# A Framework for Adaptive Cluster Computing using JavaSpaces[1]

Jyoti Batheja and Manish Parashar
The Applied Software Systems Laboratory
Department of Electrical and Computer Engineering,
Rutgers, The State University of New Jersey
94 Brett Road, Piscataway, NJ 08854
{jbatheja, parashar}@caip.rutgers.edu

## Abstract

Heterogeneous networked clusters are being increasingly used as platforms for resource-intensive parallel and distributed applications. The fundamental underlying 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 paper we present the design, implementation and evaluation of a framework that uses JavaSpaces to support this type of opportunistic adaptive parallel/distributed computing over networked clusters in a non-intrusive manner. The framework targets applications exhibiting coarse grained parallelism and has three key features: (1) portability across heterogeneous platforms, (2) minimal configuration overheads for participating nodes, and (3) automated system state monitoring (using SNMP) to ensure non-intrusive behavior. Experimental results presented in this paper demonstrate that for applications that can be broken into coarse-grained, relatively independent tasks, the opportunistic adaptive parallel computing framework can provide performance gains. Furthermore, the results indicate that monitoring and reacting to the current system state minimizes the intrusiveness of the framework.

**Keywords:** Adaptive cluster computing, Parallel/Distributed computing, JavaSpaces, Jini, SNMP.

## 1. Introduction

This paper presents the design, implementation and evaluation of a framework that uses JavaSpaces [1] to aggregate networked computing resources, and non-intrusively exploits idle resources for parallel/distributed computing.

---

[1] The research presented in this paper is based upon work supported by the National Science Foundation under Grant Number ACI 9984357 (CAREERS) awarded to Manish Parashar.

Traditional High Performance Computing (HPC) is based on massively parallel processors, supercomputers or high-end workstation clusters connected by high-speed networks. These resources are relatively expensive, and are dedicated to specialized parallel and distributed applications. Exploiting available idle resources in a networked system can provide a more cost effective alternative for certain applications. For example, a large percentage of the resources at an enterprise are idle after regular working hours and can be exploited by stealing their idle computational cycles for useful work. This option allows the enterprise to leverage its existing computational resources, rather than investing in dedicated parallel/distributed systems. However, there are a number of challenges that must be addressed before such opportunistic adaptive cluster computing can be a truly viable option. These include:

- **Heterogeneity:** Cluster environments are typically heterogeneous in the type of resources, the configurations and capabilities of these resources, and the available software, services and tools on the systems. This heterogeneity must be hidden from the application and addressed in a seamless manner, so that the application can uniformly exploit available parallelism.
- **Intrusiveness:** A key requirement for exploiting idle resources is that the effect of stealing computational cycles from resources on local applications should be minimized. That is, a local user should not be able to perceive that local resources are being stolen for foreign computations. Furthermore, inclusion into the framework should require minimal modifications to existing (legacy) code or standard practices.
- **System configuration and management overhead:** Incorporating a new resource into the cycle stealing resource cluster may require system configuration and software installation. This includes installing and configuring system software to enable application execution, as well as installing the application software itself. These modifications and overheads must be minimized so that cluster can be expanded on the fly to utilize all available resources.
- **Adaptability to system and network dynamics:** The availability and state of system and network resources in a cluster can be unpredictable and highly dynamic. These dynamics must be handled to ensure reliable application execution. This requires monitoring system/network state, and reacting to changes by adding or removing a resource from the available pool, modifying scheduling policies, changing task configurations, etc.
- **Security and privacy:** Secure and safe access to resources in the cluster must be guaranteed so as to provide assurance to users making their systems available for external computations. Policies must be defined and enforced to ensure that external application tasks adhere to the limits and restrictions set on resource/data access and utilization.

This paper presents the design, implementation and evaluation of a framework for adaptive and opportunistic cluster computing that address these issues. The framework builds on Java and JavaSpaces technologies. Java provides support for naturally managing the heterogeneity in the cluster. Furthermore, its "sandbox" execution model enables us to handle security and privacy concerns. JavaSpaces provides mechanisms for scheduling, global deployment, and execution of tasks on remote hosts. It also provides support for coordination and communication between processes and tasks. The application task deployment protocol is designed to minimize configuration and management costs. Cluster nodes are remotely configured and application code is dynamically loaded at runtime. Finally, the framework supports a sustained monitoring of the state of the cluster resources using SNMP (Simple Network Management Protocol) [2][3], and provides policies and mechanisms to automatically handle system/network dynamics so as to minimize intrusiveness at the cluster nodes. The framework is evaluated using 3 real-world applications, viz. ray tracing, stock option pricing, and web page pre-fetching. The evaluation demonstrates that for this class of coarse-grained applications, adaptive cluster computing can provide significant performance advantages. Furthermore, the evaluation shows that monitoring and reacting to the state of the cluster resources enables the framework to non-intrusively maximize its utilization of the cluster resources.

The rest of this paper is organized as follows. Section 2 presents background material and discusses related work in adaptive cluster computing. Section 3 provides an overview of the Jini/JavaSpaces infrastructure and describes its use for adaptive computing. Section 4 describes the architecture and operation of the proposed framework. Section 5 presents an experimental evaluation of the framework. Section 6 presents our conclusions and outlines current and future work.

## 2. Background and Related Work

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. Recent advances in cluster computing have followed two approaches:

**Job level parallelism:** In this approach, entire application jobs are allocated to available idle resources for computation. The state of the resources are monitored, and if a resource becomes unavailable (e.g. an interactive user logs in), the job(s) executing on it are migrated to a different resource that is available. Systems supporting job level parallelism are required to support

mechanisms for check-pointing the state of an application job on one machine and restoring the state on a different machine, to enable job migration. The Condor system [4][5] supports cluster-based job level parallelism.

**Adaptive Parallelism[2]:** In the adaptive parallelism approach, an application job is broken into smaller tasks, and scheduling and load balancing schemes are used to distribute these tasks across idle resources in the cluster. This approach targets applications that can be easily decomposed into independent tasks. In this case, the available processors are treated as part of a resource pool. Each processor in the pool aggressively competes for application tasks. Under adaptive parallelism, the pool of processors executing a parallel program may grow or shrink during the program execution based upon the processor availability. Adaptive computing techniques for parallel/distributed processing can be divided into *cluster based* and *web based* approaches. *Cluster based* systems revolve around deploying computational tasks across loosely coupled networked resources. These architectures exploit available resources within a networked cluster, such as a LAN, for parallel/distributed computing. *Web based* approach extends this model to resources over the Internet. The collection of machines connected over the Internet; if aggregated, can potentially provide unparalleled processing capacity. Web based approaches attempt to exploit this potential.

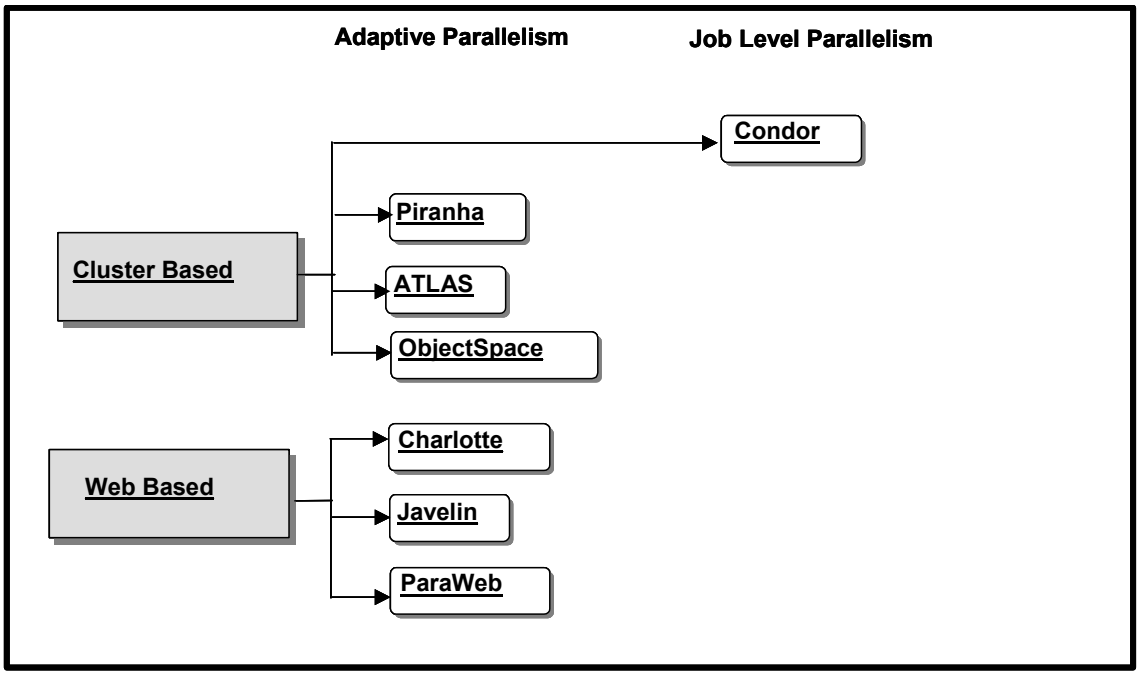



**Figure 1 – Frameworks for Opportunistic Cluster Computing**

[2] To best of our knowledge, the term "adaptive parallelism" was coined by the Piranha project [7].

