SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 8, Number 2, pp. 209–226. http://www.scpe.org
ISSN 1895-1767

A PERFORMANCE AND PROGRAMMING ANALYSIS OF JAVA COMMUNICATION MECHANISMS IN A DISTRIBUTED ENVIRONMENT

SHAHRAM RAHIMI, MICHAEL WAINER AND DELANO LEWIS*

Abstract. Distributed computing offers increased performance over single machine systems by spreading computations among several networked machines. Converting a problem to run on a distributed system is not trivial and often involves many trade-offs. Many higher level communication packages exist but for a variety of reasons(portability, performance, ease of development etc.), developers may choose to implement a distributed algorithm using one of the four Java API communication mechanisms (RMI with Serializable or Externalizable Interface, socket and datagram socket). This paper provides a performance programming complexity analysis of these communication mechanisms based upon experimental results using well known algorithms to provide data points. Numerical results and insights offer guidance towards understanding the communication and computational trade-offs as well as the programming complexities encountered when adapting an algorithm to a distributed system.

1. Introduction. The purpose of this paper is to provide a detailed performance and programming complexity analysis of four Java network communication mechanisms. These four mechanisms are: RMI using the *Serializable* interface and the *Externalizable* interface, the socket, and datagram socket. Higher level distributed communication packages have been created by others, but they in turn must also use one the four basic communications mechanisms in their implementations. Our emphasis here is on practical experimental data which can provide insights on the appropriateness of each of these mechanisms. As this is the case, rather than create or utilize a formal model, we select well known algorithms to implement and collect experimental data on. The algorithms themselves are not being compared but instead are used as a way of comparing different Java communication mechanisms and their impact upon distributed algorithms.

To help illuminate the trade-offs involved in finding the most efficient communication mechanism, the well known problem of matrix multiplication is selected. Multiplication of large matrices requires a significant amount of computation time as its complexity is $\Theta(n^3)$, where n denotes the dimension of the matrix. Because the majority of current scientific applications demand higher computational throughputs, researchers have tried to improve the performance of matrix operations (such as multiplication). Strassen's algorithm, devised by Volker Strassen in 1969, reduces the complexity of matrix multiplication to $\Theta(n^{2.807})$. Even with improvements provided by algorithms such as Strassen's, performance improvement has been limited. Consequently, parallel and distributed matrix multiplication algorithms have been developed of which two, a simple parallel algorithm (SPA), and a modified Cannon's algorithm (MCA) [14] are implemented in this paper (more on the appropriateness of this selection is given later in the paper).

The rest of this paper is as follows. Section 2 provides background introducing distributed computing on the Java platform, and the specific distributed matrix multiplication algorithms implemented in this paper. In section 3, the experimental setup and implementation are discussed. Section 4 provides an extensive performance analysis of the experiment results. Finally, the overall summary and conclusion are given in section 5. An appendix contains additional code listings.

2. Background. This work is first put into perspective in comparison to other related works. Next we review the basic elements of Java distributed programming discussing the four basic communication mechanisms two of which are variants of Remote Method Invocation (RMI) and two which are variants of sockets. Lastly we provide details of the matrix multiplication algorithms used in our experiments.

2.1. Comparison to Previous Work. Others have sought to improve Javas distributed computing performance by focusing on more efficient RMI implementations or better serialization techniques. For example, Maassen et.al. designed Manta, a new compiler-based Java system, to support efficient RMI on parallel computer systems [1]. Manta utilizes information gathered at compile time to make the run time protocol as efficient as possible [1]. Although Manta supports efficient RMI, it does not feature an efficient RMI package. Manta is based on a transparent extension of Java for distributed environments. Unfortunately, the RMI is part of that environment and cannot be used as a separate RMI package.

Furthermore, Fabian Breg and Constantine D. Polychronopoulos determined object serialization as a significant bottleneck in RMI [2]. They introduced "a native code implementation of the object serialization protocol" that resulted in performance improvements. Far more efficient implementations can be obtained when performing object serialization at the JVM level [2].

Research devoted to the improvement of Java's distributed performance, whether it involves new mechanisms or implementations, tends to focus on the problems caused by the generic marshalling and demarshalling of the serialization

*Department of Computer Science, Southern Illinois University, Carbondale, Illinois 62901.

algorithm. We discuss the *Externalizable* interface which eliminates the generic marshalling and demarshalling of the serialization algorithm but do not introduce any new techniques or implementations to the Java distributed environment. Our work examines the performance of existing communication mechanisms inherent to the Java platform rather than proposing new packages, implementations, etc., This is important to programmers who wish to obtain the best possible performance without sacrificing the user-friendliness, portability, ease of deployment, and other advantages offered by the Java programming language.

2.2. Java Remote Method Invocation (RMI). RMI is the distributed object model used for remote procedure calls. A remote procedure call mechanism allows amethod or function to be executed on a remote machine as if it were local. The syntax of remote method invocations is very similar to local method invocations. RMI also implements distributed garbage collection thereby removing the burden of memory management. All of these features make RMI development simple and natural, an excellent decision for developing 100 percent Pure Java client/server, peer-to-peer, or agent-based applications [7]. Unfortunately, RMI implementations tend to have a major Achilles' heel, communication overhead. RMI is designed for Web based client/server applications, where network latencies on the order of several milliseconds are typical [8].

Implementations using RMI must have a mechanism to convert objects to and from byte streams so that they may be communicated over a network. Two variants are possible, the more automatic *Serializable* Interface method or the *Externalizable* Interface technique which allows for more control.

2.2.1. Serializable Interface. The first communication mechanism examined for this project is RMI using the *Serializable* interface. Serialization is a "generic marshalling and demarshalling algorithm, with many hooks for customization" [9]. It is the process used to convert object instances into a stream of bytes for network transfer. There is also a deserialization mechanism which reverses the process. Some of the issues handled by serialization are: byte-ordering between different architectures, data type alignment, and handling of object references.

Serialization is a significant source of overhead in existing RMI implementations [1]. Its three main performance problems are: "it depends on reflection, it has an incredibly verbose data format, and it is easy to send more data than is required" [9]. Since it is a generic method, with hooks for customization, it has a tendency to be both inefficient and bandwidth-intensive. Declaring a class *Externalizable* helps solve some of the performance problems associated with making a class *Serializable*.

2.2.2. Externalizable Interface. RMI using the *Externalizable* interface is the second communication mechanism considered in this project. Implementing the externalization mechanism is more difficult and less flexible than the serialization mechanism. The Java *Externalizable* interface requires two methods to be created, *readExternal()* and *writeExternal()* (playing similar roles to the *readObject()* and*writeObject()* in the *Serializable* interface). This code, containing the specific marshalling and demarshalling instructions, must be rewritten whenever changes to a class's definitions are made. Because the programmer has complete control over the marshalling and demarshalling process, all of the reflective calls associated with the generic serialization mechanism may be eliminated dramatically improving performance [9].

