A DISTRIBUTED AND PARALLEL APPLICATION DEVELOPMENT FRAMEWORK FOR A NETWORK OF HETEROGENEOUS WORKSTATIONS

Hamza GÖLYERİ
M.S. in Computer Engineering
Supervisor: Prof. Dr. Müslim BOZYİĞİT
November 2003

Computer Engineering Department
Middle East Technical University
Ankara, TURKEY

Abstract

With the rapid advances in high-speed network technologies and steady decrease in the cost of hardware involved, network of workstation (NOW) environments began to attract attention as competitors against special purpose, tightly coupled, high performance parallel processing environments. NOWs attract attention as parallel and distributed computing environments because they provide high scalability in terms of computation power and memory size and have much smaller cost/performance ratios with high availability. However, they are harder to program for parallel and distributed applications because of the issues involved due to their loosely coupled nature.

Some of the issues to be considered are the heterogeneity in the software and hardware architectures, uncontrolled and uneven external loads, network communication overheads, frequently changing system characteristics like workload on processors and network links, and security of applications and hosts. The general objective of this work is to design and implement a Java-based, high performance and flexible platform i.e. a framework that will facilitate development of wide range of parallel and distributed applications on a network of heterogeneous workstations (NOW). Parallel and distributed application developers will be provided an infrastructure consisting of software developed in Java and a Java application programming interface (API) that will allow them to build and run their distributed applications on their heterogeneous NOW in a load-balanced manner without worrying about the issues specific to the NOW environments.

Table of Contents

1. BACKGROUND AND MOTIVATIONS

Although the capacity and performance of computing systems improve in a high rate the computing requirements of some of scientific and commercial applications is constantly growing. Many fundamental problems called *grand challenge problems* in science and engineering that have broad economic and scientific significance require such massive amounts of computational resources and performance that their solution on single or sequential systems is unacceptably slow. Even, some problems simply cannot be solved sequentially due to the nature of the problem. Some of the problems that require high performance computational resources are related to:

- Earth sciences
  - Numerical weather modeling and forecasting
  - Seismic exploration
  - Oceanography
- Life sciences
  - Cancer research
  - Drug research
  - CAT (Computer aided tomography)
- Engineering applications
  - Finite element analysis
  - Computational aerodynamics
  - Particle physics
- Artificial Intelligence
  - Image and speech processing
  - Computer vision

High Performance Computing (HPC) and *Parallel Computing* deal with the solution of these grand challenge problems. Traditional HPC is based on dedicated architectures called Massively Parallel Processors (MPP) and supercomputers that consist of large numbers of high performance, tightly coupled processing elements. A relatively recent trend is based on connecting low-end individual computing systems (such as workstations or PCs) over high-speed computer networks to form high-end *workstation clusters* and *network parallel computing systems*. These solutions are relatively expensive with high cost/performance ratios and of limited utility as they are designed and dedicated to limited, specialized parallel and distributed applications. Moreover they are available only to limited number of researches due to their cost and physical availability, further limiting their utilization.

Continuous and rapid advances in telecommunication and computer technologies (especially in processing power of microprocessors and in communication network capacity) combined with steady decrease in costs of these technologies made networked computers a commodity. It is observed that for more then a decade the processing capacity of microprocessors is doubled every 18 months, and this trend is expected to continue for at least another decade. At the same time communication network capacity increased exceedingly at the local, metropolitan, national and even global level. All these advances allowed almost all institutions build workspaces in which people use many computers connected to each other with high-speed networks. Also the number of personal computers connected to the worldwide computer network i.e. the Internet grew exponentially and reached hundreds of millions.

On the other hand, studies on these widely available networked computer environments, especially LANs, have shown that for the most of their operation times, the

computers in such environments are used for tasks that are not computationally intensive such as file editing, e-mail reading and Web surfing. In other words, these systems are idle and doing no computation at all for most of their lifetimes. A typical machine on the average has a 90% idle processor time even during peak times of its use.

All the above discussion i.e. existence of grand challenge problems, widespread use of networked computers, and the fact that these systems are idle most of the time motivated many recent researches in development of systems for harnessing the aggregate idle, under-utilized power of well-networked computers to form powerful, parallel and distributed computing systems.