Systems supporting Adaptive parallelism include cluster-based systems such as Piranha [6][7], Atlas [8], and ObjectSpace/Anaconda [9], and web-based systems such as Charlotte [10][11], Javelin [12], and ParaWeb [13]. Systems such as Entropia [14] and SETI@home [15] have been widely successful in effectively solving real world problems by aggregating vast unused processing cycles from resources spread across the Internet.

### 2.1. Related Work in Opportunistic Cluster Computing

Frameworks supporting opportunistic cluster computing are presented in Figure 1. These frameworks are summarized in Table 1. Note that most of these systems (including the framework presented in this paper) implement the master worker paradigm in one form or the other – the master is responsible for generating "work" and managing its execution, while workers consume the work and perform the computations. A key shortcoming of existing systems is that they 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 user's part. The framework presented in this paper makes three key contributions:

- It uses Java as the core programming language to ensure portability across heterogeneous systems.
- It uses remote worker configuration and dynamic loading of application code to minimize setup and configuration overheads.
- It provides a dedicated network management module that uses SNMP to identify idle resources, monitors system state, and enables the system to automatically adapt to the current system state to minimize intrusiveness.

**Table 1 Comparison of Opportunistic Cluster Computing Frameworks**

| Architecture | Category | Approach | Key Contributions | Sample Applications |
|---|---|---|---|---|
| Condor [4][5] | Cluster based | Job level parallelism, Asynchronous offloading of work to idle resources on a request basis | Queuing mechanism, Check pointing and process migration, Remote procedure call and matchmaking | High Throughput Monte Carlo Simulations |
| Piranha [6][7] | 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 [8] | Cluster based | Adaptive Parallelism, Hierarchical work stealing | Fault Tolerance, improved scalability due to hierarchical model, safe heterogeneous execution, | Double recursion to compute Fibonacci numbers, (POV-Ray) Ray Tracing |

| | | | | |
|---|---|---|---|---|
| | | | no administration effort | |
| ObjectSpace/ Anaconda [9] | Cluster based | Job level parallelism at brokers | Eager scheduling, automated/manual job replication | Traveling Salesman Problem |
| Charlotte [10][11] | 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 [12] | Web based | Adaptive parallelism, Centralized work distribution using downloadable applets | Heterogeneous, secure execution environment, portability to include Linda and SPMD programming models | The Mersenne Prime Application: a primality test |
| ParaWeb [13] | 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 |

## 3. Jini & JavaSpaces: An Overview

Jini [16] technology, developed by Sun Microsystems, is a runtime infrastructure that assists in building and deploying truly distributed systems that are organized as a *federation of services* [17]. It provides a set of APIs, mechanisms and network protocols that enable addition, discovery, access and removal of services. Jini presents a service-based model of distributed computing; a Jini service joins a federation to share its services with clients while a Jini client joins a federation to gain access to services. Discovery of services by clients is facilitated by a 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 its information to the map. A Jini client can request the lookup service for a list of Jini servers that match its desired attributes.

A typical interaction in a Jini based distributed system is as follows. Jini service providers first locate the Jini lookup service. This is done using the Jini *discovery protocol*. The protocol consists of broadcasting a presence announcement by dropping a multicast packet on a well-known port. This packet contains the host'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 host. Once it has located the lookup service, the service provider uses the *join* protocol to become a part of the federation and register itself with the lookup service. When a Jini client connects to the system, it first locates the lookup service using the Jini discovery protocol. The client then requests a service

by sending the desired list of attributes to the lookup server. The lookup server does an associative lookup and returns a list of services matching these attributes. It is then up to the Jini client to select a specific Jini server from the returned list and directly request service.

JavaSpaces is a Java implementation of a tuple-space [6][7] system, and is provided as a Jini service. A JavaSpaces is a shared, network accessible repository for Java objects [18], and 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. The JavaSpace API leverages the Java type semantics and provides operations to store, retrieve and lookup Java objects in the space.

Several aspects of distributed computing are inherently handled by JavaSpaces. It provides associative lookup of persistent objects. It also addresses fault-tolerance and data integrity through transactions. All access operations to objects in the space such as read/write/take can be executed within a transaction. In event of a partial failure, the transaction either completes successfully or does not execute at all. Using a JavaSpaces-based implementation allows transacting executable content across the network. The local instances of the Java objects retrieved from the space are active – i.e. their methods can be invoked and attributes modified. JavaSpaces provides mechanisms for decoupling the semantics of distributed computing from the semantics of the problem domain. This separation of concerns allows the two elements to be managed and developed independently [19]. For example, the application designer does not have to worry about issues such as multithreaded server implementation, low level synchronization, or network communication protocols.

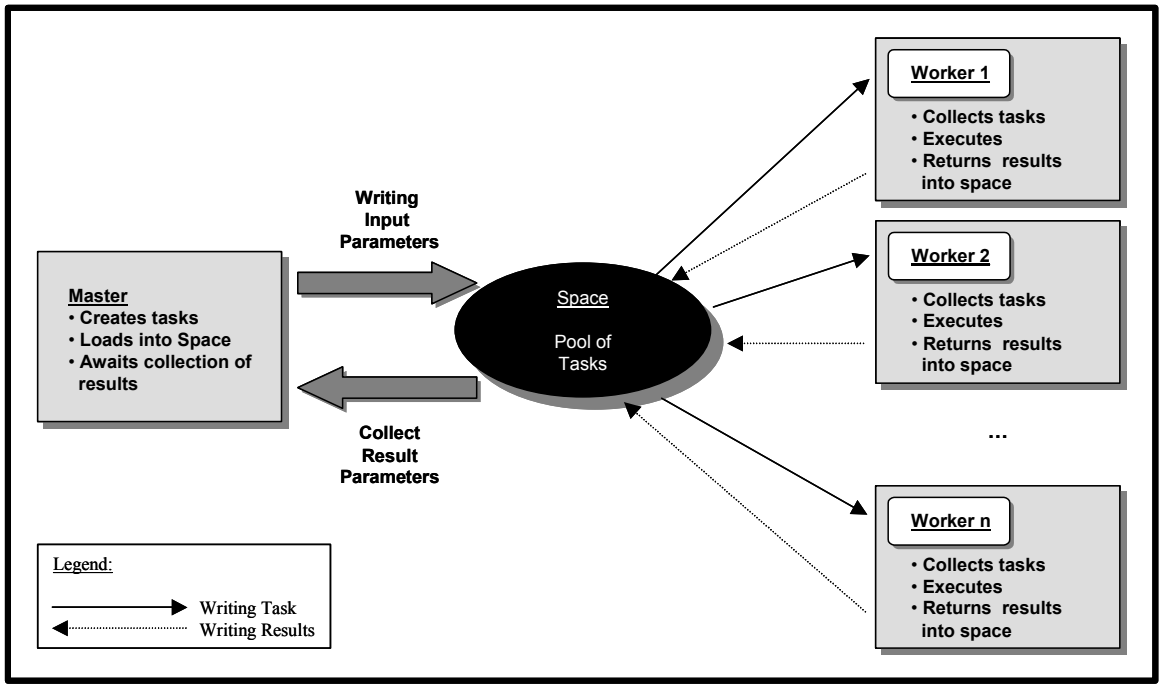


**Figure 2 – Master-Worker Parallel Computing using JavaSpaces**

### 3.1. JavaSpaces and Parallel Computation

JavaSpaces naturally supports the master-worker parallel/distributed-computing paradigm. In this paradigm (see Figure 2), the application is distributed across available worker nodes using the *bag-of-tasks* model. The master produces independent application tasks and puts them into the space. The worker consumes the tasks from the space, computes on these tasks, and places results back into the space. This approach supports coarse-grained applications that can be partitioned into relatively independent tasks. It offers two key advantages. (1) The model is naturally load-balanced. Load distribution in this model is worker driven. As longs as there is work to be done, and the worker is available to do work, it can keep busy. (2) The model is naturally scalable. Since the tasks are relatively independent, as long as there are a sufficient number of task, adding workers improves performance.