2.2.3. Serializable Interface vs. Externalizable Interface. The *Serializable* interface can be used by stating that a class implements *Serializable* and creating a zero-argument constructor. In addition, the serialization mechanism adapts automatically to changes in the application. Generally, all that is required is a recompilation of the program. To support this ease of use, it is necessary for the serialization mechanism to always write out class descriptions of all the *Serializable* superclasses [9]. Furthermore, the serialization mechanism always writes out the data associated with the instance when viewed as an instance of each individual superclass [9].

Externalization can be made to avoid much of this overhead by shifting the burden of specifying the marshalling and demarshalling details to the programmer. This can be seen by comparing code for the methods *readObject()*, *writeObject()*, versus *readExternal()*,and *writeExternal()*. The two write methods are responsible for writing a remote object to a stream, while the read methods are responsible for reading a remote object from a stream. Code listings illustrating these approaches are given in the appendix.

Fig. 5.1 (in appendix) illustrates the use of *Serializable* Interface. Notice the *readObject()* and *writeObject()* methods are not implemented. This is because all the locally defined variables are either primitive data types or serializable objects. Therefore the serialization mechanism will work without any further effort by the programmer. This makes distributed programming much easier but less efficient.

The *Externalizable* interface implementation is shown in Fig. 5.2 (in appendix). Methods *readExternal()*, and *writeExternal()* are required by the interface. The method, *writeExternal()* systematically writes each object to a stream while the corresponding *readExternal()* method reads in each object in the exact order as it was written. In this implementation,

it is the programmer's responsibility to forward the size of the matrices. Thus distributed programming using the *Externalizable* interface requires more effort on the part of the developer but ideally will compensate by increasing performance significantly.

2.3. Java Socket Communications. The third and fourth communication mechanisms examined for this project are based upon sockets. Sockets first appeared on Unix Systems as an abstraction for network communication in the mid-1970s. Since that time, the socket interface has become a foundation in the field of distributed programming.

There are two communication mechanisms that can be used for socket programming: stream communication and datagram communication. The stream communication protocol is known as TCP (transfer control protocol). TCP is a connection-oriented protocol in which a connection must first be established between a pair of sockets. It is important to note that sockets are used by both the RMI mechanisms.

2.3.1. Java Sockets (stream communication). The java.net package defines the classes and interfaces used to support stream communications with sockets. *Socket* and *ServerSocket* are the two Java classes used when reliable communication between two remote processes is required. The *Socket* class provides a single connection between two remote processes. The *ServerSocket* class is responsible for the initial connections between a client and server. The server socket listens for a connection request while the client asks for a connection. Once two sockets have been successfully connected they can be used to transmit and receive data in one or both directions.

2.3.2. Java Datagram Socket. The fourth communication mechanism examined for this project is the datagram socket. The datagram socket uses a connectionless protocol, known as UDP (User Datagram Protocol). Data are transmitted in discrete blocks called packets. Packets may either be fixed or variable in length. The network determines the maximum size of the packet therefore large data transmissions maybe divided into multiple packets.

In the datagram method, routes from the source machine to the destination machine are not created in advance. Each packet transmitted is routed independently of previous packets thereby allowing the datagram method to adapt to changing network conditions. Since packets may take different routes, each packet must include the source and destination addresses. It is important to note that it is possible for packets to be lost or arrive at the destination machine out of order due to their ability to take different routes.

The Java datagram socket implemented within this project required the use of a packet synchronization scheme to prevent the loss of packets. As the size of the matrices surpassed 500×500, the frequency of lost packets increased. Because lost packets caused the matrix multiplication application to lock up, an acknowledgment mechanism was utilized. After a packet is sent out, no other packets are transmitted until an acknowledgment packet is received. To prevent deadlocks or indefinite waiting due to lost packets, a timer counts down until it reaches zero. If an acknowledgement packet does not arrive before it reaches zero, a duplicate packet is retransmitted in hopes that it will reach its destination. Obviously, synchronization overhead may seriously affect the overall performance of this method.

2.4. Matrix Multiplication. Matrix multiplication is one of the most central operations performed in scientific computing [12]. Multiplication of large matrices requires a significant amount of computation time as its complexity is $\Theta(n^3)$, where n denotes the dimension of the matrix. More efficient sequential matrix multiplication algorithms have been implemented. Strassen's algorithm, devised by Volker Strassen in 1969, reduces complexity to $\Theta(n^{2.807})$. Even with improvements such as Strassen's algorithm, performance is limited.

A parallel implementation of matrix multiplication offers another alternative to increase performance. Matrix multiplication is composed of more basic computations which makes it an appropriate vehicle for evaluating the performance of the four Java communication mechanisms analyzed in this paper. Two well known parallel matrix multiplication algorithms were selected for implementation: a simple parallel algorithm (SPA), and a modified Cannon's algorithm (MCA). These algorithms help to illustrate the trade-offs required when implementing distributed algorithms.

2.4.1. Parallel and Distributed Implementations. When implementing an algorithm across multiple processors it is important to define the terms, manager process and worker process as these are used in the discussion of the matrix multiplication algorithms. Generally speaking, the manager process issues commands to be performed by the worker process. In this context, the manager process contains the graphical user interface (GUI) and is responsible for the issuing of matrix multiplication requests, partitioning of the input matrices, transmission of the submatrices, reception of the results, and time keeping duties. The worker process accepts matrix multiplication commands from the manager process, performs all of the actual matrix calculations, and returns the results, in the form of submatrices, to the manager process.

2.4.2. Simple Parallel Algorithm (SPA). Consider two input matrices, *A* and *B*, and the product matrix *C*. The value in position *C[r, c]* of the product matrix is the dot product of *row r* of matrix *A* and column *c* of the second matrix *B*,

given by the summation equation:

$$C[r, c] = \sum_{k=1}^{N} A[r,k] \times B[k,c] \tag{2.1}$$

The simple parallel algorithm used in this paper merely partitions this idea among multiple processes. Consider the following scenario, two square matrices *A* and *B* with dimensions $n \times n$, to be multiplied in parallel using *k* worker processes. First, matrix *A* is divided into *k* submatrices by the manager process. Second, each worker process then receives a submatrix of *A*, and the entire matrix *B* from the manager process. Next, sequential multiplication is performed to compute an output submatrix. Finally, each worker process returns its output submatrix to the manager process forming the resulting product matrix.

This algorithm has *p* iterations where $p = \frac{n}{k}$. During an iteration, a process multiplies a $(\frac{n}{p}) \times (\frac{n}{p})$ block of matrix *A* by a $(\frac{n}{p}) \times n$ block of matrix *B* resulting in $\Theta(\frac{n^3}{p^2})$. Therefore the total computation time is $\Theta(\frac{n^3}{p})$.