2. RELEATED WORK

Starting at early 90's a large number of researches have attempted to exploit the intrinsic parallelism and latent computation power in distributed system such as heterogeneous LANs and even the Internet for running intensive computations, which were traditionally undertook with specially designed high-performance MPPs and supercomputers. However, these distributed, heterogeneous computing environments are much harder to program and maintain for parallel and distributed applications because of the issues involved due to their loosely coupled nature. Some of the issues to be considered are: the heterogeneity in the software and hardware architectures, uncontrolled dynamic execution environment, less efficient network communication (high latency, low bandwidth), low reliability, priority for workstation ownership, and security and privacy of applications and hosts.

A large number of software infrastructures and libraries were developed that dealt with these issues of loosely coupled network environments to provide services or desirable execution environment properties for facilitating development of distributed and parallel applications on these system.

Earlier works like PVM and MPI tried to solve basic heterogeneity problems and provided a low level, complete set of explicit message passing primitives for communication and coordination of tasks (or distributed processes), and new distributed computing constructs like remote process creation and execution. Some other low-level libraries such as Linda derived systems and Agora [1] provided Distributed Shared Memory (DSM) based communication facilities. Later works were build on these low-level frameworks to provide some other higher level facilities like load monitoring, adaptive load balancing, fault tolerance, check-pointing, and process migration subsystems for creating high performance, easily programmable, reliable frameworks. Although these systems enabled development of distributed and parallel applications on heterogeneous workstation networks, they showed some limitations in getting a flexible parallel programming environment. Due to the low-level programming interface (API) they provide they are relatively difficult to program. They failed in totally shielding the programmer from heterogeneity of the underlying system. For example programmers needed to develop, compile and maintain and distribute different versions of their executable code for each different architecture in their system. Also these systems have high setup, configuration and management overhead. Moreover, these systems are not automatically, transparently and easily scalable i.e. they are usually limited to a single network domain. All in all, these frameworks are not flexible in many respects.

The Java programming language is rapidly being adopted as one of the most important and widely used languages for parallel and distributed application development due to the excellent features it offers that are lacking in traditional languages such as C, C++ and Fortran. Java's attraction is mainly due to its clear and effective solution to the portability and interoperability problem associated with heterogeneous machines and operating systems with

its standardized virtual machine architecture whose implementation is available for almost all systems. A Java program once compiled can be run on any system that has a Java Virtual Machine (JVM) installed. Moreover its features such as automatic garbage collection, rigid security infrastructure, extensive support for network programming, multi-threaded programming, synchronization mechanisms, object serialization, and code mobility makes it an excellent choice for distributed, network-parallel programming. Some other frameworks like Remote Method Invocation (RMI), Jini, and JavaSpaces that are build on standard Java to extend platform's capabilities further simplifies development of large scale distributed and parallel applications.

Recently, significant number of Java-based systems has been developed to support distributed and parallel programming on heterogeneous workstation networks and even on the Internet. SuperWeb [2], Javelin [3], Janet [4], Charlotte [5], KnittingFactory [6], Ajents [7], JavaParty [8], ParaWeb [9], JavaSymphony [10], JPVM [11], JavaNOW [12], JAVM [13], ALTAS [14], and Ninflet [15] are some of the works carried out in the academia with different properties in terms of objectives, architecture, scalability, supported computational model, type of programming model provided etc. In the following we give an overview of the works with significant contribution and that attracted most attention.

2.1. SuperWeb, Javelin, JANET (University of California, Santa Barbara)

Javelin, originally a prototype of SuperWeb, was reported in 1997 as a Java-based infrastructure for global computing. The goal of Javelin is to harness the Internet's vast, growing, computational capacity for ultra-large, coarse-grained parallel applications. The work that was started with SuperWeb has been continued with new versions named Javelin++, Javelin 2.0, Javelin 3.0, CX, JICOS, and currently JANET, each with improvements in performance, scalability, computation and programming model. SuperWeb and the first version of Javelin were designed based on Java applets and web browsers. Starting with Javelin++ the system is based on standalone Java applications instead of applets due to various limitations of applet-based architecture.