## 4. A Framework for Opportunistic Adaptive Parallel Computing on Clusters

The framework presented in this paper employs JavaSpaces to facilitate adaptive (opportunistic) master-worker parallel computing on networked clusters. It targets applications that are sufficiently complex and require parallel computing, that are divisible into relatively coarse-grained subtasks that can be solved independently, and where the subtasks have small input/output sizes. Real world applications having these characteristics span many disciplines including those evaluated in this paper, viz. financial modeling (stock-option pricing), visualization (ray-tracing) and web-serving (page-rank based prefetching).

### 4.1. Framework Architecture

A schematic overview of the framework architecture is shown in Figure 3. The framework consists of 3 key components: the Master Module, the Worker Module, and the Network Management Module.

**Master Module:** The master module runs as an application level process on the master processor. It hosts the JavaSpaces service and registers it with the Jini substrate. The master module defines the problem domain for a given application. It decomposes the application into independent tasks that are JavaSpaces enabled[3], and places these tasks into the space.

---

[3] 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.

**Worker Module:** The worker module runs as an application level process on the workers nodes. It is a thin module and is configured at runtime using the *Remote Node Configuration Engine* – i.e. worker classes, providing the solution content for the application, are loaded at runtime. This minimized the overheads of deploying application code. Note that the workers need not be Jini aware in order to interact with the master module. Interaction between the master and the worker processes is via virtual, shared JavaSpaces. The worker module operation is controlled by the network management module using the *Rule-Base Protocol* as described in the following section.

**Network Management Module:** The network management module serves two functions, viz. monitoring the state of worker machines, and providing a decision-making mechanism to facilitate the utilization of idle resources and ensure non-intrusive code execution. In order to exploit idle resources while maintaining non-intrusiveness at the remote nodes, it is critical that the framework monitors the state of the worker nodes, and uses this state information to drive the scheduling of tasks on workers. The *Monitoring Agent* component of the network management module performs this task. It monitors the state of registered workers and uses defined policies to decide on the worker's availability. The policies are maintained by the *Inference Engine* component and enforced using the *Rule-Base Protocol*.

The monitoring agent uses SNMP [2][3] to monitor remote worker nodes. The SNMP layer consists of two components: the manager component that runs on the SNMP server and the worker-agent component that runs on the worker nodes to be monitored. The monitoring agent uses JNI (Java Native Interface) to access the SNMP layer and query the worker-agents to get relevant system parameters such as CPU load and available memory.

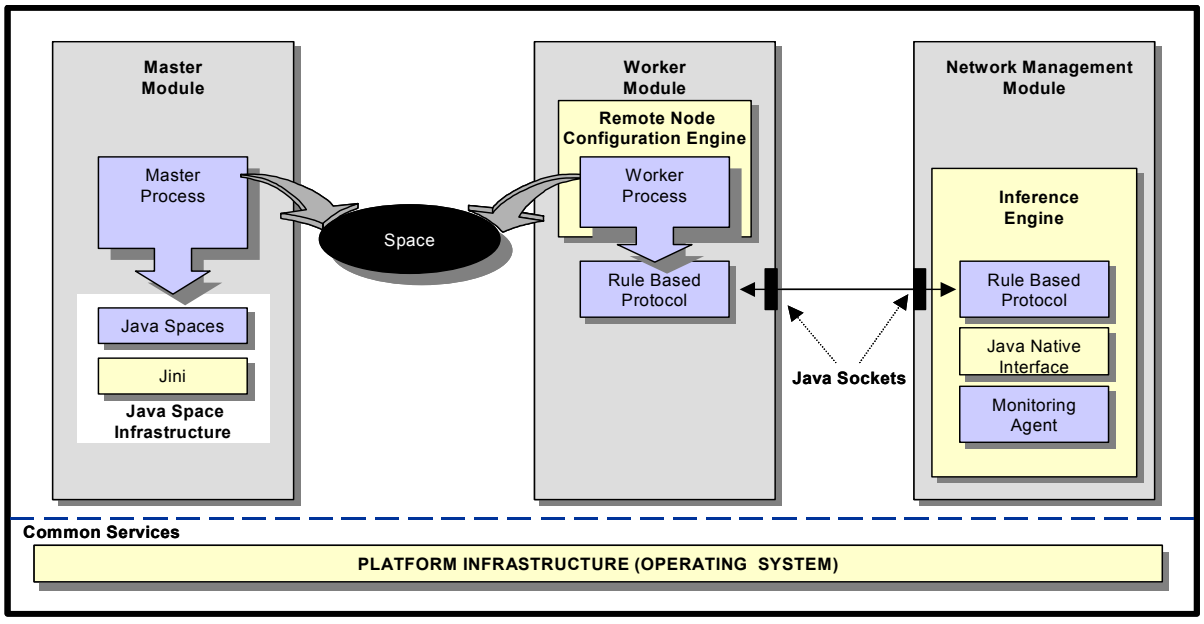


**Figure 3 – Framework Architecture**

### 4.2. Framework Operation

The framework implements the master-worker pattern with JavaSpaces as the backbone. The overall operation of the framework consists of three potentially overlapping phases, viz. task-planning, compute, and result-aggregation. During the *task-planning phase*, the master process first decomposes the application problem into sub tasks. It then iterates through the application tasks, creates a task entry for each task, and writes the tasks entry into the JavaSpaces. During the *compute phase,* the worker process retrieves these tasks from the space using a simple value-based look up. Each task object is identified by a unique ID and the space in which it resides. If a matching task object is not available immediately, the worker process waits until one arrives. The worker classes are downloaded at runtime using the Remote Node Configuration Engine. Results obtained from executing the computations are returned to the space. Worker operation during the compute phase is monitored and managed by the network management module using the rule-base protocol as described in section 4.4. During the *result aggregation* phase, the master module removes results written into the space by the workers, and aggregates them into the final solution.

If the resource utilization on a worker node becomes intolerable, the network management module sends a stop/pause signal to the worker process. On receiving the signal, the worker completes the execution of the current task and returns its results into the space. It then enters the stop/pause state and does not accept tasks until it receives a start/resume signal.

### 4.3. Remote Node Configuration

Remote node configuration uses the dynamic class loading mechanism provided by the Java Virtual Machine, which supports locating and fetching the required class files, consulting the security policy, and defining the class object with the appropriate permissions. The implementation of the remote configuration engine consists of a simple Java application starter program that can load jar and class files from URL at runtime. Required worker classes are downloaded from a web server residing at the master in the form of executable jar files.

Our modification of the network launcher [20] provides mechanisms to intercept calls from the inference engine component of the network management module and use them to signal the executing worker code. This enables the network management module to manage workers using the rule-base protocol. The signals are handled by remote node configuration engine. As preempting worker execution to process the signal may result in the current task being lost, the node configuration engine waits for the worker to complete its current task, and forwards the signal before the worker fetches the next task.

