Adaptive Cluster Computing using JavaSpaces[1]

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

Abstract

In this paper we present the design, implementation and evaluation of a framework that uses JavaSpaces [1] 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

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. 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: 1) **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. 2) **Intrusiveness:** 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. 3) **System configuration and management overhead:** Incorporating a new resource into the cycle stealing resource cluster may require system configuration and software installation. These modifications and overheads must be minimized so that clusters can be expanded on the fly to utilize all available resources. 4) **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. 5) **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.

Recent advances in opportunistic computing have followed two approaches, *Job level parallelism* and *Adaptive parallelism*. In the job level parallelism approach, entire application jobs are allocated to available idle resources for computation, and are migrated across resources as resources become unavailable. The Condor system [2] supports cluster based job level parallelism. In the adaptive approach, the available processors are treated as part of a dynamic resource pool. Each processor in the pool aggressively competes for application tasks. This

[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.

approach targets applications that can be decomposed into independent tasks. Adaptive computing techniques can be cluster based or web based. Cluster based systems exploit available resources within a local networked cluster. Web based approach extends this model to exploit resources connected over the Internet. Systems supporting adaptive parallelism include cluster-based systems such as Piranha [3][4], Atlas [5] and ObjectSpace/Anaconda [6], and web based systems such as Charlotte [7][8], Javelin [9], and ParaWeb [10].

This paper presents the design, implementation and evaluation of a framework for adaptive and opportunistic cluster computing based on JavaSpaces that address the issues outlined above. The framework has three key features: (1) portability across heterogeneous platforms, (2) minimal configuration overheads and runtime class loading at the participating nodes, and (3) automated system sate monitoring (using SNMP – Simple Network Management Protocol [11][12]) to ensure non-intrusive behavior.

The rest of this paper is organized as follows. Section 2 describes the architecture and operation of the proposed framework. Section 3 presents an experimental evaluation of the framework. Section 4 presents our conclusions and outlines current and future work.

2. A Framework for 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.

2.1. Framework Architecture

A schematic overview of the framework architecture is shown in Figure 2. The framework consists of 3 key components: the Master Module, the Worker Module, and the Network Management Module.

JavaSpaces and Parallel Computation: JavaSpaces is a Java implementation of a tuple-space [3][4] system, and is provided as a Jini service. A JavaSpaces is a shared, network accessible repository for Java objects [13], 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 JavaSpaces API leverages the Java type semantics and provides operations to store, retrieve and lookup Java objects in the space.

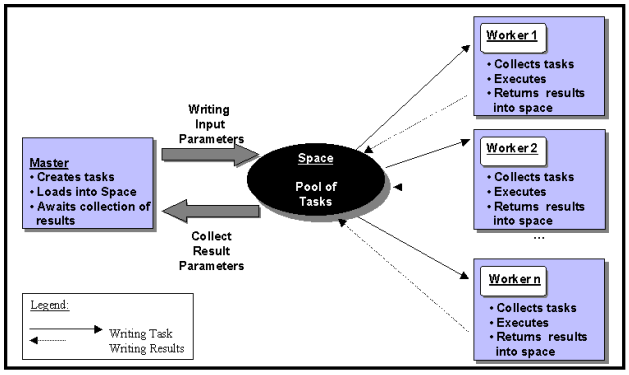

Figure 1. Master-worker parallel computing using JavaSpaces

JavaSpaces naturally supports the master-worker parallel/distributed-computing paradigm. In this paradigm (see Figure 1), 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.

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[2], and places these tasks 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*, which enables worker classes, providing the solution content for the application, to be loaded at runtime. This minimizes the overheads of deploying application code. The worker module operation is controlled by the network management module using the *Rule-Base Protocol* as described later. 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

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-

[2] 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.

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*.

