for pricing of derivative securities.

An option is defined by the underlying security, the option type (call or put), the strike price and the expiration date. Additionally, there are various factors that affect the pricing of an option such as interest rate and volatility. These financial terms are explained in greater depth in Glossary of Financial terms [26]. For our implementation we account for the various factors, and model the behavior of options using Monte Carlo (MC) simulations based on the Broadie and Glasserman MC algorithm [27].

Implementation: The main MC simulation based on the input parameters is the core parallel computation in our experiments. Input parameters are defined via a GUI as provided in our implementation. The simulation domain is divided into tasks and MC simulations are performed in parallel on these tasks. The number of simulations performed can change for each task. High and low estimates are obtained over a wide range of simulations. For the experimental evaluation presented below, the Number of Simulations is set to 5000. This is the seed of the MC simulation. The problem domain is divided into 50 sub tasks, each comprising of 100 simulations. The MC simulation consists of two iterations to compute the estimated sum. Since there is no inter-iteration dependency, 50 sub tasks spread over two iterations results in 100 sub tasks. These tasks are made available in a JavaSpace pool. Each worker process selects a sub task from the pool and performed the MC simulation on the task.

5.1.2 Parallel Ray Tracing

Ray Tracing is an image generation technique that simulates light behavior in a scene by following light rays from the observer's eyes as they interact with the scene and the light sources. The algorithms estimate the intensity and wavelengths of light entering the lens of a virtual camera in a simulated environment. These quantities are estimated at discreet points in the image plane that corresponds to pixels. The estimates are taken by sending rays out of the camera and into the scene to approximate the light reflected back to the camera. This process requires identifying points of intersection among rays and objects in the environment, a common geometrical problem known as *ray casting*. The cost to compute individual pixels can vary dramatically, depending on the complexity of the model being rendered and the algorithm employed. In the present context, the problem is to distribute the calculations for a set of pixels in an image in order to minimize the elapsed rendering time. Therefore the basic computation performed by the ray tracer is the calculation of the intersection points between rays and the objects.

Implementation: Ray Tracing [28] begins with a model of the scene, and an image plane in front of the model that is divided into pixels. Rendering an image involves iterating through all the pixels in the plane and computing a color value for each pixel. The task of computing the color value of each pixel involves tracing the rays of light that pass from a viewpoint (such as your eye) through the pixel in the image plane, and to the model. The computation is identical for all pixels, except that the parameters describing the pixel's position differ. Image ray tracing is an ideal candidate for the replicated-worker pattern, since it is made up of a number of independent tasks that are computationally identical. For our implementation we divide the 600X600-image plane into rectangular slices of 25X600 pixels thus creating 24 independent tasks. Thus the input is small comprising of the four coordinates that describe the region of computation whereas the output is large comprising of an array of pixel values. The master process generates these tasks comprising of the X and Y coordinates for computation and drops it into the JavaSpaces pool. Each worker computes the scan lines and returns a result array of pixel points. The master then collects the results and combines them to compose an image.

5.1.3 Web Page Pre-fetching Scheme Based on Page Rank

The objective of this application is to optimize web access time incurred by the web user. Web pre-fetching builds on web caching to improve file access time at the web server. The Page Rank-based pre-fetching approach [29], [30], [31] uses the link structure of a requested page to determine the "most important" linked pages and to identify the page(s) to be pre-fetched. The underlying premise of the approach is that the next page requested by the users is typically based on the current and previous pages requested. Furthermore, if the requested pages have a lot of links to some "important" page, that page has a higher probability of being the next one requested. The relative importance of pages is calculated using the Page Rank method as described above. The important pages are identified and then pre-fetched into the cache for faster accessibility. Thus web pages can be pre-fetched in clusters, thus reducing the server fetch times. Clustered accesses are accesses to closely related pages. For example, access to the pages of a single company or research group.

Implementation: For each page requested, the Page Rank algorithm performs the following operations. First the URL is scanned to see if it belongs to a cluster. If it does, the contents of retrieved pages are used to populate or update that cluster's matrix. The cluster's matrix referred here is a stochastic matrix constructed to calculate the actual rank of a page. The matrix is constructed as follows:

1. Each page *i* corresponds to row *i* and column *i* of the matrix.
2. If page *j* has *n* successors (links), then the *ij*th entry is 1/*n* if page *i* is one of those *n* successors of page *j*, 0 otherwise.

After the update operation, Page Rank calculations are performed to determine the most important pages among those requested or pointed to in the cluster. The core of the Page Ranking methodology comprises of basic matrix manipulations such as addition and multiplication of two matrices in a While loop until a specified condition is fulfilled. However there is an inter-iteration dependency in the DO-WHILE loop for Page Rank calculation. Hence we do not expect to see a speed up with parallel processing. The two matrices under consideration are 500x500 and 500x1 in size. The matrices are split to perform strip block manipulation. The segment size for each strip is kept to be 20 leading to 25 sub tasks for each iteration. Each page request spawns off a new Master thread hence multiple cluster matrices can be handled simultaneously.

5.2 Experimental Environment

The framework was evaluated on the Windows NT (version 4.0) operating system substrate. The pre-fetching scheme and the parallel ray tracing applications were tested on five

Intel Pentium III running on 256 MB RAM at 800 MHz. The option-pricing scheme was tested on a larger cluster of thirteen PCs. Due to the high memory required to host the Jini infrastructure Pentium III processor with 256 MB RAM and 800 MHz was used for the master machine, while the other worker and SNMP Server machines were configured as 64 MB RAM and 300 MHz Pentium III processors.

5.3 Experiments and Results

Three experiments were conduced to analyze the performance of our framework. The aim of the first experiment was to study the scalability of the application and our framework, and to demonstrate the potential advantage of using clusters for parallel computing. Scalability analysis for the Option Pricing scheme shows that the framework scales over 13 machines under consideration. The other two applications were tested with 4 machines to study the scalability of the applications executing within the framework. The second experiment measured the costs of adapting to system state. It measured the overheads of monitoring the workers, signaling, and state-transitions at the workers. We used a set of synthetic load generators to simulate dynamic load conditions at different worker nodes. Finally, the third experiment demonstrates the ability of our framework to adapt to the cluster dynamics. Results for the three experiments using the applications described above are presented in the following sub sections.

The following sub sections are organized as follows. For each experiment, we present results from the option pricing application, followed by the ray tracing application; finally presenting results from the pre-fetching scheme.

