An Agent-Based Infrastructure for Parallel Java on Heterogeneous Clusters

Jameela Al-Jaroodi, Nader Mohamed, Hong Jiang, and David Swanson
Department of Computer Science and Engineering
University of Nebraska – Lincoln
Lincoln, NE 68588-0115
[jaljaroo, nmohamed, jiang, dswanson@cse.unl.edu]

Abstract
In this paper, we introduce an agent-based infrastructure that provides software services and functions for developing and deploying high performance programming models and applications on clusters. A Java-based prototype, based on this architecture, has been developed. Since this system is written completely in Java, it is portable and allows executing programs in parallel across multiple heterogeneous platforms. With the agent-based infrastructure, users need not deal with the mechanisms of deploying and loading user classes on the heterogeneous cluster. Moreover, details of scheduling, controlling, monitoring, and executing user jobs are hidden. In addition, the management of system resources is made transparent to the user. Such uniform services, when rendered available in a ubiquitous manner, are essential for facilitating the development and deployment of scalable high performance Java applications on clusters. An initial deployment over a heterogeneous, distributed cluster results in significantly enhanced performance; absolute performance compared to C (MPI) improves with increased granularity of the algorithms.

Key Words: Parallel Java, distributed Java, cluster, heterogeneous systems, software agents, middleware.

1. Introduction
An agent-based infrastructure is developed to provide services for high performance-programming environments and applications that can be used on clusters or heterogeneous systems. Software agents enhance expandability, allowing the number of machines involved to grow easily, by providing services that include job distribution, monitoring, and controlling for the system. Agent technology has been used in many systems to enhance the performance and quality of service provided. One example is in communications systems where software agents provide powerful means to let systems inter-operate and manage large heterogeneous networks [6]. In this study, we utilize software agents for a parallel and distributed Java programming environment. This provides flexibility and ease of managing the different resources available.

A new choice of architecture for high performance computing has emerged to be clusters of PCs/workstations [3] and heterogeneous networked systems. Such systems provide processing power comparable to special purpose high-end multi-processor systems for a fraction of the cost. However, these systems require system and software services that can support the distributed architectures and provide transparent and efficient utilization of the multiple machines available. Moreover, as Java becomes more popular and available on almost any platform, there has been an increasing interest in using Java for high performance computing and distributed applications on clusters. This has generated intensive research in the area of parallel, distributed and multithreaded programming environments and tools for Java on distributed systems.

In the rest of this paper, we first discuss parallel Java programming models and identify common infrastructure service requirements on clusters and heterogeneous systems in section 2. Then, in section 3, we describe the architecture and features of the proposed infrastructure. Section 4 discusses an example, the Java Object-Passing Interface (JOPI), that utilizes the services provided by the agents, and presents the results of experiments conducted to evaluate the performance. In section 5, the advantages of the agent-based infrastructure are discussed. Finally, Section 6 concludes the paper with remarks about the current and future work.

2. Parallel Java on Clusters
Java’s popularity among developers is increasing steadily and many research groups have started exploring the possibilities of using Java on multi-processor systems.
Since Java is machine independent, it is possible to have the same Java programs run on any platform that has a Java virtual machine (JVM). In addition, Java has been constantly improved and optimized for performance over the last few years. Very recently, much work has been done to provide high performance capabilities in Java for distributed systems. Many research groups have worked on providing parallel Java using many different approaches and programming models. Our recent study [2] provides useful information about many of these projects, their models and approaches. In this section, a number of parallel Java programming models on clusters are reviewed. In addition, requirements to implement and deploy these models on clusters are studied. The aim is to identify the generic services and functions needed by different programming models.

2.1 Java Programming Models on Clusters

Providing parallel capabilities in Java can be achieved by following the known parallel programming models. These modes are divided into four layers (categories) based on the level of user involvement in the parallelization process and the achievable levels of efficiency. In addition, the implementation dependencies can be observed among these layers. They are described below in decreasing level of user involvement and system efficiency.

