An Agent-Based Architecture for High Performance Computing over the Internet

Shahram Rahimi, Yung-Chuan Lee, and Mehdi Zargham

Abstract—The need for parallel processing has been increasingly recognized and is now regarded as an indispensable tool for various problem domains. Distributed object computing systems are widely envisioned to be the desired software development paradigm due to their higher performance and capability of handling machines and operating systems heterogeneity. The design, implementation and system evaluation of a novel parallel-processing Java multi-agent architecture, called MPIAB, is presented here. The system aims to combine the interoperability of the traditional parallel processing model Parallel Virtual Machine along with the high performance of Message Passing Interface. A preliminary analysis of the features and performance of MPIAB looks promising. Empowered with the features of fault-tolerance, dynamic load balancing and grid computing, we envision this system having a bright prospect in the field of distributed high throughput computing.

Keywords—Agent, distributed computing, high performance computing, Java, MPI.

I. INTRODUCTION

The discipline of parallel processing has seen unparalleled growth since its inception. Various spheres in science and computing have come to recognize it as an essential tool for solving complicated problems. Distributed object computing systems are widely envisioned to be the desired software development paradigms due to their higher performance and capability of handling machines’ and operating systems’ heterogeneity.

The existing de facto standards for parallel processing are PVM and MPI [15]. PVM stands for Parallel Virtual Machine. It enables a collection of different computer systems to be viewed as a single parallel virtual machine and communicate by message passing. It operates on a collection of homogeneous or heterogeneous UNIX computer systems in single or multiple networks. MPI stands for Message Passing Interface. It is a message-passing library specification for parallel computers. The goal of MPI, simply stated, is to develop a widely used standard for writing message-passing programs. It attempts to establish a practical, portable, efficient, and flexible standard for message passing in a homogeneous environment.

The possibility for advancing these traditional parallel models paves the path for the introduction of MPIAB (Message Passing Interface-Agent Based) agent architecture for parallel processing.

The proposed novel grid architecture for parallel computing addresses five major problems faced by both MPI and PVM traditional models. Firstly, MPI and PVM are not fully platform independent; therefore, interoperability between different kinds of implementations is not easily possible. Secondly, codes are not portable. Secondly, as the number of job submissions to the system increases, the importance of the existence of a load balancing mechanism in a parallel computing environment intensifies. Lack of elaborate scheduling mechanism for efficient submission of processes in MPI and PVM, leads to inefficient parallel execution of codes. They also do not offer automatic node selection for the participating computers, so load balancing for the network is not possible. Thirdly, these models are not designed to support grid computing. The possibility of a high performance grid environment for parallel computing is not addressed in either of these architectures. Fourthly, MPI and PVM do not comprise any fault tolerant mechanism. If one of the participating nodes crashes during computation, the job could not be concluded and has to be resubmitted to the system. Moreover, if a particular node underperforms during a parallel computation, MPI and PVM do not have any mechanism to prevent the performance decline of the system. Finally, the capability of dynamic load balancing which involves the dynamic reallocation of a process from one node to another, on finding itself heavily loaded, is not supported by PVM or MPI.

The platform independence is achieved by the selection of Java as the language support for this architecture. Java is a primary language for distributed environments. Its portability is primarily due to the mechanism involved in compiling and executing Java code [16]. Java source code is compiled using the Java compiler to generate Java byte code which is similar to normal machine code, except it is platform neutral. It is compiled for the Java Virtual Machine rather than specific target hardware. The JVM is an abstract computer implemented in software on top of a real hardware platform and operating system. This provides an important level of
abstraction - as Java byte code is created for and executed on the JVM, it is extremely portable, allowing codes to run on any system which implements a virtual machine. In addition to the portability of Java byte code, Java source code is also extremely portable. The language specification has been written to ensure that no platform dependent aspects exist. For example, a common portability issue arising with C and Fortran is differences in primitive data type sizes depending on the platform. In Java the language specification explicitly specifies the size of these data types, eliminating this problem. The choice of this language lends multithreading concurrency to the system making Java an attractive candidate for writing portable computationally intensive parallel applications. The security architecture of Java makes it reasonably safe to host an alien agent. Another major advantage of Java is that it provides communication mechanisms inside the language environment, whereas other languages like C++ require external mechanisms (libraries) like message passing. Java is available on almost every platform, including PC (windows and Linux), Sun, SGI, Compaq and Intel PC.

In a grid-based environment, as the system load increases by receiving high number of jobs, a judicious load balancing mechanism is vital for efficient distribution of the workload among different participating nodes in the system. MPIAB also aims to optimize the scheduling mechanism and to provide an improved load balancing scheme via the use of resource management agents that select the least busy nodes based on a threshold value. MPIAB uses remote agent creation which subsumes agent migration with less overhead. This is because remote agent creation does not require the agent state to be transferred to the destination.

MPIAB thus combines the interoperability of PVM along with the high performance of MPI using intelligent agents for distributed computing in a fault-tolerant and dynamically load balanced environment.

II. SYSTEM ARCHITECTURE

In general, the MPIAB architecture [19] employs Java agents at different functional levels to accomplish parallel processing tasks in a heterogeneous environment. There are three main types of agents in this architecture.