Upon examination of this algorithm, it is clear that a significant amount of data is transmitted during each communication call. Initially,$(n \times n) + \frac{n \times n}{k}$ data is transferred to each worker process. After each worker process calculates its output submatrix, $\frac{n \times n}{k}$ data is returned to the manager process. Thus the total amount of data transferred is:

$$k[(n \times n) + \frac{n \times n}{k} + \frac{n \times n}{k}] = k(n \times n) + 2(n \times n) = (n \times n)(k + 2) \tag{2.2}$$

In addition, this algorithm performs one communication call to each of the *k* worker processes, and each worker process performs one communication call to the manager process for a total of *2k* calls. Thus the average amount of data transferred per communication call is:

$$\frac{(n \times n)(k + 2)}{2k} \tag{2.3}$$

This assumes that there is no broadcast mode that would enable matrix A to be broadcast to all the worker processes simultaneously. Unfortunately, Java RMI does not support broadcasting [13].

2.4.3. Modified Cannon's Algorithm (MCA). The previous multiplication algorithm not only transfers large amounts of data per communication call, but also requires a significant amount of memory. Cannon's algorithm was developed to be a memory-efficient matrix multiplication algorithm [14]. The idea behind this algorithm is as follows: two matrices *A* and *B* are partitioned into square blocks and transferred to *k* worker processes. If a schedule can been forced, so that after each submatrix multiplication these partitioned blocks can be rotated among the worker processes, then each process contains only one block of each matrix at any given time [14].

The modified Cannon's algorithm used in this paper differs in two distinct ways from the traditional Cannon's algorithm. The first difference concerns the initial pre-skewing of matrices *A* and *B* which is now done entirely within the manager process. This change was made to help decrease the average amount of data transferred per communication call. The second difference is that only one dimensional submatrix blocks are transferred between the worker processes (rather than square submatrices). This change was made to increase the total number of communication calls between the worker processes.

Both of these changes appear to negatively affect distributed computing performance. Why were they implemented? The purpose of this paper is to provide a performance analysis of four Java network communication mechanisms using two parallel matrix multiplication algorithms. As discussed earlier, the first parallel algorithm transmits a large amount of data per communication call, but requires only a small number of communication calls. To get a better understanding of the performance characteristics of each communication mechanism, it seems logical to contrast the first algorithm with one that transmits a small amount of data per call, but requires many calls. This is the reason why the above changes were made to Cannon's algorithm.

Consider the following scenario, two square matrices *A* and *B* with dimensions $n \times n$, to be computed in parallel using one manager process and *k* worker processes. First, matrices *A* and *B* are pre-skewed within the manager process. Second, matrices *A* and *B* are partitioned into square blocks and one block from each matrix is transferred to each worker process. Third, each worker process performs matrix multiplication (by element) on the square blocks to obtain an output submatrix *C*. Next, communication between the worker processes begins as each process sends the first column of submatrix *A* to its left neighbor and sends the first row of submatrix *B* to its top neighbor. At the same time each process receives a new column and row from its right and bottom neighbors respectively.

TABLE 2.1
Simple Parallel Algorithm vs. Modified Cannon's Algorithm (k=4, n=500)

Parallel Algorithm	Total Number of Calls	Average Amount of Data per Call	Total Number of Calls	Average Amount of Data per Call
Simple Parallel Algorithm	$2k$	$\frac{(n\times n)(k+2)}{2k}$	8	187,500 elements
Modified Cannon's Algorithm	$2k + 2k(n-1)$	$\frac{3(n\times n)+2kt(n-1)}{2k+2k(n-1)}$	4000	437 elements

At this point, each worker process shifts its *A* and *B* submatrices to include the newly received row and column and discards the row and column that were transferred. Again, each worker process performs matrix multiplication to obtain an updated product submatrix *C*, followed by another round of data exchange and matrix multiplication. This cycle is performed a total of $n-1$ times. Finally, each worker process returns its product submatrix to the manager process to form the resulting product matrix.

This algorithm has $\sqrt{p}$ iterations where $p = \frac{n\times n}{k}$. During an iteration, a process multiplies a $(\frac{n}{\sqrt{p}}) \times (\frac{n}{\sqrt{p}})$ block of matrix *A* by a $(\frac{n}{\sqrt{p}}) \times (\frac{n}{\sqrt{p}})$ block of matrix *B* resulting in $\Theta(\frac{n^3}{p^{\frac{3}{2}}})$. Therefore the total computation time is $\Theta(\frac{n^3}{p})$.

Let's examine the communication characteristics of the modified Cannon's algorithm. Initially, $\frac{(n\times n)}{k} + \frac{(n\times n)}{k}$ data is transferred to each worker process. Let $t = \sqrt{\frac{(n\times n)}{k}}$ denotes the row and column sizes of the *A*, *B*, and *C* submatrices. After each worker process calculates its first submatrix, a total of $t(n-1) + t(n-1)$ data are transferred during the worker process communication phase. Finally each worker process returns $\frac{n\times n}{k}$ to the manager process. Thus the total amount of data transferred is:

$$k[\frac{2(n\times n)}{k} + 2t(n-1) + \frac{(n\times n)}{k}] = 3(n\times n) + 2kt(n-1),\ where\ t = \sqrt{\frac{(n\times n)}{k}} \tag{2.4}$$

In addition, this algorithm performs one communication call to each of the *k* worker processes. Each worker process performs $2(n-1)$ communication calls during the worker communication phase and one call to the manager process to return its result. This results in a total of $2k + 2k(n-1)$ calls. Thus the average amount of data transferred per communication call is:

$$\frac{3(n\times n) + 2kt(n-1)}{2k + 2k(n-1)} \tag{2.5}$$

2.4.4. SPA vs. MCA. Recall, the simple parallel and modified Cannon's algorithms presented in this section were chosen because of their distinct characteristics with respect to the total number of communication calls required and the average amount of data transferred per communication call. The simple parallel algorithm transmits a large amount of data per communication call, but requires only a small number of communication calls. Conversely, the modified Cannon's algorithm transfers a small amount of data per communication call, but requires a large number of communication calls.

To better demonstrate the differences between these two algorithms, Table 2.1 is provided for $k = 4$ and $n = 500$.

3. Experiment Implementation. As was mentioned before, the purpose of this paper is to provide a performance and programming complexity analysis of Java communication mechanisms using matrix multiplication as the experimental framework. Experiments were performed using five identical computers connected to a 100 MHz LAN (the specifications are given in the following subsection.) One computer acted as the manager process, while the remaining four computers performed as worker processes (k = 4). The manager process was responsible for partitioning the matrices, transmission of the submatrices, reception of the results, and time keeping duties. The worker processes performed all of the actual calculations. The data matrices used in this experiment contained randomly generated integers with values ranging from 0 to 20. The size of the matrices multiplied ranged from 100×100 to 1000×1000 in increments of 100. The data reported in this paper represent an average from five trial runs.

