A Mobile Agent Based Push Methodology for Global Parallel Computing

Cheng-Zhong Xu and Brian Wims

Department of Electrical and Computer Engineering
Wayne State University, Detroit,
MI 48202, USA
czxu@ece.eng.wayne.edu

Abstract

The 1990s are seeing the explosive growth of the Internet and Web-based information sharing and dissemination systems. The Internet is also showing a potential of forming of a supercomputing resource out of networked computers. Parallel computing on the Internet often works in a machine-centric "pull" execution model. That is, a coordinator machine maintains a pool of tasks and distributes the tasks to other participants on demands. This paper proposes a novel mobile agent based "push" methodology from the perspective of applications. In the method, users declare their computation-bound jobs as autonomous agents. The computational agents will roam on the Internet to find servers to run. Since the agents can be programmed to satisfy their goals, even if they move and lose contact with their creators, they can survive intermittent or unreliable network connection. During their lifetime, the agents can also move themselves autonomously from one machine to another for load balancing, enhancing data locality, and tolerating faults.

We present an agent-oriented programming and resource brokerage infrastructure, TRAVELER, in support of global parallel computing. The TRAVELER provides a mechanism for clients to wrap their parallel applications as mobile agents. The agents are dispatched to a resource broker. The broker forms a parallel virtual machine atop available servers to execute the agents. TRAVELER relies on an integrated distributed shared array (DSA) runtime system to support inter-agent communication and synchronization on clusters of servers. We demonstrate the feasibility of the TRAVELER in parallel sorting and LU factorization problems.

Keywords: Distributed shared array, global computing, mobile agents, parallel computing, push technology, Traveler,.

1 Introduction

The 1990s are seeing the explosive growth of Internet and Web-based information sharing and dissemination systems. The Internet is also showing a potential of forming a supercomputing resource out of networked computers. This potential was recently demonstrated in parallel applications like cracking messages encoded by 56-bit DES algorithms in 1998 [9] and finding the first million digit prime in 1999 [29]. The applications are mostly run in a brute-force parallel search model. That is, one machine maintains a pool of parallel tasks and dispatches these tasks to other participants on demand. It is essentially an on-demand "pull" execution model. There are Web-based metacomputing infrastructures, such as Charlotte of NYU [1], Javelin of UCSB [8], and Bayanihan of MIT [30], which advance the "pull" execution model by taking advantage of the latest Web technologies. The infrastructures allow users to define tasks as Java applets and post them in WWW homepages. Prospective servers shall access the homepages and download the task applets for execution in their secure browsing environments. Relying on voluntary participants, the "pull" execution model works well for applications that are of common interest to the Internet community. However, it cannot provide any guarantee of the service quality from the perspective of end users. To harness the computational power of the Internet for general applications, a well defined global computing infrastructure is necessary to facilitate resource registrations and utilization.

The idea of harnessing computational power of networked computers is not new. It has long been an active area of research. Job scheduling systems have covered a wide range of needs, from traditional batch job queuing, to load sharing, and cycle stealing; see [18] for an excellent review of leading packages. There are also parallel programming environments that provide task scheduling, load balancing, and even fault tolerant services in support of parallel applications on clusters [3][25]. Internet-based global computing is a natural extension of LAN-based cluster computing. Its objective is to seamlessly integrate networked computers to form a computational grid so as to provide dependable, consistent, pervasive, and inexpensive access to high-end computational capacities on the Internet [10]. Unlike cluster computing where users have access, in a dedicated or multi-programmed mode, to all cluster-wide resources and are able to perform privileged operations, global computing shall assume clients run their codes on virtually any machines (servers) and prepare servers to execute programs from anonymous users. The Internet is characteristic of unreliable connection and unpredictable traffic on link. Global computing infrastructures shall also provide programs a reliable and adaptive execution environment on the Internet.

Anonymous accessibility raises many concerns in the construction of a wide area parallel computing infrastructure. Of the foremost are security and interoperability. Since machines in different administrative domains do not necessarily trust one another, the infrastructure needs to protect servers against potentially hostile actions of client codes under execution and the client codes against tampering attempts by the executing server [7]. Although few solutions are able to protect computation integrity and privacy of the client codes from servers, recent advent of secure languages like Java ensures the server security by restricting the alien codes to be executed in a secure sandbox. Due to its maturing security framework and strong "write-once-run-everywhere" commitment, Java is widely recognized as one of the major programming tools for global computing on the Internet. There is also increasing effort to advance Java for high performance computing; see [13] for the latest workshop devoted to this topic.

Java-based global parallel computing relies on an easy-to-use programming paradigm and a high performance execution model that can adapt to the uncertainty of the Internet. In this paper, we propose a novel mobile agent "push" methodology for high performance computing on the Internet. An agent is a special object type that has autonomy. Mobile agents extend the model of Java applet-like mobile codes. Like an applet, the code for an agent can migrate across a network. But a mobile agent can also carries its state when it migrates. An applet tends to move from a server to a client on demand. The "push" method allows users to dispatch their jobs as agents and enable the agents to roam on the Internet to find servers to run. It reverses the logic of task distribution in the "pull" execution model and provides a way of high performance computing from the perspective of applications. Due to the autonomy and mobility of the agents, the "push" methodology has the following characteristics.

- *Survival of intermittent or unreliable Internet connections*. Traditional distributed applications rely on reliable network connections throughout their lifetime. If the connection goes down, the client often have to re-start the application from the beginning. Since an agent can be programmed to satisfy one or more goals, even if the object moves and loses contact with its creator, the infrastructure will allow clients to dispatch computational agents into the Internet and then go off-line. The agent will re-establish the connection to its originator and present results back when it finishes its assigned task. Survival of intermittent connections is especially desirable for long-lasting compute-bound agents.
- *Adaptive and fault tolerant execution model*. Traditional client/server applications need to specify the roles of the client and the server very precisely, based on some predicted network traffic information, at their design time. Due to the instability of the Internet, performance of the applications often fluctuates unpredictably. The proposed infrastructure will allow computational agents to move themselves autonomously from one machine to another to harness the idle CPU cycles, balance the workload of machines, and enhance data locality. Persistent state associated with a mobile agent will also help recover computational tasks from their failures.

We present an agent-oriented resource brokerage infrastructure, TRAVELER, in support of wider-area parallel computing. The infrastructure provides an agent wrapper to simplify agent-oriented programming. It allows clients to dispatch their computational agents to a resource broker. The broker collects workload information of registered servers via its own agent. Based on the workload distribution information, the broker forms a virtual machine over available servers to execute the computational agent. Due to the commonplace of symmetric multiprocessor (SMP) servers and the increasing popularity of SMP clusters, the TRAVELER supports multithreaded agents for high performance computing on clusters of servers. Agents are cloned on each server and run in single-program-multiple-data (SPMD) paradigm. Inter-agent communication and synchronization are supported by an integrated distributed shared array (DSA) run-time support system. This paper demonstrates the feasibility of the TRAVELER in two applications: parallel sorting and LU factorization on an ATM-connected heterogeneous cluster of servers. Clients are run in a remote local area network. Although the current prototype has not been deliberately refined for performance, benefits from parallel computing have been observed in both applications. Preliminary results were reported in [38][39].