- Manager Agent (MA)
- Resource Agent (RA)
- Task Agent (TA)

Below we give a description of these agents. In the next section, we express the behavior of these agents considering various functional processing scenarios.

A. Manager Agent (MA)

The Manager Agent is a persistent stationary agent which resides on the root node (terminal with which the user interacts). Every participating node in the network where users are allowed to submit jobs from contains an MA. The MA receives tasks (parallel program to be executed) from the user through a Graphical User Interface (GUI). Then it generates the Task Agents (described below) with the help of the local agency and distributes them to the dynamically selected least busy nodes in the network, the number of which has been specified by the user. To allocate the least busy nodes for the task, the MA sends requests to the Resource Agents, which are located on the hosts available to the system (described below). If the response performance value (free CPU) returned by a Resource Agent is greater than the selected threshold value then the corresponding node is chosen to be a prospective candidate for node allocation. The threshold value is an adjustable value and is calculated based on the CPU utilization of the system when the task code is submitted to the node. The heuristic stage of MPIAB’s load balancing module then selects a set of nodes from the list of the prospective candidates (historically the least busy nodes) to participate in the process. The MA retrieves physical addresses of the selected hosts from a local data structure called Host-Registry (HR) that contains tuples of the hostname and IP address for each node in the network. It then generates Task Agents, give them the task code to be executed and disperses them to the corresponding selected hosts. We discuss more about MAs and TAs when we describe the behavior of the system.

B. Resource Agent (RA)

On each participating node in MPIAB grid environment an RA (a stationary agent) exists. On receiving a request from an MA for node allocation, the RA computes a performance value for the node on which it resides based on CPU utilization and returns this value as a response to the MA.

C. Task Agent (TA)

Created by a MA, a TA contains a data structure called SubHostRegistry referred to as SHR (which is a sub-domain of the HostRegistry data structure and contains tuples of type [IP, rank]) and an executable task code. The SHR contains information about the selected nodes (computers participating in the process). The TA is also associated with an Array of LIST data structure for storing the received data, which is described below. The TA initially creates a Collecting Thread (CT) to collect the incoming data from the sender nodes. Here Java Socket technology is utilized for sending/receiving data between different nodes in the environment. Hence, the next function of the TA is to create sockets for sending data to other nodes of the network. In this communication paradigm, the TA processes are directly responsible for sending data (non blocking send) and Collecting Thread (generated by TAs) responsible for receiving data (blocking receive). Collecting Threads utilize the Socket object from the ServerSocket class of java.net for listening and accepting connections.

Moreover, the TA also creates a MAServerProxy Object for the purpose of printing the results on the GUI located at the originating node. Associated with each socket created is an output stream (output) to send information to the destination node. The TA then initializes the various parameters of the Environment Object for the current run of the task code which include (the current rank, the total number of nodes in the network, the DataOutputStream, the THL data structure described below, and the MAProxy). It then begins the
execution of its task code. After the task code is executed to completion, the task agent suspends the Collecting Thread and returns the computed results back to the MA to be displayed to the user via the GUI. TA is then terminated.

D. Transfer Handler Lists (THL) Data Structure

This data repository is a serializable data structure which holds an array of doubly-linked list data structures and a set of methods for its manipulation. Both the CT and the TA have access to the THL. Consider a single list of the THL object; it includes two pointers: head pointer and tail pointer (Figure 1). These two pointers are used to keep track of the oldest and the most recent data inserted to the list by a message sender TA. The data structure is composed of an array of multiple such list data structures where each TA participating in a particular parallel execution is assigned one list to handle messages sent by it. Multiple receive requests for the same source TA are identified by a tag (a label). The TA retrieves the data (and deletes the data node from the list data structure) by traversing the list in a first in first out (FIFO) manner.

E. Collecting Thread (CT)

Each TA generates a CT to monitor its THL data structure and be responsible for receiving sent messages by other TA. The Collecting Thread when generated creates a serverSocket to listen for connections at the specified port (formed as [portBase+senderRank]). When a TA calls Send function (or any function that utilizes Send function), it transmits a packet to the destination TA’s CT. This sent packet is received by the Collecting Thread, of the destination TA and inserted into the THL data structure. The destination TA will retrieve the data from THL when it executes Receive function for the source TA.

III. BEHAVIOR OF THE SYSTEM

In this section, the behavior of the system for some common MPI functions has been presented.

A. Send and Receive

Consider a scenario where a user runs a parallel program on node $N_i$ in the network. We refer the reader to Figure 2.1 and Figure 2.2 for the illustration of this scenario. Let us consider Figure 2.1 first. The user interacts with the MPIAB GUI by passing the executable task code and the number of tasks which should execute it to the system. $MA_i$ sends a request for node allocation to the other nodes in the network. To respond to the request, the RA (already residing) on each node does a performance evaluation for their local computers and sends the values to $MA_i$. Comparing these performance values with a corresponding threshold, $MA_i$ selects a set of least-busy nodes for the process (this comprises a set of tasks which will be discussed later). Then, the MA generates a TA for each selected node (nodes participating in this parallel computing) and writes the list of all the selected nodes to the SHRs of the TAs. If the root node is one of the selected nodes for the computation, the system also generates a local TA (LTA) on the root node ($N_i$). The TAs, after reaching their destination, binds with the local agencies, generates their CT’s to mediate access to the THL data repositories and start execution of their task codes.