1. Message Passing: In this category, the system provides some form of information exchange mechanism among distributed processes. It provides a number of library functions to exchange messages among processes such as point-to-point and group communication primitives, synchronization and other functions. The user program, with a unique ID, runs in multiple processes distributed on the cluster nodes. This approach requires the utilization of a run-time support system to handle the message exchange between the participating machines. The message passing library run-time support can be implemented in many different ways such as using a pure Java implementation based on socket programming, native marshaling, and remote method invocation (RMI) [14] or by utilizing Java native interface (JNI), Java-to-C interface (JCI), parallel virtual machine (PVM) and other parallel infrastructures. A number of projects have tried to comply with message-passing interface (MPI) [10] and the proposed standard for message passing in Java MPJ [4], while others were based on a new set of class libraries and object-passing interfaces.

2. Distributed Shared Address Space: In distributed shared address space or distributed shared object model, the system presents an illusion to the user of having a single address space where all or some data/objects are available to all participating processes. The underlying infrastructure can be implemented in different ways. One example is using an existing distributed shared memory (DSM) system; another is utilizing a message-passing infrastructure. The environment should handle the different issues of shared space such as information (data or objects) integrity and coherence, synchronization and consistency.

3. Automatic Parallelization of Multi-Threaded Programs: This model aims to provide seamless utilization of a distributed environment to execute multi-threaded programs on multiple machines. The main goal here is to be able to run concurrent multi-threaded applications in parallel without having to modify or rewrite them. In this case, the implementation issues are similar to the shared space model in the sense that all data and objects used by the threads need to be sharable. This also means that the underlying run-time support requires data sharing or data exchange mechanisms to provide thread distribution and information sharing.

4. Transparent (Automatic) Parallelization: Here, some systems provide transparent parallelization of Java programs written in standard Java by modifying the JVM, while others utilize preprocessors to achieve this goal. In both cases, the main goals are to relieve the developer of the burden of explicitly parallelizing the application and to run current applications in parallel without (or with minor) modifications. Again, we need to have a run-time support to execute the automatically generated parallel programs. The run-time support could be built from scratch or could utilize any of the infrastructures described in the above three categories.

More detailed information about the classification, implementations, and comparison of parallel and distributed Java projects for heterogeneous systems can be found in [2].

2.2 Identifying Common Requirements

Standard Java technology such as JVM and JINI [8] provide a variety of features to develop and implement distributed Java applications. However, there are some key features lacking in standard JVM when directly applied as an underlying infrastructure for constructing and running parallel Java on clusters. The missing features, needed by parallel Java programming models on clusters, are:

1. Loading User Programs onto the Remote JVMs of the Cluster Nodes: Java does not provide mechanisms to remotely load user classes on more than one JVM in parallel. A user program needs to be loaded onto all JVMs of the nodes where it is scheduled. Thus, the
A parallel Java environment needs mechanisms to remotely load user classes onto the selected cluster nodes before starting the parallel program.

2. Managing Cluster Resources and Scheduling User Jobs: In order to efficiently run parallel user programs, the system needs to schedule the user program based on the availability of cluster nodes and of resources in each node. Thus, a mechanism is needed to monitor, manage and maintain the resources of the entire cluster. Available resources on a node may include the number of idle or sharable processors, unoccupied memory space, current workload, the number of free communication ports, and sharable data and objects. Running user programs may require holding some of these resources.

3. Security: Some resources on the cluster nodes or distributed system may need to be protected from running a remote user job. For example, while a user is allowed to run a program on a remote node, he/she is not allowed to access any files on the remote nodes or change their system properties. Although basic security mechanism is available in Java to protect selected node resources, it would nevertheless be preferable if high-level functions provide the security mechanism.

4. Job and Thread Naming: For an environment supporting multiple parallel jobs, a unique job ID is needed to assign to each active parallel job. In addition, each parallel job may consist of multiple threads distributed on the cluster nodes; therefore, a thread ID is needed for each thread. Thread IDs are numbered, starting from 0, as master thread, then 1, 2, 3, ..., n-1, where n is the number of threads. The job ID is needed to control a specific job, for example to kill a job. The thread ID is used to distinguish one thread from another and to control the flow of the parallel program in the message-passing model. To allow user threads to communicate, a mechanism to provide a mapping between logical thread IDs and actual network addresses such as IP address and network port is needed.