The rest of the paper is organized as follows. Section 2 discusses related work with an emphasis on the distinctive features of the TRAVELER. Section 3 presents the TRAVELER's architecture of mobile agents and agent-oriented programming environments. Section 4 presents the DSA run-time support for dynamic virtual machines on clusters of servers. Programming interfaces to the TRAVELER are presented in Section 5. Section 6 contains the experimental results from a comprehensive evaluation of the TRAVELER. The paper is concluded in Section 7 with remarks on future work.

2 Background and Related Work

Global computing is a natural extension of cluster computing. The later is rooted in systems as old as the VAX VMS cluster [20]. Traditionally, clusters have been used to provide fault tolerant services for high availability and to serve multiprogramming workloads for high throughput computing. Availability clusters, such as Microsoft SQL Server/ClusterServer [23] and Tandem NonStop Server Cluster [35], utilize mirroring techniques to allow a service on one machine to “fail over” to another machine and continue operation upon occurrence of a hardware or operating system failure. High throughput clusters rely on job queuing and scheduling services to harness computing power of lightly loaded computers. Job queuing or scheduling systems like Condor [23] has covered a wide range of needs, from traditional batch queuing to load sharing and cycle stealing. Recent research efforts are to seamlessly integrate networked computers together to form a computational grid and to provide dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities on the Internet [10][32].

2.1 Wide Area Parallel Computing Infrastructures

A number of recent research projects were dedicated to providing programming models and a range of services that programmers can call upon when developing wide area applications [10]. Legion [16] provided a single coherent virtual machine by addressing key issues like security, scalability, programmability, fault tolerance, and site autonomy within a reflective object-based meta-system. It was implemented in Mentat programming language (MPL) and based on a customized security model. It supported parallel applications in MPL and executes MPI and PVM programs via simulation. By contrast, the infrastructure we propose will be based on the platform-neutral Java programming environment and on a de facto industry standard Java security model. It will also support multithreaded applications on clusters of symmetric multiprocessors.

Globus toolkit [11] provided a “bag of services”, including multi-party security, resource location, resource management (GRAM), and multiprotocol communication (Nexus) [12], while leaving programming models to be addressed by application developers. These services complement to the agent-oriented programming model. As a matter of fact, GRAM can be integrated into the infrastructure due to its standard interface to existing Condor- and LSF-like cluster-wide resource management systems.

The infrastructure we propose shares a need for similar global parallel computing infrastructure with “pull” model based systems. In Charlotte [1], when a program reaches a parallel step, it registers itself with specific daemon process, which creates an applet for each parallel task and maintains a list of the applets in a URL homepage. Any servers can visit this homepage using browsers and download applets for execution if they wish to donate some of its cycles. Since parallel tasks are declared by clients, it is the computational server that initiates parallel computation on demands. The machine-centric model provides no guarantee of service qualities from the viewpoint of applications.

In Javelin[8], servers register their CPU by visiting a broker web page and download a Java applet that keeps in contact with the broker. Clients submit tasks by visiting another Web page and submitting a form to request resources. As clients request resources, the broker maps these to the next available server, using a simple round-robin algorithm. The broker does not ship actual client tasks, but rather sends a URL pointer to the tasks. The servers then directly retrieve the chosen task from the client. As in Charlotte, tasks are declared as applets and maintained in an HTTP server accessible to the computing servers.

A defining characteristic of “pull” execution model is passive. Tasks declared as Java applets are just waiting for visits from voluntary servers. The “pull’’ model was also referred to as volunteer computing in Bayanihan [30]. It is similar to the idea of global work stealing in Atlas [1]. Atlas provided a framework for idle servers to steal threads from those that are busy. Unlike other Web-based global computing infrastructure, any Atlas machine can be either a server or a client and clients are closely coupled with servers. The passive “pull” or volunteer execution model was demonstrated to be simple and effective for applications that were of common interest to a group of users [9][29]. However, it cannot provide any guarantee of the service quality from the perspective of applications. By contrast, the proposed agent-based approach will activate the tasks and enable them to locate execution resources.

There were other types of Internet-based metacomputing infrastructures. NetSolve [6] and Ninf [31] assume that programs persistently reside on servers. Users request services by sending data to the server, which executes the code correspondingly and sends results back to the users. Such network-enabled solvers allow users to invoke advanced numerical solution methods without having to install sophisticated software. In [6], the authors categorized client/server architectures into three classes: proxy computing, code shipping, and remote computing. Proxy computing require clients to send both the code and data to servers; code shipping stores the code on a server and requires clients to download the code on demand; remote computing allows clients to invoke remote services by sending data to the server. In this classification, NetSolve and Ninf belong to remote computing and Java applet-oriented systems like Javelin, Charlotte, and Bayanihan are of code shipping. The mobile agent approach goes beyond the way of proxy computing because the running state is to be migrated, together with the code and data.

All these infrastructures were intended to provide user-level services for building and running wide-area applications. WebOS[36] represented an effort to offer system level services for simplifying the development of wide-area applications and improving the utilization efficiency of global resources.

2.2 Mobile Agent and Push Technologies

The word "agent", or software agent, has found its way into a number of technologies. It has been applied to artificial intelligence, information gathering and services on the Internet, computer supported coordinated work, etc[17]. Although there is no single definition of an agent, all definitions agree that an agent is essentially a special software component (object) type that has autonomy. It behaves like a human agent, working for some clients in pursuit of its own agenda. Mobile agents have as their defining trait the ability to travel from machine to machine. When traveling, the software agent packages its code, data, and its running state and moves to a new site. Once arriving at a new site, the agent continues executing its code from where it left off.

Mobile agents grew out earlier technologies of mobile codes and remote evaluation. Code mobility has long been an active research topic in distributed systems; see [14] for a classification and comprehensive review. Process or fine-grained thread migration concerns the transfer of user-level processes or threads across different address spaces for load balancing, fault masking, and improving data locality. It is the autonomy of the agents that makes high performance metacomputing different from cluster-wide process/thread migrations. Mobile agents extend the model of Java applet-like remote evaluation. Like an applet, the code for an agent can migrate across a network. But a mobile agent can also carries its state when it migrates. An applet tends to move from a server to a client on demand. An agent can be pushed from one machine to another on a network. This provides the agents the ability to travel and gather information at different sites, and negotiate with other agents on behalf of their clients [19]. This paper intends to employ mobile agent technologies in a non-traditional way. Within the infrastructure, computational tasks are defined as agents. Instead of gathering data or servicing machines, they roam in the network from one machine to the other to find appropriate machines to run.