Now let us assume that during the course of the computation, $N_i$ desires to send a message to $N_j$ (see Figure 2.2 - $N_i$ doing a corresponding receive). On encountering a Send function call in its task code, $TA_{13}$ uses a DataOutputStream of java.io associated with its Socket object to send its message (in the form of a formatted packet) to the CT of the destination TA ($TA_{13}$), to be placed in its THL TA2. On executing the corresponding Receive function call in its task code, collects the sent message from its THL data structure based on the rank of the source TA. If the Receive is encountered in the code before the data is actually put in the THL, then the destination TA must periodically check the THL Data Structure until the message is available. This is because the Receive function is a blocking function.

During the execution of the Send function, an identification tag is generated and assigned to the message being sent (sendtag [dest]). This is to keep track of the messages sent to a particular destination. The Send function forms a packet (send packet), which comprises the current source node’s rank (ID of...
Figure 2.1. Send – receive communication scenario (I)

Figure 2.2. Send – receive communication scenario (II)

CT: Collecting thread, HR: Host registry, LTA: Local task agent, MA: Manager agent, N: Node, RA: Resource agent, SHR: Subh ost registry, TA: Task agent, THL: Transfer handler list

The node during the current execution, the send tag (ID tag of the message), which is generated in corresponding to the destination node, and the message itself (send packet=source rank+"-"+send tag+"-"+message). The Collection Thread of the destination TA receives the sent packet and inserts it into its Transfer Handler Lists (THL) data structure by invoking its “putdata” method. The non-blocking send in the system is maintained by the usage of the Collecting Thread which
simultaneously runs with the TA and has a listener Socket for collecting data from the sender nodes. When a TA needs to communicate with another TA, it uses a previously initialized socket connection to the destination address.

When a Receive function is encountered by a TA, the receive [source] tag is incremented for a particular source node, the data first inserted is collected from the list Data Structure (deleting that node) and returned. The collection of data is done on the basis of FIFO or queue structure. The data inserted first, is collected first after doing a check of the source. (By checking the equality of send [destination] tag and receive [source] tag). The send [destination] tag is appended to the data and incremented every time the data is sent to the same destination node and receive [source] tag is incremented every time the data sent by a particular source is collected. At the end of the process, the computed result is returned to MA.

B. Broadcast

This function broadcasts a message from the process with the rank ‘root’ to all other processes involved in the computation. Consider the MPI function MPI_Bcast as a case study which has the signature:

```c
int MPI_Bcast (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

This function is executed in MPIAB as follows:

On the “root” node, when the TA encounters an MPIAB_Bcast function call, it sends the content of its “buffer” over its socket based communication links to all the other TAs. In every other node, when the TA encounters an MPIAB_Bcast function call, it retrieves the data from its THT and places it in its “buffer”.

C. Gather

This call gathers values from a group of processes. Consider the MPI function MPI_Gather as a case study which has the signature:

```c
int MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int.recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

This function is executed in MPIAB as follows:

When the TA on a node with rank equal to “root” encounters an MPIAB_Gather function call, it collects data sent by all other participating TAs from its THT. On the nodes with rank other than “root”, the TAs, on encountering the same function, uses the corresponding OutputStream for the root node to send the data in their “sendbuf” variable to the root residing TA.

