10-2003

MPIAB: A Novel Agent Architecture for Parallel Processing

Shahram Rahimi
Southern Illinois University Carbondale, rahimi@cs.siu.edu

Ajay Narayanan
Southern Illinois University Carbondale

Meha Sabharwal
Southern Illinois University Carbondale

Published in Rahimi, S., Narayanan, A., & Sabharwal, M. (2003). MPIAB: a novel agent architecture for parallel processing. IEEE/WIC International Conference on Intelligent Agent Technology, 2003. IAT 2003, 554-557.

MPIAB: A Novel Agent Architecture for Parallel Processing

Shahram Rahimi, Ajay Narayanan, Meha Sabharwal
Department of Computer Science, Southern Illinois University
MailCode 4511, Carbondale, Illinois 62901
{rahimi,anara,meha}@cs.siu.edu

Abstract

This paper presents MPIAB, an agent based architecture for parallel processing. The architecture is developed to model the functions of standard MPI using java agents. It remedies the deficiencies that exist in MPI, including the incapability to operate in heterogeneous environments. The architecture also addresses other issues such as effective utilization of system resources by dynamically selecting the least busy computing nodes through the computation of a threshold barrier value. Our proposed agent architecture would integrate the power of Java technology and agents to support efficient communication and synchronization of the nodes over the network for parallel processing.

1. Introduction

The need for parallel processing is increasingly recognized and is now regarded as an indispensable tool for various problem domains. The existing de facto standards for parallel processing are PVM and MPI. 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. 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 [1].

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. MPIAB addresses two major problems faced by both MPI and PVM traditional models. Firstly, they do not offer automatic node selection for participating computers in the parallel process, so load balancing of the network is not possible. Secondly, they are not fully platform independent; therefore, interoperability between different kinds of implementations is not easily possible, hence codes are not portable.

Lack of elaborate scheduling mechanism for efficient submission of processes in MPI and PVM, leads to inefficient parallel execution of codes. MPIAB addresses the automatic node selection problem by the use of agents, which select the least busy nodes to participate in the process. The platform independence is achieved by the selection of Java as the language support for this architecture.

The major design objective of MPIAB is 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 agent cloning which subsumes agent migration with less overhead. This is because agent cloning does not require the agent state to be transferred to the destination. MPIAB models MPI which is more widely used standard for parallel computing compared to PVM. The closest related work to MPIAB is Mobile Agent Team System (MATS) [2]. MATS is developed to extend the concept of PVM architecture. MATS architecture is quite impressive but lacks the support for automatic load balancing (selection of the least busy nodes) which our architecture provides.

The remaining parts of the paper are organized as follows: section 2 presents the proposed agent architecture. Section 3 and 4 discuss the behavior of the system and its design and implementation specifications. Finally, section 5 summarizes the discussion and sets a course for future work.

2. MPIAB architecture

In general, the architecture employs Java agents at different functional levels to accomplish parallel processing tasks in a heterogeneous environment. The application, we are developing using the architecture, has a GUI as the front end through which a user interacts with the MPIAB environment to submit a parallel processing task.

Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology (IAT'03)
0-7695-1931-8/03 $ 17.00 © 2003 IEEE

There are four main types of agents in this architecture:

- Manager Agent
- Resource Agent
- Task Agent
- Collecting Agent

2.1 Manager agent (MA)

The MA 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, has a MA. The MA receives tasks (parallel program to be executed) from the user through a GUI, 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 specified by the user.

To allocate the least busy nodes for the task, the MA sends requests to the resource agents, located on the hosts available to the system (described below). If the response performance value 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 which can be incremented or decremented by the user and is calculated based on the memory and CPU utilization of the system. In this way, the system randomly selects a set of nodes from the list of the prospective candidates (the least busy nodes) to participate in the process.

The MA retrieves physical addresses of the selected hosts from a local data structure called HostRegistry (HR) that contains tuples of the hostname and IP address for each node in the network. It then generates task agents and disperses them to the corresponding selected hosts. For the root node, MA generates a Local Task Agent (LTA).We talk more about MAs and LTAs when we describe the behavior of the system.

2.2 Resource agent (RA)

On receiving a request from the MA for node allocation, RA computes a performance value for the node on which it resides based on memory and CPU utilization and returns this value as a response to the MA.

2.3 Task agent (TA)

Created by the MA, the 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 a Transfer Handler Tree Structure (THT) for storing the received data, which is described below.

The TA initially creates a collecting agent to collect the incoming data from the sender threads of other TAs and places them in the local THT. It then begins the execution of its task code. On encountering certain communication function calls, such as `Send` in the task code, it generates the sub-task threads (discussed in the next paragraph). After the task code runs to completion, the task agent suspends the CA and sends the computed results back to the MA to be displayed to the user via the GUI. The TA is then terminated.