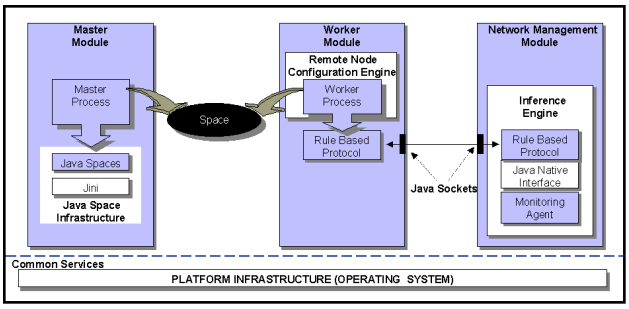

Figure 2. Framework architecture

The monitoring agent uses SNMP [11][12] 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 state parameters such as CPU load and available memory.

2.2. 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 2.2.1. 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.

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. 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 [14] 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. 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.

2.2.1. Dynamic Worker Management for Adaptive Cluster Computing. 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 (see Figure 3).

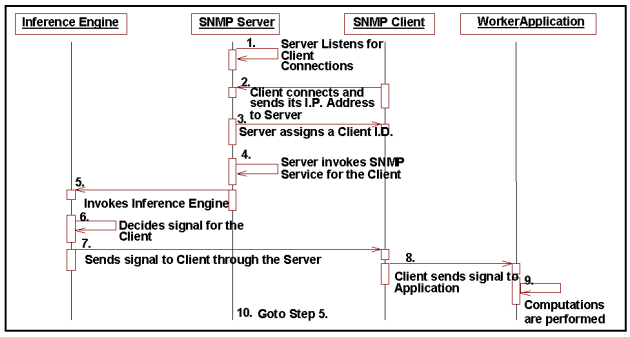

Figure 3. Rule-base protocol for adaptive worker management

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 4 and are described below.

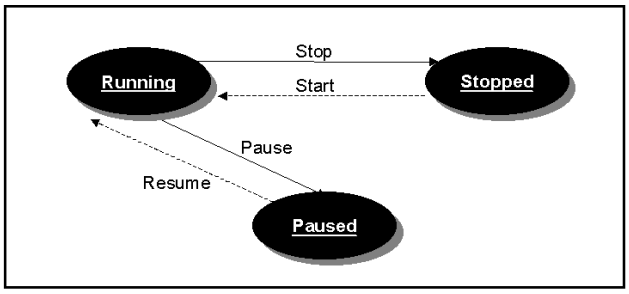

Figure 4. Worker state transition diagram

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.

3. Experimental Evaluation of the Framework

The JavaSpaces-based opportunistic cluster-computing framework is experimentally evaluated using two "real-world" applications: (1) a financial application that uses Monte Carlo (MC) simulation for option pricing, and (2) a scientific ray tracing application. 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 measures 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 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.

3.1. Parallel Monte Carlo Simulation for Stock Option Pricing

A stock option is a derivative, that is, its pricing value is derived from something else such as the underlying security, the option type (call or put), the strike price, and an expiration date. These financial terms are explained in greater depth in [15]. Parameters such as varying interest

rates and complex contingencies can prohibit analytical computation of options and other derivative prices. Monte Carlo (MC) simulation, using statistical properties of assumed random sequences is an established tool for pricing of derivative securities. In our implementation we use MC simulations, based on the Broadie and Glasserman MC algorithm [16], to model the behavior of options and account for the various factors affecting its price.

Implementation Overview: 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 consists 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 take the task from the space and perform the MC simulations.

3.2. Parallel Ray Tracing

Ray tracing [17] 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. Such 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: Rendering an image involves iterating through all the pixels in the plane and computing a color value for each pixel. 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.

3.3. Experimental Results

3.3.1. Scalability Analysis. This experiment measures the overall scalability of the application and the framework. In this experiment, we measure the Max Worker Time, Task Planning Time, Task Aggregation Time, and 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.

Parallel Monte Carlo Simulation for Stock Option Pricing: The results of the scalability experiment for the option pricing application are plotted in Figure 5.