D. Scatter

This function sends data blocks from one task to all other tasks in a group. Consider the MPI function MPI_Scatter as a case study which has the signature:

```c
int MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

This function is executed in MPIAB as follows:

For the “root” node, the TA generates sockets for sending blocks of data of size ‘sendcnt’ from ‘sendbuf’ when the MPIAB_Scatter function is invoked. For every other node, the TA retrieves the data from THL with the source node equal to “root” and ‘recvbuf’ as the receiving buffer of size ‘recvcnt’.

E. Graphic Use Interface

Figure 3 illustrates a snapshot of the MPIAB 1.0 graphical user interface. Every participating computer in the network with the MPIAB application installed will have a data input screen as shown above. To execute a parallel task code written in Java on the network, a user would first need to compile his task code. Then using the GUI, the user can select (using the ‘Open’ button) the program class file (.class file type) from the local file system of the ‘root’ machine. The name of the selected java task code class is displayed in a text box. The user is then required to input the number of processors the task code is to be executed on in a text box provided. A list box on the GUI provides a list of currently registered processors in the domain. For task code execution the ‘Start’ button has to be pressed. The output panel displays the results of execution of the task code.

IV. DESIGN & IMPLEMENTATION

The design of MPIAB architecture, although seemingly simple, has undergone several iterations. A few during the design phase and some during the implementation phase. Revisiting the design during the implementation stage was necessary to simplify and enhance the system further by exploiting Java’s key features.

A. Design and Implementation Specifications

The first version of MPIAB (1.0) was implemented on the Grasshopper platform (note that this platform is no longer supported) [9]. As an initial step to develop the complete system, we have worked out a design strategy for send-receive communication scenario.

A UML sequence diagram depicting the flow of control between modules is shown in Figure 4. This diagram describes the timing sequence of method calls between different classes. The flow of control is initiated by the user who is passing the class, to be executed, to the manager agent. The arrows in the sequence diagram correspond to the method calls. The design outline presented in this section gives the reader a better understanding of the MPIAB application. The first step is the creation of the MA on the host node on which a user will submit the parallel task code through the MA GUI. The MA next creates the RA on all participating nodes. It then calls the computePerfVal() method of the RA to retrieve the CPU performance of the participating node. The MA then computes the computational complexity of the received task code and maps it to the appropriate CPU utilization, thereby selecting optimal nodes for parallel processing of the task. Once this is done, the Task Agents (TAs) are created on the selected hosts and executed.
Following this is the creation of Collecting Thread corresponding to all other nodes except the CT generating node, so that the generating node can collect the data sent from other nodes. The collecting Threads of all the nodes set up serverSocket to listen for incoming data. After the initialization of the package via the MPIAB_Init() call, the task code is executed by the TA. While the task code is executed if a MPIAB_Send() call is encountered, the data is written through sockets DataOutputStream to the destination node. The Collecting Thread that was created is still running and so it collects data continuously via the DataOutputStream() method and puts it into its THT data structure via the putData() method.

When the task code encounters a receive method call through the MPIAB_pargRecv() method it retrieves the data that was put in the THL Data Structure. When a MPIAB_pargPrint() is encountered, the results are printed on the MPIAB (MA-associated) GUI. This is accomplished via a call to the printBuffer() method of the MA.

Finally, the results of the processing nodes are returned by TA to the MA and which are displayed on the user GUI. The TA’s then terminate their Collecting Threads and remove themselves from the participating node. The user can terminate the MA by closing the GUI.

B. Design-Implementation Issues

The key design and implementation issues that arose during the nascent stages of this project may be broadly classified as - Core Functionality and Performance Enhancement issues.

1) Core Functionality: The design of the MPIAB architecture had to be revisited several times during initial implementation phase only to keep it as simple as possible and aiming to achieve a high performance platform. Some of the problems encountered while building the core functionality of the system are listed below –

- Remote agent creation improved the simplicity of the design by allowing the Manager Agent to remotely create the Task Agents on the participating host machines instead of first creating them on the user host and moving them to respective nodes in the network.

- Use of Collecting Thread instead of dedicating an agent called the Collecting Agent to listen to a socket for receiving data from other nodes. The initial design visualized a Collecting Agent that would be generated by the Task Agent once created on the selected parallel processing node. This reduced the dependency on the agent platform and reduced amount of agent information to be stored or transferred from node to node by making use of Java’s multithreading ability.

- Task Code to reside on Network Server. One of the main issues in making this architecture possible was to be able to execute the parallel task code submitted by the user through the Manager Agent GUI on the participating nodes. The parallel task code is in the form of Java byte code generated by the compilation of the user Java class file. The primary version of MPIAB makes use of a network server accessible to all the processing nodes. The user has to upload the task byte code on to this server to be able to execute on the other nodes. Once the Task Agents were created on the selected nodes, doing a URL look up of the task code on the network server would allow them to load the byte code as a 'Class' object into system memory using the URLClassLoader and execute it using a method to call the constructor of the object 'Class' to instantiate the object. The central network server acts as a common repository for all task codes to be executed. This however is a potential bottleneck. In the upcoming revised version of MPIAB, an alternate solution has been devised to allow the users to select and transfer the task code from their host machine.
Synchronization of task code execution – Due to variation in creation time of the Task Agents (due to processing node distance from source Manager Agent node) on the processing nodes, the task code execution initiated at different instants of time on different nodes. This caused discrepancy in communication timings and sometimes premature termination of program as it wasn’t able to find the Task Agent on the receiving node. The agent platform’s createGroup method of ProxyGenerator class allows us to group several agent classes into a single group and by using the MulticastResult class (AND function), we can force the Task Agents to wait for a signal (method call) from other Task Agents before proceeding with the execution of task code.