5.3.1 Scalability Analysis

This experiment measures the overall scalability of the application under the framework using a cluster of Windows NT workstations. Results are plotted with timing measurements as the number of worker nodes participating in the computation is increased. For each run of the application four curves were plotted namely: Max Worker time, Task Planning time, Task Aggregation time and Parallel Time. For each worker, total worker computation time is measured starting from the first access of task from the space until it puts the final result back into the space. The max worker time is the maximum of the total worker computation times for all workers participating during a given run. The Task Planning time is measured at the master process and measures the time required for the task planning phase. This involves division of the problem domain into sub tasks and inputting each sub task as an entry into space. The Task aggregation time is also measured at the master process and measures the time required for collecting results from the space and aggregating them into a meaningful solution. The Parallel time is measured at the master process and measures the entire computation time from start to finish. The task aggregation curve is expected to follows the Max worker curve, since the master needs to wait for the last task computation in order to aggregate the resultant data into a meaningful solution.

5.3.1.1 Parallel Monte Carlo Simulation for Stock Option Pricing

Scalability of the Option Pricing scheme is plotted in Figure 5.1.

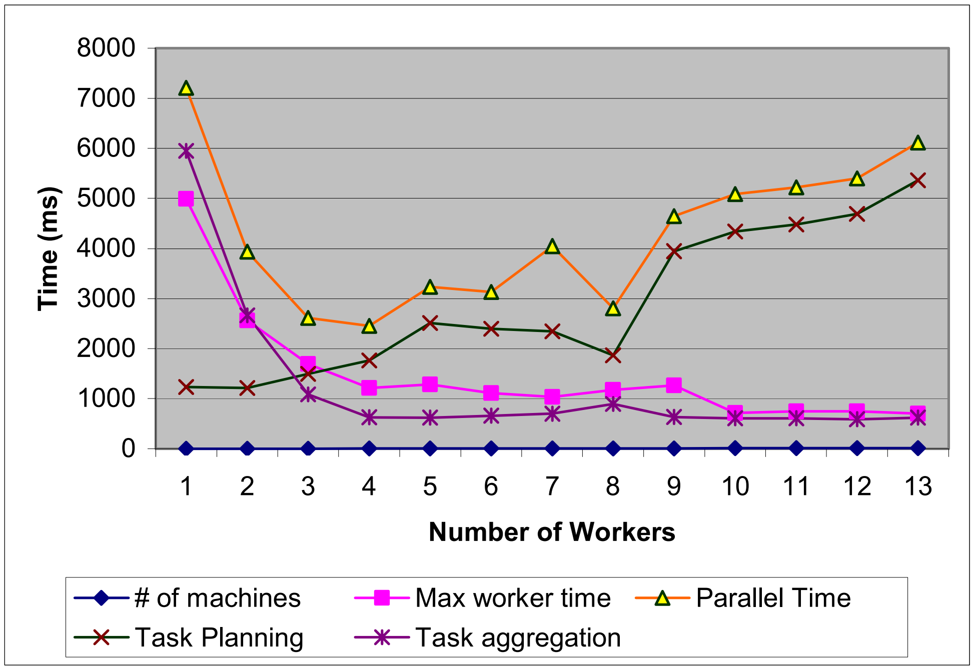


Figure 5.1 Graph depicting the scalability analysis for option pricing scheme

As shown in the figure 5.1 an initial speedup is obtained as the number of workers is increased. The application gains considerable speedup until 4 processors beyond which it deteriorates. The part of the total parallel time curve from 0 to 4 processors closely follows the maximum worker time. As the number of workers increases the model spreads the total tasks more evenly across the available workers. Hence the maximum (Max) worker time evens out as the number of workers increases. However, after a point we notice that the total parallel time is dominated by the Task Planning time. That is the workers are able to complete the assigned task and return to the space much before the master gets a chance to plan a new task and put it into the space. Hence the workers remain unused until the task is made available. As a result the scalability deteriorates. This indicates that the framework favors coarse-grained tasks that are compute-intensive.

5.3.1.2 Parallel Ray Tracing

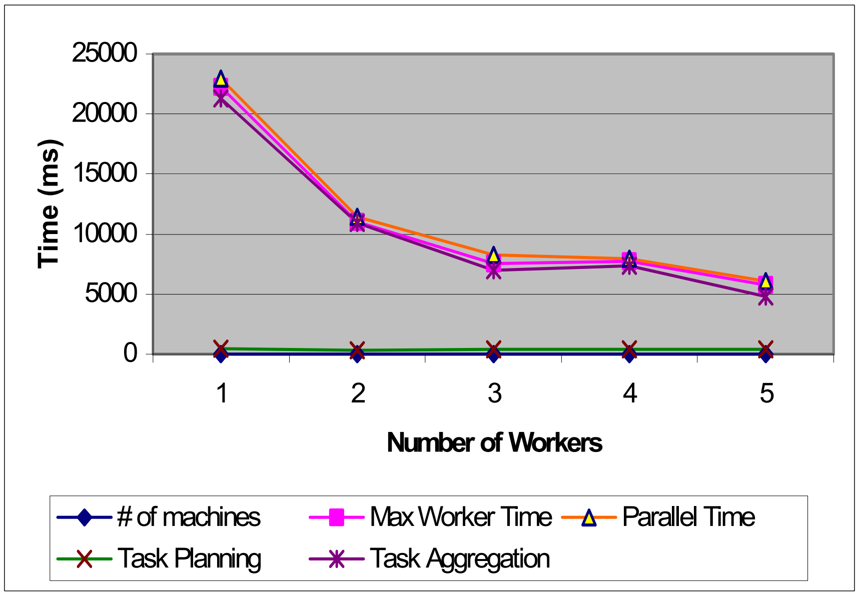


Figure 5.2 Graph depicting the scalability analysis for ray tracing scheme

Scalability results for the parallel ray tracing application are plotted in Figure 5.2. As shown in the figure, the task-planning curve is markedly consistent at 500 ms. The worker time scales well resulting in a speedup as the number of workers increases. The worker computation for the Ray Tracing application is resource intensive. Hence the worker time dominates the total parallel time experienced master. The task aggregation curve follows the Max. Worker curve. Embarrassingly parallel applications such as the Ray Tracing scheme described scale well and are suited to the parallel computing framework developed.