5. User Commands and Collective Operation: Users need commands to run, schedule, and monitor their parallel Java programs and to control the environment from a single point on the cluster. Examples of these commands are to report available resources and currently running parallel Java jobs. These commands should provide the user with a single system image.

3. An Agent-Based Infrastructure

The proposed agent-based infrastructure is designed to satisfy the requirements discussed above. This system provides a pure Java infrastructure based on a distributed memory model. This makes the system portable, secure and capable of handling different programming models. The system has a number of components that collectively provide middleware services for a high performance Java environment on cluster and heterogeneous systems (see figure 1), including some of the common requirements described above. These components include the software agents and client services. In this section, we discuss the design and functions of these components and show the system’s main features that satisfy the requirements to support parallel Java programming models.

![Figure 1](image_url). A block diagram of the system architecture. Each workstation maintains one agent that manages multiple threads. Dynamically created threads communicate directly with one another. Users use client services to interact with the environment.

3.1 Agents

Software agent technology has been used in many systems to enhance the performance and quality of services provided by those systems [6]. The proposed infrastructure utilizes software agents for a high-performance Java environment, which provides flexibility, expandability and ease of managing a variety of resources. The main functions of the agents are to deploy, schedule, and support the execution of the Java code, in addition to managing, controlling, monitoring, and scheduling the available resources. Agents can manage and deploy user jobs on a single cluster or on a collection of heterogeneous networked systems. When a user job (parallel Java application) is submitted, an agent will perform the following tasks:

1. Examine the available resources and schedule the job for execution on the required set of processors. The scheduler balances the load among processors by assigning jobs to the least loaded group of processors.
2. Convert scheduled user classes into threads, remotely upload the threads to processors, and start execution directly from main memory.

3. Continue to monitor and control system resources, in addition to providing some monitoring and control functions to the user.

For high throughput, the agents are designed to be multithreaded, where each thread serves a client’s request. Once user threads are deployed, they will directly communicate with one another to perform parallel tasks thus freeing the agents to deal with other requests more efficiently and avoiding introducing overhead on the user programs. Each agent, shown in Figure 2, consists of a number of components whose functions are described below:

1) **The Request Manager** takes care of user job requests such as deploying user classes, starting a job, stopping it and checking agents or threads status. It receives the request object from the client services and provides the requested service.

2) **The Resource Manager** provides methods to manage, schedule, and maintain the resources of the machine where the agent is working. It keeps records of all running threads, machine and communication resources utilization, and performance information. In addition, it is responsible for reclaiming system resources after the termination of a user job.

3) **The Security Manager** provides security measures for the system (see Section 3.4 for details).

4) **The Class Loader** remotely loads user classes onto the JVMs in preparation for execution. To load classes on different remote machines for job deployment, the original Java class loader was replaced by one written specifically for this environment.

5) **The Scheduler** selects the machines to run a user job on the requested number of processors. To generate a schedule, the scheduler executes a test program, whose results are sorted in order of increasing response time. Based on the number of processors requested, the scheduler assigns an appropriate number of processes per agent, starting with the fastest. The assignment is based on the number and speed of processors available in the system. This method provides a basic level of load balancing among the processors.

In addition to the agents, some support classes are needed to complete the system’s functionality. The `agentClient` class provides API for client/agent and agent/agent communication and `AgentGroup` class provides API for communication with a group of agents.

### 3.2 Communication among System Components

To maintain the portability and pure Java implementation of the agent, we used the mechanisms provided by the standard Java for communication among system components. However, standard Java provides two mechanisms for communication among remote processes: 1- Socket Programming and 2- Remote Method Invocation (RMI). RMI provides advanced services such as support for data serialization, remote class loading, and socket manipulation, which is inefficient and inflexible compared to socket programming for system level and infrastructure programming where more control is needed. For example, RMI does not provide asynchronous or local-area multicast communication. RMI, however, provides a simple communication mechanism in the form of method invocation, but it compromises efficiency by adding an extra layer between the application and the primitive communication mechanisms. Consequently, socket programming was adopted as a communication mechanism in the current infrastructure implementation.