Until recently, mobile agent systems were developed primarily based on research languages like Tcl, Scheme, and Telescript. Current explosion of interest in mobile agent systems is due almost entirely to the widespread adoption of Java. The Java virtual machine and Java's class loading model, coupled with several of Java other features---most importantly serialization, remote method invocation, multithreading, and reflection---have made building first-pass mobile agent systems a fairly simple task. Over the past years, over more than a dozen of Java-based agent frameworks (e.g. Voyage [28], Aglet [21], and Odyssey [15]) were announced for developers to choose from; see [40] for an excellent review of Java-based mobile agent frameworks. They were pure Java implementations and based on Java remote method invocations. While they have proven effectiveness in support of information retrieval, collaboration, and negotiation agents, none of them supports multithreaded mobile agents for high performance computing on the Internet.

Finally, we note that the "push" model, in contrast to "pull" (i.e. download or browsing), mostly referred to as push publishing in Web technology [33]. The pushing server is essentially a CGI program. It enables content providers to create channels and associate them with particular web pages on a site. Channel subscribers will be notified of any content change with the related pages. The agent based push execution model goes beyond the conventional push technologies by allowing users to distribute their codes to servers for remote execution.

3 Architecture of The TRAVELER

The TRAVELER relies on mobile agent technologies to realize ubiquitous global computing. Its implementation is based on Java's remote method invocation (RMI) and object serialization. The RMI system allows remote-procedure-call (RPC) like access to remote objects and supports mobile behaviors [34].

Like traditional RPC, RMI is enabled by declaring a remote interface of an object to expose its methods to remote objects. The following AgentServer interface enables clients to create mobile agents and dispatch them to a server that implements the interface.

```
import  java.rmi.*
public  interface AgentServer extends Remote {
  Object execute( Agent agent ) throws RemoteException;
}
```

A basic implementation of the AgentServer is as follows. The execute method takes the agent object and starts the execution of the object. RMI provides for secure channels including encrypted sockets between client and server. It also uses built-in Java security mechanisms to protect servers from possible attacks by malicious clients. It is realized by installing a security manager before exporting any server object or invoking any method on a server. RMI provides a RMISecurityManager type that is as restrictive as those used for applets. Each server can also define and install its own security manager object to enforce different security constraints. For example, a server can open a tmp directory for an agent to store intermediate results. The server should also allow alien agents to open connections to the broker and the servers of the same virtual machine

```
import  java.rmi.*;
import  java.rmi.server.*;
public class AgentServerImpl implements AgentServer {
  public AgentServerImpl() throws RemoteException{}
  public Object execute( Agent agent) throws RemoteException {
    return agent.execute();
  }
  public static void main( String args[] ) {
    System.setSecurityManager( new RMISecurityManager() );
    try {
      AgentServer as = new AgentServerImpl();
      Naming.rebind( "AgentServer", as );
    } catch ( IOException ){};
  }
}
```

3.1 The Broker Architecture

The TRAVELER, as shown in Figure 1, is essentially an agent oriented broker system. The broker executes trades between clients and servers and forms a parallel virtual machine out of the available servers upon receiving an agent task. The agent is then cloned for each server. The cloned agents are run in a single-program-multiple-data paradigm. They are executed on the virtual machine independently of the broker. Servers of the machine can report results to the broker or directly to clients. Notice that the system may comprise of more than one broker. Each broker serves regional clients and servers or nation wide domain-specific clients and servers. Brokers are organized in a hierarchical way for a wide area computational grid.

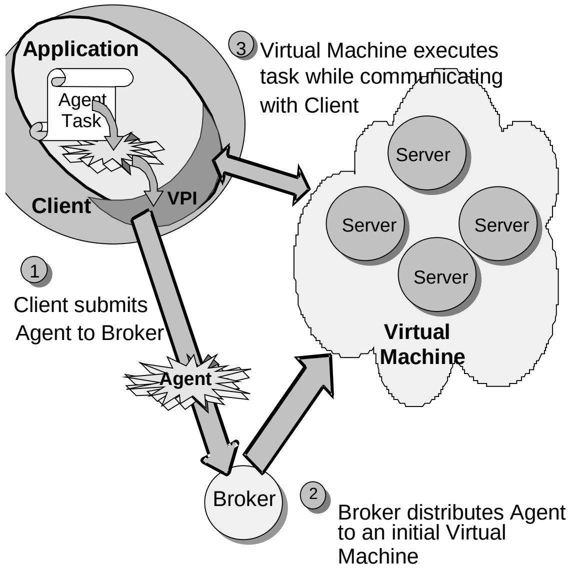
Application
Agent Task
VPI
Client
3 Virtual Machine executes task while communicating with Client
Server
Server
Server
Server
Virtual Machine
1 Client submits Agent to Broker
Agent
Broker
2 Broker distributes Agent to an initial Virtual Machine

Figure 1. Architecture of Traveler

Specifically, a client defines a computational task as an AgentTask and meanwhile creates a Virtual Processor Interface (VPI) for communication between the client, the broker and servers. The VPI creates a ParallelAgent to wrap the AgentTask object. The VPI then sends out the ParallelAgent object (or a reference to the object) to a broker via RMI. The broker collects states of the registered servers and forms a virtual machine out of the servers for the ParallelAgent object. On each server, the task agent spawns threads for multiprocessing. A monitoring agent can be created to oversee the execution of the code on the virtual machine. Multithreaded agents are run on the virtual machine, supported by an integrated distributed shared array (DSA) run-time support system. Agents communicate to one another via accessing user-defined shared regions of distributed arrays. Servers can contact clients for input data or returning results through callback handlers carried with the task agent. The virtual machine is finally destroyed upon completion of the computation.

The broker constructs a virtual machine based on the workload information of registered servers. The information can be either polled by the broker or reported by the servers. Information polling is realized by a normal *information-collection agent*. It is dispatched by the broker and is migrated from server to server to collect the workload information of servers periodically. Since there is no agreement about a single metric for server workload in the literature, agent-based information collector provides the broker a way to define customized services (workload indices) from the servers.

3.2 Client Architecture: Virtual Processor Interface

The client architecture provides user task agents with a perception of running on a machine with an unlimited number of processors. It is accomplished via a Virtual Processor Interface (VPI) class. Each VPI object provides a communication channel between user task agents, brokers, and parallel virtual machines.