5.3.1.3 Web Page Pre-fetching Scheme Based on Page Rank

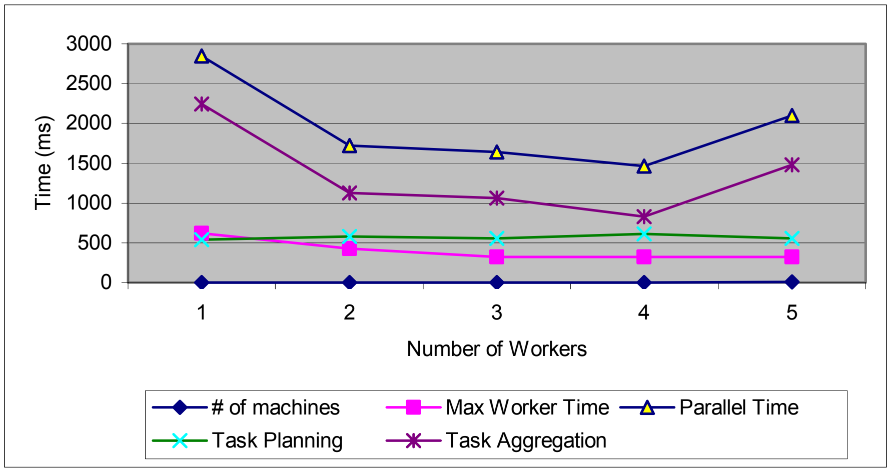


Figure 5.3 Graph depicting the scalability analysis for pre-fetching scheme

Scalability results for the pre-fetching scheme based on Page Rank are plotted in Figure 5.3. As shown in the figure the application scales well until 4 processors. The speedup obtained until 4 processors deteriorates as the number of workers is further increased. From the experiment we observed that the number of parameters being passed to and from the space are very limited thus limiting the task planning overhead. The low task planning overhead is attributed to the minimal data transfer between the master and worker processes. However once the results are collected from the space, assimilation of data into the resultant matrix takes bulk of the time. The increased task aggregation times as shown in the plot illustrate this fact. Thus the total parallel time is mainly dominated by the Task Aggregation time.

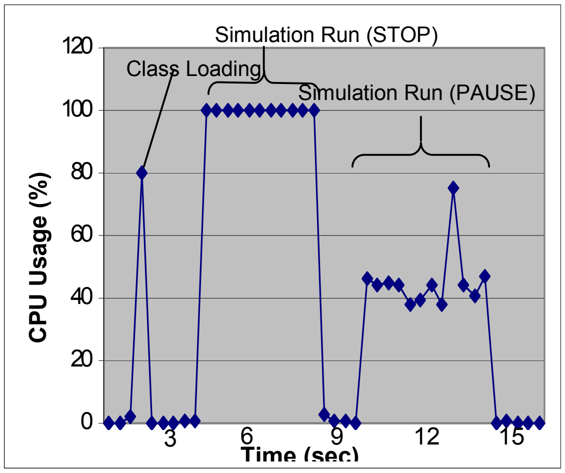
The segment size of the strips can be further optimized in order to facilitate faster computing.

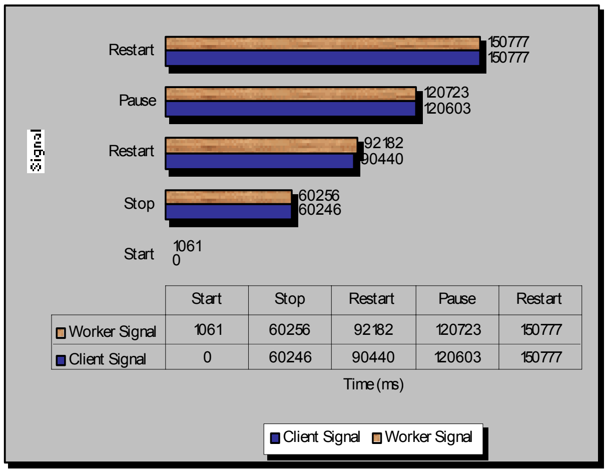
5.3.2 Adaptation Protocol Analysis

In this experiment, we provide a time analysis to illustrate the overhead involved in signaling worker nodes and adapting to their current CPU load. As a part of the experimental setup, we built two sets of load simulators: load simulator 1 simulated varied types of data transfers such as RTP Packets of Voice Traffic, HTTP Traffic, and Multimedia traffic over HTTP via Java sockets originating from the worker hosts. This load simulator was designed to raise the CPU usage level on the worker from 30% to 50% utilization. The second load simulator (load simulator 2) raised the CPU utilization of the worker machines to 100%. Part (a) of the results plotted illustrates the CPU usage history on the worker machine throughout the simulation run. Part (b) provides timing results of Client Signal times and worker signal times. Client signal time is the time at which the SNMP client on the worker machine receives the signal. The Worker signal time is time taken for the signal to reach the executing worker code and manifest itself into the respective action. The Start time of the Client Signal is taken as a reference point in this experiment. We expect the adaptation overhead to be minimal in all cases. Furthermore, the large overhead associated with remote class loading is avoided in the case of transient load increase at the node using the pause/resume states.

5.3.2.1 Parallel Monte Carlo Simulation for Stock Option Pricing

Figure 5.4(a) and 5.4(b) depict the Worker behavior for the Option Pricing scheme under simulated load conditions. In figure 5.4(a), the peaks are identified as times where the worker reacts to the signals sent to it. The first peak at 80% CPU usage occurs when the worker is started. This sudden load increase is attributed to the remote class loading of the worker implementation. Next, load simulator 2 is started which sends the CPU usage to 100%. This causes a Stop signal to be sent to the worker node. The load simulator 2 is then stopped and load simulator 1 is started which raises the CPU load to 46%. As shown in Figure 5.4(b) the worker reaction times to the signal received is minimal in all cases.