Although various improvements made, the essential architecture remained almost same: The whole system is based on three system entities named clients, brokers, and hosts. A *client* is a process seeking computing resources, a *host* is a process offering computing resources, and a *broker* is a process that coordinates the allocation of computing resources. Hosts offer their resources to the world by registering to the brokers. Brokers essentially form a directory of available hosts and they coordinate resource consumption across clients and hosts. As the computational model, Javelin supports *piecework computations* (also called master/slave, manager/worker, or bag-of-tasks): adaptively parallel computations that decompose into a set of sub-computations, each of which is autonomous (does not require communication or coordination with other sub-computations), apart from scheduling work and communicating results. Parallel matrix multiplication, ray tracing, and Monte Carlo simulations are some of good examples of piecework computations. Javelin also supports *branch-and-bound* computations. Javelin achieves scalability and fault-tolerance by its network of brokers architecture and by integrating distributed deterministic *work stealing* with a distributed deterministic *eager scheduling* algorithms.

2.2. Charlotte (New York University)

Charlotte is one of the first Java-based frameworks of its kind. Similar to Javelin, its architecture is mainly based on Java applets embedded into Web pages. It aims to utilize the Web as a parallel meta-computer but it clearly does not scale much and experiences bottlenecks due to the limitations of its applet-based architecture.

Charlotte provides distributed shared memory (DSM) architecture within the language, at the data type level i.e. through classes so it does not modify the Java Virtual Machine (JVM), nor does it rely on a preprocessor or require any kind of external runtime support (ex. OS support). For every basic data type in Java, there is a corresponding Charlotte data type implementing its distributed version. The consistency and coherence of the distributed data is maintained by the Charlotte runtime systems. The provided DSM memory-consistency semantics is Concurrent Read, Concurrent Write Common (CRCW-Common). This means that one or more entities can read a shared variable, and one or more entities can write a variable as long as they write the same value.

Charlotte programs are written for a virtual parallel machine with an unbounded number of processors sharing a common namespace i.e. the programmer has no knowledge of how many machines will execute a computation. The main entities in a Charlotte program are a manager (i.e. a master task) executing serial steps and one or more workers executing parallel steps. The manager process creates an entry in well-known Web page for the active computation and volunteer users load and execute the worker processes as Java applets embedded into web pages by pointing their browsers to this page. In essence the programming model supported by Charlotte is master-slave (master-worker, or bag-of-tasks) programming model. The computation is first divided into a large number of small computational units, or task. Then participating machines pickup and execute a task one at a time until every task has been executed.

Charlotte achieves load balancing and fault masking by implementing two concepts: *eager scheduling* and *two-phase idempotent execution* (TIES). Eager scheduling aggressively assigns and re-assigns tasks until all tasks are completed when the number of remaining task becomes less then the available machines. Concurrent assignment of tasks to multiple machines prevents slow, un-accessible, or faulty machines from slowing down the progress of the computation. Multiple executions of a task (which is possible when using eager scheduling) can result in incorrect program state. TIES ensures idempotent memory semantics in the presence of multiple executions. TIES guarantees correct execution of shared memory by reading data from the master and writing it locally in the workers' memory space. Upon completion of a worker the dirty data is written back to the master who invalidates all successive writes, thus maintaining only one copy of the resulting data. Moreover Charlotte employs *dynamic granularity management* (bunching) to mask latencies associated with the process of assigning tasks to machines. Bunching is achieved by assigning a set of task to a single machine at once. The size of bunches is computed dynamically based on the number of remaining tasks and number of available machines.

Charlotte has a number of problems. The primary function of the manager is scheduling and distributing works. But it is also responsible for communication of workers (i.e. it implements the DSM) as all applet-to-applet communication is routed through it. So it is a potential bottleneck. Secondly, Charlotte requires that a manager run on a host with an HTTP server. If only one machine with an HHTP server is available and if more than one application needs to be run in the system, multiple managers are needed to run on this machine, making it a possible communication bottleneck. Moreover multiple assignment and execution of the same task with eager scheduling might cause excessive data traffic further leading to performance problems. All in all, Charlotte does not seem to scale enough to meet its goal of utilizing the Web as a big meta-computing platform.

2.3. KnittingFactory (New York University)

KnittingFactory is the successor Charlotte from the same research group. With KnittingFactory some of the limitations and deficiencies of Charlotte is eliminated. KnittingFactory addresses the problems:

- Searching and finding other members of a collaboration session.
- Having to run distributed application on a machine also running a HTTP server.
- Direct communication between applets.