The Agent interface and ParallelAgent Class

User task agents are wrapped by a ParallelAgent object and dispatched via a VPI. When the object arrives at a server, it spawns user task agents to each available CPU. ParallelAgent objects implement a serializable Agent interface. A serializable object means its internal state, excluding stack, can be sent as a message to another Java virtual machine and be reconstructured at the remote JVM. The Agent interface declares three primary abstract methods initialize(), execute(), and terminate(). The initialize method creates an Agent object. The execute() starts the execution of an Agent object when it reaches a server. The Agent object spawns threads to the available CPUs on the server. The terminate method stops the execution of the Agent object for either transfer, storage, or termination. It suspends each of its threads on the server.

```
public interface Agent implements Serializable {
  abstract public initialize();
  abstract public execute();
  abstract public terminate();
}
```

Virtual Processor Interface

A VPI object, extending from UnicastRemoteObject, provides methods for synchronous or asynchronous communication between AgentTask objects, the broker, and servers. The VPI is run at each client, through which agents are dispatched.

```
public class VPI() extends UnicastRemoteObject implements Listener{
  public VPI() {...}
  public VirtualMachine sendAgent(ParallelAgent pa) {...}
  public SharedArray lookUpArray(String name){...}
  public boolean complete(){...}
  public void waitUntilComplete(){...}
  public String ps() {...}
  public int kill(int brokerAgent){}
  public synchronized void endAgent(Object o) throws RemoteException{}
}
```

The sendAgent method packages a new ParallelAgent object and submits it to the broker. The object contains a callback reference of a receiving server to contact its matching sender. When the broker successfully creates a virtual machine, it returns to the VPI object the location of the Virtual Machine. With this information, the user task agent can interact with shared data and monitor the status of the operation, if needed. Method lookUpArray returns a reference to a SharedArray object in the virtual machine, as shown in the subsequent section. Methods complete and ps asynchronously check the status of a running task and kill allows a client to kill its virtual machine. Method waitUntilComplete will suspend the client thread itself until the virtual machine completes the task. The VPI object also provides a remote method endAgent so that a virtual machine can report results and completion status.

4 Dynamic Virtual Machine on Clusters of Servers

Throughout the lifetime of an agent, availability of the computational resources of its servers may change with time. Servers may also stop their services due to some unexpected events. In both scenarios, the TRAVELER’s virtual machine must be reconfigured to adapt to the change of resource supplies. It is a self-adaptive process. The virtual machine decides whether or not to change its configuration based on the availability of resources from the broker. The broker should be informed of the new occupancy status once the virtual machine has finished its reconfiguration.

4.1 Strong Mobility of Multithreaded Agents

In theory, a software agent should be able to migrate with all its state: heap, execution stack, and registers. Since Java virtual machine does not allow a program to directly access and manipulate execution stacks for security reasons, the execution state and program counters of the threads are not serializable. In other words, Java-based agents are weak in mobility[14]. They are unable to carry the state of their execution stacks with threads as they migrate. On arrival at a new site, each thread will start over at the beginning of its `run()`[1] method. It is because of this reason that none of the Java-based agent systems we reviewed in Section 3.2 has support for strong mobility.

While strong mobility for arbitrary Java-based agents is hardly viable without changing the Java Virtual Machine, we propose an application-level transformation mechanism in support of multithreaded mobile agents. We introduce a `onMove()` method in the `AgentTask` base class to record the execution state in a number of Agent’s instance variables. The instance variables can be serialized and migrated with the agent. When `run()` is invoked to start the agent’s new life, the `run()` method would check the instance variables to determine where to resume. The `onMove()` method will be called when there is an impending serialization. For example, in a multithreaded LU factorization code using the row-wise block decomposition approach, as shown below, only a single instance variable is needed to keep the current pivot row if we assume no migration is allowed within an iteration sweep.

```
Public class luDemo extends AgentTask Implements Runnable
int[1] curPivotRow;  // curPivotRow is to be stored in heap
double[N][N] A;
private void lu() {
  for (int k=curPivotRow[0]; k<N; k++)
                  // Migration occurs between iteration sweeps
    for (int j=k+1; j<N; j++) A[k][j] = A[k][j]/A[k][k];
    for (int i=k+1; i<N; i++)
      for (int j=k+1; j<N; j++)
        A[i][j] = A[i][j] - A[i][k]*A[k][j];
    onMove();             // restore the value of k into curPivotRow
}
public void run() { lu(); }
```

Due to the complexity in the migration of Java-based multithreaded agents, in general, we gear reconfigured virtual machines toward to a popular bulk synchronous SPMD computational model. Such a computation proceeds in phases. During each phase, agents perform calculations independently and then communicate with their data-dependent peers. The phases are separated by global synchronization operations.

[1] Not that `run()` method represents the entry point for each Java thread.

For simplicity in implementation, we restrict a virtual machine to be reconfigured only in between phases of a computation. Since little information needs to be recorded for the computation to proceed into the subsequent phase, threads of an agent can be re-started easily from its limited instance variables.

The Traveler's virtual machine can be reconfigured in two ways:

- It moves its running agent to new servers in the case that some current servers of the virtual machine become unavailable. If no additional servers are available to continue the execution, the virtual machine just transfers data.
- It changes its configuration, expanding or shrinking the size to adapt to the varying resource requirements of the agent or to the change of the availability of server resources. Expanding a virtual machine needs to duplicate the running agent in new servers.

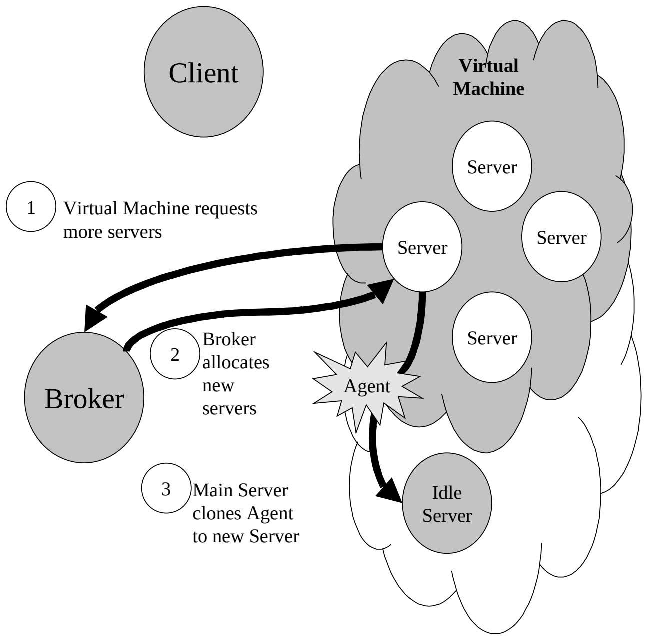
Client
Virtual Machine
Server
Server
Server
Server
1 Virtual Machine requests more servers
2 Broker allocates new servers
Broker
Agent
3 Main Server clones Agent to new Server
Idle Server

Figure 2. Mobile and Adaptive Virtual Machine

Details of the mobile and adaptive procedures of a virtual machine are in Figure 2. It starts with a request from the virtual machine to its broker. Upon receiving a request for additional servers, the broker checks the states of the registered servers and allocates required servers to the virtual machine, if possible. The agent then suspends its execution in between phases of bulk synchronous computations. It is the main server of the virtual machine that clones its agent task for the new server. If the request from a virtual machine is a migration to a new server (for the purpose of enhancing data locality or fault masking), the agent running at the original machine will be killed after it clones for the new site. The expanding, shrinking, or migration process ends with a status report to the broker and/or the client.