```java
serverGroup = ProxyGenerator.createGroup(ITaskAgent.class);
((IGroup)serverGroup).setResult(MulticastResult.AND);
(ITaskAgent)serverGroup.numberUpdate();
mResult = ((IGroup)serverGroup.getResult());
```

Access of Common Data Structure. A data structure for storing data sent by other nodes was required, as a send and receive may not necessarily be encountered in that order on two different nodes. The data sent through a socket had to be collected as and when it was received. The Collecting Thread serves this purpose. However, the Data Structure had to be made accessible to both the Collecting Thread (for storing the data after retrieving through socket) and to the executing Task code by the Task Agent (for the receive method encountered) which are two separate threads of execution. This was made possible using Java’s pass by reference concept which allows a reference to the memory location of a Java Object to be passed as a parameter. Here references to the Transfer Handler List data structure was passed to both the Collecting Thread (‘tail’) and the executing Task Code (‘head’). Using these references, the Collecting Thread could now insert data received from other nodes in to the List while the Task Code retrieved it from the List on encountering a ‘Receive’ function.
Synchronization of Data Structure – To handle the situation where a ‘Receive’ was encountered by the Task Code on one node prior to the other node sending it the data, the MPIAB receive function does a polling of the data structure to check if any data was received.

2) Performance Enhancement: With the chief goal of delivering a high performance parallel processing platform, this project has had to deal with several performance hindering issues. Some of the intricate performance techniques used to make this java agent platform as efficient as possible are –

- Socket communication was found to be the fastest communication mechanism when compared to remote method invocation of Java or remote Agent method invocation.
- Sockets TcpNoDelay for faster communication. The java.net.Socket class in Java provides a method setTcpNoDelay which allows us to enable or disable Nagle's algorithm, the purpose of which is to conserve bandwidth by minimizing the number of data segments that are sent. Nagle algorithm automatically concatenates a number of small buffer messages; this process (called nagling) increases the efficiency of a network application system by decreasing the number of packets that must be sent. When TcpNoDelay is enabled, data will not be sent immediately, instead it waits for more write calls and tries to squeeze in as much as possible into a packet before sending.