A *distributed registry* is implemented based on Web servers and standard Web browsers as search-engines. A registry accepts requests for partners, stores these requests, and deletes then again upon request. A user who wants to participate in a distributed computation simply points his/her browser to one of these registries. Charlotte required that master processes of applications run on machine with a running HTTP server. KnittingFactory factory solves this problem by automatically embedding a HTTP server in each application. So now applications can run on any machine. In Charlotte, all the communication between applets is routed through the master process. KnittingFactory supports direct communications between worker processes (applets) by exploits a non-standard (maybe a bug) of Sun JVM to pass RMI object references between applets.

2.4. Ajents (York University, University of Waterloo)

Ajents is a Java based framework that provides necessary infrastructure for building parallel and distributed applications. Unlike many other similar frameworks Ajents does not make modifications to the Java language or to the JVM and no preprocessors, special compilers, or special stub compilers are required. It is a collection of pure Java classes (a library) and servers (also implemented in Java) so it runs on any standard compliant JVM. Every host in the system runs a simple lightweight server called *Ajents Server* as a daemon process.

Ajents greatly simplifies the task of writing distributed application by providing features that are not available in standard Java. Actually it is build on standard Java RMI technology but extends the distributed object model provided RMI by providing support for:

- **Remote Object Creation:** While RMI allows local referencing and remote method invocation on statically created remote object, Ajents supports dynamic creation and referencing of object on remote hosts that are running Ajents Server.
- **Remote Class Loading:** To be able to instantiate objects on remote hosts or migrate objects to remote hosts, Ajents transparently loads the binary executable code of the object to the remote host.
- **Asynchronous Remote Method Invocation:** Java RMI only allows invocation of methods on remote objects synchronously (or serially). Ajents supports asynchronous RMI i.e. a process can call a method of a remote object and continue execution until an arbitrary time when the result of the method call is available.
- **Object Migration:** Ajents allows migration of objects between heterogeneous hosts without a preprocessor, and without modification to the virtual machine, compiler or stub compiler. This is implemented using check-pointing, rollback and restarting mechanisms. Any object can be migrated while it is executing by interrupting its execution, moving the most recent check-pointed state of the object and restarting the currently executing method.

The main idea for programming in Ajents environment is to have Ajents Server running on every host in a heterogeneous environment and writing applications in Java using the Ajents class library calls to use resources on these servers by creating remote objects, invoking their methods, and migrating them between host as necessary.

Ajents framework does not support transparent dynamic load balancing but of course users can implement custom load balancing on top of Ajents framework. Although not clearly explained, it has some scheduling mechanism that allows selection of an appropriate host to migrate objects.

Unlike Javelin, Charlotte and KnittingFactory type of frameworks, Ajents does not dictate any high level distributed or parallel computation model. Instead, it provides relatively low level and complete, easy to use framework for building any kind of distributed applications or even higher-level frameworks like Javelin on top of Ajents. In another view, it's simply an extension of distributed object model of RMI technology with new distributed programming facilities.

2.5. JavaParty (University of Karlsruhe, Germany)

Although Java platform includes RMI technology for distributed programming it is not easy and straightforward to write distributed and parallel applications for distributed shared memory architectures like clusters of workstations. JavaParty transparently adds distributed, remote objects to Java simply by declaration, avoiding complexity, disadvantage and programming overhead of socket based, or RMI based programming. JavaParty is targeted towards and implemented on clusters of workstations. It extends the Java language simply and transparently with a pre-processor and a runtime system for distributed parallel programming in heterogeneous workstation clusters.

JavaParty adds the *remote* keyword to the Java language. The programmer simply attributes classes that should be spread across the distributed environment with the *remote* keyword. JavaParty uses a preprocessor, which converts *remote* classes into pure Java code with RMI hooks. The change to the language is designed to simplify RMI programming, placing the burden of creating and handling remote proxies upon the preprocessor, simplifying the programming task.

Objects of these *remote* classes can transparently be created on remote hosts and referenced locally as if they are local objects. JavaParty is location transparent i.e. it maps created remote objects to hosts transparently. The compiler and runtime system deals with locality and communication optimization using pluggable distribution strategies. The programmer can only influence distribution and mapping of object to hosts by developing and inserting code that directs the strategy's placement decisions. Besides distribution strategies at object creation time, JavaParty monitors the interaction of remote objects and if it is appropriate (or the programmer) it can schedule object migration between hosts to enhance locality.

2.6. JavaSymphony (University of Vienna)

Basically JavaSymphony extends the distributed object frameworks provided by Java RMI technology with new high level construct, similar to Ajents and JavaParty.