The agents manage, control, allocate, and reclaim communication resources such as port numbers. Java serializable objects were used as a communication unit among the system’s components where each object represents a specific type of request or reply.

![Figure 2: The Agent Structure](image)

**Figure 2** The Agent Structure. (Although scheduler shown separately, it is functionally part of the agent).

### 3.3 Client Services and Environment APIs

The client services and environment APIs provide commands for users to interact with the environment. Requests are accepted from the user, and passed to the agent after they are encapsulated as an object with the
The following commands are available for the user through the client services:

1) \texttt{pjava} initiates a parallel user job,
2) \texttt{pingAgent} lists available agent(s) and their status,
3) \texttt{listThreads} lists all active threads on the agent(s),
4) \texttt{testAgents} reports performance information about the machines where the agents are running,
5) \texttt{killJob} terminates a user job (like kill in UNIX),
6) \texttt{startAgents} starts agents on appropriate nodes in the system,
7) \texttt{resetAgent} resets agent(s), and
8) \texttt{killAgent} kills (terminate) agent(s).

The client services class uses two types of classes for communication between clients and agents and among agents:

1) \texttt{agentClient}: provides API to manage, control, and send request for an agent. This class is used for direct communication between the client and a given agent or among agents.

2) \texttt{agentGroup}: provides API to manage, control, and send requests for a group of agents. This class can use \texttt{agentClient} to individually communicate information to all agents in the group. For example, when a user job is initiated, the request object and the schedule object for that job are passed to the \texttt{agentGroup} that can then use the \texttt{agentClient} to pass them to the agents.

Both \texttt{agentClient} and \texttt{agentGroup} can be used as API for users to develop distributed and high-performance applications that utilize resources available on clusters or heterogeneous systems.

### 3.4 Multi-user and Security Issues

The system allows multiple users to run multiple jobs at the same time. To properly manage these jobs, each job has multiple levels of identification, starting with a unique job ID assigned by the system. The user ID and the program name further distinguish different jobs. Within each job, thread IDs are used to identify the remote threads of the job. Executing user threads on remote machines exposes these machines to many “alien” threats, raising security and integrity concerns. A mechanism to protect these machines must be utilized to ensure safe execution. Java provides a default security manager that provides some level of protection for the system resources. When the security manager is set for an application, many operations will be checked against the defined security policy before they are executed. The security manager in Java has some restrictions, thus many functions were modified or rewritten for this system. More specifically, two modes of execution are used to provide a robust and secure environment in this system:

1) **The Agent Mode** in which no restrictions are imposed. A thread running in this mode will have full control of all the resources and operations in the system. All agents run in agent mode.

2) **The User Mode** in which restrictions are applied to limit the user access to the system resources. Some operations are disabled in this mode such as deleting files, creating a sub-process or using system calls, modifying system properties, and writing files.

With the security modes in place, the user processes have full access to resources on their local machine (where the user job was initiated). In addition, user processes have very limited access to all remote machines’ resources (since they are running in user mode). To provide users with access to necessary resources for their application, the root (master) process is forced to execute on the user’s local machine. However, the user has the option to override this setting and allow the root process to execute on a remote machine, but the application will have no access to any system’s resources. Nevertheless, this policy can be adapted to provide different levels of access control on the available machines. For example, a user on a cluster may be given full access to all cluster nodes, but limited access to external systems. Another example is deploying an authentication policy to allow different access modes on the different participating machines.

### 4 Example of Using the Infrastructure

The agent-based infrastructure is capable of supporting different parallel and distributed programming models. Examples of this support include the implementation of a distributed shared object API’s for Java (work in progress) and that of a Java object-passing interface (JOPI) [1]. In addition, distributed applications can utilize this infrastructure to facilitate their operation. JOPI was implemented on top of the services provided by the infrastructure and a set of experiments was conducted to measure its performance. JOPI is an object-passing interface with primitives similar to MPI, where information is exchanged by means of passing objects. Using JOPI, users can initiate point-to-point communications in synchronous and asynchronous modes, group communications and synchronization in addition to a few other supporting functions and attributes.