4.2 Distributed Shared Array Runtime Support

To support multithreaded agents on virtual machines, the TRAVELER implements a distributed shared array (DSA) runtime support system, as an integral part. The DSA system provides a Java-compliant interface to extend multithreaded programming to clusters. It supports SPMD programming paradigm. It exposes the hierarchical organization to programmers and allows programmers to explicitly specify globally shared arrays and their distributions. It shares a common objective with Global Array [27] to combine the better features of message passing and shared-memory for a fairly large class of regular applications. Shared arrays are distributed between threads, either regularly or as the Cartesian product of irregular distributions on each axis. It makes a better tradeoff between ease of programming and execution efficiency. We present its distinctive features in this paper. Details of the model can be found in [39].

The DSA system provides a transparent interface with two distributed variables, SharedArray and SharedPmtVar, to user programmers. The interface comprises of a collection of access primitives, including asynchronous read and write methods. A distributed object is created by methods createSharedArray and createSharedPmtVar of the AgentTask object. Upon arriving at a virtual machine, the object is distributed among the local and global threads (within a server and across servers) of the machine. Each server owns its partition. Accordingly, array access operations are distinguished between local and remote. Access to any array items is through a DSA run-time support system.

The DSA support system consists of a group of threads (DSA threads) running at each server. The DSA threads are spawned on demands. We designate one of the DSA threads as a root. During the creation of a virtual machine, all newly created threads report to the root with information about the number of threads assigned by the broker. Once the virtual machine is set up, the root replies to all other threads with the configuration of the virtual machine (i.e. the number of nodes, number of threads per node, node IP addresses, etc.). On receiving an access request to an item from an application thread, the local DSA thread first determines the owner of the item according to the item index. Remote access is implemented using Java RMI, too. To tolerate remote access latency, the DSA support system fetches a chunk of data items at a time and caches them on non-owner sites. The chunk size can be various in different implementations. For simplicity, we set the chunk unit as the entire partition containing the requested item.

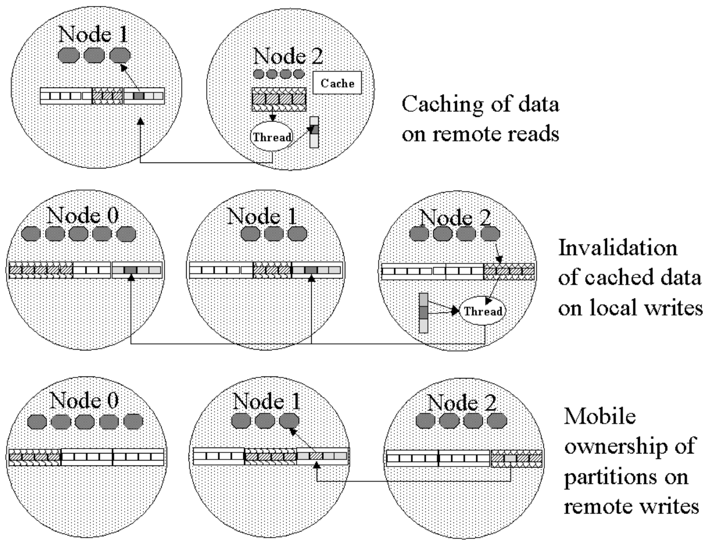
Node 1
Node 2
Cache
Thread
Caching of data on remote reads
Node 0
Node 1
Node 2
Thread
Invalidation of cached data on local writes
Node 0
Node 1
Node 2
Mobile ownership of partitions on remote writes

Figure 3. Mobile ownership of partitions

The DSA system deploys an SCI-like directory-based invalidation protocol, together with a mobile partition ownership, to ensure cache coherence. As shown in Figure 2, the DSA thread owning a block maintains a linked list of sharers for the block and a pointer to the head of the list. The head thread has both read and write permissions on its cached block whereas the others have only read permission. The DSA thread for a local write will invalidate remote copies. It will first request the ownership of the remote partition by tracing the ownership trail. Invalidation occurs after the current owner grants the ownership. The invalidation and mobile ownership events are transparent to applications.

To provide multi-step atomic operations, DSA allows a thread to lock out other threads from accessing an index. When a thread has a combined read and write operation, the thread can lock out other threads from writing in the middle of the operation. The mutual exclusive operation can occur at either index or partition level.

We conclude this section by a summary of the features of the DSA run-time support system, in comparison with the Global Array. Both the DSA and GA provide a shared array programming model. However, the DSA run-time support features a thread-oriented, adaptive, and Java-based implementation.

- The DSA provides a shared address space to threads running over a cluster of servers. The DSA itself is implemented as a collection of threads that are dynamically spawned on demands. It matches well with the server architecture and has potential to deliver higher performance than GA's process model.
- The DSA supports dynamic re-distribution of a SharedArray object and hence facilitates masking server faults, balancing workload, and improving data locality.
- The DSA exploits spatial and temporal localities by duplicating array partitions. Its mobile ownership mechanism helps improve data locality further.
- The DSA is based on platform-neural Java language and its remote array access is implemented in Java RMI. It enables parallel global computing.

5 The TRAVELER API

The DSA runtime support system provides a Java-compliant programming interface to extend multithreaded programming to clusters. It exposes the hierarchical organization to programmers and allows programmers to explicitly specify globally shared arrays and their distributions. It is realized by extending their codes from the AgentTask class.

5.1 AgentTask: A Base Class of Parallel Applications

The AgentTask class refers a server to the machine that has one or more processors. Parallel programs extended from the class should be supplemented with a list of servers and the number of threads to be created at each server. In TRAVELER, it is the broker that provides the server list when a DSA-based virtual machine is formed. We designate the Thread-0 of Server-0 as the main thread.

```
Public class AgentTask implements Agent {
  SharedArray createSharedArray (String name, int size, int grain)
  SharedArray createSharedArray (String name, int size)
  SharedPmtVar createPmtVar (String name)
  void globalBarrier()
  void localBarrier()
  Barrier createBarrier(String name, int size, String type)

  int numServer(), myServerID(),setServerID()
  int numGlobalThreads(), numLocalThreads(), myLocalThreadID()

  int fileOpen(Sring filename, char type) // return a file id
  int fileReadLine(int fid)
  int fileWrite(int fid, String line)
  void endJob(Object o)
  void setStatus(String status)
}
```

The DSA system defines two distributed variables: SharedArray and SharedPmtVar. The method createSharedArray() creates a SharedArray object. It can be distributed between threads in different ways. Currently, block decomposition is supported. The optional parameter `grain` specifies the granularity of coherence for data replication. The DSA actually provides three extensions of SharedArrays: SharedIntArray, SharedFloatArray, and SharedDoubleArray. Similarly, the object SharedPmtVar refers to a base of shared objects of primitive type. The method createPmtVar creates shared singular variable of types SharedInteger, SharedFloat, and SharedDouble for synchronization purposes. Operations over the distributed arrays and shared variables include synchronous and asynchronous read and write.

