AN EMPIRICAL STUDY OF EFFICIENCY IN DISTRIBUTED PARALLEL PROCESSING

DR. ROGER EGGEN
Department of Computer and Information Sciences
University of North Florida
Jacksonville, Florida 32224 USA
ree@unf.edu
904.620.2985
904.620.2988 fax

DR. MAURICE EGGEN
Department of Computer Science
Trinity University
San Antonio, Texas 78212 USA
meggen@trinity.edu
210.999.7487
210.999.7477 fax

ABSTRACT
Software development is proceeding at a remarkable rate. Many new tools are available to the researcher in parallel and distributed processing. These tools include PVM, MPI, CORBA, and Java. Since Java provides an object oriented programming environment, programmers have flocked to its use. This paper describes benefits of using the object-oriented approach with enhanced efficiency using RMI and CORBA. CORBA provides a language independent communication methodology while RMI requires Java to communicate with Java. This paper compares the benefits of Java to C++ communication using CORBA compared to Java to Java with RMI where a native method is used to improve efficiency.

KEY WORDS
JNI, RMI, CORBA, LAM, Parallel Programming Paradigms

1 INTRODUCTION
Computer scientists, along with software engineers, are enjoying a remarkable period of increased processing speeds and declining hardware prices. These new machines present unique challenges to the computing professional to develop software that adequately takes advantage of these computing systems. While techniques and opportunities exist to create sophisticated programs, there is an increasing supply of challenging problems requiring ever faster and more capable computers. The software developer can now afford clusters of efficiently networked machines providing new challenges to use parallel distributed systems.

The development of software tools to take advantage of fast, new hardware traditionally lags behind hardware production and development. However, new software programming tools have been implemented in an attempt to take advantage of the programming environments. Most notably, the object-oriented paradigm through Java is recognized as a major component of web based distributed programming. The Java Development Environment (JDE) includes several features which facilitate the production of stable, robust code for parallel processing. Threads provide an efficient and effective paradigm for utilizing tightly coupled systems, remote method invocation (RMI), and the common object request broker architecture (CORBA) provide convenient tools for utilizing distributed systems.

CORBA allows applications implemented in differing languages the ability to communicate while RMI is a strictly Java to Java feature. However, Java methods can invoke native C routines which will provide enhanced efficiency. Using native methods reduces the portability of Java, but efficiency is improved while remaining within the object-oriented paradigm. This paper is the result of research studying the efficiency and ease of use of CORBA and RMI using native methods. Included for a baseline comparison is a comparable algorithm using LAM.

2 HARDWARE
The hardware for these experiments consists of a cluster of homogeneous workstations all running RedHat Linux v7.0. The machines are all Intel based PCs consisting of single 500 MHz processors connected by 100 megabit fast ethernet.

3 SOFTWARE
The software for these experiments is Java (TM) 2 Runtime Environment, Standard Edition (build 1.3.0) available free from Sun. The CORBA tools are Iona Corporation's ORBacus free from Iona. LAM is free from the University of Notre Dame.

4 MESSAGE PASSING SYSTEMS
When a distributed cluster of workstations cooperates on the solution of a parallel application, a messaging system is involved. The details of the communication vary between software methods, but fundamentally messaging systems must all operate in a similar manner.

Messaging systems are subject to a common set of requirements in order to provide the capability to perform concurrent execution of processes.
The code in a process is inherently sequential. A process must be able to send and receive messages. A send operation should be synchronous. A receive operation should be asynchronous. A process should be able to perform dynamic task creation and allocation. Messaging should be able to be created and destroyed dynamically.

When a process, say process one, sends a message to a process, process two, the following must happen:

- Process one must prepare a send buffer.
- Process one must pack information into the send buffer.
- Process one must initiate a send.
- Process one sends the buffer.
- Process two receives the buffer.
- Process two unpacks the information from the received buffer.
- Process two performs the appropriate computations on the information.
- Process two prepares a send buffer.
- Process two packs the new information into the send buffer.
- Process two sends the buffer, etc.

Whether this procedure is handled implicitly by the programmer, as one might do using sockets or explicitly by the programming environment as in LAM, CORBA, and RMI, is irrelevant. The point is that the above steps must be done for communication to occur.

In general, we used 3 communication techniques to measure relative efficiency. The first communication technique is given to us by the University of Notre Dame's implementation of MPI. MPI is a C/C++ to C/C++ message passing set of functions which are efficient and widely used. This provides a base line of how fast our network and machines can communicate to solve a problem. Next, Java's RMI technique is used to pass data between Java methods. Since Java is roughly 10 times slower than C, a native method is invoked to do the processing. So, the communication between client and the servers is accomplished using Java's RMI, but to enhance efficiency the processing is done by invoking a native method implemented in C/C++. The client is threaded allowing asynchronous communication to each of the server processes. The third comparison is between a Java client and C/C++ servers using Iona's ORBacus CORBA. One of the advantages of CORBA is that different languages can communicate directly. Implementing the servers in C/C++ enhances efficiency over Java implementations, but retains the flexibility offered by CORBA. This paper studies and compares the relative efficiency of:

- LAM C/C++ client to C/C++ servers
- RMI Java client to Java servers, each server invokes a C/C++ native method
- CORBA Java client to C/C++ servers

The experiment consisted of generating a large set of integers. This set of integers was divided equally by the number of servers. Each set of integers was then sent to the respective server which sorted and returned the data. RMI is a Java to Java feature, thus to enhance efficiency, the server invoked native method implemented in C to do the sorting. Essentially, the Java server method called a C sorting routine, passing it the array of numbers. In the second phase of the research, Java passed the data via CORBA to a C function, which did the sorting. CORBA is reasonably language independent, thus Java can communicate directly with C. In the final phase of the research, LAM was used. LAM is strictly C to C, which we expect to be faster, but how these all compare is the research.

5 LOCAL AREA MACHINE - LAM

The University of Norte Dame has implemented MPI, the "standard message passing" interface. LAM is regarded as an efficient communication set of functions that is used as a benchmark for this study. LAM is relatively easy to use requiring understanding of 5 or 6 functions in the simplest form. LAM consists of over 100 functions which provide flexibility and robustness. We restricted the study to the basic functions commonly used. LAM is efficient, but rather restrictive in that it communicates strictly C or C++ to C or C++ or from FORTRAN to FORTRAN. LAM is reasonably easy to set up and use. There are six primary functions that are required to use LAM: MPI_Init, MPI_Comm_size, MPI_Comm_rank, MPI_Send, MPI_Recv, MPI_Finalize. After initializing the environment, MPI_Comm_size indicates the number of processes desired. MPI_Comm_rank returns a process identification. Communication is accomplished through the Send and Receive functions. Upon completion, the Finalize call cleans up the environment. LAM is a set of function calls allowing C/C++ to C/C++ or FORTRAN to FORTRAN communication. It is used in this study to provide a baseline of how quickly the problem can be solved. A client process distributes integers to several servers, each of whom sort and return their sorted portion of the data. The LAM environment is used strictly as a benchmark to determine relative efficiency of Java’s communication techniques.

6 JAVA MESSAGE PASSING - RMI

Java RMI is a high form of communication where the messaging is handled by the environment rather than explicitly by the programmer. Java remote method invocation establishes interobject communication that is fundamental to the Java model. Remote methods, after the environment is established, are invoked as if the
method was local. Thus, Java makes remote method invocation transparent to the user.

To invoke a method on a remote machine, stubs and skeletons are used to support the invocation. The stubs and skeletons are local code that serves as a proxy for the remote object. Fortunately, the programmer never has to work with stubs and skeletons directly. They are created by running the rmic – the RMI compiler. After compiling your Java server program and your Java client program, the rmic is executed on the Java files that define the interface between the remote methods.

One final piece of the puzzle must be introduced, the RMI registry. Your code must use the registry, which provides a look up reference to the remote object on the remote machine. An analogy may be wishing to send a letter to a friend. Before the letter can be sent, you must look up the address in your address book. RMI must be able to look up the remote object in the registry. The rmiregistry must be running on each of the server machines involved in the application.

On the server side, our remote Java method invokes a C++ method to process the work. A C method runs approximately 10 times faster than a corresponding Java method. Our study includes a client invoking several servers, each of which solves their portion of the problem by invoking a C/C++ method realizing enhanced efficiency. Implementing the client involved setting up the communication in corresponding threads, each of which communicate to an individual server. Each server is a Java method that invokes a C/C++ method. Surprisingly, the implementation, while reasonably complicated, was implemented with only minor complications and surprises. The sample programs will be made available upon request.

7 OBJECT REQUEST BROKER - CORBA

CORBA, like Java RMI, was developed so that different computer applications could work together over networks. Working with CORBA is similar to working with RMI, Java's own Object Request Broker (ORB). However, CORBA was designed so that any applications implemented in any language which adhere to the CORBA standard are able to communicate. CORBA translators are provided for C/C++, Java, Perl, Smalltalk, and even COBOL, to name just a few.

CORBA applications, like Java applications, are composed of objects. Fundamentally, distributed applications require the ability to request the invocation of a remote object, very similar to Java's RMI. CORBA is built around three building blocks:

- The Object Management Group Interface Definition Language, OMG IDL.
- The Object Request Broker (ORB).
- The Internet Inter-ORB Protocol (IIOP).

Each object type must have an interface defined in OMG IDL to indicate how the remote method is going to be invoked and the parameters required. The IDL file is at the heart of ability to invoke methods or functions from different languages. This Interface Definition must be independent of the implementation language of the object.

However, each of the popular programming languages have OMG standards by which they will be known. C, C++, Java, etc. all have been defined. The interface is defined very strictly, but the underlying implementation of the object is encapsulated, providing the ability for implementation in the language chosen by the programmer.

