A Platform for the Development of Parallel Applications using Java

Erick Pinacho Rodríguez, Darnes Vilariño Ayala

Benemérita Universidad Autónoma de Puebla, Facultad de Ciencias de la Computación
epinacho@gmail.com, darnes@cs.buap.mx

Abstract. The search for solutions for real problems generally involves a lot of calculations. This brings itself a great amount of time consumed when generating a response. The only solution for this problem is offered by Parallelism. By now, the hardware is not a trouble anymore. It has become more important the development of software tools that take advantage of the hardware possibilities.
One of the most widely used function library for developing parallel applications is Message Passing Interface (MPI) [1, 2]. This library has limits about memory handling, so the developer must handle complex synchronization mechanisms for using shared memory.
Java language offers upgrades about concurrency mechanisms and technologies for developing distributed applications [3, 4]. Then it is a good language to create a platform to develop parallel applications. Even today, there are no major efforts in the development of utilities to create parallel software on this language.
Our developed Platform is oriented for clusters or Local Area Networks (LAN). The design applied has considered slaves and a master process, which is in charge of controlling the tasks executed by each of the slaves. Both the master and slave processes have their own architecture. The master architecture is composed by three layers, while the slave architecture is composed by five layers.
For the communication between master and slaves inside the platform, a multicast User Datagram Protocol socket (UDP) is used. The communication is transparent to developer, who disposes of functions to send and receive messages to develop his applications.
This Platform offers a shared memory. This one is paged, so the amount of memory used is not a problem.
Remote Method Invocation (RMI), has permitted to distribute the classes between all of the nodes. This aspect is controlled by the master process, in order to guarantee an adequate work balance.

Keywords: Parallel Computing, Distributed Computing.

1 Introduction

The performance of the computers has increased exponentially from 1945 until now, on an average factor of 10 every 5 years [5]. The performance of a computer is

A. Gelbukh, S. Suárez, H. Calvo (Eds.)
Advances in Computer Science and Engineering
Research in Computing Science 29, 2007, pp. 202-210

Received 01/07/07
Accepted 19/10/07
Final version 24/10/07

directly dependant to the time required to perform a basic operation, and the number of basic operations that can be performed concurrently. It is not about faster processors at all to improve the computer performance.

Two important features are time and space, it means the number of steps that an algorithm requires -in the worst case scenario- to generate a result, and the memory required to solve the current problem [6].

The calculations over a network of computers -also known as “distributed computing”- are not only a subfield of the parallel computing. Distributed computing is deeply related with troubles such as reliability, security and heterogeneity that are considered as ___ in the parallel computing research.

A distributed system is where all of the computers that belong to the network must work as one. Once one of the computers that form part of the system fails, even when the user does not know which computers form this 'system', an application running may not be performed correctly [5].

The concurrency theory has been an active research field on the computer sciences, since the results presented by Carl Adam about the Petri Webs in 1960 [7]. Since then, a wide variery of theoretical and logical models and tools have been developed to understand the concurrent systems.

The clusters are gaining a wide area of the current applications, due to they offer acceptable performance levels for a relative low cost.

Regarding the software, there are a wide variety of tools that help to the programmer to develop distributed and/or parallel tools. Some of these tools are currently being standarized, while some others are gaining certain acceptance from the developer community.

A great part of the dedicated effort on the development of parallel or distributed applications is put over (a) the handling of structures or communication channels offered by the current tools; (b) the management of the platform resources; and (c) the control of events of the system, amongst others.

Particularly, and regarding the Java Language, those efforts to develop parallel or distributed tools are minimal. Some of these tools are on a mature stage, and the others are still under development.

Thanks to the creation of the Forum "Java Grande", the interest and effort has been focused on the standarization and development of more tools over the Java Language.

Nevertheless, already disposing of these tools, there are missing higher abstraction levels that ease the programming task to the developer.

More tools and applications are required; these tools must help to the developer to focus their efforts into solving the initial problem. These tools must also help to decrease the time dedicated to solve communication details, event control and management or resources.

The platform presented in this paper is executed over a cluster of computers with Linux as OS. The migrative nature of java allows to use any OS with the minimal adequations needed.

This platform is directed to the development of Java parallel applications that require the use of shared or distributed memory, with transparency to the user. In the parallel applications, the use of shared memory becomes an important problem. Another important advantage offered by this platform is the message passing, because the programmer can forget completely about the protocols for this matter.

The platform supports and manages the following tasks:

1. Support to the mpiJava implementation. The MPI libraries are not originally designed to support concurrence. The platform makes the programmer free of the concurrent handling of the MPI libraries.
2. Shared memory. The developer can use shared memory. The platform is in charge of maintaining the memory consistent and of the automatic delivery and refreshment of the data.
3. Support for a single model of pool threads for each of the execution nodes, considering also mechanisms that guarantee the good use and performance of the model.
4. Support for the delivery of packages -not related to the MPI- in a transparent manner to the user.
5. Support for the execution of N parallel tasks using M nodes of a cluster.
6. Management tasks for the nodes that form the network of the platform.