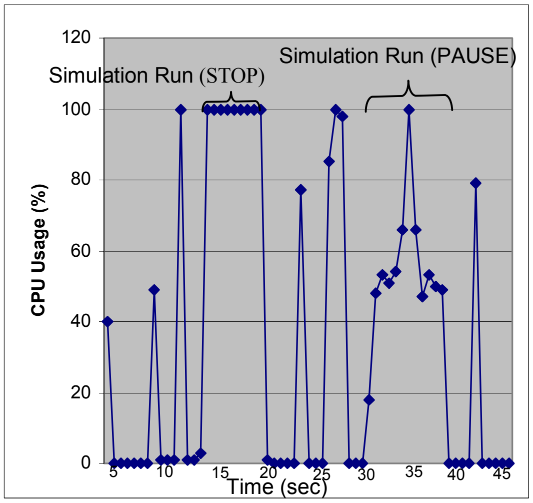


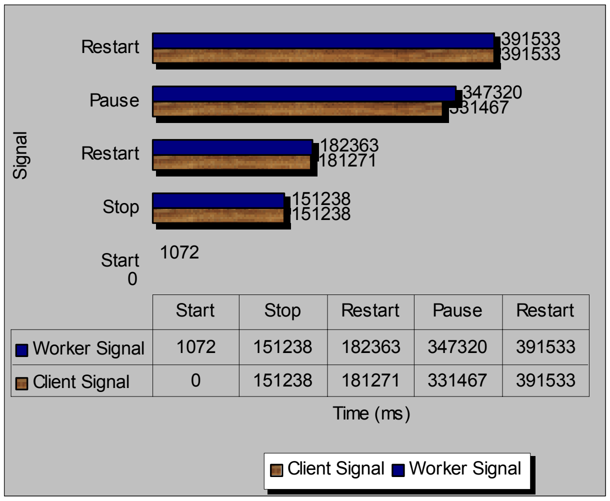
	Start	Stop	Restart	Pause	Restart
▨ Worker Signal	1061	60256	92182	120723	150777
■ Client Signal	0	60246	90440	120603	150777

Figure 5.4(a) Graph of CPU usage **Figure 5.4(b) Graph of worker reaction times**

Adaptation Protocol Analysis for option pricing scheme

5.3.2.2 Parallel Ray Tracing





	Start	Stop	Restart	Pause	Restart
■ Worker Signal	1072	151238	182363	347320	391533
▨ Client Signal	0	151238	181271	331467	391533

Figure 5.5(a) Graph of CPU usage **Figure 5.5(b) Graph of worker reaction times**

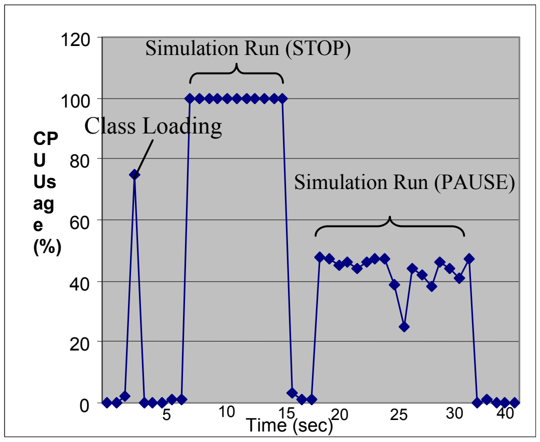
Adaptation Protocol Analysis for ray tracing scheme

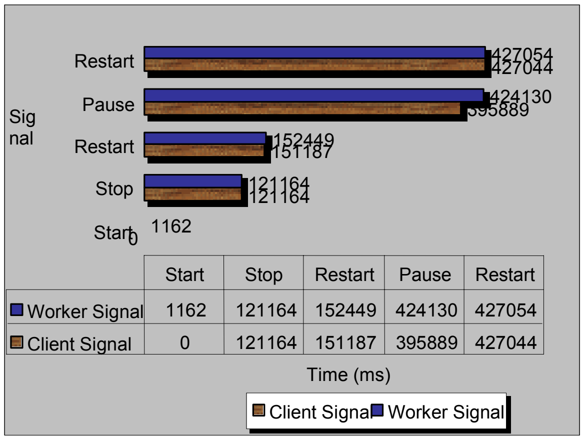
Figure 5.5(a) and 5.5(b) depict the Worker behavior of the parallel ray tracing application under simulated load conditions. In figure 5.5(a), the first peak at 42% CPU usage occurs when the worker is started. This sudden load increase is attributed to the remote class loading of the worker implementation. Next, load simulator 2 is started which sends the CPU usage to 100%.

This causes a Stop signal to be sent to the worker node. The load simulator 2 is then stopped and load simulator 1 is started which raises the CPU load in the range of 50 to 55%. As shown in Figure 5.5(b) the worker reaction times to the signal received is minimal in all cases. The Ray Tracing application is truly a resource intensive application. The various intermittent peaks falling in the range of 78 to 100% CPU usage illustrate this. These spikes of load increase occur when the actual task is being computed at the worker node.

5.3.2.3 Web Page Pre-fetching Scheme Based on Page Rank

Figure 5.6(a) and 5.6(b) depict the Worker behavior under the simulation conditions.





	Start	Stop	Restart	Pause	Restart
▨ Worker Signal	1162	121164	152449	424130	427054
▨ Client Signal	0	121164	151187	395889	427044

Figure 5.6(a) Graph of CPU usage **Figure 5.6(b) Graph of worker reaction times**

Adaptation Protocol Analysis for pre-fetching scheme

As shown in figure 5.6(a), the first peak at 75% CPU usage occurs when the worker is started. This sudden load increase is attributed to the remote class loading of the worker implementation. Next, load simulator 2 is started which sends the CPU usage to 100%. This causes a Stop signal to be sent to the worker node. The load simulator 2 is then stopped and load simulator 1 is started which raises the CPU load to 50%. As shown in Figure 5.6(b) the worker reaction times to the signal received is minimal as expected.

5.3.3 Dynamic Behavior Patterns under Varying Load Conditions

This experiment studies the dynamic behavior patterns under varying load conditions. It consists of three runs: In the first run none of the workers were loaded. In the second and third runs, the load simulator was run to simulate high CPU loads of 25% and 50% on the available workers respectively. Two plots are obtained for each application run under this experiment. The first plot presents a time analysis describing the application behavior under the influence of the simulator loads. The four parameters measured are Maximum Worker Time, Maximum Master Overhead, Task Planning and Aggregation time, Total Parallel time. The maximum worker time is the maximum instantaneous value for worker computation for a given task for all workers participating during a given run. The maximum master overhead is the maximum time taken by the master for task planning or aggregation. Both the maximum worker time and the maximum master overhead are expected to remain constant for all three runs of the experiment. The task planning and aggregation is total time taken by the master during the task planning and aggregation phases. The Total Parallel time is measured at the master process and measures the entire computation time from start to finish. Both the Task Planning and Aggregation times and the Total Parallel times is expected to increase with an increase in load on the worker machines. The second plot illustrates the work distribution among all the workers for the three runs.