3.1. Settings. For the socket implementation, buffered input and output streams were used. For the datagram socket implementation, a proper packet size had to be determined. Packet size has an important impact on system efficiency. Smaller packet sizes allow for improved pipelining, but have a higher header overhead due to a fixed header size. Larger packet sizes have a lower header overhead, but have worse pipelining as the number of hops increases. Because the maximum number of hops was limited to one due to the LAN topology, a larger packet size was selected. The Internet Protocol restricts datagram packets to 64KB [9].

```
try
{
    theManager.dataNodeArray[index].getWorkerInterface().requestMUL(theMULNode);
}

catch (Exception e)
{
    System.out.println("Serialization Exception " + e);
}
```

FIG. 3.1. *Serialization Implementation*

Since the maximum allowable packet size is 64 KB, a data packet size of 63 KB (allowing for overhead) was chosen for this experiment. This does not imply that every packet transmitted was 63 KB. It only specifies an upper bound on a packets size. Consider the scenario where two 100×100 matrices are to be multiplied using the modified Cannon's algorithm.

After the initial pre-skewing, each of the four worker processes receives two 25×25 matrices or 5 KB of data. Obviously it would be a tremendous waste of time and bandwidth to send a 63 KB packet with 5KB of useful data. Therefore the datagram packet generation code (Fig. 3.5) determines the appropriate size of a data packet. If the size of the data to be transmitted is 5 KB, then the packet size will be 5 KB. If the size of the data is 75 KB, then a 63 KB and a 12 KB packet will be sent. This applies to all forms of communication, whether it is between a manager process and worker process or between two worker processes.

In terms of the computer specifications, each computer had an Intel Pentium 4 clocked at 1.7 GHz, 512 MB of RAM, Microsoft Windows XP Professional, and Java version 1.4.2. In addition, like many computers connected to a LAN, these five computers had typical programs running in the background such as virus and client programs. These background programs were not disabled during the experiment. The manager process computer used an initial heap size of 64 MB and max heap size of 96 MB. The four worker process computers had their initial and max heap size set to 64 MB.

3.2. Comparison of Implementation Difficulty. To aid in the programming complexity analysis, a series of coded examples are included. All of these examples involve a data transmission from the manager process to a worker process during the execution of the simple parallel algorithm. Remember with the simple parallel algorithm, each worker process receives one full matrix and a submatrix.

The least difficult communication mechanism used in the design and development of this distributed application was RMI using the *Serializable* interface. Since the syntax of remote method invocations is very similar to that of local method invocations, RMI shields Java programmers from low level communication concerns. In addition, since RMI uses the *Serializable* interface to automatically handle the marshalling and demarshalling of remote objects, implementing the simple parallel and modified Cannon's algorithms in a distributed environment becomes a straight forward process. Fig. 3.1 shows the actual code to invoke a remote method using serialization.

Basically two lines of code are all that is required to send the necessary data to a worker process. The portion, *theManager.dataNodeArray[index].getWorkerInterface()*, specifies the particular worker process to communicate with. The portion, *requestMUL(theMULNode)*, identifies the specific remote method to be invoked. In this case, the remote method is a multiplication request with one argument of type class *MultiplyDataNode*(see Fig. 5.1 in appendix).

RMI using the *Externalizable* interface was only slightly more difficult to implement when compared to the *Serializable* interface. The added complexity came as a result of the burden placed on the programmer to implement the *readExternal()*, and *writeExternal()* methods. In this scenario, converting object instances into a stream of bytes and vice versa was no longer the responsibility of Java, but the programmer's. Fig. 3.2 displays the code to invoke a remote method using externalization. The only difference lies in the programmer's responsibility to code the *readExternal()*, and *writeExternal()* methods (see Fig. 5.2 in appendix). .

The third most difficult communication mechanism to work with was the socket implementation. The socket personifies low level communication. Similar to RMI using the *Externalizable* interface, object instances must be converted to and from a stream of bytes by the programmer. Dissimilar to RMI using the *Externalizable* interface, remote method invocations are not supported. Therefore extra work is needed to properly encode and decode remote methods. Fig. 3.3 illustrates the socket implementation.

```
try
{
    theManager.dataNodeArray[index].getWorkerInterface().requestMUL(theMULNode);
}

catch (Exception e)
{
    System.out.println("Externalization Exception " + e);
}
```

FIG. 3.2. *Externalization Implementation*

```
try
{
   theManager.socket[i] = new Socket(theManager.dataNodeArray[i].getWorkerHostName(),
                                     theManager.dataNodeArray[i].getWorkerPortNumber() + 1);

   theManager.iStream[i] = theManager.socket[i].getInputStream();
   theManager.oStream[i] = theManager.socket[i].getOutputStream();

   theManager.in[i] = new DataInputStream(newBufferedInputStream(
                                theManager.iStream[i]));

   theManager.out[i] = new DataOutputStream(new BufferedOutputStream(
                                theManager.oStream[i], bufferSize));

   theManager.out[i].writeInt(i);
   theManager.out[i].writeInt(command);
   theMULNode.writeExternal(theManager.out[i]);
}

catch (Exception e)
{
   System.out.println("Socket Exception " + e);
}
```

FIG. 3.3. *Socket Implementation*

The first line of the code creates a stream socket and connects it to the specified port number at the specified IP address. The next four lines of code create buffered input and output streams. After the I/O streams have been initialized, the process number and job command are written to the socket. The process number is used for record keeping purposes. The job command however is very important. Because remote method invocations do not exist, a worker process does not know what method it should perform. The job command provides the necessary information. The last line of code writes the class, of type *MultiplyDataNode*, to the socket. This method is nearly identical to the *writeExternal()* method discussed in the previous section. The only distinctions are a different argument and the inclusion of a flush command.

The datagram socket was the most difficult communication mechanism to implement since it uses a connectionless protocol, which by definition is unreliable. Thus to ensure reliability, some form of packet synchronization is required. Introducing packet synchronization involves the design and coordination of multithreaded processes to read packets as they become available and deal with timeout issues. Fig. 3.4 highlights the difficulty involved with the datagram socket implementation.

The first method, *setupDataTransmission()*, creates an output stream in which data may be written into a byte array. The byte array automatically grows as data is written to it. The next line of code writes the class, of type *Multiply-*

```
try
{
    setupDataTransmission();
    theMULNode.writeExternal(theManager.out[clientIndex]);
    sendData();
}

catch (Exception e)
{
    System.out.println("Datagram Socket " + e);
}
```

FIG. 3.4. *Datagram Socket Implementation*

DataNode, to the newly created byte array. As before, this method is very similar to previous methods. The challenge of implementing the datagram socket lies in the *sendData()* method as shown in Fig. 3.5.