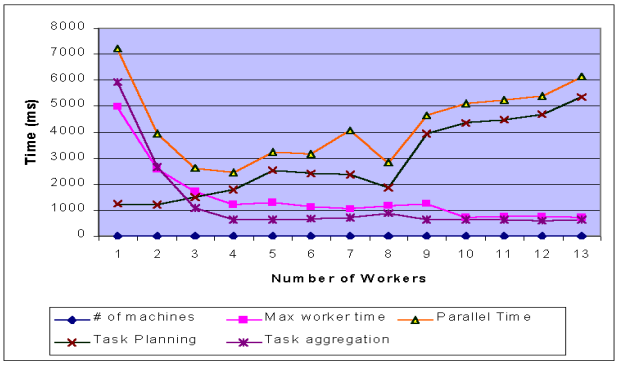

Figure 5. Scalability analysis - option pricing application

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 workers 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.

Parallel Ray Tracing: The results of the scalability experiment for the parallel ray tracing application are plotted in Figure 6.

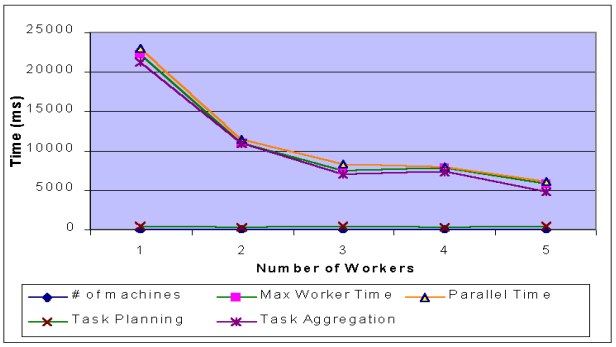

Figure 6. Scalability analysis – ray tracing application

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.

3.3.2. 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.

Parallel Monte Carlo Simulation for Stock Option Pricing:

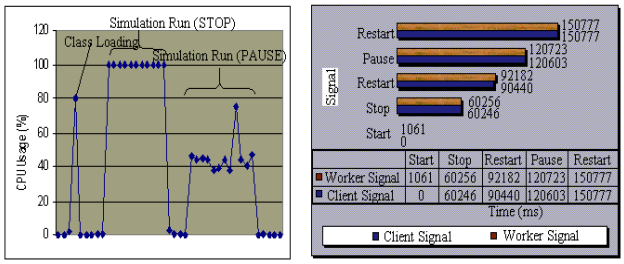

Figure 7. Adaptation protocol analysis - option pricing application (7a. worker CPU usage; 7b. worker reaction times)

The results of this experiment for the stock option pricing application are plotted in Figures 7(a) and 7(b). These plots show the worker behavior under simulated load conditions. In Figure 7(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 7(b) the worker reaction times to the signal received is minimal in each case.

Parallel Ray Tracing: The results of this experiment for the ray tracing application are plotted in Figures 8(a) and 8(b). As seen in Figure 8(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 8(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.

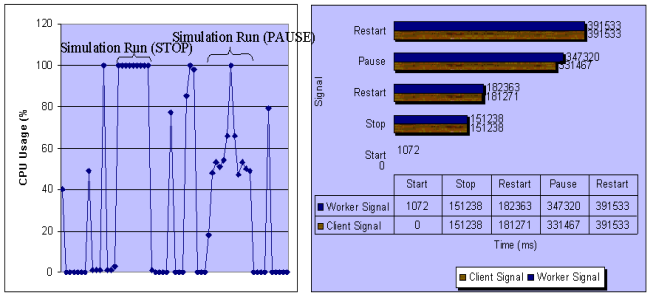

Figure 8. Adaptation protocol analysis - ray tracing application (8a. worker CPU usage; 8b. worker reaction times)

3.3.3. 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.

Parallel Monte Carlo Simulation for Stock Option Pricing:

The results of this experiment for the option pricing application are plotted in Figures 9(a) and 9(b).