The methods globalBarrier and localBarrier methods provide barrier synchronization between global threads of the entire virtual machine (across servers) and local threads (within a server), respectively. Programmers can also create their own barrier objects, via the method createBarrier, for synchronization between any group of threads. The next method group returns the total number of servers of a virtual machine, local server identifier with respect to a thread, total number of local threads within a server, thread identifier within a server. The information helps programmers to distribute and redistribute data between threads. The methods regarding file read and write are for the virtual machine to contact the VPI of the running agent for input data or returning results.

5.2 An Example – Parallel Inner Product of Vectors

In TRAVELER, a user program starts with a master thread, which calls a VPI method to create an application-specific VPI object and create a ParallelAgent object to wrap a AgentTask. ParallelInnerProduct, defined in the following, is a AgentTask, which performs inner product of a vector. The TRAVELER's broker will clone the agent for each server of a virtual machine. One each server, a number of threads, either specified by programmers or provided by the broker, will be spawned to perform the operations defined in the run method. On a dedicated server, the number of threads is set to the number of processors of the server.

```
import traveler.agent.*;
public class ParallelInnerProduct  extends AgentTask {
  private int vecSize;
  private String fileName;
  private SharedFloatArray vec;            // input vector
  private SharedFloatArray result;         // temporary vector for intermediate results

  public ParallelInnerProduct(String fn, int sz) {
    fileName = fn;
    vecSize = sz;
  }

  public void run() {
    vec = createSharedFloatArray("Inner Product Vector", vecSize);
    result = createSharedFloatArray("Immediate Result Vec", numServer());
    globalBarrier();
    if ( mainThread() )  read fileName to initialize the vector
```

```
                                   // mainThread is Thread-0 at Server-0
    int blkSize = vec.length/(numServer() * numLocalThread());
                                   // Assume same thread number per server
    int myMinIndex =
          (myServerId()*numLocalThread()+myLocalThreadId())*blkSize;
                                           // Assume block decomp.
    float sum = 0.0;
    for (int k = myMinIndex; k < myMinIndex + blkSize; k++)
    sum  + = vec.read(k) * vec.read(k);
    result.write(myServerId, sum);
    globalBarrier();

    if ( mainThread() ) {
       float sum = 0.0;
       for (int k = 0; k < numServer; k++)
          sum  += result.read(k);
       fileWrite( VpiStdout, sum ); // Output results to VPI at the client machine
    }
  }
}
```

The ParallelInnerProduct is first wrapped by ParallelAgent, and then dispatched via a VPI. The following is a method innerProductVPI, which establishes an application-specific VPI object (ipVpi) and creates a ParallelAgent object (pa) to wrap the task agent ParallelInnerProduct.

```
public void innerProductVPI() {
  VPI ipVpi = new VPI();
  ParallelAgent pa = new ParallelAgent();
  ParallelInnerProduct ft = new ParallelinnerProduct("Datafile", 1000);
                                 // Assume data is in "Datafile" and size=1000
  pa.addTask(ft);
  vpi.addAgent(pa, "Datafile");
                                 // "Datafile" is provided to VPI for its communication with agents.
  pi.sendAgent();
  float answer = (float[]) ipVpi.waitUntilComplete();
}
```

6 Experiments

The evaluation of the TRAVELER was done in three major aspects. First was the time for establishing a parallel virtual machine, including the cost of RMI. Second was the cost of local (within a server) and remote (across servers) data access over the DSA. Last was about TRAVELER's overall performance in the solution of two applications: sorting and LU factorization.

All the experiments were conducted on a cluster of four SUN Enterprise Servers. One machine was 6-way E4000 with 1.5 Gbytes of memory and the other three were 4-way E3000 with 512 Mbytes of memory. Each processor module had one 250MHz UltraSPARCII and 4 Mbytes of cache. The machines were connected by a 155 Mbs ATM switch. We designated one 4-way machine as the broker and others for parallel virtual machines. Clients were run either in the same machine as the broker or in workstations of a remote local area network. All codes were written in Java and compiled in JDK 1.1.6.

6.1 Cost of Creating a Virtual Machine

Establishing a parallel virtual machine on a cluster of servers involves three major steps: a client submits task agents to a broker, the broker executes trades, and the broker dispatches the agents to selected servers. The client and broker can be run either on the same machine or on different machines. Figure 4 shows the overall time and a breakdown of the time for creating a virtual machine comprising up to three servers.

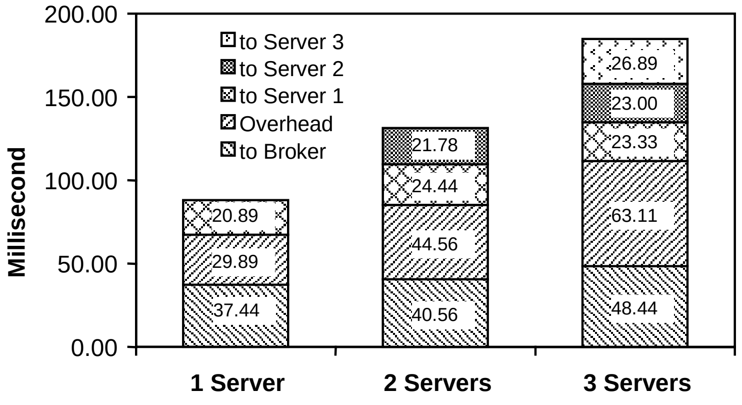

Figure 4. Cost of Creating Virtual Machine

The time from client to broker (to Broker) and from broker to each server (to Server 1, Server 2, and Server 3) is basically the cost of RMI and object serialization. We measured the round-trip time and then took half of it. Since the broker and servers are located in an ATM network, the cost of RMI and object serialization between a pair of machines is from 20 to 27 milliseconds.

The broker overhead (Overhead) includes the time for selecting servers, duplicating the task agent for each server, and initializing data for the virtual machine and the waiting time for server acknowledgements. Since each server would not acknowledge its status until it finishes thread spawn and DSA object distribution, the waiting period is actually the time required by the servers to construct a virtual machine plus one-way RMI. From the figure, it can be seen that the cost of RMI is still a large portion of the broker overhead.

Since the TRAVELER is targeted global computing on the Internet, we also experimented with remote submission of task agents. We set up a client in a different local area network from the broker. The two LANs were connected by an Ethernet. It was observed that an agent submission from clients on a different LAN took about 37 to 48 milliseconds. Compared with the cost for a local submission, RMI and object serialization dominate the total cost.

6.2 Overhead of the DSA

In the second experiment, we measured the access time of an array item via the DSA within a server and across servers. An array of 10,000 integers was assumed to be distributed equally among the whole threads spawned by a computational agent.