The first step is to calculate the total number of packets needed to transfer the required data. Once inside the *while* loop, a single packet is created and sent during each iteration. Once a worker process receives a data packet,it will send an acknowledgment. An important note, at this point another thread is running with the sole purpose of reading acknowledgments from the receiving worker process. After an individual packet is transmitted, a new thread of type class *PacketTimer* is created. This thread acts as a timer and counts down to zero. If the timer reaches zero, a duplicate packet is sent, the timer is reset, and the count down starts again. Each time it reaches zero, a duplicate packet is transmitted.

The variable *controlPacketReceived* is a semaphore with an initial value of zero. The line, *controlPacketReceived.p()*, is a wait operation. Therefore this process halts and waits until the listening thread receives an acknowledgment. Once an acknowledgment is received, a signal operation is performed on the semaphore, and the halted process continues execution. At this point the timer thread is destroyed and if there are more packets to send, execution of the next iteration begins.

Looking at Fig. 3.5 again, the method *getTotalPacketNum(int)* looks at the data to be transmitted and calculates the total number of packets needed. The method *getBufferSize(int)* determines the correct packet size by looking at the total number of packets needed. If only one packet is needed, then a packet size is created that matches the size of the data to be sent. If more than one packet is needed, then a 63KB packet size is chosen during the current iteration.

4. EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS. First, the performance of the four Java communication mechanisms, when using the simple parallel algorithm, is considered. Recall the simple parallel algorithm transmits a large amount of data per communication call, but requires only a small number of communication calls. The results are shown in Fig. 4.1.

Fig. 4.1 shows the socket implementation provides the best performance for the simple parallel algorithm. The externalization implementation offers performance that is nearly as good. The serialization implementation comes in third with the datagram socket implementation providing the worst performance of the group. In general, the socket, externalization, and the serialization implementations perform similarly with respect to each other. The datagram socket however, clearly provides the worst performance.

Next, the performance of the four Java communication mechanisms are evaluated, when using the modified Cannon's algorithm. Unlike the simple parallel algorithm, the modified Cannon's algorithm transfers a small amount of data per communication call, but requires a large number of communication calls. Does the socket implementation achieve the best performance in this scenario? The results when executing the modified Cannon's algorithm are displayed in Fig. 4.2.

As before, Fig. 4.2 illustrates the best performance is achieved using the socket implementation. Again the externalization implementation offers similar performance to that of the socket implementation, but still comes in second. The serialization implementation comes in third with the datagram socket implementation providing the worst performance of the group. Table 4.1 provides a numerical representation of Fig. 4.1 and 4.2.

4.1. Communication Time vs. Overall Computation Time. All of the results discussed to this point, have represented the overall computation time; the total time it takes, once a command is entered in the manager process, until a final result is achieved by the manager process. Communication time, however, only encompasses the total amount of time spent communicating during the execution of a particular algorithm. Fig. 4.3 and Fig. 4.4 display these communication times.

```
public void sendData()
        {
            int bufferSize, limit, offset;

            totalBufferSize = theManager.oStream[clientIndex].size();
            totalPackets = getTotalPacketNum(totalBufferSize);

            limit = totalPackets;
            offset = 0;

            try
            {
                while (limit > 0)
                {
                    theManager.packetNum =getNextPacketNumber(theManager.packetNum);
                    bufferSize = getBufferSize(limit);
                    packetOutBuffer = createPacketBuffer(offset, bufferSize,
                                      theManager.oStream[clientIndex].toByteArray(),
                                      theManager.packetNum);

                    dataPacket = new DatagramPacket(packetOutBuffer, packetOutBuffer.length,
                                 theManager.dataNodeArray[clientIndex].getWorkerAddress(),
                                 theManager.dataNodeArray[clientIndex].
                                 getWorkerPortNumber() + 1);

                    theManager.socket[clientIndex].send(dataPacket);

                    pTimer = new PacketTimer(theManager.socket[clientIndex], dataPacket,
                                 packetDelay);
                    pTimer.start();

                    offset = offset + bufferSize;
                    limit--;

                    controlPacketReceived.p();
                    pTimer.pleaseStop();
                }

            }

            catch (Exception e)
            {
                System.out.println("Send Exception " + e);
            }

        }
```

FIG. 3.5. *The SendData() Method*

For the simple parallel algorithm, Fig. 4.3 shows the socket implementation has the lowest communication times followed closely by the externalization and serialization implementations. As expected, the datagram socket implementation has the highest communication times.

For the modified Cannon's algorithm, Fig. 4.4 shows that the socket implementation has the lowest communication times followed closely by the externalization and serialization implementations. Again, the datagram socket implementation has the highest communication times of the group. Table 4.2 provides a numerical representation of Fig. 4.3 and Fig. 4.4.