### 4.4. Dynamic Worker Management for Adaptive Cluster Computing

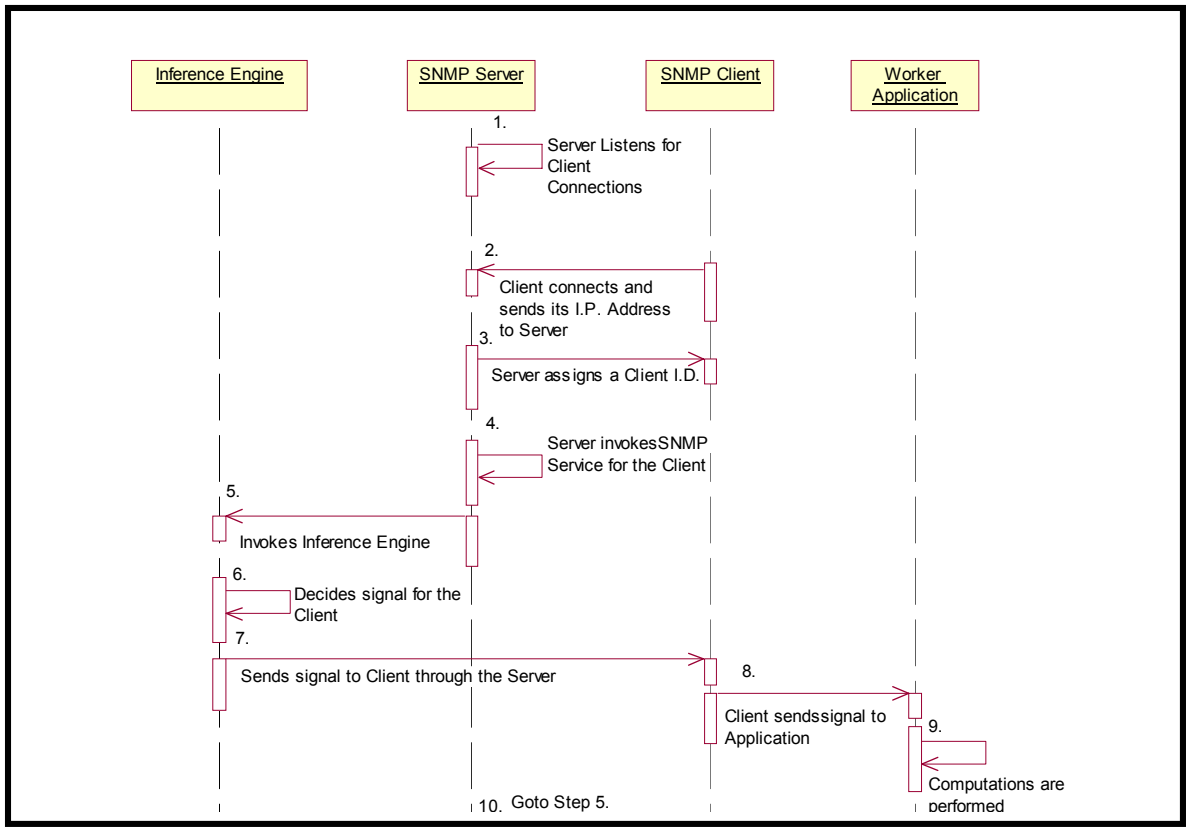



**Figure 4 – Rule-Base Protocol for Adaptive Worker Management**

The rule-base protocol manages worker execution, and defines the interaction between the network management module and the worker module (see Figure 4) to enable the worker to react to changes in its system state. It operates as follows:

The SNMP client, which is part of the worker module, initiates the workers participation in the parallel computation by registering with the SNMP server at the network management module. The inference engine, also at the network management module, maintains a list of registered workers. It assigns a unique ID to the new worker and adds its IP address to the list. The SNMP server then continues to monitor the state of workers in its list.

The primary SNMP parameter monitored is the average worker CPU utilization. As these values are returned they are added to the respective entry in the worker list. Based on this return value and programmed threshold ranges, the inference engine makes a decision on the worker's current availability status and passes an appropriate signal back to the worker. Threshold values are based on heuristics. The rule-base currently defines 4 types of signals in response to the varying load conditions at a worker, viz. *Start, Stop, Pause* and *Resume.* Based on the signal it receives, the

worker can be in 3 possible states: *Running*, *Paused*, or *Stopped*. The worker state transitions are shown in Figure 5 and are described below.

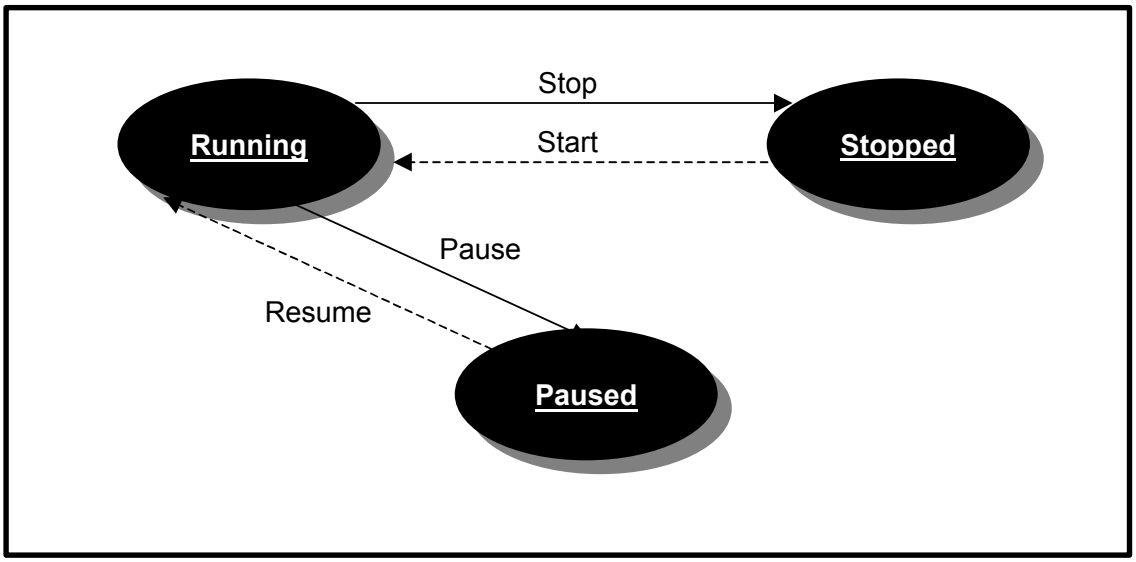



**Figure 5 - Worker State Transition Diagram**

**Running:** The worker enters the running state in response to a *Start* or *Resume* signal. *Start* or *Resume* signals are sent when the CPU load at the worker is in the range of 0% - 25%. While in this state, worker is considered idle and can start participating in the parallel application. On receiving a *Start* signal, the worker initiates a new runtime process. The new thread first goes through the remote class loading phase and then starts off a worker thread for task execution. If the worker receives a *Resume* signal, however, it does not require loading of the worker classes since they are already loaded into the worker's memory. It simply removes the lock on the interrupted execution thread and resumes computations.

**Stopped:** The worker enters this state in response to a *Stop* signal. 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 *Stopped* state is in the range of 50% to 100%. While in this state the node can no longer be used for computations. On receiving the *Stop* signal, the executing worker thread is interrupted and shutdown/cleanup mechanisms are initiated. 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. The next time this worker becomes available, a transition to the *Running* state will require the worker classes to be reloaded.

**Paused:** The worker enters this state in response to the *Pause* signal. This state indicates that the worker node is experiencing increased CPU loads and is not currently idle, and hence it should temporarily not be used for computation. However, the load increase might be transient and the node could be reused for computation in the near future. Threshold values for the *Paused* state are in the range of 25% - 50%. Upon receiving this signal, the worker backs off, but unlike the stop state the back off is temporary, i.e. until it gets the resume signal. This minimizes worker initialization and class loading overheads for transient load fluctuations. 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 but only interrupts the execution until the resume signal is received hence bypassing the overhead associated with remote node configuration.

## 5. Experimental Evaluation of the Framework

The JavaSpaces-based opportunistic cluster-computing framework is experimentally evaluated using three "real-world" applications: (1) a financial application that uses Monte Carlo (MC) simulation for option pricing, (2) a scientific ray tracing application, and (3) a web page pre-fetching technique for server optimization. The evaluation consists of three experiments. The objective of the first experiment is to study the scalability of the application and our framework, and to demonstrate the potential advantage of using clusters for parallel computing. The second experiment measures 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.

The experiments are conducted on PC clusters running Windows NT (version 4.0). The web page pre-fetching and parallel ray tracing applications are evaluated on a five PC cluster, with an 800MHz. Intel Pentium III processors and 256 MB RAM. The option-pricing scheme is evaluated on a larger cluster with thirteen PCs. The PCs in this cluster had 300 MHz. processors and 64MB RAM. Due to the high memory requirements of the Jini infrastructure, the master module in both cases runs on an 800 MHz. Intel Pentium III processor PC with 256 MB RAM.

### 5.1 Application Description

The characteristics of the three applications are summarized in Table 2. The applications and their implementations within the framework are briefly described below.

**Table 2 Classification of the Evaluated Applications**

| 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 |
| Task Dependency | No | No | Yes |

### 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. A stock option is defined by the underlying security, the option type (call or put), the strike price, and an expiration date. Furthermore, factors such as interest rate and volatility, affect the pricing of an option. These financial terms are explained in greater depth in [21]. In our implementation we use Monte Carlo (MC) simulations, based on the Broadie and Glasserman MC algorithm [22], to model the behavior of options and account for the various factors affecting its price.

**Implementation Overview:** In our implementation, the main MC simulation is the core parallel computation that is distributed. Input parameters are fed in using a simple GUI provided in the implementation. The simulation domain is divided into independent tasks and the MC simulations are performed in parallel on these tasks. The total number of simulations is defined by an external input. Each MC task consist of two iterations, the first one obtains a high estimate and the second one obtains a low estimate. For the experimental evaluation presented below, the number of simulations was set to 5000. The problem domain is divided into 50 tasks, each comprising of 100 simulations. As each MC simulation consists of two independent iterations, a total of 100 sub-tasks were created and put into the JavaSpaces. The workers took the task from the space and performed the MC simulations.

### 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 an observer as they interact with the scene and the light sources. Ray tracing algorithms estimate the intensity and wavelengths of light entering the lens of a virtual camera in a simulated environment. The quantities are estimated at discreet points in the image plane that correspond to pixels. These 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 technique known as *ray casting*. The cost for computing individual pixels can vary dramatically, and depends on the complexity of the model being rendered and the algorithm employed. Parallel implementations of ray casting algorithms typically distribute the calculations for a set of pixels in an image in order to minimize the overall rendering time. These applications are ideal candidates for the replicated-worker pattern as they are made up of a number of independent and computationally identical tasks.