Figures 5 and 6 plot remote and local access time versus the number of threads spawned by the agent on each server. The figures show that a remote access takes around 5 milliseconds. It is huge compared with 1.7 microsecond local access time because remote data access is realized via Java RMI. Recall that the last experiment showed the cost of a RMI-based agent submission was between 24 to 34 milliseconds. The big difference between two RMI costs should be due to the complexity of the transmitted objects. Since RMI-based remote array access only involves primitive objects (indices and item values), their serializations can be done quickly. By contrast, serialization of a complicated agent object will take much longer. The difference between the two RMI cost also implies that object serialization dominates the cost in Java RMI.

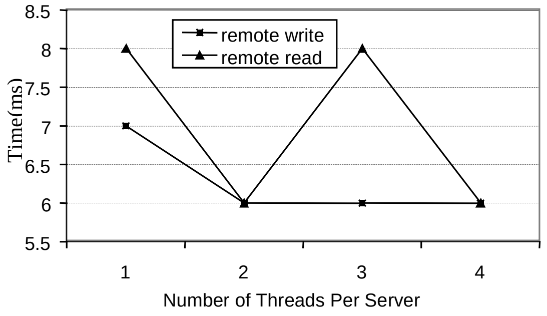

Figure 5. Cost of Remote DSA Access

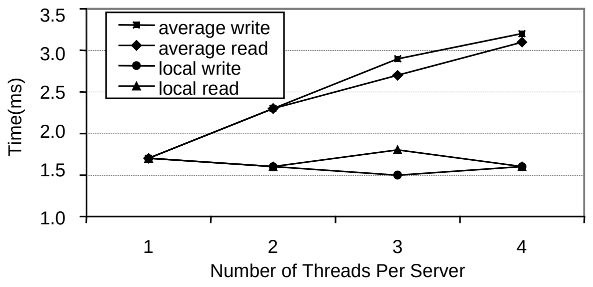

Figure 6. Cost of Local and Average DSA Access

In addition to local access time plots, there are also two plots of average access time in Figure 6. They were measured by scanning the whole array from the beginning to the end. Since the partition size of each thread decreases with the increase of the number of threads per server, the figure shows that the average access time becomes close to local access time, in particular in the case of large partition size. It is resulted from the DSA's data caching and mobile ownership strategies.

Success of the DSA mechanism also relies critically on the performance of synchronization operations. The TRAVELER implements barrier synchronization of threads across servers by designating one of the local application threads to communicate with remote threads. Figure 7 plots the cost for barriers between threads within a server and across servers as the number of thread increases. Expectedly, the figure shows the cost of a local barrier within a server increases with the number of threads. Synchronization with remote threads will cause a big jump in cost due to the overhead of remote memory access.

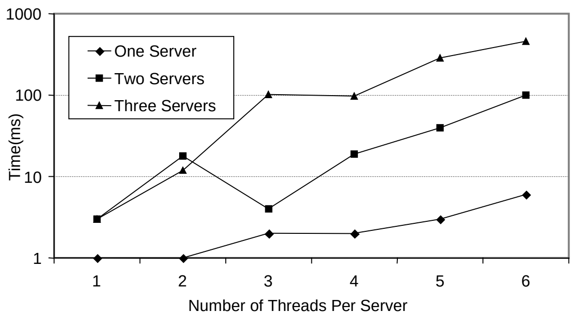

Figure 7. Cost of Barrier Synchronization

6.3 Distributed LU Factorization and Sorting

Finally, we evaluated the overall performance of the TRAVELER in the solution of two problems: odd-even sorting and LU factorization.

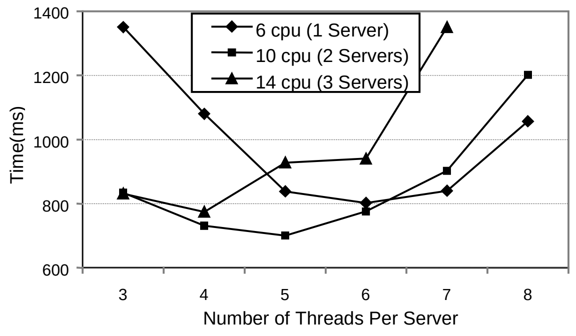

Figure 8. Timing of LU Factorization

We considered LU factorization of a 100 x 100 integer array. It was partitioned into threads in the simplest row-wise block decomposition way. Figure 8 shows the total execution time on virtual machines of one, two, and three servers as the number of threads per server increases. From the figure, benefits from a parallel virtual machine with multiple servers can be seen clearly. The machine with two servers outperforms the others. In each case, the machine performs best when 4 to 5 threads exist. It is because all the servers, except one, have four CPUs. One server has 6 CPUs. That is why a virtual machine with the single 6-way SMP wouldn't saturate until 6 threads.

In the parallel sorting application, an input array was block decomposed. Threads proceed independently over their array partitions. They are then synchronized to combine their sorted results in parallel. We selected the odd-even sorting due to its simple algorithmic structure. Figure 9 shows the total sorting time of an array of 10,000 integers on virtual machines with one, two, and three SMPs. The figure clearly indicates the performance improvement due to the use of multiple servers. The breakdown of the execution time demonstrates again the efficiency of TRAVELER's thread synchronization.

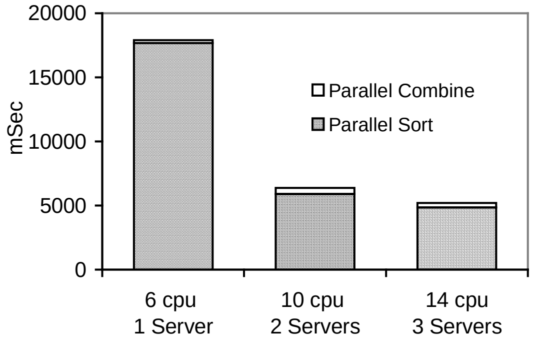

Figure 9. Timing of Distributed Sorting

7 Concluding Remarks

In this paper, we have proposed a novel mobile agent based "push" methodology for wide area parallel computing. In the method, clients declare their applications as mobile agents that roam on the network to find servers to run. Since the agents can be programmed to satisfy their goals, even if they move and lose contact with their creators, they can survive intermittent or unreliable network connection. During their lifetime, the agents can also move themselves autonomously from one machine to another for load balancing, enhancing data locality, and tolerating faults. We have presented an agent-oriented programming and resource brokerage infrastructure, TRAVELER, to support wide area parallel applications. Traveler provides an agent wrapper to simplify agent programming. Agents are dispatched to a resource broker for services. Upon receiving an agent, the broker executes trades and forms a parallel virtual machine over available servers to execute the computational agent. TRAVELER also provides an integrated distributed shared array run-time support system to support agent communications on clusters of servers. Java RMI provides for secure channels between clients and servers. Its built-in security mechanisms also protect servers from possible attacks by malicious clients.