A client requests an object identified by the IDL stub which is the object's proxy in a similar manner as Java's RMI objects are referenced. The ORB handles the request, which is passed along to the server through an IDL skeleton, which invokes the object's implementation. In CORBA, every object instance must have its own unique object reference, which clients use to direct their invocations. In this way, different object instances are identified.

We note that in order for the object reference to be passed from client to server and returned, the data is marshalled, in the same way as in the Java RMI model. Thus, the CORBA model can be viewed as a message
passing system, as described earlier. ORBacus provides an IDL to Java (jidl) and an IDL to C (idl) translators, which create the appropriate stubs and skeletons, required for communication.

8 THE RESULTS

To evaluate software systems for performing distributed parallel processing, it is important to note that the programs used for the testing should be real applications. For this reason, the authors of this paper have chosen an experiment involving sorting. Since sorting is ubiquitous in many applications, it represents a problem with real world applicability.

Figure 1 gives the general configuration of the system. Figure 2 shows the relative efficiency of processing using 8 processors. Three programs were implemented in a consistent manner utilizing the features provided by each of LAM, RMI, and CORBA.

Data was distributed from the client to each of several servers. The servers processed the data and returned the results to the client as shown in Figure 1. The project was implemented in a threaded Java environment for the CORBA and RMI portion, allowing the client to communicate in a timely manner with each of the servers.

The research evaluates performance of LAM, RMI, and CORBA using 2, 4, and 8 servers over data ranges of 10,000 to 80,000 integers. The servers do all the work while the client distributes and receives data to and from the servers. All tests were executed under exactly the same conditions for each of the communication protocols. The table in Figure 3 summarizes the results.

From Figures 2 and 3 we see the implementations of RMI and CORBA provide very nearly the same performance. This is due in part to the way each communicate and the service each provides. CORBA averages 17% longer than RMI or sockets.

<table>
<thead>
<tr>
<th>COMMUNICATION and COMPUTATION TIME</th>
<th>Figure 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of servers</td>
<td>8</td>
</tr>
<tr>
<td>DATA</td>
<td>LAM</td>
</tr>
<tr>
<td>10000</td>
<td>0.20</td>
</tr>
<tr>
<td>20000</td>
<td>0.41</td>
</tr>
<tr>
<td>40000</td>
<td>0.83</td>
</tr>
<tr>
<td>80000</td>
<td>1.61</td>
</tr>
</tbody>
</table>
9 SUMMARY AND CONCLUSIONS

It is apparent that CORBA and RMI are consistently slower than LAM. This is to be expected since C/C++ is faster than Java. In all cases the computing time is dominated by the communication where doubling the amount of data doubles the time to solve the problem. Linear growth is typically acceptable when considering algorithm development. C/C++ is a constant amount faster than RMI where the constant varies between 13 and 20 depending on the number of processors (which effects the amount of data transferred to each processor). We see communication is dominant causing the problem solved to require large grain parallelism, but the modern object-oriented paradigms and communication techniques can effectively be used to do parallel distributed processing. The benefits of Object Oriented programming can be and probably should be realized when doing parallel distributed processing. Portability and flexibility continue to benefit application development in the parallel arena just as it does in the sequential program development. While Java communication is not as efficient as C, some of the processing speed can be regained through using native methods, as in this paper. In comparison to a similar study [4], invoking a native method in RMI yields a 100% to 138% increase in speed. The more data processed, the more speed up realized. For CORBA, processing the data with C rather than Java, again compared to the results in a study done in 2001 [4] yields the expected speed up 114% to 1.67%. Higher speed increases are again realized when more data is processed. We might see greater speed gains, but communication is still a major factor in efficiency.

The amount of work required to set up LAM is roughly equivalent to setting up RMI and using either involves simply function invocation. There is more to be done to communicate with the ORB in a CORBA application since the data interface is defined in IDL. IDL is a language requiring knowledge of syntax and data types. IDL is only an interface language therefore there is no execution component, but knowledge of data type mapping between IDL and C/C++ or Java is required. CORBA allows the flexibility of applications implemented in different languages to communicate.

For raw speed, one probably would not choose Java in the first place, but for ease of programming, Java is a very good choice. RMI is easy to program and requires a one time communication setup. CORBA is harder to program, but also requires a one-time communication setup and provides the opportunity to communicate between applications implemented in different languages. CORBA requires a minor understanding of IDL and experiences a slight performance hit from this generality.

10 FUTURE RESEARCH

There are other distributed parallel program environments, each with their own advantages and disadvantages, that should be considered. Two of these are PVM (parallel virtual machine) and MPICH (message passing interface alternate implementation).

There are a variety of implementations of the CORBA standard, each requiring a slightly different application programming interface. Some of these are Visibroker, Java2 version 1.3, Orbix and OMNIORB. It would be interesting to determine which of these implementations are most efficient and characteristics required for their usage.

JavaSpaces provide a Linda like environment. Efficiency considerations of JavaSpaces in comparison with the above techniques should be studied.

11 REFERENCES