High-level distributed and parallel programming frameworks that does not provide programmer control over locality of data by automatically distributing and migrating objects can easily lead to performance loose as the underlying runtime system has little information about the distributed computation. In contrast to most existing systems, JavaParty provides

the programmer with the flexibility to control data locality and load balancing by explicitly mapping of object to computing nodes.

The key features of JavaSymphony that greatly simplifies performance-oriented distributed and parallel programming are:

- **Dynamic Virtual Distributed Architectures:** The programmer can dynamically define and modify virtual distributed architectures that impose a virtual hierarchy on distributed system of physical computing nodes. Virtual architectures consist of a set of components: computing node, clusters (collection of nodes), sites (collection of clusters), and domains (collection of sites). Virtual architectures effect how automatic mapping and migration of objects are done for locality and load balancing decisions.
- **Access to system parameters:** JavaSymphony provides access to a large variety of periodically monitored system parameters such as CPU load, idle times, available memory, network latency and bandwidth, etc. Programmer can use these parameters for load balancing decisions.
- **Automatic and User-Controlled Mapping of Objects:** The programmer can control the creation and mapping of objects to specific components of the virtual architectures. If the programmer does not provide explicit mapping of objects the JavaSymphony runtime offers automatic mapping based on periodically monitored systems parameters.
- **Automatic and User-Controlled Object Migration:** JavaSymphony supports both automatics and user-controlled migration of objects based on periodically monitored systems parameters.
- **Asynchronous, Remote, and One-sided Method Invocation:** JavaSymphony supports both synchronous and asynchronous remote method invocation. Moreover for methods that do not return any value it supports one-sided method invocation.
- **Selective Remote Class Loading:** JavaSymphony automatically loads binary class codes only to the nodes on which they are actually needed. This feature can reduce overall memory requirements of an application.

JavaSymphony has been influenced by Ajents programming model for remote object creation, asynchronous remote method invocation and remote class loading. Its most significant contributions over Ajents are its support for virtual architectures, one-sided method invocations and access to system parameters.

2.7. JavaNOW (DePaul University)

JavaNOW aims to provide an environment for parallel computing on an ordinary network of workstations that is both expressive and reliable. It creates a virtual parallel machine similar to the Message Passing Interface (MPI) model, and provides distributed shared memory (DSM) similar to Linda memory model but with a flexible set of primitive operations. It can be view as a hybrid system of PVM, MPI, and Linda. It provides a simple mechanism to start tasks on remote hosts (as found in PVM), has a small number of expressive and complete primitives to support produces/consumer style communication (as found in Linda) and finally has collective operation that can be performed on shared objects (as found in MPI).

In a JavaNOW system, processes coordinate and communicate through a distributed associative shared memory similar to tuple-space model and using PVM and MPI like primitives that are build solely on this share memory model. The Linda like DSM is

implemented as a totally distributed and load balanced data structure (actually a distributed hashtable). So it differs from MPI and PVM in that it does not provide direct point-to-point IPC primitives but rather provides a producer/consumer model of IPC. Mutually exclusive access operations to the shared memory make it easy to implement distributed synchronization primitives like locks, mutexes, and semaphores.

JavaNOW does not provide a dynamic resource management but requires that the user statically specify the list of machines on which the application will run. Also it does not provide dynamic load-balancing at work distribution. It just uses a simple hashing scheme for load balancing the distributed shared memory.

3. WORK OBJECTIVES

Main objective of this work is to design and implement a framework (software libraries and supporting infrastructure) that is easy to use (in terms of programming, configuration and management) for building distributed and parallel applications on a network of heterogeneous workstations. The key features of the framework will be:

- **Support for Heterogeneity:** The framework should have uniform support for different software (OS) and hardware (processor, machine, network) architectures. Users of the framework should be totally abstracted from all kinds of interoperability problems associated with heterogeneous systems and be able to develop and run their applications without needing to deal with any heterogeneity issue.
- **Virtualization:** The framework should provide the programmers and applications a single virtual parallel machine view of the whole network with high levels of location transparency, fault tolerance, reliability, and robustness.
- **Adaptability:** The runtime systems should easily, transparently and automatically adapt to the dynamics of a general workstation network. The participating machines should be able to easily and transparently join and leave the system at any time for any reason including machine and network failures or due workstation owner preferences. The framework should shield programmers and applications from dynamically changing properties of the system; for example the programmer should not deal with machine failures.
- **Transparent Load-Balancing:** The framework should provide pluggable and configurable (possibly dynamic) load balancing infrastructure transparent to the programmer and applications.
- **Accessibility:** The services of the framework and runtime system should be easily accessible by users from anywhere on the network. For example, the programmers can be able to run their applications on the systems from any machine on the network.
- **Minimized Overhead:** The framework should keep the overhead minimum associated with installation, configuration, and management of the software infrastructure. For example new machines should be easily and transparently added to a running system.
- **Security and Privacy:** Security and privacy concerns of users and especially workstation owners should be answered with a sound security infrastructure. Access to resources should be limited by configurable security policies.
- **Highly Configurable:** The owners of workstations that participate in the system should be able to easily make configurations that effect how and when their workstations used by the system.

The framework will be made up of two parts: a high level API that allows programmers to explicitly express their parallel and distributed applications as a set of independent tasks to be executed in parallel, and a set of software (daemons processes on workstations, brokerage services etc.) components. The primary computation model will be piecework computations (master-worker, or bag-of-tasks) in which the computation is explicitly divided into many sub-computations that can be performed in parallel at different nodes of the system. Scheduling and load-balancing will be performed by the runtime automatically and transparent to the application. In fact, programmer will not be able to effect how sub-computations are mapped to the processing nodes because programs will be written for a virtual parallel machine with an unbounded, unknown and dynamically changing number of processing nodes.

4. REFERENCES

[1] R. Bisani and A. Forin, "*Multilanguage Parallel Programming of Heterogeneous Machines*", IEEE Trans. Computers, Vol. 37, No. 8: pp. 930-945, Aug 1988.

[2] A. Alexandrov, M. Ibel, K. Schauser, and C. Scheiman, "*Super Web: Towards a Global Web-based Parallel Computing Infrastructure*", In 11th International Parallel Processing Symposium, pp. 100-106, April 1997.

[3] B. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K. Schanuser and D. Wu. "*Javelin: Internet Based Parallel Computing Using Java*", Concurrency: Practice and Experience, Vol. 9, No. 11, pp. 1139-1160, Nov. 1997.

[4] P. Cappello, "*Janet's Abstract Distributed Service Component*", Proc. 15th IASTED Int. Conf. on Parallel and Distributed Computing and Systems, pages 751 - 756, Marina del Rey, California, Nov. 2003.

[5] A Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, "*Charlotte: Metacomputing on the Web*", In Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems, 1996.

[6] A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem, "*KnittingFactory: An Infrastructure for Network Computing with Java Applets*", In ACM 1998 Java Grande Conference, 1999.

[7] M. Izzat, Patrick Chan and Tim Brecht. "*Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications*", In ACM 1999 Java Grande Conference, 1999.

[8] M. Phillippsen and M. Zenger. "*JavaParty: Transparent Remote Objects in Java*", In ACM 1997 Workshop on Java for Science and Engineering Computation, June 1997.

[9] T. Brecht, H. Sandhu, J. Talbot, and M. Shan. "*ParaWeb: Towards Worldwide Supercomputing*", In European Symposium on Operating System Principles, October 1996.

[10] T. Fahringer. "*JavaSymphony: A system for development of locality-oriented distributed and parallel Java applications*" In Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000), 2000.

[11] A. J. Ferrari, "*JPVM: Network Parallel Computing in Java*", Technical Report CS-97-29, Dept. of Comp. Sci, University of Virginia, Charlottesville, VA 22903, USA.

[12] G. K. Thiruvathukal, P. M. Dickens and S. Bhatti, "*Java on Networks of Workstations (JavaNOW): A Parallel Computing Framework Inspired by Linda and the Message Passing Interface (MPI)*",

[13] L. F. Lau, A. L. Ananda, G. Tan, W. F. Wong, "*JAVM: Internet-based Parallel Computing Using Java*" School of Computing, National University of Singapore

[14] 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, 1996.

[15] H. Takagi, S. Matsuoka, H. Nakada, S. Sekiguchi, M. Satoh, and U. Nagashima. "*Ninflet: A Migratable Parallel Objects Framework using Java*", in Proc. of ACM 1998 Workshop on Java for High-Performance Network Computing Feb./Mar. 1998.