**Implementation Overview:** The ray tracing [23] application 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. This computation involves tracing the rays of light that pass from a viewpoint (such as an eye or the virtual camera) to the model, through the pixel in the image plane. The computation is identical for all pixels - only the parameters describing the pixel's position differ. In our experiments the 600X600 image plane was divided into rectangular slices of 25X600 thus creating 24 independent tasks. The input for each task consisted of the four coordinates describing the region of computation. The output produced by each task was relatively large, consisting of an array of pixel values. In our implementation, the master generated the tasks and put them into the JavaSpaces. Each worker took a task, computed the scan lines for the pixel and returned the resultant array of pixel points to the JavaSpaces. The master then collected the results and combined them to compose the image.

#### 5.1.3 Web Page Pre-fetching based on Page Rank

The overall objective of this application is to optimize access time experienced by the web user by pre-fetching web pages that are likely to be requested by the user. The page rank-based pre-fetching approach [24], [25] uses the link structure of pages requested to determine the "most important" pages they link to, and to identify the page(s) to be pre-fetched. This scheme targets access to web page cluster, i.e. groups of closely related pages such as pages of a single company. The underlying premise of the approach is that the next page requested by the user is typically based on the current and previous pages requested. Furthermore, if the requested pages link to an "important" page, that page has a higher probability of being the next one requested. The relative importance of the linked pages is calculated using the Page Rank technique [25]. The important pages are then pre-fetched into the cache for faster access.

**Implementation Overview:** For each web page requested, the Page Rank algorithm performs the following operations. First the page's URL is scanned to see if it belongs to a web page cluster. If it does, the link contained in the page to other pages on the local server are parsed out and used to populate a stochastic matrix 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.

The stochastic matrix is then used to compute the ranks of the linked pages. The core of the Page Rank algorithm consists of matrix operations and iterative eigenvector computations [26]. Parallelism is achieved by distributing the matrix and performing the computation on local portions in parallel. Inter-iteration dependencies in these computations have to be resolved as it limits the overall speedup. Note that Page Rank computations for different web page clusters are independent and can also be performed in parallel. In our experiments, the two matrices used are of sizes 500 x 500 and 500 x 1. Tasks are created by dividing the matrices into strips of size 20, leading to 25 tasks. The workers take these tasks from the JavaSpace and perform the iterations in parallel.

## 5.2 Experimental Results

### 5.2.1 Scalability Analysis

This experiment measures the overall scalability of the application and the framework. In this experiment, we measure the maximum worker time (Max Worker Time), total task planning time (Task Planning Time), total task aggregation time (Task Aggregation Time), and overall parallel execution time (Parallel Time) for the different applications as the number of worker nodes is increased. The computation time at a worker is measured from the time it first accesses a task from the space to the time it puts its final result back into the space. Max Worker Time is the maximum of the worker computation times among all workers participating in the application. Task Planning Time is measured at the master process and is the time required for the task-planning phase. This involves dividing the application into tasks and placing these tasks into the space. Task Aggregation Time is also measured at the master process and is the time required for collecting the results returned by the workers from the space, and aggregating them into a meaningful solution. The task aggregation time is expected to follow the maximum worker time, since the master needs to wait for the last task to complete before it finishes aggregating the results. Finally, Parallel Time is measured at the master process and is the time required for the entire application computation from start to finish. It includes the times listed above.

#### *5.2.1.1 Parallel Monte Carlo Simulation for Stock Option Pricing*

The results of the scalability experiment for the option pricing application are plotted in Figure 6. As seen in the figure, there is an initial speedup as the number of workers is increased to 4. The speedup deteriorates after that. Initially, as the number of workers increase, the total tasks are more evenly distributed across the available workers causing the maximum worker time to decrease. As

expected, the initial part of the Parallel Time curve (up to 4 processors) closely follows the Maximum Worker Time curve. As the number of worker increase beyond 4, the amount of work is no longer sufficient to keep the workers busy, and the Task Planning Time now dominates Parallel Time. Here, the workers are able to complete their tasks and return the results to the space much faster than the master is able to create new tasks and put them into the space. As a result, the workers remain idle waiting for a task to become available, causing the scalability to deteriorate. This indicates that the framework favors computationally intensive coarse-grained tasks.

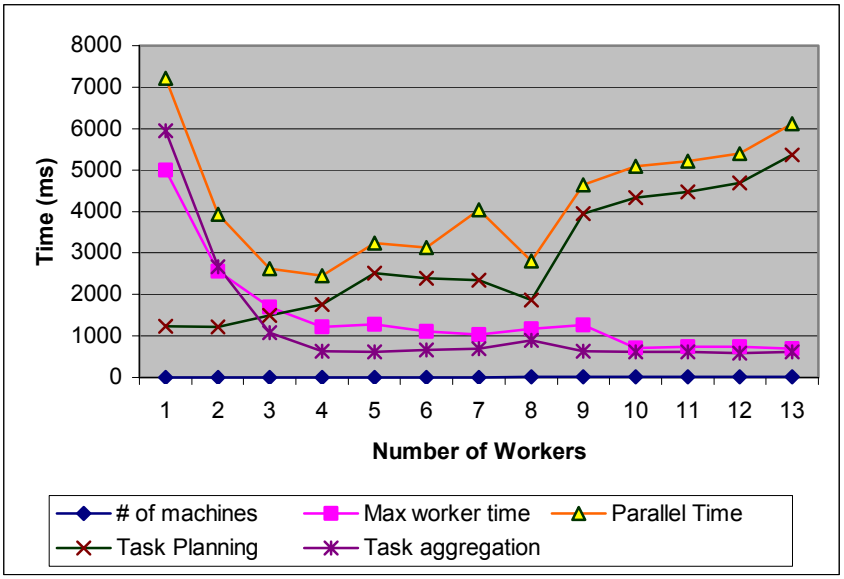



**Figure 6 - Scalability Analysis - Option Pricing Application**

*5.2.1.2 Parallel Ray Tracing*

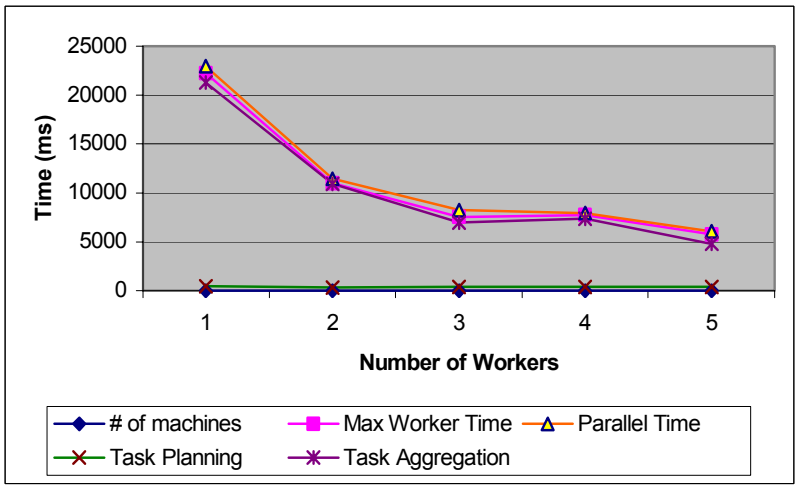



**Figure 7 – Scalability Analysis – Ray Tracing Application**

The results of the scalability experiment for the parallel ray tracing application are plotted in Figure 7. As seen in the figure, Max Worker Time scales reasonably well for this application. This is because worker computations in this application are computationally intensive. The Parallel Time is dominated by the maximum worker time and results in good overall application scalability. Note that the Task Planning Time curve is constant at 500 ms. in this case. The Task Aggregation Time curve follows the Max Worker Time curve as expected. Embarrassingly parallel applications with coarse grained computationally intensive task, such as the parallel ray tracing application, scale well and are suited to the JavaSpaces-based cluster computing framework presented.

*5.2.1.3 Web Page Pre-fetching based on Page Rank*

The results of the scalability experiment for the web page pre-fetching application are plotted in Figure 8. As seen in the figure, the application scales up to 4 processors. This application has a low task planning overhead. This is primarily due to the small amount of data that needs to be communicated between the master and the workers. Task Aggregation Time dominates the Parallel Time in this case. This involves assimilating the results returned by the workers and creating the resultant matrix. The increased task aggregation times as shown in the plot illustrate this fact. The segment size of the strips, and hence the task size can be further optimized to improve scalability.