```java
public void setTcpNoDelay(boolean on)
throws SocketException
```

- Underlying data type for data transfer is String – On running preliminary tests on MPIAB, it was observed that using the basic data types such as int and double, as is, in communication (send-receive) caused the performance to depreciate even for small sizes of data. But this did not happen to a very large extent with the String data type. As a workaround to this problem, all data communication between nodes involve passing of String from one node to another. Since java provides the ability to convert any data type to a String and vice-versa, concatenation of arrays of a data type other than a String for message passing becomes viable.

Fragmenting larger data in to smaller size for faster communication - Since even the String data type has a threshold on the communication time based on its size; for larger chunks of data, the concatenated String is broken down into smaller sized String's and sent separately thereby increasing the number of sends for a single large array of any data type. Testing this approach proved it to be far more efficient than sending the entire array as a single String.

C. Implementation tools

The foremost requirement while considering tools to adopt was the selection of an implementation language, another concern in implementation was to find an appropriate distributed environment.

After Java was selected as the programming medium for its key distinct features of portability, security and multi-threading capability, the next step was to select the agent environment and also select a methodology to obtain the on-the-fly system resources for the load balancing module.

Grasshopper, a mobile agent platform, is used as an agent environment for the development of the MPIAB. Java Native Interface (JNI) is used for extracting the system CPU usage in the networked environment. Both the technologies are described in the subsections, below.

V. SYSTEM EVALUATION

In this chapter, we illustrate how the MPIAB system prototype developed so far achieves the desired features discussed in the introduction section and supports future enhancements for those not included in the prototype. A performance evaluation is also covered.

A. System Feature Evaluation

Reflecting back and bringing to view our goals driving the inception of MPIAB, we derived the following requirements for creating a distributed high throughput computing system–

1. Simple and Efficient
2. Interoperability & Security in a heterogeneous environment
3. Distributed Dynamic load balancing
4. Fault-tolerance providing for high availability
5. Grid computing

The choice of having an architecture based on software agents and implemented in Java has been crucial in achieving most of the above goals. MPIAB employs Java mobile agents in a Grasshopper environment for fault tolerant, secure and dynamically load balanced distributed high throughput computing. The MPIAB prototype does not currently support fault-tolerance, dynamic load balancing and grid computing mechanisms though a primary version of a load balancing system has been developed. Nevertheless, we will discuss here how we can incorporate these additional features in to the existing system seamlessly.

The MPIAB prototype was implemented on Grasshopper version 2.2.4b using Sun’s JDK version 1.4.2_02 and running on both Intel P4 (Windows NT OS) and Sun UNIX machines.

1) Simple and Efficient: Mobile agents characteristically consume less network resources as they move the computation to the data and not vice versa. Modeling the message passing functionality based on MPI's function calls also adds the familiarity of a popular standard to the system. The architecture maintains its simplicity by exploiting the generic inherent parallelism and navigational autonomy of agents allowing for distributed job management.

2) Interoperability and Security: Portability and security of agent execution [11] are the most fundamental requirements for mobile agent platforms, portability being an issue because mobile agents should be able to move in heterogeneous networks to be really useful, and security being at stake because the agent's host effectively hands over control to a foreign host. Grasshopper being MASIF (Mobile Agent System Interoperability Facility) compliant is interoperable between agent platforms of different vendors. Java also offers a higher level of platform independence than the traditional High Performance Computing languages such as C and Fortran, making it a natural choice for the Computational Grid. Java is suited for distributed applications as it provides...
mechanisms to simplify the process of writing distributed applications by hiding the differences in the connections and communication mechanisms between distributed resources. We employ Java socket communication in MPIAB which is the standard API for TCP communication.

The Grasshopper [11] security service supports two different kinds of security mechanisms: external and internal security. External security protects remote interactions between the distributed Grasshopper components. The external security mechanisms are based on the use of X.509 certificates and the Secure Sockets Layer (SSL). SSL is an industry-standard protocol that makes substantial use of both symmetric and asymmetric cryptography. Internal security protects resources of an agency from unauthorized access by agents. Furthermore, it is useful to protect agents from each other.

When running an application obtained from a remote system, it is important to either restrict what the code can do or have some mechanism which ensures the code comes from a trusted source before it may be executed. Java has mechanisms for implementing both these scenarios, in addition to a number of other features such as byte-code verification. When an untrusted piece of code is executed on a system, this is run within a sandbox, which simply means that a number of restrictions are placed on what the code can do. For example, the code cannot have any direct access to the file system. In this way, the potential damage an untrusted code can do is limited. This does of course place limitations on what an application can do, which may be an issue in some applications. Hence Java allows a digital signature to be attached to a piece of Java code, if the attached digital signature is from a trusted organization the code may be run without the sandbox restrictions. In addition to these security measures Java also provides a byte-code verification process that is performed when untrusted code is executed. This ensures that all code is well-formed, ensuring that corrupted or virus byte-code cannot be executed. Lastly, the lack of pointers in Java offers some security, as this prevents direct access to memory, which is a large source of malicious attacks.

3) Fault Tolerance: Inherently, mobile agents are free from the need of a continuous connection between machines which also results in lower network traffic. Also, when moving through a large and unreliable network such as the Internet, mobile agents may fall a prey to manifold accidents, e.g. host crashes or line breakdowns. An agent (the Task Agent) can create a checkpoint, i.e. a complete record of its current internal state, at any time in its execution. Checkpoints may be stored on some persistent media and can be used to later restore the agent to its state at the time of check-pointing [17]. There are several check-pointing strategies to choose from but the deciding factor would be the amount of complexity added to the agent code and to the system in its entirety. We will not discuss them here.

4) Dynamic Load Balancing: Load balancing in a distributed system is a process of sharing computational resources by transparently distributing the system load. The combined strengths of improved bandwidth and reduced latency over client-server applications as well as reduction of vulnerability due to network failure make mobile agents an ideal choice for implementing dynamic load scheduling. In practice, each Task Agent in the system could spawn a resource monitoring thread which polls the Resource Agent’s on each of the other hosts for its CPU-Memory performance values and based on certain other criteria such as time of execution on current host will determine to migrate to an under-utilized machine such that the system-wide performance is optimized. The possible strategies for implementing dynamic scheduling are numerous and will not be discussed here.