5.3.3.1 Parallel Monte Carlo Simulation for Stock Option Pricing

As illustrated in Figure 5.7(a), as the number of worker hosts being loaded increases, the total parallel computation time increases. The computational tasks that would have been executed normally are now off loaded from the loaded workers and picked up by other executing workers. The task planning and aggregation times also increase since the master needs to wait for the worker with the maximum number of tasks to return the last task back into the space. The maximum master overhead and the maximum worker time remains the same across all three runs as expected.

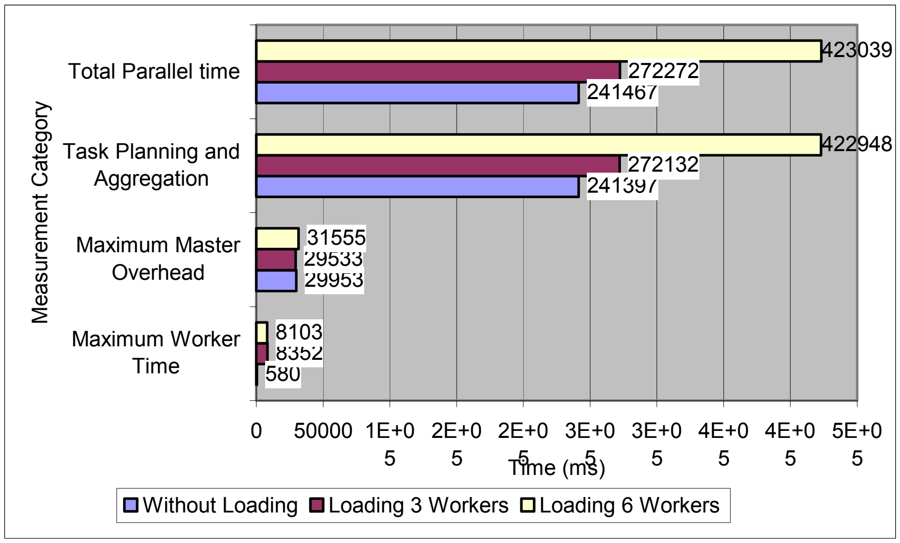


Figure 5.7(a) Graph depicting time measurements across 12 Workers for option pricing scheme

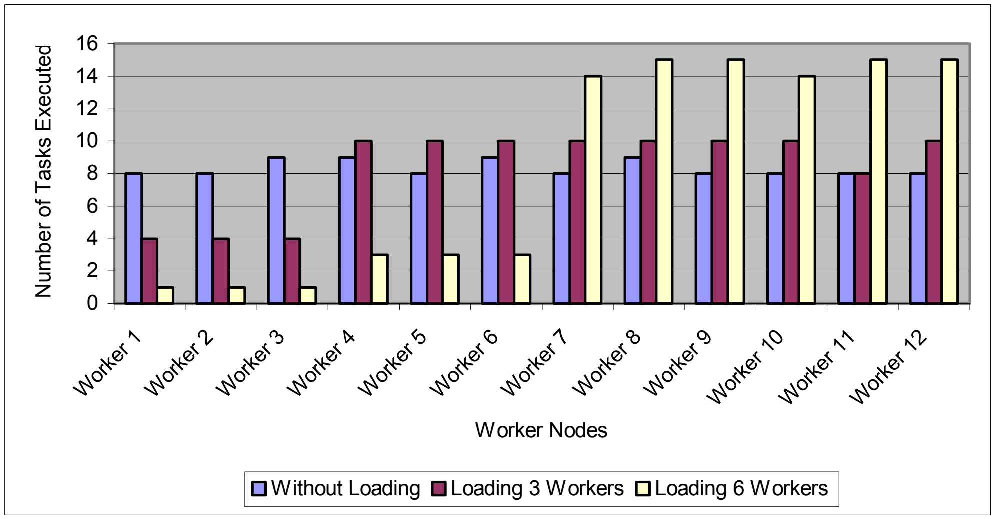


Figure 5.7(b) Measurements of number of tasks executed across 12 Workers for option pricing scheme

Figure 5.7(b) illustrates how task scheduling adapts to varying load conditions. It shows that the number of task executed by each worker depends on its current load. Loaded workers execute fewer tasks causing the available workers to execute larger number of tasks.

5.3.3.2 Parallel Ray Tracing

As illustrated in Figure 5.8(a), the task planning and aggregation times and total parallel time increase as the number of loaded workers increase. However, we see a sharp increase in the

Max. Worker time and the Maximum master time. This is attributed to three tasks that have delayed the overall completion time. Probable cause for this latency is the system or network conditions at that instant. It may be noted that Jini being a network-based protocol does not offer any real-time guarantees. Figure 5.8(b) illustrates the task distribution across the worker hosts.

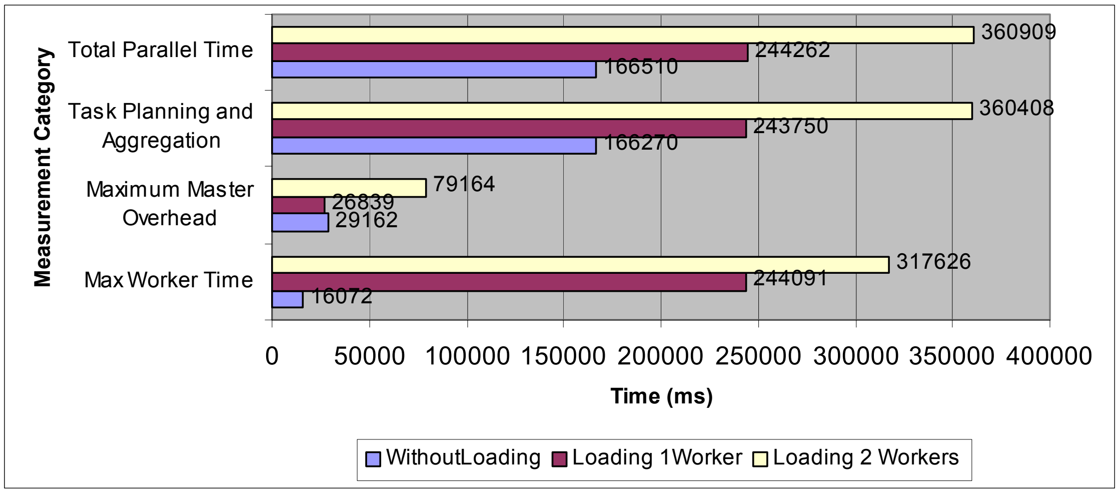


Figure 5.8(a) Graph depicting time measurements across 4 Workers for ray tracing scheme