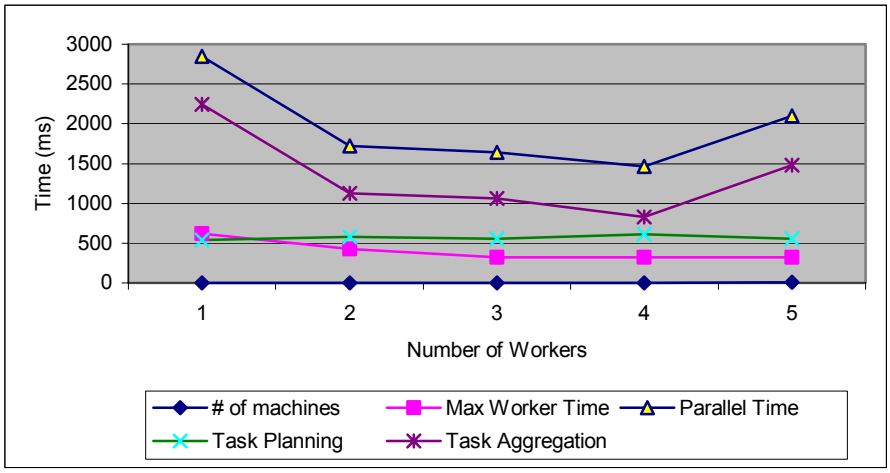


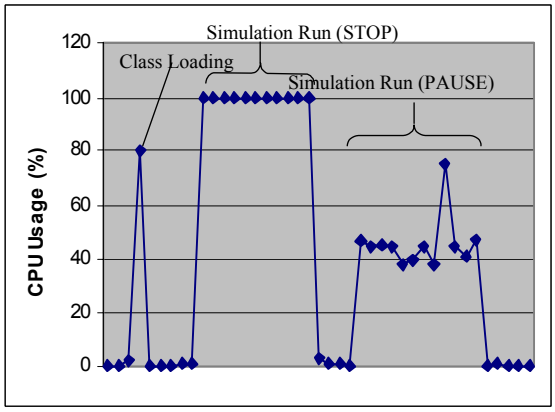

**Figure 8 - Scalability Analysis –Web Page Pre-fetching Application**

5.2.2 Adaptation Protocol Analysis

In this experiment, we analyze the overheads involved in signaling worker nodes and adapting to their current CPU load. In order to enable repeatable loading sequences for the experiments, we implemented two load simulators as part of the experimental setup. Load simulator 1 simulates different types of data transfers, such as RTP packets for voice traffic, HTTP traffic, and multimedia traffic over HTTP via Java sockets, originating at the workers. This load simulator was

designed to raise the CPU usage level on the worker from 30% to 50%. The second load simulator (load simulator 2) raised the CPU utilization of the worker machines to 100%. The results of this experiment for the three applications are presented below. Each result consists of two parts: Part (a) plots the CPU usage history on the worker machine throughout the experiment. Part (b) provides an analysis of the signaling times, and lists the Client Signal and Worker Signal times. Client Signal time is the time at which the SNMP client on the worker machine receives the signal. Worker Signal time is the time taken for the signal to be interpreted by the worker and the required action completed. The key observation in this experiment is that the adaptation overhead is minimal in all cases. Furthermore, the large remote class loading overhead at the workers is avoided in the case of transient load increases using the pause/resume states.

*5.2.2.1 Parallel Monte Carlo Simulation for Stock Option Pricing*



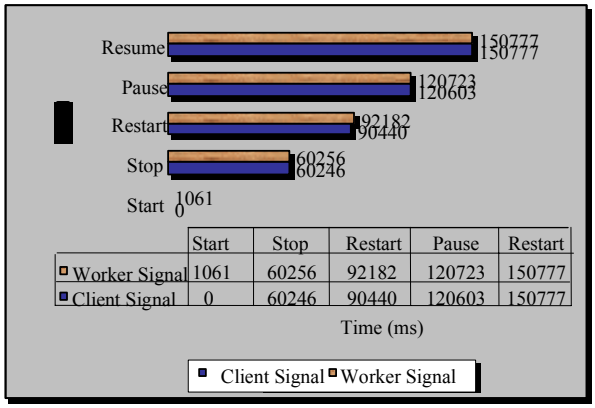




**Figure 9(a) - Worker CPU Usage** **Figure 9(b) - Worker Reaction Times**

**Figure 9 – Adaptation Protocol Analysis - Option Pricing Application**

The results of this experiment for the stock option pricing application are plotted in Figures 9(a) and 9(b). These plots show the worker behavior under simulated load conditions. In Figure 9(a), the peaks represent the times when the worker receives and reacts to the signals. The first peak is at 80% CPU usage and occurs when the worker is started (i.e. a *Start* signal). This sudden load increase is due to the remote loading of the worker implementation. The next peak at 100% CPU usage occurs when load simulator 2 is started on the worker. This causes a *Stop* signal to be sent to the worker and directs the worker to back off. Load simulator 2 is then stopped allowing the worker to once again become available and to do work. Load simulator 1 is now started causing the next peak at 46% CPU usage. A *Pause* signal is now sent to the worker temporarily suspending work execution. Finally, the simulator 1 is stopped causing a *Resume* signal to be sent to the

worker. As seen in Figure 9(b) the worker reaction times to the signal received is minimal in each case.

*5.2.2.2 Parallel Ray Tracing*

The results of this experiment for the ray tracing application are plotted in Figures 10(a) and 10(b). As seen in Figure 10(a), the first peak is at 42% CPU usage and occurs when the worker is started (i.e. a *Start* signal). This sudden load increase is once again due to the remote loading of the worker implementation. The next peak at 100% CPU usage occurs when load simulator 2 is started on the worker. This causes a *Stop* signal to be sent to the worker and directs the worker to back off. Load simulator 2 is then stopped allowing the worker to once again become available and to do work. Load simulator 1 is now started raising the CPU load to 50% to 55%. A *Pause* signal is now sent to the worker temporarily suspending work execution. Finally, the simulator 1 is stopped causing a *Resume* signal to be sent to the worker. As seen in Figure 10(b) the worker reaction times to the signal received is once again minimal in each case. The Ray Tracing application is resource intensive as illustrated by the various intermittent peaks at 78 to 100% CPU usage. These spikes occur when the task is being computed at the worker node.

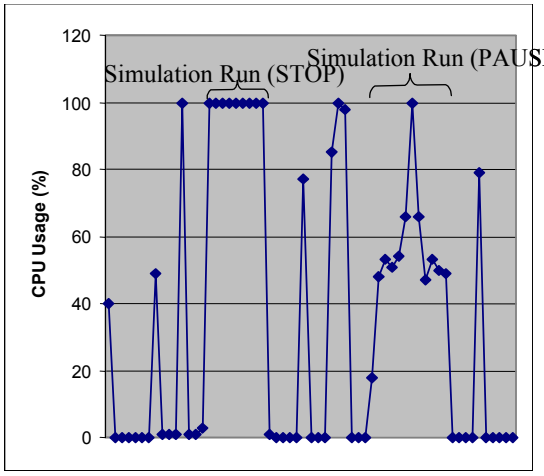



**Figure 10(a) – Worker CPU Usage**

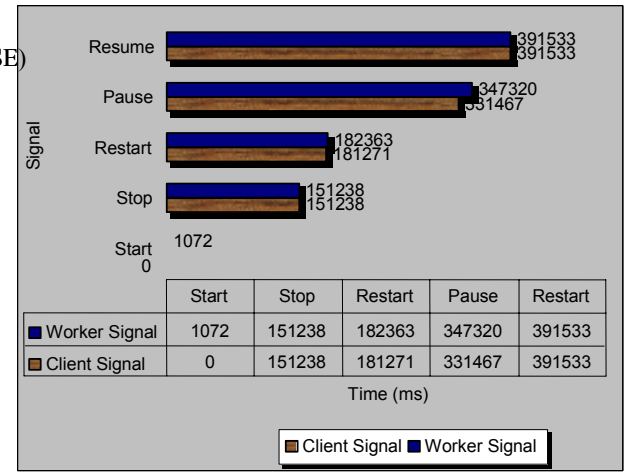



**Figure 10(b) - Worker Reaction Times**

**Figure 10 - Adaptation Protocol Analysis - Ray Tracing Application**

*5.2.2.3 Web Page Pre-fetching based on Page Rank*

The results of this experiment for the pre-fetching application are plotted in Figures 11(a) and 11(b). As seen in Figure 11(a), the first peak is at 75% CPU usage and occurs when the worker is started (i.e. a *Start* signal). This sudden load increase is once again due to the remote loading of the worker implementation. The next peak at 100% CPU usage occurs when load simulator 2 is started

on the worker. This causes a *Stop* signal to be sent to the worker and the worker backs off. Load simulator 2 is then stopped allowing the worker to once again become available and to do work. Load simulator 1 is now started raising the CPU load to 50%. A *Pause* signal is now sent to the worker temporarily suspending work execution. Finally, the simulator 1 is stopped causing a *Resume* signal to be sent to the worker. As seen in Figure 11(b) the worker reaction times to the signal received is once again minimal in each case.