We have demonstrated the feasibility of the TRAVELER in parallel sorting and LU factorization problems. The broker and servers were run on a cluster of SUN Enterprise servers. Remote clients programmed their applications as agents by extending the TRAVELER API's `AgentTask` class and submitted them via a Virtual Processor Interface. We have evaluated the overall performance of the TRAVELER as well as the cost of its two major components: creation of a parallel virtual machine and DSA data access. Although the current prototype was presented as a proof-of-concept and has not been deliberately refined, benefits in performance from parallel global computing have been observed in both applications. It was found that the cost of RMI and its related object serialization were two major sources of performance loss. Their optimization [37] are expected to boost the overall performance of the TRAVELER further.

Future work will primarily be on refining the TRAVELER system for performance and demonstrating the feasibility of the system with more applications. First, we will investigate the mobility of computational agents. Due to the needs of multithreading for high performance computing on multiprocessor servers, our focus will be on the mobility of multithreaded agents. Second, we will refine the DSA runtime system with an implementation of variable shared granularities so as to make agent communication more efficient. Recent studies explored the potential of fine-grained mobile agents [24]. We will also implement a Java TCP/UDP-based DSA version for a cluster of machines that belong to the same administrative domain. Third, we will improve on the current virtual machine to support fully adaptive parallelism of bulk synchronous SPMD applications.

Acknowledgements

This work is supported in part by U.S. National Science Foundation Grant EIA-9729828. We would like to thank the anonymous referees for their valuable comments and suggestions.

References

[1] J. Baldeschwidler, R. Blumofe, and E. Brewer, Atlas: An infrastructure for global computing, in *Proc. of the 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, 1996.

[2] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff, Charlotte: Metacomputing on the Web. In *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems, 1996.* http://www.cs.nyu.edu/milan

[3] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman (eds.*) The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Pub. August 1998.

[4] J. Bredin, D. Kotz, and D. Rus. Market-based resource control for mobile agents. In *Proc. of Autonomous Agents*. Pages 197—204, May 1998.

[5] B. Brewington, R. Gray, K. Moizumi, D. Kotz, G. Cybenko, and D. Rus. Mobile agents in distributed information retrieval. In Matthias Klusch, editor, *Intelligent Information Agents*, chapter 12, Springer-Verlag, 1999.

[6] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. Of Supercomputing'96.* and "Using agent-based software for scientific computing in the NetSolve system", *Parallel Computing*, 24(1998), pages 1777—1790.

[7] D. M. Chess. Security issues in mobile code systems. In *Mobile Agents and Security*, G. Vigna (Ed.), Springer-Verlag, 1998, pages 1--14.

[8] B. Christiansen, P. Cappello, M. F. Ionescu, M. Neary, K. Schauser, and D. Wu. Javelin: Internet-based parallel computing using Java. *Tech. Report, University of California, Santa Barbara*, 1997.

[9] DESCHALL. A Brute Force Search of DES KeySpace. http://www.interhack.net/projects/deschall

[10] Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Pub, August 1998.

[11] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. http://www.globus.org

[12] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1): 70—82, 1996.

[13] G. Fox, K. Schauser, and M. Snir (eds). ACM 1999 Java Grande Conference. June 1999.

[14] A. Fuggetta, G. Picco, G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, vol. 24, No. 5, May 1998, pages 352—361.

[15] General Magic Odissey page, http://www.genmagic.com/agents/odyssey.html.

[16] A. Grimshaw, W. wulf, and the Legion team. Legion: The next logical step toward the world-wide virtual computer. http://www.cs.virginia.edu/~legion.

[17] M. Huhns and M. Singh. Internet-based Agents: Applications and Infrastructure. *IEEE Internet Computing, Special Issue on Internet-based Agents*, 1(4), July/August 1997.

[18] J. Jones and C. Crickell. Second evaluation of job queuing/scheduling software. *Tech. Report NAS-97-013, NASA Ames Research Center*, 1997.

[19] N. M. Karnik, and A. R. Tripathi. Design issues in mobile-agent programming systems. *IEEE Concurrency*, July—September 1998, pages 52—61.

[20] N. Kronenberg, H. Levy, and W. Strecker. VAXcluster: A closely coupled distributed system. *ACM Trans. on Computer Systems*, 4(2), May 1986.

[21] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.

[22] M. Livny, R. Raman. High-throughput resource management. In I. Foster and C. Kesselaman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, Pub. 1998.

[23] M. Litzkow, M. Livny, and M. Mutka. Condor – A hunter of idle workstations. In *Proc. of the 8th Int. Conf. on Distributed Computing Systems*. Pages 104—111, 1988.

[24] Microsoft. Clustering Support for Microsoft SQL Server: High Availability for Tomorrow's Mission Critical Applications (white paper). 1997.

[25] C. Mascolo, G. P. Picco, and G.-C. Roman. A fine-grained model for code mobility. Tech. Report WUCS-99-07, Department of Computer Science, Washington University in St. Louis, March 1999.

[26] J. Nagib, A. Beguelin, et al. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. Report CMU-CS-95-137, School of Computer Science, CMU, 1995.

[27] J. Nieplocha, R. Harison, and R. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, Vol. 10, 1996, pages 169--189.

[28] ObjectSpace, Inc. ObjectSpace Voyager Core Technology. http://www.objectspace.com

[29] Primenet Server. The GREATE Internet Mersenne Prime Search. http://www.mersenne.org/ prime.html

[30] L. Sarmenta and S. Hirano. Bayanihan: Building and studying web-based volunteer computing systems using Java. *Future Generation Computer Systems, Special Issue on Metacomputing*, Vol. 15(5/6), 1999.

[31] S. Sekiguchi, M. Sato, H. Nakada, S. Matsuoka, and U. Nagashima. Ninf: Network based Information Library for globally High Performance Computing. In *Proc. Of POOMA'96*.

[32] L. Smarr. Special issue on computational infrastructure: Toward the 21st century. *Communication of ACM*, 40(11), Nov. 1997.

[33] D. Strom. Push Publishing Technologies. http://www.strom.com, May 1999.

[34] Sun Microsystems. Java Remote Invocation --- Distributed Computing for Java. http://java.sun.com (white paper).

[35] Tandem Computers, Inc. Making Enterprise-Class Clusters Come Alive (white paper), 1997.

[36] A. Vahdat, et al. WebOS: Operating system services for wide area applications. *http://now.cs .berkeley. edu/WebOS*

[37] G. Thiruvathukal, L. Thomas, and A. Korczynski. Reflective remote method invocation. http://www. jphc.org

[38] B. Wims and C. Xu. Traveler: A mobile agent based Infrastructure for global parallel computing. In *Proc. of First Joint Symposium: Int. Symp. on Agent Systems and Applications (ASA'99) and Third Int. Symp. on Mobile Agents (MA'99*), October 1999.

[39] C. Xu, B. Wims and R. Basharahil. Distributed Shared Array: An integration of message passing and multitithreading on SMP clusters. *In Proc. of the 11th Int'l Conf. on Parallel and Distributed Computing and Systems*, November 1999.

[40] D. Wong, N. Paciorek, and D. Moore. Java-based mobile agents. *Communication of ACM*, Vol. 42, No. 3, March 1999, pages 92—101.