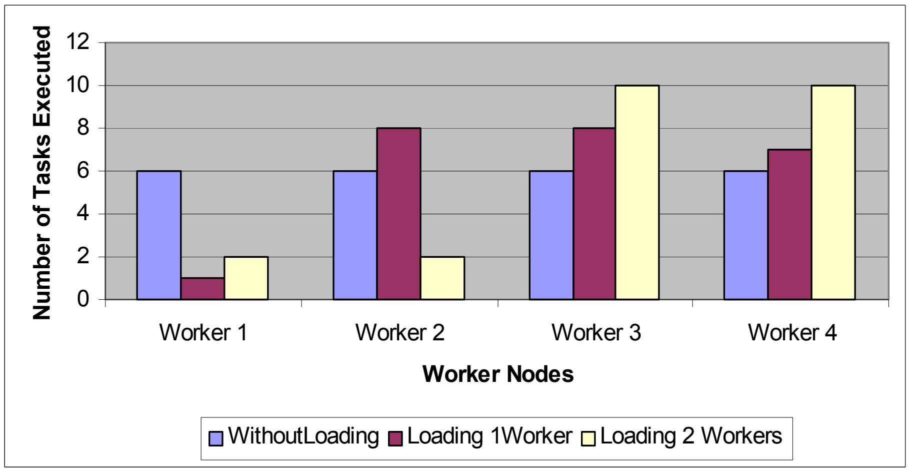


Figure 5.8(b) Measurements of number of tasks executed across 4 Workers for ray tracing scheme

5.3.3.3 Web Page Pre-fetching Scheme Based on Page Rank

As illustrated in Figure 5.9(a), as the number of worker hosts being loaded increases, the total parallel computation time increases. The task planning and aggregation times also increase since the master needs to wait for the worker with the maximum number of tasks to return the last task back into the space. The maximum master overhead and the maximum worker time remains the same across all three runs as expected.

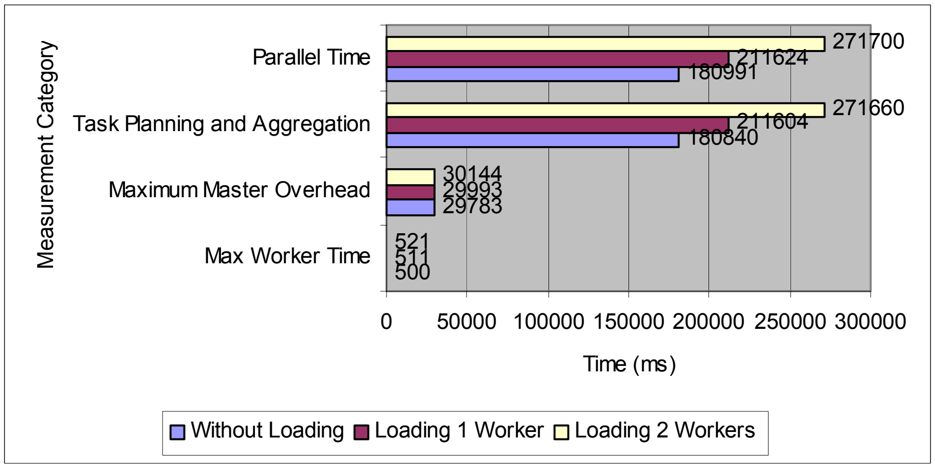


Figure 5.9(a) Graph depicting time measurements across 4 Workers for pre-fetching scheme

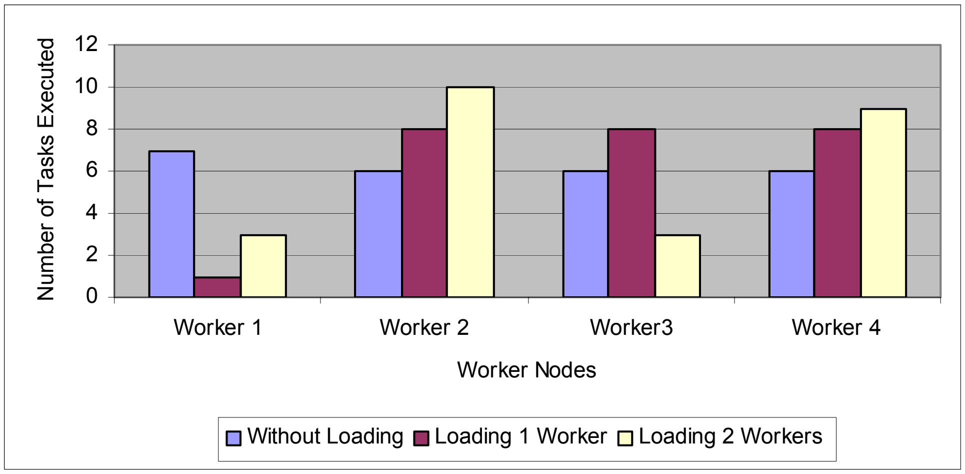


Figure 5.9(b) Measurements of number of tasks executed across 4 Workers for pre-fetching scheme

Figure 5.9(b) illustrates how tasks are scheduled with changing load conditions. Results are obtained as expected.

From the above experiments it can be clearly seen that it is possible to fit a wide spectrum of applications into the framework. The experiments also demonstrate the ability of the framework to react to the varying System State dynamically. The easiness of adaptation to the System State is demonstrated by the minimal overheads and reaction times to the signals.

Chapter 6

Conclusions and Future Work

6.1 Summary

This thesis presented the design, implementation and evaluation of a framework for opportunistic parallel computing on networked clusters using JavaSpace. This framework enables coarse-grained applications to be distributed across and exploit existing heterogeneous clusters. The framework builds on Jini and JavaSpaces technologies. It provides support for global deployment, configuration management and uses an SNMP system state monitor to ensure non-intrusiveness.

The experimental evaluation shows that the framework provides good scalability for coarse-grained tasks. Further, using the system state monitor and triggering heuristics the framework can support adaptive parallelism and minimize intrusiveness. It demonstrates that idle workstations can be effectively used to compute large parallel jobs that would typically require dedicated supercomputers.

6.2 Challenges Addresses in this Thesis

In Chapter 1 we enlisted the challenges faced in utilizing available idle resources on a networked cluster for parallel computing. We now revisit these issues and highlight how our framework addresses them.