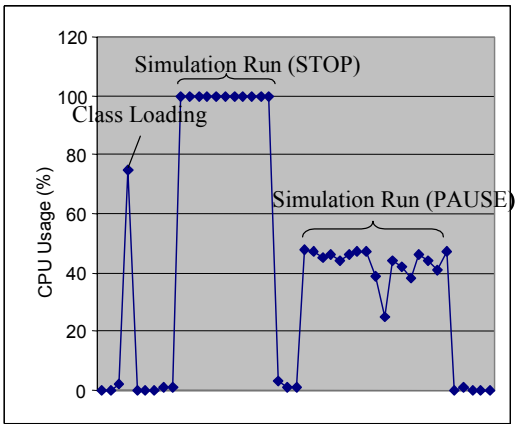



**Figure 11(a) - Worker CPU usage**

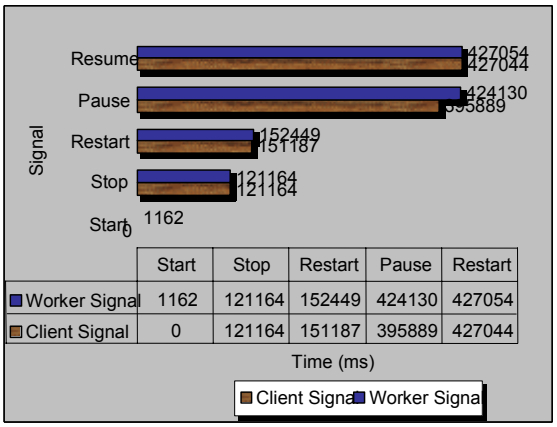



**Figure 11(b) - Worker Reaction Times**

**Figure 11 - Adaptation Protocol Analysis – Web Page Pre-fetching Application**

5.2.3 Analysis of Dynamic Worker Behavior Patterns under Varying Load Conditions

This experiment studies the dynamic behavior patterns at the workers 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 used to simulate high CPU loads are run on 25% and 50% of available workers respectively. Two plots are presented for each application run in this experiment. The first plot presents an analysis of the application behavior under the different load conditions. The four parameters measured are Maximum Worker Time, Maximum Master Overhead, Task Planning and Aggregation Time, and Total Parallel Time. Maximum Worker Time is the maximum value for worker computation time across all workers participating in the application. Maximum Master Overhead is the maximum instantaneous time taken by the master for task planning and aggregation for a particular task. Both the maximum worker time and the maximum master overhead are expected to remain constant for all three runs of the experiment. Task Planning and Aggregation Time is total time taken by the master during the task planning and aggregation phases. Finally Total Parallel Time is the time taken for the execution of the entire application and is measured at the master processor. Both the Task Planning and Aggregation Time and the Total

Parallel Time are expected to increase with increased load on the worker machines. The second plot shows the work distribution among all the workers for the three cases.

*5.2.3.1 Parallel Monte Carlo Simulation for Stock Option Pricing*

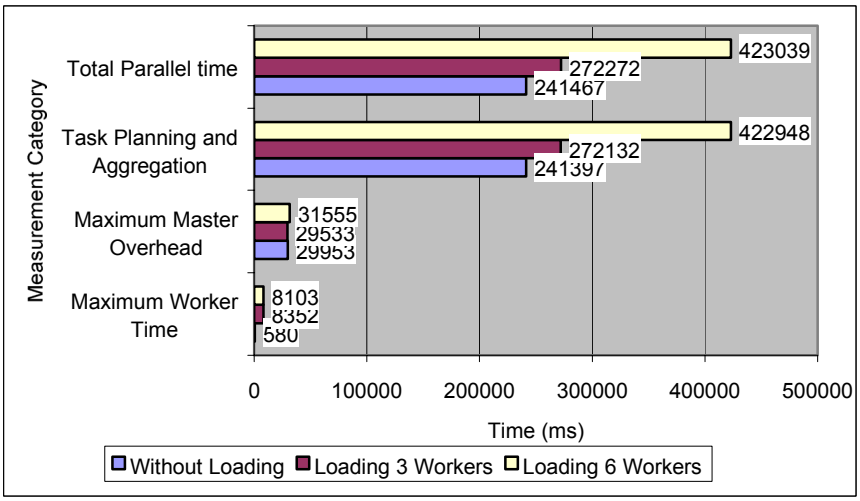



**Figure 12(a) – Execution Time Analysis (12 Workers) - Option Pricing Application**

The results of this experiment for the option pricing application are plotted in Figures 12(a) and 12(b). As seen in Figure 12(a), as the number of workers being loaded increases, the total parallel computation time increases. This is because the computational tasks that would have been normally executed by the loaded workers are now offloaded and picked up by the available workers. The task planning and aggregation times also increase, as the master now has to wait for the worker with the maximum number of tasks to return all its results into the space. The maximum master overhead and the maximum worker time remains the same across all three runs as expected.

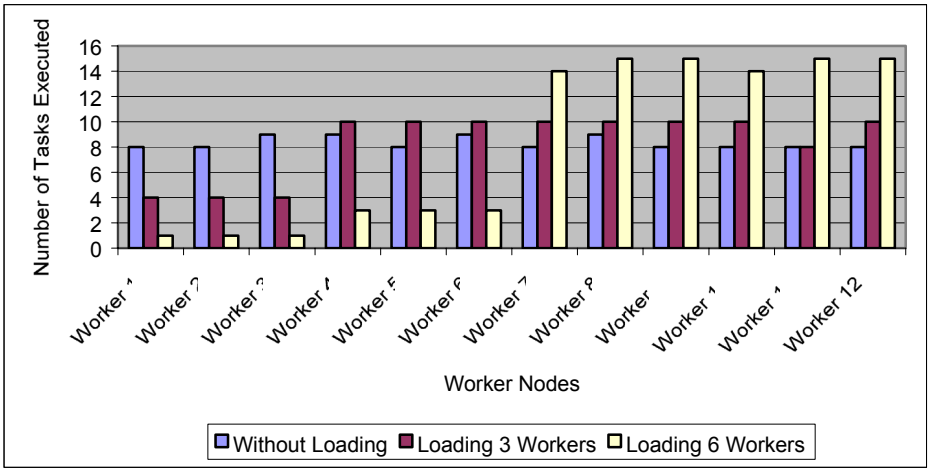



**Figure 12 (b) – Tasks Executed per Worker (12 Workers) - Option Pricing Application**

Figure 12(b) illustrates how task scheduling adapts to the varying load conditions. It shows that the number of task executed by a worker depends on its current load. Loaded workers execute fewer tasks causing the available workers to execute larger number of tasks.