The paper is organized as follows: Section 2 describes the considerations made to the design and construction of the platform. Section 3 shows the architecture of the platform, particularly the master and slave daemons, and how they comunícate. Section 4 presents the implementation of the platform, the programmed structure of the messages sent on it, how to handle memory block and how to synchronize them. Section 5 shows the conclusions obtained during the development of this research project.

2 Design Considerations

The platform is executed over a network interconnected by a TCP/IP protocol. Each node has its own IP address on the same address range, in other words they form the same subnetwork. On execution time, the platform is formed by an instance of a class named MasterDaemon, and one or more instances of a class named SlaveDaemon.

When the MasterDaemon is uploaded, it opens a multicast socket, which is an UDP socket with the ability of joining to socket groups which are also multicast. In this kind of socket, when any of the participants sends a package, all the group -including the emitter- receives the same package. This is important and is done over again many times in the platform, specially on Shared Memory mode.

Once the remaining daemons are uploading, they will be joining to the multicast group.

It has been mentioned that more than one instance of the SlaveDaemon class can be uploaded inside the same node. For the platform effects, each of these instances will become a logical node added to the platform. When a daemon is uploaded, a whole instance of the Java virtual machine is created. When a second daemon is uploaded, the JVM is not completely created again, but a fraction of it. For each daemon subsequently uploaded, there will be a partial copy of the JVM. These actions consume mainly memory resources.

3 Architecture of the Platform

The planning and modeling phases for the platform has brought sustantiable benefits. The first one is the possibility of changes in the implementation of the services for the sublayers without affecting the superior layers or other services. The latter benefit is that new services can be added in any layer, enriching it and extending its capabilities.

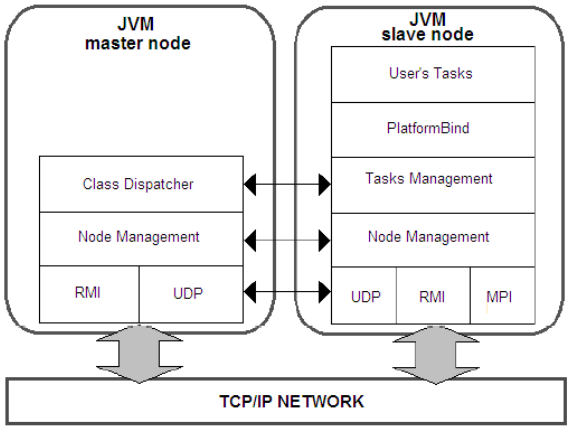


Fig. 1. Architecture of the daemons.

Master Daemon. The stack has 3 service layers:
1. Communication Channel Layer
2. Node Management Layer
3. Classes Dispatcher Layer

In the first layer the services that the master daemon uses to communicate with all the components of the platform are located. This layer is subdivided in two parts, UDP and RMI. The UDP part provides the service of message delivery and an easy structure to send them; both services are widely used in the entire platform. The RMI part offers the service of distributing the classes which are exposed thorough a remote interface.

The Node Management layer registers all the daemon processes that are uploaded in the platform. Also it makes polls on the nodes to know if they are available.

The Class Dispatcher layer makes the User Class Definition to balance the load. It uses the information of nodes to indicate to each node which classes to obtain and execute.

Slave Daemon. This stack has 5 service layers:
1. Communication Channel Layer
2. Node Management Layer
3. Tasks Management Layer
4. Platform Bind

5. Tasks of the user

The first layer is similar to the one in the Master Daemon; however this layer adds MPI services. The RMI part provides some facilities to obtain the master daemon classes. The MPI part has the MPI functions properly codified, solving the concurrency problems found in the normal library.

The Node Management layer creates the name of the node, also can request the name of each node.

The Tasks Management layer, as its name refers, treats the tasks received from the Master Daemon. This daemon indicates to each slave daemon which class obtains by assigning an ID for each task.

The PlatformBind acts as an interface for the Platform and the classes of the user. The layer provides communication services among tasks using messages with an UDP socket, passing messages with MPI, management of threads and the service of memory sharing.

The last layer performs the tasks of the user. Though the tasks are managed on the layer 3, in this layer they are executed. The distinction of layer 3 and 5 is about the services that each layer offers, such as putting the tasks in a pool. This brings up the extensibility concept.

4 Implementation

The implementation of the platform has been made in layers through the codification of the diverse classes. Next, the most important considerations for the development of each layer are presented.