The sub-task threads are spawned by the TAs to perform `Send` related functions. The need for threads as a medium of communication between TAs arises in the case that a TA is delayed by the communication process. By spawning a separate thread to take care of the `Send` function, the execution of the code base and the communication can take place simultaneously thereby maintaining the characteristic parallelism of the system. When a TA needs to communicate with another TA, it generates a thread with the destination address and data to be sent. The thread looks up the destination from the local SHR to retrieve the physical address of the destination node. The `Send` thread, spawned by the source TA, passes the message to the remote 'putData' method of the destination CA and invokes it. This method puts the data in the THT structure. When a `Receive` function is encountered then the left most node's data is collected from the THT (deleting that node) and stored in the buffer. At the end of the process, the computed result is passed back to the MA.

Transfer Handler Tree (THT). This data repository class holds a tree data structure and synchronized member functions for its manipulation. Both the CA and the TA have access to the THT. In this structure, the root node at the top level is followed by a level of nodes, each representing a participating computer in the process (called the source node). Multiple receive requests for the same source node are identified by a tag or label at the second level. At the next level, each node represents the data sent by the respective source. The CA stores the incoming data in the tree data structure and then notifies the TA. The TA retrieves the data (and deletes the data node) by traversing the tree in pre-order (the left most node is read first). We refer the reader to figure 1 for the THT structure.

2.4 Collecting agent (CA)

Generated by the TA, the CA monitors the THT for synchronizing communication between TA's. The CA has access to the tree structure, requested by the TA threads.

The CA collects the data from the thread of the message-sender TA. It creates a child node holding the data and appends it to the corresponding parent source node of the tree structure, and also notifies the TA. At the end of the communication operation CA is terminated by its TA when the task agent completes execution.

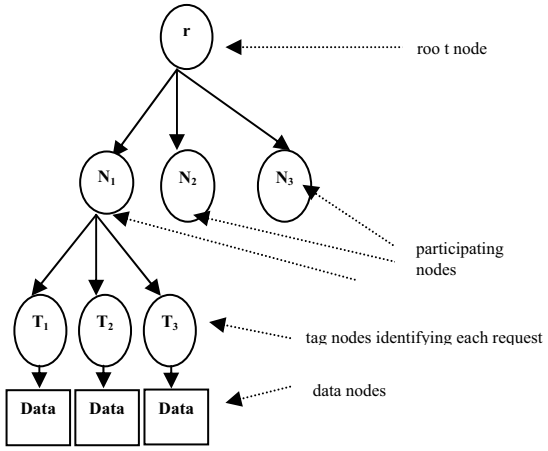


Figure 1. Transfer handler tree

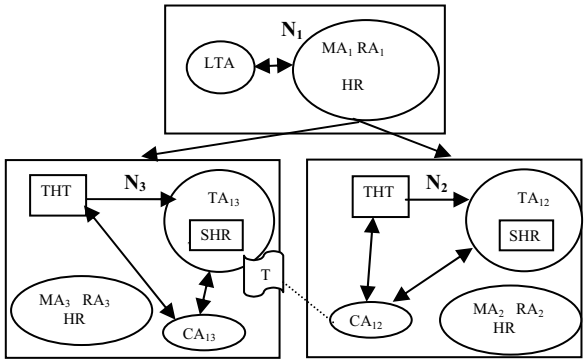
3. Behavior of the system

In this section, the behavior of the system for more common MPI functions is presented.

Scenario 1: Send-receive function call

Consider a scenario where a user runs a parallel program on node N_1 in the network. We refer the reader to figure 2 for the illustration of the scenario. The user interacts with the MPIAB GUI by passing the executable task code and the number of nodes to the system. The MA_1 sends a request for node allocation to the other nodes in the network. To respond to the request, the RA (created by the MA) on each node computes a performance value for the local node and sends the value to MA_1. Comparing these response performance values with the corresponding threshold, the MA selects a set of least-busy nodes for the process. Then, the MA generates a TA for each selected node (nodes participating in the parallel process) and writes the list of all the selected nodes to the SHRs of the TAs. The system also generates a LTA on the root node (N_1). The TAs, after reaching the destination nodes, binds with the local agencies. Each TA then generates a CA to mediate access to the data repository class THT and starts execution of its task code.

Consider the case when N_3 wants to send data to N_2 (N_2 doing a corresponding receive).On encountering a `Send` function call in its task code, TA_{13}[1] spawns a thread that holds the message to be sent and also the destination rank of the node to be sent to (obtained by referencing the SHR). The send thread invokes the CA's method to put the data in the THT. Then, TA_{12}, on finding the corresponding `Receive` function call in its task code, collects the data from the tree data structure (THT) based on the rank of the source computer. If the Receive is encountered in the code before the data is actually put in the THT by the `Send` thread, then the destination TA must periodically check the tree for the messages until the data is available.