*5.2.3.2 Parallel Ray Tracing*

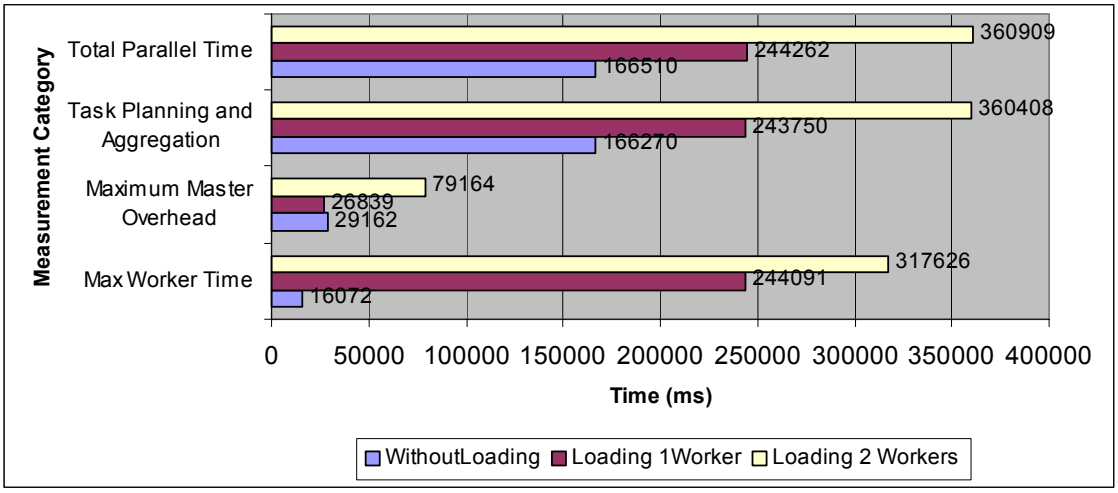



**Figure 13(a) - Execution Time Analysis (4 Workers) – Ray Tracing Application**

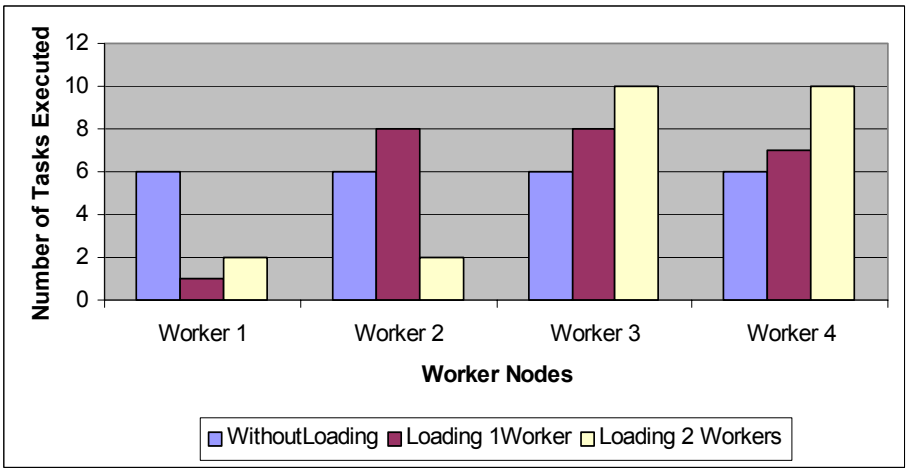



**Figure 13(b) - Tasks Executed per Worker (4 Workers) – Ray Tracing Application**

The results of this experiment for the ray tracing application are plotted in Figures 13(a) and 13(b). As expected (see Figure 13(a)), the total parallel computation time increases as the number of workers loaded increases. The task planning and aggregation times also increase as before. In this application, the Max Worker Time and the Maximum Master Overhead also increase. This is increase was due to an increased latency experienced by the worker while returning tasks to the space. A possible cause for this latency is the system or network conditions at that instant. Note that

Jini being a network-based protocol does not offer any real-time guarantees. Figure 13(b) illustrates the task distribution across the workers for the three cases.

*5.2.3.3 Web Page Pre-fetching Scheme based on Page Rank*

The results of this experiment for the web page pre-fetching application are plotted in Figures 14(a) and 14(b). As expected (see Figure 14(a)), the total parallel computation time increases as the number of workers loaded increases. The task planning and aggregation times also increase as before. The maximum master overhead and the maximum worker time remains the same across all three runs. Figure 14(b) illustrates how tasks are scheduled with changing load conditions.

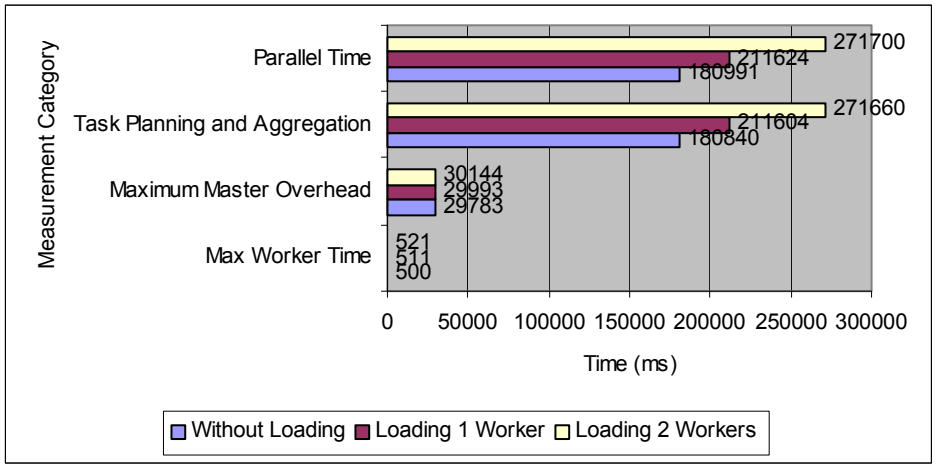



**Figure 14(a) - Execution Time Analysis (4 Workers) – Web Page Pre-fetching Application**

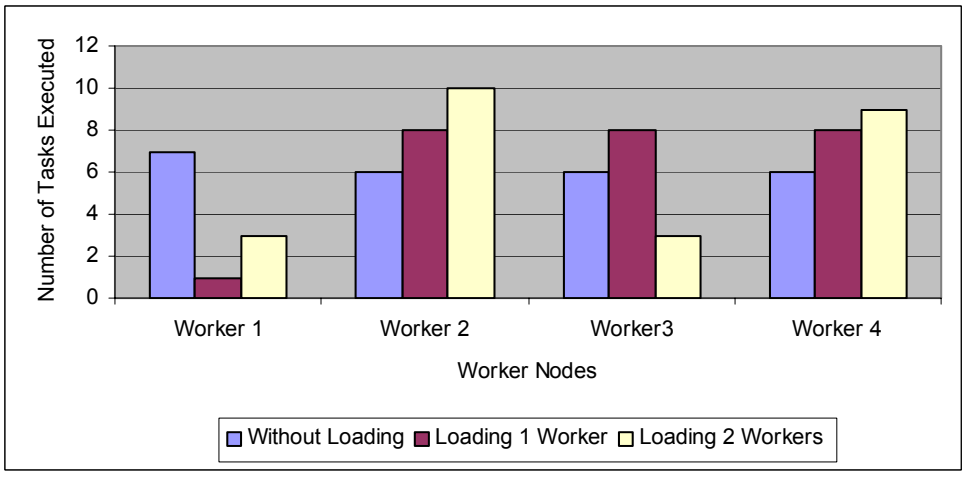



**Figure 14(b) - Tasks Executed per Worker (4 Workers) – Web Page Pre-fetching Application**

#### 5.2.4 Discussion

The experiments presented in this section show that the JavaSpaces-based cluster-computing framework could support to fit a family of applications from varied domains. The experiments also demonstrate the ability of the framework to dynamically react to the varying system state with minimal overheads and reaction times.

## 6. Conclusions

This paper presented the design, implementation and evaluation of a framework for opportunistic parallel computing on networked clusters using JavaSpaces. 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 of application code and remote configuration management of worker nodes, and uses an SNMP system state monitor to ensure non-intrusiveness. The experimental evaluation, using 3 "real-world" applications, shows that the framework v provides good scalability for coarse-grained tasks. Furthermore, using the system state monitor and triggering heuristics the framework can support adaptive parallelism and minimize intrusiveness. The results also show that the signaling times between the worker and network management modules and the overheads for adaptation to cluster dynamics is insignificant.

We are currently investigating ways to reduce the overheads during task planning and allocation phases. Furthermore, several application-specific optimizations can be introduced to improve performance. As future work, we envision incorporating a distributed JavaSpaces model to avoid a single point of resource contention or failure. The Jini community is also investigating this area. Finally, the current implementation of the framework does not provide fault tolerance. We are investigating the transaction management service provided by Jini to address this issue.

## 7. References


[1] Sun Microsystems. Javaspaces specification, http://www.javasoft.com/products/javaspaces/specs/index.html.

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

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

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

[5] 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.

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

[7] 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.

[8] 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.

[9] P. Ledru. Adaptive Parallelism: An Early Experiment with Java™ Remote Method Invocation. *Operating system Review*, Vol. 3, No. 4, October 1997.

[10] 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.

[11] A. Baratloo, M. Karaul, H. Karl, Zvi M. Kedem. An Infrastructure for Network Computing with Java Applets. *Concurrency: Practice and Experience*, Vol. 10, No.11-13, pp 1029-1041, September 1998.

[12] 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, No. 11, pp. 1139-1160, November 1997.

[13] 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.

[14] Entropia: Harnessing your PC Network, www.entropia.com.

[15] SETI@home: The Search for Extraterrestrial Intelligence, http://setiathome.ssl.berkeley.edu/.

[16] 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.

[17] Objects, the Network, and Jini, http://www.artima.com/jini/jiniology/intro.html.

[18] E. Freeman, S. Hupfer, K. Arnold. JavaSpaces Principles, Patterns, and Practice. *Addison-Wesley*, June 1999.

[19] JavaSpaces: Making Distributed Computing Easier, http://www.byte.com/feature/BYT19990915S0001.

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

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

[22] MC algorithm for Option Pricing, http://www.puc-rio.br/marco.ind/monte-carlo.html.

[23] A. Heirich and J. Arvo. A Competitve analysis of Load Balancing Strategies for Parallel Ray Tracing. *In Journal of Supercomputing*, Vol. 12, No. 1-2, pp. 57-58,1998.

[24] 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.

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

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