- ✓ *Adaptability to cluster dynamics:* Our framework achieves a natural load balancing in that each worker computes the number of tasks that it can handle. Thus the worker computation adapts to the changing system needs doing less work on heavily loaded machines and more on lightly loaded machines. Additionally, if the system constraints become intolerable explicit signals are sent to the worker process to offload the computation to other idle worker nodes. Our framework reacts to these signals with minimal reaction times.

- ✓ *System configuration and management overhead:* In an effort to minimize the system configuration and management overhead, our framework provides a remote node configuration mechanism. This mechanism facilitates remote class loading techniques. Thus the actual worker computation code resides on the machine hosting the Jini infrastructure, and leads to a thin client model at the worker nodes.
- ✓ *Heterogeneity:* Our framework has been evaluated to handle heterogeneity in terms of processor speeds and capacity. The framework was evaluated on a Windows NT platform, however use of a Java centric approach guarantees portability across various platforms. It was noted that tools for performance measurement are not standardized across all versions and platforms. Use of SNMP as a third party network-monitoring tool provides us with standardized hooks into the System State. SNMP protocol being solely dedicated to the purpose of network monitoring and management provides a more comprehensive tool with room for expansion if the Inference Engine is made more complex and sophisticated.
- ✓ *Security and Privacy concerns:* Remote class loading raises security concerns both for the downloaded code and the machine running the downloaded code. Enforcing restricting policies protects the machine hosting the worker processes. The policy file is read at the time of download and any code violating the policy is denied access. The sandbox model of Java protects the downloaded worker code. The access to the worker code is restricted to the parent process that invokes the remote class loader.
- ✓ *Non intrusiveness:* Since the worker code is remotely loaded from the master machine, the worker nodes are fairly lightweight components. Since all the computation occurs at the application layer the framework does not warrant any changes in system setting. Memory requirements for the system to take effect are also kept to a minimum. The problem domain is divided such that the worker calculation does not impose on the overall functioning of the worker node. It has been observed with the evaluation

applications that the CPU usage during the worker computation is in the range of 0 to 3% with the exception of the Ray Tracing application.

6.3 Future Work

There are two main directions for future work on the presented framework, at the application level and at the framework infrastructure level. While evaluating our framework prototype, we identified a number of issues for further study. For example, in the pre-fetching application small matrix sizes are not beneficial for this type of parallel processing. Secondly, the segment size of the strips needs to be optimized in an effort to minimize task aggregation overhead and facilitate faster computing. Thirdly, the problem to be distributed should be free from any dependencies. The inter-iteration dependency in the pre-fetching algorithm limits speed up. Further optimization can be achieved with use of proper measurement and prioritizing tools on worker threads. We have room to facilitate this optimization in our current infrastructure. The results obtained from reaction times between the Worker and Network Management Modules are promising and prove that the overhead incurred in including adaptability features is not significant.

The framework infrastructure can be improved by using Transaction Management for fault tolerance. Partial failures are not addressed as a part of this thesis. However, the Transaction Management service offered by Jini can be effectively used as a means of securing the write and take operations on the JavaSpace. As future work, we envision incorporating a distributed JavaSpaces model to avoid a single point of resource contention or failure. The Jini community is investigating this area of research.

APPENDIX A

(This work is contributed by Pravin Bhandarkar from The Applied Software Systems Lab (TASSL), CAIP Center)

The System State component is a generic component that can be used to determine the state of the system. Since the software was evaluated on NT hosts, Windows based Simple Network Management Protocol (SNMP) was used to build the network management module. There are two components in network management that have to work in conjunction with each other to enable acquisition of system related data. The two components are the manager component that runs on the management station and the agent component that runs on the network element to be monitored. To monitor NT hosts there is a need to build agents that run on the hosts and continuously obtain network management data. This data is then forwarded to management stations upon request and used to monitor the behavior of the network element. The agents on the NT hosts have to be customized to obtain the data parameters as per the requirement and have to be built separately. Such agents are known as extension agents.

The following sections are organized as follows. First we describe the support for windows based SNMP. Section A.2 describes the construction of Extension Agent using NT SNMP followed by the implementation details, finally concluding with salient point regarding the implementation.

A.1 Windows SNMP

Under Windows NT there are two SNMP services namely the agent service ***(SNMP.exe)*** and the SNMP trap service ***(SNMPTRAP.exe)***. The SNMP agent service processes requests from the SNMP management systems and sends GetResponse messages in the reply. The agent handles the interface with the Windows Socket API, SNMP message parsing and ANS.1 and BER encoding and decoding. The agent is responsible for sending the trap messages to SNMP management systems.

The SNMP trap service listens for the trap sent to the NT host and then passes the data to the management API. The SNMP service allows the user to build an extension NT agent that allows the MIB information to be dynamically added and supported as required. The extension agent resides within the SNMP service. It receives the SNMP messages across the network using the Winsock API, and passes the message data to one or more extension agents for processing.

A.2 Extension Agent Using NT SNMP

Managed devices such as hosts and routers contain monitoring and (possibly) control instrumentation. The NT agent provides the instrumentation of some critical information such as CPU load for the manager. The NT agent represents hosts access to this instrumentation to the manager via a MIB, filtered by the SNMP security mechanisms. The manager communicates with the NT agent via SNMP to monitor and (possibly) control managed hosts.

The NT agent is based on the Microsoft SNMP Extension API that provides a basic functionality for constructing an extension agent dynamic link library (DLL) capable of communicating with the SNMP service and interacting with network management application using SNMP.

Once the SNMP service is activated on a host, the NT agent DLL is loaded as an extension agent DLL by the SNMP service. The DLL entry point function DllMain() is called first, then the initialization functions such as SnmpExtensionInit() and SnmpExtensionInitEx() are called to load the primary supported MIB subtree, the handle used by the NT agent to assert that it needs to send the trap message, and additional MIB subtrees if appropriate.

When a request message from a manager is received, then the querying process is invoked. Each request message will contain one or more variable bindings. The NT agent iterates through each binding and applies the Get, GetNext, and Set operation specified by the message type to the OID and the data value present in each binding. For processing each variable binding, the matching between the OID of the NT agent MIB variable and OID specified in the variable binding is checked first, then the attributes are compared. Finally actions will be taken if all the SNMP security checks have been passed.

A.3 Implementation

The network abstraction is a simple class called ***snmp***, this class accepts the input rule base from the inference engine. The *snmp* class accepts the IP address of the network element to be queried, community string and the object id. This class interfaces with a dynamic link library (DLL) ***javamgr.dll*** that holds the native code, which in turn interfaces winsnmp.dll to query the network element. For the sake of the initial testing we used the CPU load parameter on an NT 4.0 machine to determine the time frame of reception of a particular stream.