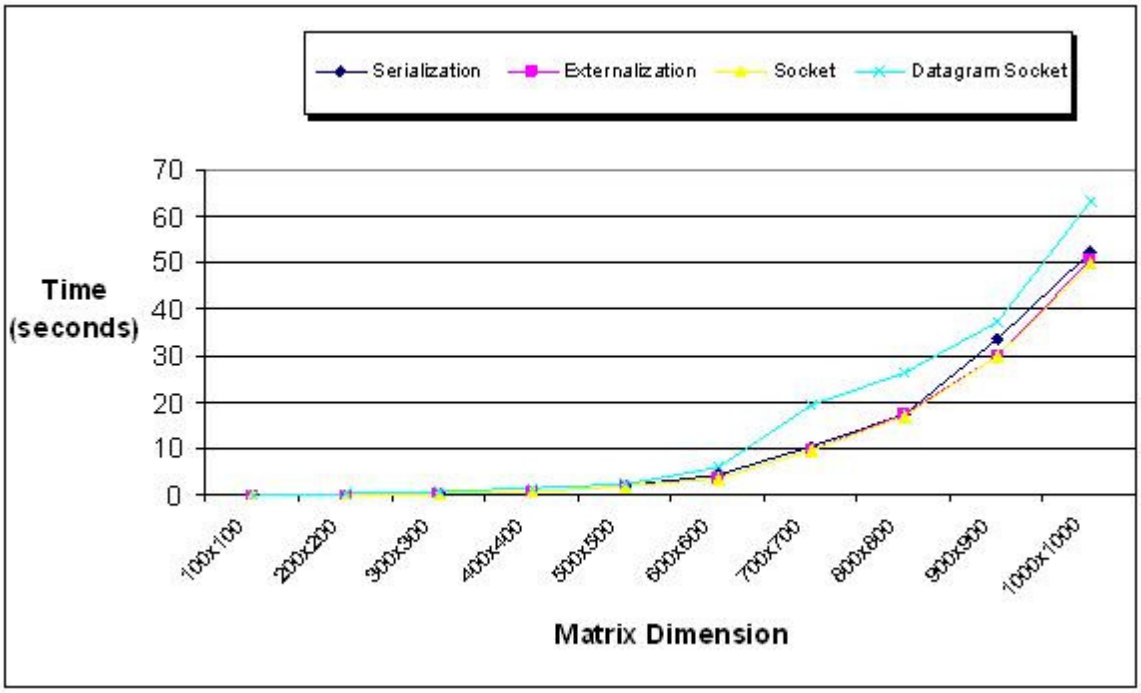

FIG. 4.1. *Simple Parallel Algorithm Time*

TABLE 4.1
Performance Results of Java Communication Mechanisms

Size $R \times C$	Serial SPA	Serial MCA	Extern SPA	Extern MCA	Socket SPA	Socket MCA	Data Socket SPA	Data Socket MCA
100×100	0.103	0.803	0.055	0.736	0.053	0.710	0.101	0.617
200×200	0.165	1.072	0.142	1.046	0.141	0.945	0.247	1.076
300×300	0.413	1.716	0.395	1.744	0.391	1.511	0.689	1.992
400×400	0.872	2.900	0.920	2.823	0.900	2.667	1.562	3.336
500×500	1.959	4.378	1.950	3.580	1.944	3.372	2.605	3.726
600×600	4.494	6.547	3.749	5.919	3.699	5.668	5.895	6.236
700×700	10.569	9.087	9.710	8.946	9.751	8.428	19.598	9.569
800×800	17.321	11.778	17.305	10.492	16.899	10.001	26.367	13.516
900×900	33.660	14.007	29.986	12.853	30.021	12.251	37.285	17.946
1000×1000	52.221	19.886	50.459	17.446	49.694	16.894	63.046	23.995

It is important to note, the overall computation time consists of four main components. The first component is the communication time as discussed earlier. The second component is the time it takes for the manager process to partition the original matrices and ready them for transmission. For example, in the case of the modified Cannon's algorithm, the initial pre-skewing is done by the manager process before any data is transmitted. The third component is the time it takes for the worker processes to perform the required calculations. And the last component is the time it takes for the manager process to recombine all the partial solutions received from the worker processes and arrive at a final answer. Therefore, subtracting Table 4.2 from 4.1 will not give an accurate measurement of the time spent on calculations alone.

4.2. Serializable Interface vs. Externalizable Interface. Java RMI using the *Externalizable* interface provided slightly better performance than Java RMI using the *Serializable* interface. This increase in performance does come at a cost since the externalization mechanism is more difficult to implement. Methods *readExternal()* and *writeExternal()* must be implemented and the marshalling and demarshalling code contained within these methods must be rewritten when

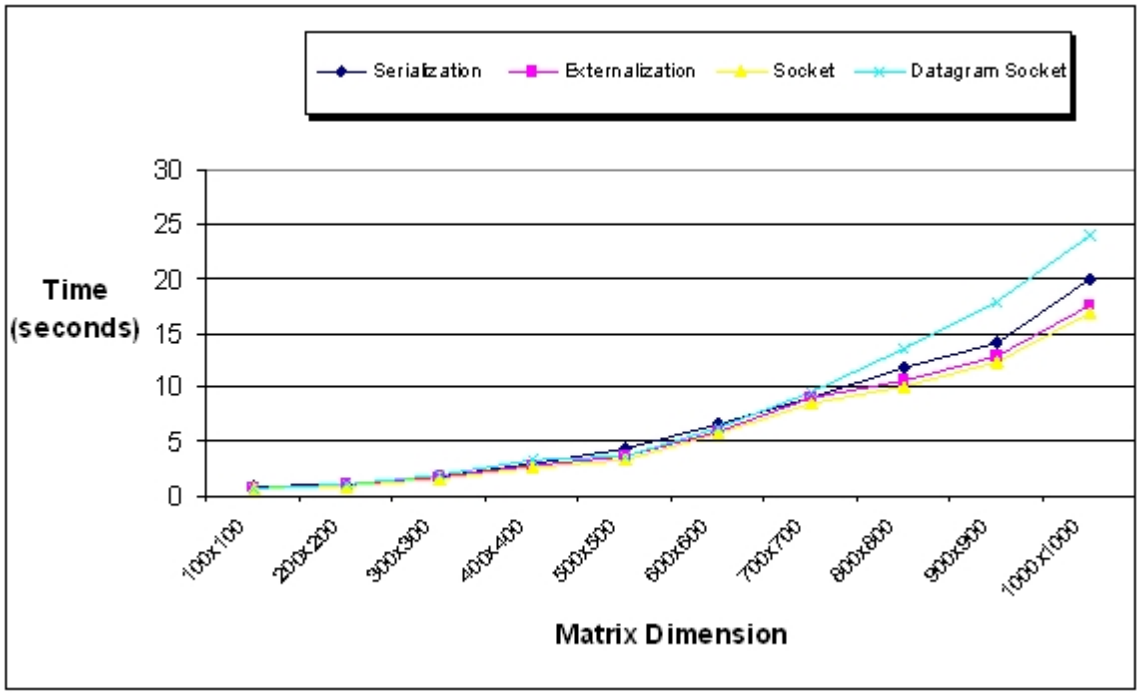

FIG. 4.2. *Modified Cannon's Algorithm Time*

TABLE 4.2
The Amount of Time Used For Communication

Size $R \times C$	Serial SPA	Serial MCA	Extern SPA	Extern MCA	Socket SPA	Socket MCA	Data Socket SPA	Data Socket MCA
100×100	0.034	0.637	0.020	0.542	0.030	0.515	0.049	0.438
200×200	0.109	0.840	0.100	0.816	0.109	0.716	0.194	0.961
300×300	0.213	1.269	0.208	1.189	0.188	0.957	0.473	1.509
400×400	0.365	1.703	0.346	1.590	0.337	1.433	0.951	1.823
500×500	0.572	2.197	0.488	1.847	0.486	1.642	1.480	2.388
600×600	0.831	2.809	0.817	2.638	0.781	2.384	2.787	3.625
700×700	1.194	3.491	1.175	3.364	1.038	2.845	5.141	4.772
800×800	1.503	4.066	1.225	3.735	1.225	3.245	7.171	6.123
900×900	1.828	4.416	1.797	4.265	1.777	3.661	9.067	8.474
1000×1000	2.452	4.937	2.341	4.833	2.109	4.281	13.261	10.467

changes to a class's definitions are made. But if a programmer wants to use a high level communication mechanism and requires the utmost performance, Java RMI using the *Externalizable* is the best choice.

4.3. Socket vs. Datagram Socket. The socket and datagram socket implementations represent the low level communication mechanisms studied in this paper. More generically, the socket method uses circuit switching whereas the datagram socket uses packet switching. Fig. 4.5 compares these basic communication methods during the transmission of a message with length X, a K link path, with a link rate of B, propagation delay D, packet prep time T, circuit setup time S, packet length P, and header length H.

Technically, packet switching will outperform circuit switching when the setup time is greater than the time required to prep $Q - 1$ packets plus the time required to prep and transmit the last packet over a K link route. We are interested in when the last packet will arrive at the destination. Because the socket method offered a much higher level of performance, it is easy to conclude that the setup time associated with the socket approach was considerably less. This makes perfect sense because the datagram method implemented in this paper used the stop-and-wait method.