JOPI utilizes most of the features provided by the agent-based infrastructure, including the scheduling mechanisms, deployment and execution of user classes, control of user threads and available resources, and the
security mechanisms. In addition, the agent client services provide uniform user access to control JOPI programs. Writing parallel programs using JOPI is generally simpler than using C and MPI because Java is object-oriented and the user can define a class for solving the sequential version first and test it, then use it as the main object in the parallel version. A few additional methods and classes can be added to the original classes to facilitate the decomposition and distribution of the problem, exchange of intermediate results, and reconstruction of the solution. Finally, the main class will be written to implement the parallelization policy such as deciding on the size of subproblem, load balancing, etc. Passing objects in JOPI preserves the object-oriented advantages in Java and simplifies the parallelization process and the information exchange mechanisms.

4.1 JOPI Application Performance

Some benchmark programs were written to evaluate the performance of the system using JOPI. All experiments, unless otherwise mentioned, were conducted on Sandhills, a cluster of 24 Dual 1.2 GHz AMD processor nodes, with 256 KB cache per processor and 1 GB RAM per node. The cluster is connected via a 100 Mbps Ethernet. For these experiments, the standard JVM version sdk 1.3.1 was used. The experiments were designed to measure the agents’ overhead, the communication performance, and the parallelization efficiency in terms of speedup.

4.1.1 Agent Overhead. To test the agent overhead, a Java program was executed independently (without the agent) and then through the agent. The average execution time for both experiments were measured and compared. Currently a small and program independent overhead (around 0.37%) was found since the agent is very lightweight. We assume that adding more functions to the agent may introduce additional but relatively minor delay.

4.1.2 Communication measurements. A Ping-Pong program was used to exchange messages of incremental sizes between two processors. One program was written in C using MPI and another was written in JOPI. The return trip time, RTT, was registered and effective bandwidth was calculated as follows:

\[
\text{Bandwidth (Mbps)} = \frac{(8 \times \text{Message size} \times 2^{20})}{\text{RTT}/2}
\]

The results (see Figure 3) show that, for small messages (less than eight KB), the RTT in MPI is lower than that in JOPI and both show some effects of overhead. With larger messages, the overhead starts to become less evident and the RTT grows linearly with the growth of the message size. Such linear growth is also evident in JOPI for larger messages (greater than 16 KB), due mainly to serialization [16]. In addition, at large message sizes the figures show that JOPI matches and eventually surpasses MPI’s performance. For example, at 64KB MPI gives RTT of 0.012 seconds (i.e. 84.6Mbps) while JOPI gives 0.013 seconds (76.9Mbps), but at 256KB, MPI gives 0.049 seconds (81.2Mbps) and JOPI gives 0.047 seconds (85.1Mbps). To summarize, the overhead of communication in JOPI affects mostly messages of small sizes, mainly due to the serialization. As measured in our experiments the serialization delay is around two milliseconds and may increase slightly as the size or contents of the object increases. Serialization and deserialization delay is independent of the application; therefore, it becomes insignificant as the application becomes larger and more computation intensive.

4.1.3 Comparing with MPI. The dense matrix multiplication algorithm [5] was used to compare the performance of the system with C and MPI. The matrix size was 720x720 and the first matrix was sent as a whole to all processes while the second matrix was divided into groups of columns that were then distributed to processes, including the root process. All programs used blocking point-to-point communication to ensure uniformity between the Java and C programs. Matrix elements are randomly generated floating point numbers. C programs were compiled with gcc and maximum optimization (-O3).

From the measurements obtained (Figure 4) we observe the following:

1) MPI is generally faster than JOPI, but the speedup gain in JOPI grows faster with the increase of number of processors. For example, the sequential C program is three times faster than Java (4.8s to 12.1s), but at eight processors, the C program is only 1.8 times faster (1.99s to 3.49s).