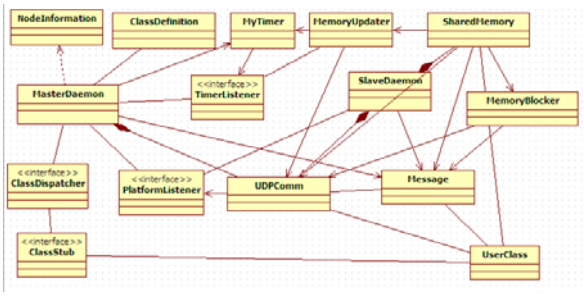
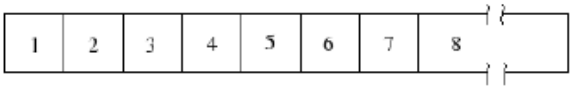


Figure 2. Class Diagram of the platform

Naming of the nodes. When a daemon is uploaded in a host, the daemon notifies to the platform that a new node has been created. A node is the instance of the SlaveDaemon class. For a user, the name of a node is not relevant, because the user does not need to know in which node its classes will be executed. However for the message passing, it is necessary to know the location of both sender and receiver. For this reason each node has a unique name in the platform. First the slave daemon takes its name directly from the node itself; if the name is localhost, the daemon renames the node on the application level. The name assigned will be NODE_XYZ, where

XYZ is a random number. If the selected name is in use, the Master Daemon notifies it and the slave tries to obtain a new name.

Message structure. A unique communication point is used by all of the functionalities of the platform, such as the memory, the platform management and the user messages. It is necessary that the message has a structure that allows recognizing the origin, destiny and purpose of the message. In the platform, the messages are represented by the class Message.



1. [0] Source Node
2. [50] Destination Node
3. [100] Source Task
4. [104] Destination Task
5. [108] Platform Tag
6. [109] Message
7. [113] Message Length
8. [117] Buffer

Figure 3. Message structure

If the field of the message Platform Tag indicates that the message comes from a user, the field Message may be used as desired. If the message comes from the platform, the field Message may have one of the following values:

1. DROP NET. This message indicates to all of the nodes of the platform to end the application.
2. ADD NODE. This message indicates that a node has registered in the platform network.
3. LIST NODES. This message is generated by the Master Daemon to advise to each node the list of available and registered nodes in the platform.
4. PING. This message is generated by the Master Daemon to poll to the nodes if they are still active.
5. PONG. This message is generated by a Slave Daemon in response to a PING message.
6. IP MASTER. This message is generated by the Master Daemon to inform the IP address of the master node to the slave nodes.
7. FETCH CLASS. It is a message to inform to the Slave Daemons the class that must be obtained to be next executed.
8. CLASS DEFINITION. This message is generated by the user to inform to the Master Daemon about the definition of the classes that will be used.
9. RENAME NODE. This message is generated once a node changes its name.
10. MEMORY MESSAGE. This message contains the changes made to the shared memory that should be informed to all of the nodes of the network.

11. BLOCKER MESSAGE. This message contains the addresses of the shared memory that have been blocked.
12. RELEASE MESSAGE. This message contains the address of the shared memory that have been liberated.
13. BARRIER MESSAGE. This message indicates that a node has been synchronized.
14. CLEAR BARRIER. This message indicates that the nodes have been liberated to continue their execution.

Communication sockets. To perform the communication through the platform, a UDP Multicast socket is employed. The use of these sockets brings simplicity and quickness to the protocol, which are not present in TCP, not even a Multicast implementation. Were TCP sockets employed instead of UDP in a network of N nodes, each node must create N-1 sockets to have communication with each of the remaining nodes of the platform. This justifies the use of UDP Multicast.

The Class-D IP addresses, which are between 224.0.0.0 and 239.255.255.255, are employed on multicast. Any application may use one of these addresses, and this assignment does not affect the address assigned by any network interface. With a correct address and a standard port UDP, the node can receive and transmit multicast messages.

Any delivery made in multicast will be received by any node which uses this socket.

Shared memory. The platform offers architecture with shared memory. Then, it is possible to offer to the developer a memory area common for all of the nodes of the platform, which is updated by replies. Each node possesses a full replica of the shared memory. This memory is designed to make pagination to the disc through small pages. The developer then disposes of a memory limited theoretically by the size of his hard drive. The data banks have an original size -configurable by the user- of 4096 bytes.

All of the instances of the user class that are found on the same node receive the same instance of shared memory. It is said that the shared memory offers concurrency in two levels, node and platform level.

The access to the shared memory is made through put*() and get*() methods, which allow to read and write primitive types of values to the memory -such as byte, int, float, double and byte arrays. All the addresses of the shared memory contain a byte, so the user shall be responsible of the movements needed to store primitive values which require more than one byte. The platform offers constants with the size of each of the primitive types. Similarly to the implementation of a hardware shared memory, the platform offers to block specific addresses. This functionality can be used as a basic synchronization mechanism. However several considerations about latency must be reviewed, because once a block is made, it will take some time before it is notified to all of the nodes of the platform.