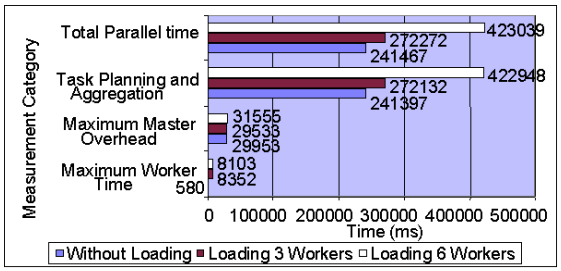

Figure 9(a). Execution time analysis (12 workers) - option pricing application

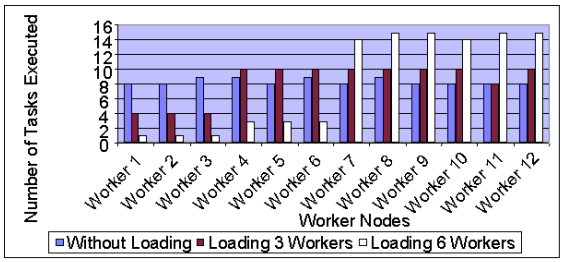

Figure 9 (b). Tasks executed per worker (12 workers) - option pricing application

As seen in Figure 9(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 9(b) illustrates how task scheduling adapts to the varying load conditions. It shows that the number of tasks executed by a worker depends on its current load. Loaded workers execute fewer tasks causing the available workers to execute larger number of tasks.

Parallel Ray Tracing: The results of this experiment for the ray tracing application are plotted in Figures 10(a) and 10(b). As expected (see Figure 10(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 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 10(b) illustrates the task distribution across the workers for the three cases.

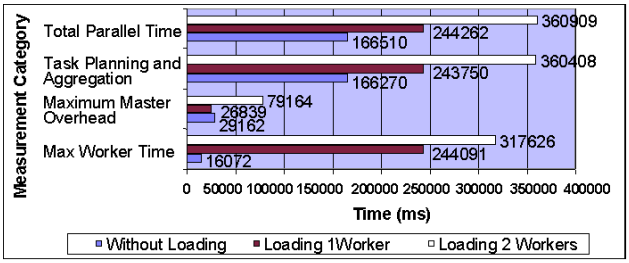

Figure 10(a). Execution time analysis (4 workers) – ray tracing application

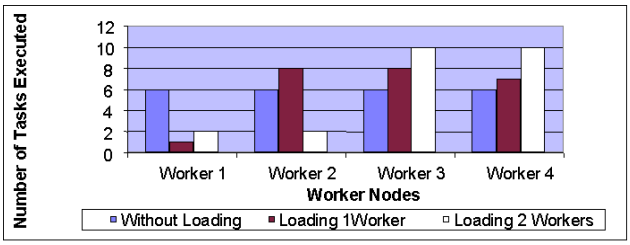

Figure 10(b). Tasks executed per worker (4 workers) – ray tracing application

4. Conclusions

This paper presented the design, implementation and evaluation of a framework for opportunistic parallel computing on networked clusters using JavaSpaces. 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 shows that the framework provides good scalability for coarse-grained tasks. Furthermore, using the system state monitor and triggering heuristics the framework can support adaptive parallelism and minimizes 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 also looking into comparing our approach with approaches that exhibit Job level parallelism through check pointing and process migration as well as comparing different paradigms for parallel processing. We are investigating the transaction management service provided by Jini to address this issue.

5. References

[1] Sun Microsystems. JavaSpaces, www.javasoft.com/products/javaspaces/specs/ (1998)

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

[3] M. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky, "Adaptive Parallelism and Piranha", IEEE Computer, Vol. 28, No. 1, January 1995, pp. 40-49.

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

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

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

[7] 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), September 1996, pp. 181-188.

[8] A. Baratloo, M. Karaul, H. Karl, and Z. Kedem, "An Infrastructure for Network Computing with Java Applets", Concurrency: Practice and Experience, Vol. 10, September 1998, pp. 1029-1041.

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

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

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

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

[13] E. Freeman, S. Hupfer, K. Arnold, "JavaSpaces Principles, Patterns, and Practice", Addison Wesley, June 1999.

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

[15] Glossary of Financial Terms http://www.centrex.xom/terms.html

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

[17] A. Heirich and J. Arvo, "A Competitive analysis of Load Balancing Strategies for Parallel Ray Tracing", In Journal of Supercomputing, 1998.