CA: Collecting agent HR: Hostregistry LTA: Local task agent
MA: Manager agent N: Node RA: Resource agent
SHR: Subhostregistry TA: Task agent T: Thread
THT: Transfer handler tree

Figure 2.Send-receive communication scenario

Scenario 2: BroadCast function call

Broadcasts a message from the process with rank 'root' to all other processes of the group. Consider the MPI function `MPI_Bcast` as a case study which has the signature:

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

This function is modeled in MPIAB as follows:

On the 'root' node, when the TA encounters an `MPIAB_Bcast` function call, it generates threads for sending the data in 'buffer' to all the other nodes present in the network. In every other node (other than the root), when the TA encounters an `MPIAB_Bcast` function call, it retrieves the data from its THT.

[1] TA_{13} refers to the task agent of node 3 created by the manager agent of the node 1.

Scenario 3: Gather function call

Gathers values from a group of processes. Consider the MPI function `MPI_Gather` as a case study which has the signature:

```
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 modeled in MPIAB as follows:

When the TA on a node with rank equal to 'root' encounters an `MPIAB_Gather` function call, it collects data from the THT for all the participating nodes. On the nodes with rank other than 'root', the TAs, on encountering the same function, generate threads for sending data in 'sendbuf' to the 'root'.

Scenario 4: Scatter function call

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:

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

This function is modeled in MPIAB as follows:

For the 'root' node, the TA generates threads 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 THT with the source node equal to 'root' and 'recvbuf' as the receiving buffer of size 'recvcnt'.

4. Design and implementation specifications

The first version of MPIAB is being implemented on the Grasshopper platform. The implementation of the scenarios of "`Send`", "`Receive`", "`BroadCast`"and "`Gather`" is complete. As an initial step to develop the complete system, we have worked out a design strategy for send-receive communication scenario. This incorporates the key steps and the flow of control between the different modules of MPIAB.

A UML sequence diagram, depicting the flow of control between modules, is shown in Figure 3. This diagram describes the timing sequence of method calls between different classes.

5. Summary

In this paper, we discussed the design of MPIAB, an agent-based heterogeneous MPI environment for parallel processing. We have addressed two major concerns: The first regards effective utilization of system resources through the selection of the least busy nodes, and the second concerns the support for heterogeneous operability through the use of Java based agents. Agent technology has the potential to manage these complexities and to produce satisfactory results. Grasshopper environment is used for the implementation of MPIAB. The performance evaluation of separate modules of MPIAB looks promising; however, a thorough evaluation of the system will be performed upon completion of the implementation.

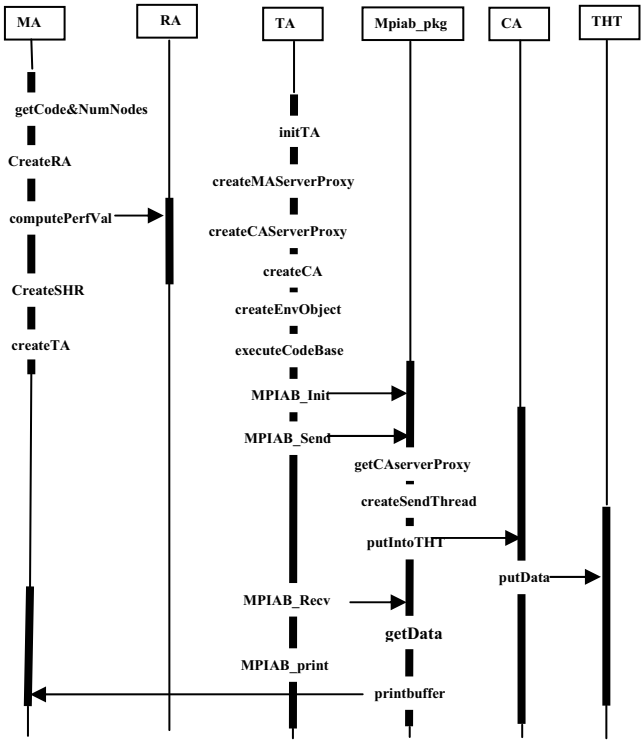


Figure 3. UML sequence diagram

6. References

[1] The MPI Forum.The MPIMessage Passsing Interface Standard , URL:http://www.mcs.anl.gov/mpi/mpi-report/mpi-report.html,Dec 1995.

[2] Hercock Ghanea-R ;Colis-J.C.;Ndumu-D.T," Co-operating Mobile Agents For Distributed Parallel Processing ".