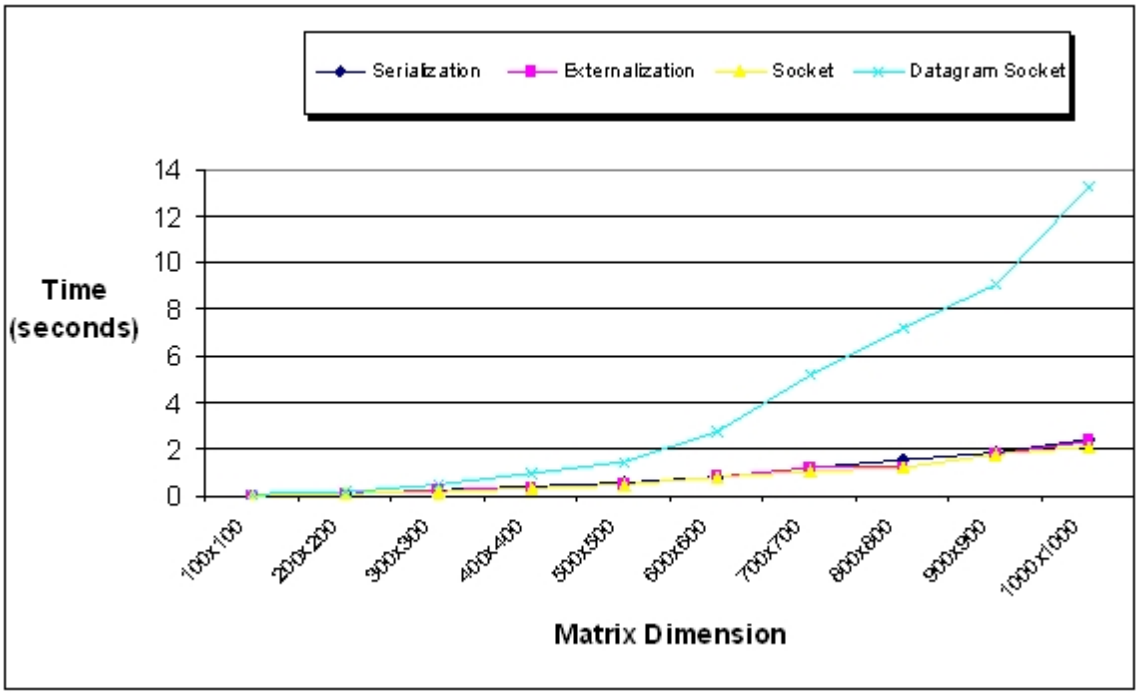

FIG. 4.3. *Simple Parallel Algorithm Communication Time*

Remember the Java datagram socket implementation used the idea of packet synchronization because it suffered from an increasing number of lost packets as the size of the matrices surpassed 500×500. The disadvantage of the stop-and-wait method is inefficiency. In the stop-and-wait method, each packet must travel all the way to the receiver and an acknowledgment must be received before the next packet may be transmitted. This ensures that all packets will arrive in proper order, but also results in a tremendous waste of bandwidth because each packet is alone on the transmission line. Therefore the time to transmit a packet becomes rather significant, especially if some packets are lost necessitating the retransmission of the lost packets.

It is interesting to note the datagram socket implementation performed slightly better than the socket implementation before packet synchronization was introduced. One possible method to increase the performance of the datagram socket implementation, using packet synchronization, is to implement a sliding window protocol. The sliding window method helps alleviate the inefficiency of the stop-and-wait method by allowing multiple packets to be transmitted before requiring an acknowledgment. This allows the transmission line to carry multiple packets at once resulting in higher bandwidth utilization and an increase in performance.

4.4. Simple Parallel Algorithm vs. Modified Cannon's Algorithm. Implementing the simple parallel algorithm within the Java environment was a rather simplistic task. Unfortunately, this algorithm required a significant amount of memory. Cannon's algorithm was developed to be a memory-efficient matrix multiplication algorithm [14]. The notion behind this algorithm is as follows: two matrices A and B, are partitioned into square blocks and transferred to q processes. If a schedule can be enforced, so that after each submatrix multiplication these partitioned blocks can be rotated among the processes, then each process contains only one block of each matrix at any given time [14]. Although a modified Cannon's algorithm was implemented in this paper, the characteristic of being memory efficient still applies.

Looking back at Table 4.1, an interesting trend can be witnessed between the SPA and MCA implementations. At matrices sizes of 600×600 and smaller, all the SPA implementations outperform their corresponding MCA implementations. But at 700×700 and larger, the roles are reversed. At these larger matrix sizes, all the MCA implementations offer better performance than their corresponding SPA implementations. In fact as the matrix size increases, so to does the degree in which the MCA outperforms the SPA, even though the MCA implementations generally require more communication time as indicated by Table 4.2. As the size of matrices approach and pass 700×700, the SPA implementations start to suffer from a memory related issue.