Grid Computing: Extending the mobile agent system to encompass grids over a vast network of computers spanning geographical areas will need to be further researched.

B. Performance Evaluation

The MPIAB architecture was developed aiming to combine the interoperability of PVM along with the high performance of MPI. It supports parallel processing modeled on MPI functions. For the purpose of evaluating the system performance, a very efficient, popular and high performance implementation of the MPI standard was chosen – MPICH, a freely available portable implementation of MPI.

The next step involved selection of a benchmark program that would be tested on both MPICH and MPIAB platforms and timed to compare how each would perform with differing sizes of the input data. Three chief factors contributing to the execution time are the time for communication between participating nodes, computation time and IO (memory access for retrieving data from data structure). In this project, preference was given to communication latency as the deciding factor when compared to the other two. For this purpose, the pingpong program was ideal. Functionality wise, the pingpong program basically sends a specified size of data from one parallel node to another which then receives and sends it back to the source node. The roundtrip time is averaged over 50 iterations, halved to get the one way trip time and the minimum of 10 test runs is taken. The input data size to this program varied from 64 bytes to 2048 bytes and was generated as double arrays that stored a random double value.

C. MPIAB vs. MPICH Timing Comparison

Table 1 gives the results of the tests for varying size of input data, of the range (64 – 2048 bytes), for both MPICH and MPIAB, in milliseconds. The following charts Figure 5 and Figure 6 depict the Size vs. Time and Size vs. Rate variation respectively as column and line graphs.

D. Observations

The lower performance of the system may be improved upon by taking advantage of new enhanced features in Java’s most recent release. The possible performance hindering components of the system are -

a) Synchronized data structure for storing data and

b) Data passing strategy (currently wrapped in a String object) between processes.
TABLE I
TIME COMPARISON MPIAB VS MPICH

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>MPICH (msec)</th>
<th>MPIAB (msec)</th>
<th>MPICH (MB/sec)</th>
<th>MPIAB (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.243</td>
<td>0.31</td>
<td>0.25</td>
<td>0.20</td>
</tr>
<tr>
<td>128</td>
<td>0.243</td>
<td>0.31</td>
<td>0.50</td>
<td>0.40</td>
</tr>
<tr>
<td>256</td>
<td>0.243</td>
<td>0.31</td>
<td>1.01</td>
<td>0.79</td>
</tr>
<tr>
<td>512</td>
<td>0.25</td>
<td>0.46</td>
<td>1.95</td>
<td>1.06</td>
</tr>
<tr>
<td>1024</td>
<td>0.364</td>
<td>0.49</td>
<td>2.68</td>
<td>1.99</td>
</tr>
<tr>
<td>2048</td>
<td>0.486</td>
<td>2.0</td>
<td>4.02</td>
<td>0.98</td>
</tr>
</tbody>
</table>

![Figure 5. Timing Comparison](image)

Figure 5. Timing Comparison

![Figure 6. Data Transfer Comparison](image)

Figure 6. Data Transfer Comparison

VI. RELATED WORKS

Given the large number of high-performance distributed computing systems built, it is impossible to mention all of them. Here we discuss the ones we feel are more related to our work and examine their architecture and functionality compared to MPIAB.

**STA:** Seamless Thinking Aid (STA) [1] is a Web aware Java-based environment which includes a number of tools for assisting parallel programming. The goal is to allow larger calculations and to couple applications with different memory or architectural requirements. This is a project of the Japanese Center for Computational Science and Engineering (CCSE) which was established within the Japanese Atomic Energy Research Institute (JAERI). Fortran 90 and MPI are used and the software is portable across many platforms including: Intel Paragon, Fujitsu VPP, Hitachi SR2201, Fujitsu AP3000, IBM SP, NEC SX4, Cray T90 etc. StaMPI is the application-layer communication interface for the Seamless Thinking Aid from JAERI. It is a meta-scheduling method which includes MPI-2 features to dynamically assign macro-tasks to heterogeneous computers using dynamic resource information and static compile time information. This kind of dynamic assignment can be done by PVM but not by MPI-1. An application program is invoked for execution as the main process on a host computer in the grid. Other processes are spawned by the main process on other computers to execute macro-tasks. MPIAB employs Java Task agents each of which would have the capability to dynamically assign itself a node in the network to balance the load, thereby removing the centralized control found in STA. It also provides fault tolerance mechanisms which is deficient in STA.

**Condor:** The goal of the Condor project is to develop, implement, deploy, and evaluate mechanisms and policies that support high-throughput computing on large collections of computing resources with distributed ownership. Condor is a distributed system developed at the University of Wisconsin that runs on a cluster of workstations. Condor provides an environment for executing serial and parallel applications on clusters. Moreover, it supports Checkpoint/Restart (C/R) in order to provide fault tolerance and process migration [21]. Condor employs the sync-and-wait protocol [18], which is coordinated, and imposes several restrictions on C/R in programs. As we mentioned before, our architecture will allow us to implement, side-by-side, both grid computing environment and load balancing facilities. Also, we believe that using MPIAB architecture we can remove most of the restrictions imposed in Condor.

**Globus:** The Globus project is developing basic software infrastructure for computations that integrate geographically distributed computational and information resources [6]. A joint project of Argonne National Laboratory and the University of Southern California's Information Sciences Institute, the project team includes groups at Argonne, USC/ISI, and the Aerospace Corporation, with significant contributions also being made by other partners. The Globus grid programming toolkit is designed to help application developers and tool builders overcome obstacles in the construction of "grid-enabled" scientific and engineering applications by providing a set of standard services for authentication, resource location, resource allocation, configuration, communication, file access, fault detection, and executable management. These services can be incorporated into applications and/or programming tools in a mix-and-match fashion to provide access to needed capabilities. While Condor is a tool for harnessing the capacity of idle workstations for computational tasks, Globus is primarily concerned with harnessing high-end computing resources. MPIAB also differs from the Globus project in this respect and provides a more simplistic and easy to use application architecture for distributed users. Globus is one of the most well known toolkit used to implement grid-computing middleware systems such as Condor.

**MPICH-G:** a grid-aware version of the free MPI
implementation is used with Globus which enables a MPI program to run in a grid environment without change.

**Legion:** Legion is an object-based meta-system [16] with the goal of providing a highly usable, efficient and scalable system designed to support large degrees of parallelism in application codes and manage the complexities of the physical system for the user. It has been built on a collection of connected hosts to provide a virtual computer that can access all types of data and physical resources. Legion arose from an object-based software project at the University of Virginia.

The project focuses on developing an object-oriented framework for grid applications. The goal is to promote the "principled" design of distributed systems software by providing sets of standard objects (for processors, data systems, etc.). Applications are then developed in terms of those objects. Globus and Legion are to some extent complementary since Globus focuses on low-level services and Legion on higher-level programming models. Legion is designed to be a worldwide virtual computer, while MPIAB is designed to be a reliable and highly available distributed system for executing message-passing applications across multiple networked domains. Legion also does not support dynamic load balancing.