SNMP

public long []_routerId=new long[11];
public String _router;
public String _community;
public long value;

public static native long snmpget(String router, String community, long[]routerId);
public snmp(String router, String community, String oid) { //**Accepting the input in the constructor**}
static {
System.loadLibrary("javamgr"); //**load native library** }
public long getSnmpData() {
//**code interfaces the c- native code to get the parameter**
…….
return (value);
}

To enable the acquisition of network management information from the local host we built a simple extension agent that runs on the local machine. The agent continuously polls the machine for local data and upon request from the *SNMP* class, which is built on top of the WinSNMP manager API, obtains the information from the instrumentation routines to reply to the SNMP get command. The native implementation first creates a session by the *SnmpOpen* function that is used to manage the link between the WinSNMP application and the WinSNMP interface implementation. Then it uses *SnmpSendMsg* and subsequent *SnmpRecvMsg* calls to process querying information from the managed devices such as routers and hosts. Finally *SnmpClose* is used to close the session.

A.4 Salient Points About Implementation

The SNMP querying component is built using WinSNMP DLL and is loaded using JNI. This component queries the various network elements to obtain network parameter related information.

The interaction between the Java based code and the network management component can result in substantial timing overheads. It was found that if this information acquisition is done in the same module as data reception and servicing then it could result in an overhead on the real-time nature of servicing data. Owing to this the SNMP parameters from the standard MIBs are considered to be averaged values.

REFERENCES

[1] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. IEEE Computer, Vol. 28, No. 1, pp. 40-49, Jan. 1995.

[2] Sun Microsystems. Javaspaces specification.
http://www.javasoft.com/products/javaspaces/specs/index.html (1998).

[3] SNMP Documentation, http://www.snmpinfo.com.

[4] J. Murray, Windows NT SNMP, O'Reilly Publications, January 1998.

[5] WinSnmp Documentation, http://www.winsnmp.com

[6] M. Lizkow, M. Livny, and M. Mukta. Condor: A hunter of idle workstations. In Proceedings of the 8th International conference on Distributed Computing Systems, June1998, pp. 104-111.

[7] M. Lizkow, M. Livny, and T. Tannenbaum. Checkpoint and Migration of UNIX Processes in the condor Distributed Environment. Technical Report 1346. University of Wisconsin-Madison, April 1997.

[8] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In Proceedings of International Parallel Processing Symposium (IPPS'95) Workshop on Job Scheduling Strategies for Parallel Processing, Vol. 949, Springer-Verlag, pp. 259-278 April 1995.

[9] J. Pruyne and M. Livney. Interfacing Condor and PVM to harness the cycles of workstation clusters. Journal on Future Generations of Computer Systems, vol. 12, no. 1, Elsevier, pp. 67-85, May 1996.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine. MIT Press, 1994.

[11] The Linda Group. http://www.cs.yale.edu/HTML/YALE/CS/Linda/linda.html

[12] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary Experience with Piranha. In Proceedings of the 6th ACM International Conference on Supercomputing, pp. 417-427, July 1992.

[13] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In Proceedings of the 7th ACM SIGOPS European Workshop: Systems support for Worldwide Applications, pp. 165-172, September 1996.

[14] P. Ledru. Adaptive Parallelism: An Early Experiment with Java™ Remote Method Invocation. Technical Report, CS Department, University of Alabama, 1997.

[15] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In Proceedings of the 9th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS), pp. 181-188 September 1996.

[16] A. Baratloo, M. Karaul, H. Karl, Zvi M. Kedem. An Infrastructure for Network Computing with Java Applets. Concurrency: Practice and Experience, Volume 10(11-13), pp 1029-1041, September 1998.

[17] B. Christiansen, P. Cappello, M.F. Ionescu, M. O. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. Concurrency: Practice and Experience, Vol. 9(11), pp. 1139-1160, November 1997.

[18] T. Brecht, H. Sandhu, J. Talbott, and M. Shan. ParaWeb: Towards world-wide supercomputing. In Proceedings of the 7th ACM SIGOPS European Workshop, pp. 181-188, September 1996.

[19] M. Stang and S. Whinston. Enterprise Computing with Jini Technology. In issue of IT Professional, IEEE Computer Society, Vol. 3, No. 1, pp. 33-38, Jan/Feb 2001.

[20] B. Venners. Objects, the Network, and Jini. Jiniology, JavaWorld. http://www.artima.com/jini/jiniology/intro.html

[21] E. Freeman, S. Hupfer, K. Arnold. JavaSpaces Principles, Patterns, and Practice, June 1999.

[22] L. Cameron. JavaSpaces: Making Distributed Computing Easier. Byte Online Magazine. http://www.byte.com/feature/BYT19990915S0001, 20 September 1999.

[23] F. Khan. Tutorial on Java, JavaBeans Components, Multithreading and JavaSpaces: Concurrency among Components Using JavaSpaces. National Center of High Performance Computing (NCHC), September 2000. http://eewww.eng.ohio-state.edu/~khan/khan/Presentations/Taiwan-9-00/ConcurrencyCompUsingJavaSpaces/ConcurrencyCompUsingJavaSpaces.html

[24] P. Bhandarkar. An Adaptive Framework for Collaboration in Heterogeneous Networks. M.S. Thesis. Department of Electrical and Computer Engineering, Rutgers University, October 1999.

[25] M. Noble. Tonic: A Java TupleSpaces Benchmark Project. http://hea-www.harvard.edu/~mnoble/tonic/doc/

[26] Glossary of Financial terms http://www.centrex.com/terms.html

[27] Broadie and Glasserman MC algorithm for Option Pricing http://www.puc-rio.br/marco.ind/monte-carlo.html

[28] A. Heirich and J. Arvo. A Competitve analysis of Load Balancing Strategies for Parallel Ray Tracing. In Journal of Supercomputing 1998.

[29] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In Proceedingsof the 7th International World Wide Web Conference, pp. 107-117, April 1998.

[30] S. Brin and L. Page. The PageRank Citation Ranking: Bringing Order to the Web. In Technical Report, http://www-db.stanford.edu/~backrub/pageranksub.ps, January 1998.

[31] S.D. Conte and C. de Boor. Elementary Numerical Analysis: An Algorithmic Approach. McGraw-Hill 1980.