Update and replicate. Once a process or task updates the shared memory of its node -by writing with the method put*()- this update must be notified to the other instances of the shared memory, so they know the new updated values.

The notification of these updates requires a great amount of messages that should be sent into the network. This provokes traffic in the network, resource consumption and extra time dedicated on attending each message.

To overcome this situation, the updates are made with delays. This is due to the writing process is regularly accompanied by another similar processes, rarely the writing is isolated. The late updates are controlled by a temporizer -with a time period configurable by the user. When a user updates the shared memory, the temporizer initiates a time counting. Once the count ends, all of the updates made on the memory are collected and then sent to all of the nodes of the platform. This mechanism reduces the network traffic and the consumption of resources. However, the execution time will be delayed and also the solution to the application of the user. This can be handled by modifying the temporizer time; when the time is reduced, the waiting time will be reduced, but the traffic will increase. It is decision of the user which factor to affect.

Address block. The shared memory allows the memory address block. Each time that an address is blocked, the block is registered with the ID of the task that made the block. From this moment, only this task may access to this address, to read or even write on it. Any other process which wants to read or write this address will be blocked until the address is liberated by the blocker task. The memory blocks are communicated through the whole platform with a message described previously. Unlike the messages that update content, the messages that indicate blocks are sent immediately when a task blocks an address, including also all of the messages prepared to be sent to the nodes -it means, this message acts as a trigger for the temporizer defined previously. With this, it is assured that the content is updated before making the block or unblock.

Regularly, the tasks will use the memory blocks as a manner to assure the concurrency, and as a manner to keep synchronization between them. This is why the block notifies or liberation of memory addresses is performed immediately, not delayed as the content update and replication.

Platform Synchronization. The synchronization points are common on the software tools that perform parallel or distributed tasks. A synchronization point consists into make all of the different processes to reach a point where they will expect the other processes to reach it. When all of the nodes reach the synchronization point, their execution will continue.

In the design of this platform, the synchronization points have been included to be programmed, and they are known as Barriers. These Barriers occur in two stages: the first stage synchronizes the node itself, because a node may be performing more than one task of the used; so the node will be synchronized once all of its performed tasks get the same defined point. The second stage synchronizes the entire platform, which will wait for all the nodes to reach the same defined point, once the first stage is reached for every node. The master node will be responsible of receiving all the notifications of each synchronized node. Once the notifications for each node are received, the master node will send the liberation message to all the nodes, and their execution will continue.

To achieve the synchronization, two classes of the concurrency tools offered by java have been employed: they are the *CyclicBarrier* and the *CountDownLatch*.

5 Conclusions

With the main objective of solving optimization problems on high scale by using parallel algorithms, a new platform has been developed. This platform is directed to take advantage of the cluster architecture.

One of the most popular tools in the scientific area is MPI. It is used to develop solutions by using a mechanism of message passing. However, a more natural model can be applied for some problems by using the concept of shared memory, which is not included in MPI. This problem has been completely solved with this platform.

This construction with shared memory has been developed by using concurrency mechanisms offered by the Java Language.

The facilities offered by the language for the creation of thread pools allows their management and reduces the traffic that implied the creation an destruction of threads. It has also support for the creation and application of policies about the execution of threads.

The utilities for concurrency of Java include an asynchronous queue. With this, it is not necessary to create critic zones or semaphores for the addition or substraction of elements, because a concurrency control is included in the same class. Asynchronous queues were employed in the platform for the message pass.

Some parts of the platform, especially in the shared memory area, include the semaphores concept of Java.

The disposal of this platform will allow to the students of the Faculty of Computer Sciences at the Benemérita Universidad Autónoma de Puebla, the development of concurrent, distributed and parallel applications through a unique programming language which requires only the classes and characteristics offered by the platform.

Acknowledgments. The heading should be treated as a 3rd level heading and should not be assigned a number.

References

1. Graham E. Faggs .Perfomance analysis of MPI collective operations. Cluster Computing. ISSN : 13867857. pag. 127-143. Kluwer Academic Publisher.
2. LAM/MPI Team at Open Systems Lab. LAM/MPI Users Guide. 2007.
3. David. D. *A Review: Java Packages and Classloaders*. 2001.
4. Goetz. B. *Concurrent Collection Classes in Java*. *IBM Home Page*. 2003
5. Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley. 1995.
6. Paul E. Black, *Algorithms and Theory of Computation*. Appearing in the Dictionary of Computer Science, Engineering and Technology. 2000. CRC. Press LLC.
7. Filman, Robert E, Daniel P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw Hill Higher Education. (1984)