2) Using large objects (e.g. large matrix) reduces the communication overhead effect, thus the performance of JOPI becomes more efficient. This is evident for the

![Figure 3 RTT vs. Message size: JOPI & MPI](image-url)
continuing gain in speedup at 6 and more processors compared to MPI

3) Although the optimized version of C was used for comparison, it is not a fair comparison for Java because Java does not do code optimization.

4.1.4 Matrix Multiplication (load balancing). The matrix multiplication algorithm [5] used here also employs blocking point-to-point communication methods. The difference here is the distribution mechanism. The first matrix is divided into blocks of rows and the second is divided into blocks of columns. Each process gets a block of rows and a block of columns and the multiplication is performed using a cache friendly algorithm. When work is done, the process sends its results to the master process and takes new blocks to work on. Here a matrix of 1800x1800 floating numbers was used, with a stripe size of 300 rows or columns.

The results are illustrated in figure 5. The initial super linear speedup can be attributed to the fact that each of the parallel processes deals with a small part of the matrix, which leads to fewer cache misses. At the same time, the sequential program has a larger data set to deal with, which slows its execution due to cache misses (even though it is using the same cache friendly algorithm). In general, this experiment shows that JOPI can achieve very good results given an efficient algorithm.

4.1.5 Traveling Salesman Problem (TSP). The algorithm is based on branch-and-bound search [9]. This problem required using many of the JOPI primitives to implement an efficient load-balanced solution. Broadcast was used to distribute the original problem object to processes and used by processes to broadcast the minimum tour value found. This allows other processes to update their minimum value to speedup their search. In addition, asynchronous communication is used by the processes to report their results to the master process, while continuing to process other parts. The results obtained (Table 1 and Figure 6) show good speedup and efficiency measures with growing number of processors and fixed problem size (22 cities). The results also indicate that many problems that require infrequent communication can be implemented efficiently using this system.

4.1.6 Heterogeneous system Experiment. This experiment shows the capabilities of this system to execute parallel applications on heterogeneous platforms with minimum user involvement. Using the matrix multiplication in 4.1.4, a cluster of different platforms was used to execute the program. The machines used were the Lunix nodes (Sandhills), RCF [13], and a Windows2000 server (see Table 2). To fairly compare performance, the sequential running time for the program was measured on each platform. As a first order approximation, speedup is calculated individually for each participating platform by dividing the sequential time on that platform by the parallel execution time on the system.

<table>
<thead>
<tr>
<th># procs.</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (sec.)</td>
<td>83986.68</td>
<td>18255.84</td>
<td>9130.18</td>
<td>4634.58</td>
<td>3136.71</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.000</td>
<td>1.862</td>
<td>3.722</td>
<td>7.333</td>
<td>10.835</td>
</tr>
<tr>
<td>Efficiency</td>
<td>100.00%</td>
<td>93.08%</td>
<td>93.06%</td>
<td>91.67%</td>
<td>90.29%</td>
</tr>
</tbody>
</table>
Due to the communication overhead imposed by the distributed setting of the cluster, the system is most suitable for applications that have high computation to communication ratio, or coarse grain parallelism, as expected. In addition, the varying specifications of the processors used indicate the possibility to achieve more speedup and faster response times by distributing tasks based on their suitability to the platform. For example, if some tasks require excessive data sharing, they can be assigned to a multiprocessor parallel machine, while relatively independent tasks can be assigned to a cluster.

Table 2 The results of the heterogeneous experiment.

<table>
<thead>
<tr>
<th>Cluster nodes Specifications</th>
<th>Operating system</th>
<th>Procs used</th>
<th>Time seq.</th>
<th>Speedup / platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 nodes, dual 1.2 MHz AthlonMP, 256KB cache, 1GB RAM per node</td>
<td>Linux 6, inc. root</td>
<td>398</td>
<td>9.258</td>
<td></td>
</tr>
<tr>
<td>RCF: Origin 2000, 32 processors, 250 MHz, 4MB cache, 8GB RAM</td>
<td>IRIX 6.5.13</td>
<td>3</td>
<td>36.030</td>
<td></td>
</tr>
<tr>
<td>CSNT-TS: 3 CPUs, Intel x86 700MHz, 1.5GB RAM</td>
<td>Windows 2000 server</td>
<td>3</td>
<td>9.746</td>
<td></td>
</tr>
<tr>
<td>Total Processors</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel time</td>
<td>42.992</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Benefits of Using Agents