**HPVM:** High-Performance Virtual Machines [20] is a distributed system that runs on a cluster of PCs with Windows NT. Developed at the University of Illinois at Urbana-Champaign, the Illinois HPVM project aims to develop shared controllable high-performance components for distributed systems and includes predictable communication, management of heterogeneity, stable performance models and adaptive resource management. This system achieves high performance communication by using modern processors (300 MHz Pentium II) and FM protocol for communication on Myrinet [3]. In addition, the system includes efficient implementation of standard scientific computing APIs such as MPI. However, HPVM does not support fault tolerance or high availability.

**Other Works:** LoadLeveler is a distributed system that runs on a cluster of workstations [10]. It provides an environment for executing serial and parallel applications with dynamic scheduling. In addition, it supports Checkpoint/Restart only with serial jobs in order to balance workload and provide process migration, and is thus incomparable to MPIAB. Millipede is a Distributed Shared Memory (DSM) system that runs on a cluster of workstations. It supports various consistency models of DSM [8], as well as thread migration inside the cluster for load-sharing and to improve the locality of memory references [13]. Millipede however, does not support fault tolerance, parallel I/O, security, and more [12]. On the other hand, MPIAB system supports load-balancing, and in other cases architectural extension compliant with grid computing environment.

The management of batch jobs within a single distributed system or domain has been addressed by many research and commercial systems, notably Condor [21], DQS [4], LSF [22], LoadLeveler [10], and PBS [14]. Some of these systems were extended with restrictive and ad hoc capabilities for routing jobs submitted in one domain to a queue in a different domain. In all cases, both domains must run the same resource management software. With the exception of Condor, they all use a resource allocation framework that is based on a system wide collection of queues—each representing a different class of service. Condor flocking [5] supports multi-domain computation management by using multiple Condor flocks to exchange load.

Recently, various research and commercial groups have developed software tools that support the harnessing of idle computers for specific computations, via the use of simple remote execution agents (workers) that, once installed on a computer, can download problems (or, in some cases, Java applications) from a central location and run them when local resources are available (i.e., SETI@home [2], Entropia, and Parabon). These tools assume a homogeneous environment where all resource management services are provided by their own system. Furthermore, a single master (i.e., a single submission point) controls the distribution of work amongst all available worker agents. Application-level scheduling techniques provide "personalized" policies for acquiring and managing collections of heterogeneous resources. These systems employ resource management services provided by batch systems to make the resources available to the application and to place elements of the application on these resources. Condor-G mechanisms complement this work by addressing issues of uniform remote access, failure, credential expiry, etc. [7]. Condor-G for grid computing represents the marriage of technologies from the Condor and Globus projects. From Globus [6] comes the use of protocols for secure inter-domain communications and standardized access to a variety of remote batch systems. From Condor comes the user concern of job submission, job allocation, error recovery, and creation of a friendly execution environment. The result is a tool that binds resources spread across many systems into a personal high-throughput computing system.

**VII. CONCLUSION & FUTURE WORK**

To summarize, the inception, design, implementation and system evaluation of the novel parallel processing Java agent architecture MPIAB have been presented in the preceding sections. The successful implementation of a primeval prototype of the Java agent architecture is witnessed. The preliminary performance evaluation of the system when compared to MPICH, a popular and high-performance implementation of the MPI standard looks promising. Further enhancements to improve on the system's performance may involve improving on a synchronized data structure to overcome memory contention issues and a more efficient data passing strategy between processes. To reiterate, the use of mobile agents in the MPIAB architecture engenders the improvement in bandwidth and latency compared to the client-server paradigm. The agent platform also materializes as a generic parallel processing framework. The power of Java's portability and Grasshoppers framework enables this agent
architecture to function in a heterogeneous environment apart from lending concurrency to the system.

The prototype version of the MPIAB system can be enhanced to support dynamic load balancing by empowering each Task Agent with the ability to migrate itself and select a host of the network based upon current system resources which is subject to change during execution of the task code. This would definitely give a more accurate estimation of the choice of selection of the hosts, resulting in a more efficient system optimization. Fault-tolerance mechanisms may also be incorporated into the system to allow it to be self-sufficient and be able to recover from network failures.

From our analysis of the features provided by MPIAB in the previous chapters, this distributed mobile agent system may definitely have a bright future as a high performance distributed computing system. Our ultimate goal or vision for the distributed architecture presented here will be to extend its computing power further by making it compliant with OGSA for Grid Computing, namely, MPIAB-Grid (MPIAB-G). This will allow the user to harness multi-domain resources as if they all belong to one personal domain. The user would define the tasks to be executed; MPIAB-G would handle all aspects of discovering and acquiring appropriate resources, regardless of their location; initiating, monitoring, and managing execution on those resources; detecting and responding to failure; and notifying the user of termination. The result would be a powerful tool for managing a variety of parallel computations in grid environments.

REFERENCES


Shahram Rahimi is the undergraduate program director and an associate professor of computer science at Southern Illinois University Carbondale. He received his PhD in 2002 from Stennis Space Center/University of Southern Mississippi in Computational Sciences. He also has Two MS degrees in Computer Science and Engineering. Rahimi is the editor in chief for International Journal of Computational Intelligence Theory and Practice and the associate editor for Informatica. He has over 130 peer reviewed publications in computational intelligence and distributed computing areas.

Yung-Chuan Lee received the B.S. and M.S. degree in computer science from the Southern Illinois University, Carbondale, in 2002 and 2005, respectively. He is currently a PhD candidate and Computer Information Specialist in computer science at the Southern Illinois University, Carbondale. His research interests include distributed computing, bio-inspired intelligence, ubiquitous computing, and trustworthy computing.

Mehdi Zargham is a professor and chair of computers science department at Southern Illinois University Carbondale. He received his PhD in computer science from Michigan State University. Zargham’s research areas include pattern recognition, clustering, soft computing, and data mining.