In this paper, we introduced an agent-based infrastructure to support high performance parallel and distributed computing in Java. The utilization of software agents provided a number of benefits, of which some are:

1) Portability: In addition to Java’s machine independence, the use of agents provides the needed functionality to make the system fully portable. This allows the system to work seamlessly across multiple platforms regardless of the underlying architecture or operating system. Thus, an application utilizing this infrastructure, such as parallel programs written using JOPI, can run simultaneously on many heterogeneous machines. Here, the agents play an important role in distributing the user processes to remote machines and deploying them remotely as threads.

2) Expandability: Agents provide the system with the flexibility to expand the hardware easily. By activating an agent on a machine, it becomes part of the system where user jobs can execute. This process is transparent to the user and does not require any changes in the applications using the system. The user will be able to access and utilize any resources that the agents have access to regardless of their location, hardware architecture or operating environment.

3) Flexibility: Since the infrastructure is detached from the applications utilizing it, it is easy to modify its components, and add more features and functions to it. In this case, it is possible, for example, to change the scheduling module, deployment or communication mechanisms without requiring changes to the applications. Additional functions, such as fault tolerance, dynamic load balancing, and resource monitoring and discovery, can also be easily integrated into the system.

4.2 Discussion

The proposed infrastructure provides a platform for parallel Java using JOPI. With JOPI, we are able to achieve good communication and speedup performances. However, JOPI in its current form is most suitable for coarse grain parallelism, but can be optimized to handle finer grain parallelism. As described earlier, JOPI utilizes the agent-based infrastructure to provide its services and, from the experiments results, we see that the agents impose a very small overhead while providing efficient and flexible functions for JOPI. The agents allow user jobs to be deployed and executed on remote machines transparently, requiring no user involvement other than specifying the number of processors needed. Therefore, the user need not know nor deal with the details of the different hardware resources being used. This has allowed easy utilization of multiple distributed platforms of different specifications to execute a single parallel application. In addition, the pure Java implementation, allowed for utilizing different heterogeneous platforms to simultaneously execute parallel Java applications.
4) Security: The agent’s security module protects the participating machines and provides a safe environment to execute user parallel/distributed applications. In addition, this module can be further enhanced to support any required security protocol to further protect the system.

5) Resources Management: Agents reside on all participating machines and could collect the necessary information about the resources of their respective home machines. This provides a distributed information base for all the system resources. The agents could exchange the information to provide efficient and comprehensive resource management tools for the system. Adding resource discovery modules that enable the detection of and adaptation to the changes in the resources of the system could further strengthen the agents.

6 Conclusion and Future Work

The proposed agent-based infrastructure provides services to support the development of high performance parallel and distributed Java applications that can be executed on clusters and heterogeneous systems. It is observed that this infrastructure has some points of uniqueness to it. One is the use of software agents for flexibility and expandability. Another is the preservation of the compatibility with available Java virtual machines. The infrastructure has security measures that make it less vulnerable to abuse and attacks. User classes are transported and executed as threads on the fly directly from memory. In addition, threads are lightweight, which means that it is an advantage to run user programs as threads instead of heavy weight processes.

The experiments conducted show that the system performs well. There are numerous opportunities for enhancing the utilization of resources and quality of service for high performance Java through the cooperation and coordination among agents. These could be collaborative resource discovery and recovery modules, collaborative garbage collection, and collaborative scheduling mechanisms for user threads. Additional functions can also be integrated to the infrastructure such as fault tolerance and dynamic load balancing. In addition, we have started to develop a distributed shared Java object space based on this infrastructure.

Acknowledgements

This project was partially supported by a National Science Foundation grant (EPS-0091900) and a Nebraska University Foundation grant (26-0511-0019), for which we are grateful. We would also like to thank other members of the secure distributed information (SDI) group [15] and the research computing facility (RCF) [13] at the University of Nebraska-Lincoln for their continuous help and support.

References