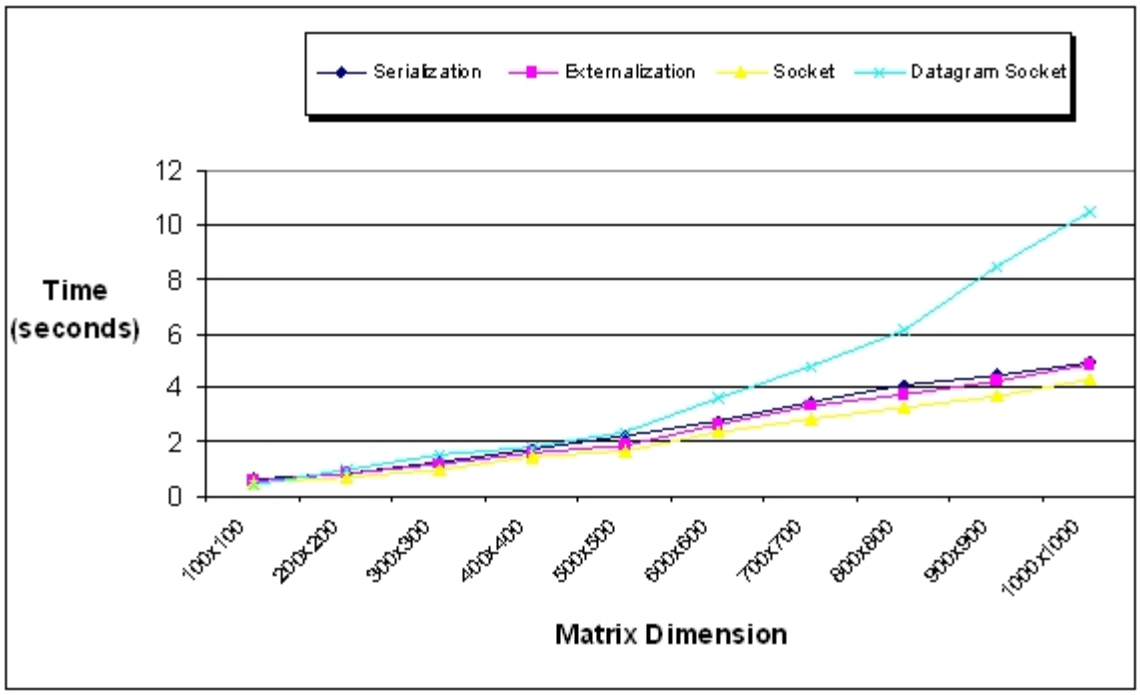

FIG. 4.4. *Modified Cannon's Algorithm CommunicationTime*

$$
\begin{aligned}
&\textit{Given:} \quad KD = \textit{Total propagation delay}\\
&\qquad \tfrac{X}{B} = \textit{Delay to put message on the link}\\
&\qquad Q = \tfrac{X}{(P-H)} = \textit{Number of packets}\\
&\qquad \left(\tfrac{P}{B}+T\right) = \textit{Delay to put packet on the link plus prep time}\\
&\qquad Q\left(\tfrac{P}{B}\right) = \tfrac{X}{B}\,(\textit{approximately})\\
&\qquad CS = S+\tfrac{X}{B}+KD\\
&\qquad PS = (Q-1)\left(\tfrac{P}{B}+T\right)+K\left(\tfrac{P}{B}+T\right)+KD
\end{aligned}
$$

$$\textit{Question:} \qquad \textit{When is } PS < CS\,?$$

$$
\begin{aligned}
&\textit{Answer:} \qquad PS < CS\\
&(Q-1)\left(\tfrac{P}{B}+T\right)+K\left(\tfrac{P}{B}+T\right)+KD < S+\tfrac{X}{B}+KD\\
&(Q-1)\left(\tfrac{P}{B}+T\right)+K\left(\tfrac{P}{B}+T\right) < S+\tfrac{X}{B}\\
&Q\left(\tfrac{P}{B}\right)+QT-\tfrac{P}{B}-T+K\left(\tfrac{P}{B}+T\right) < S+\tfrac{X}{B}\\
&QT-\tfrac{P}{B}-T+K\left(\tfrac{P}{B}+T\right) < S\\
&(Q-1)T+K\left(\tfrac{P}{B}+T\right)-\tfrac{P}{B} < S
\end{aligned}
$$

FIG. 4.5. *Circuit Switching vs. Packet Switching*

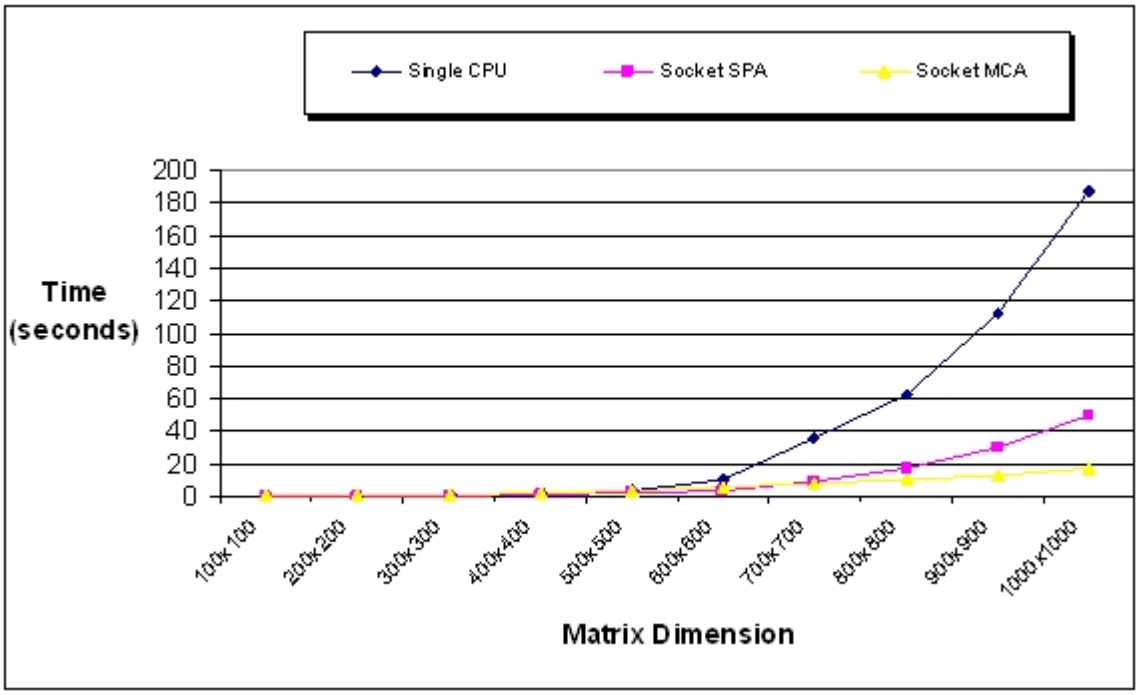

FIG. 4.6. *Sequential Algorithm Time vs. Parallel AlgorithmTime*

TABLE 4.3
Sequential Algorithm Time vs. Parallel AlgorithmTime

Size $R \times C$	Single CPU	Socket SPA	Socket MCA
100×100	0.022	0.053	0.710
200×200	0.141	0.141	0.945
300×300	0.694	0.391	1.511
400×400	1.756	0.900	2.667
500×500	4.031	1.944	3.372
600×600	10.316	3.699	5.668
700×700	35.396	9.751	8.428
800×800	62.219	16.899	10.001
900×900	111.678	30.021	12.251
1000×1000	187.745	49.694	16.894

4.4.1. Speedup. As mentioned earlier, one advantage of distributed computing is computational speedup as defined by the equation:

$$Computation\ speedup = \frac{fastest\ sequential\ algorithm\ execution\ time}{parallel\ algorithm\ execution\ time} \tag{4.1}$$

Implementing a distributed algorithm for a problem that underperforms a sequential algorithm for the same problem is a waste of time. Therefore sequential execution times were measured so that computation speedup could be calculated. The sequential algorithm used was the basic matrix multiplication algorithm with complexity $\Theta(n^3)$.

Fig. 4.6 shows the execution times of the sequential matrix multiplication algorithm and the two parallel matrix multiplication algorithms using sockets. Only the socket execution times were compared since they achieved the best performance. Table 4.3 provides a numerical representation of Fig. 4.6 and Fig. 4.7 illustrates the speedup results achieved during this experiment.

As mentioned previously, to correctly compute computation speedup, the fastest sequential algorithm must be used. In this paper, the generic sequential matrix multiplication algorithm $\Theta(n^3)$ was used to calculate computation speedup

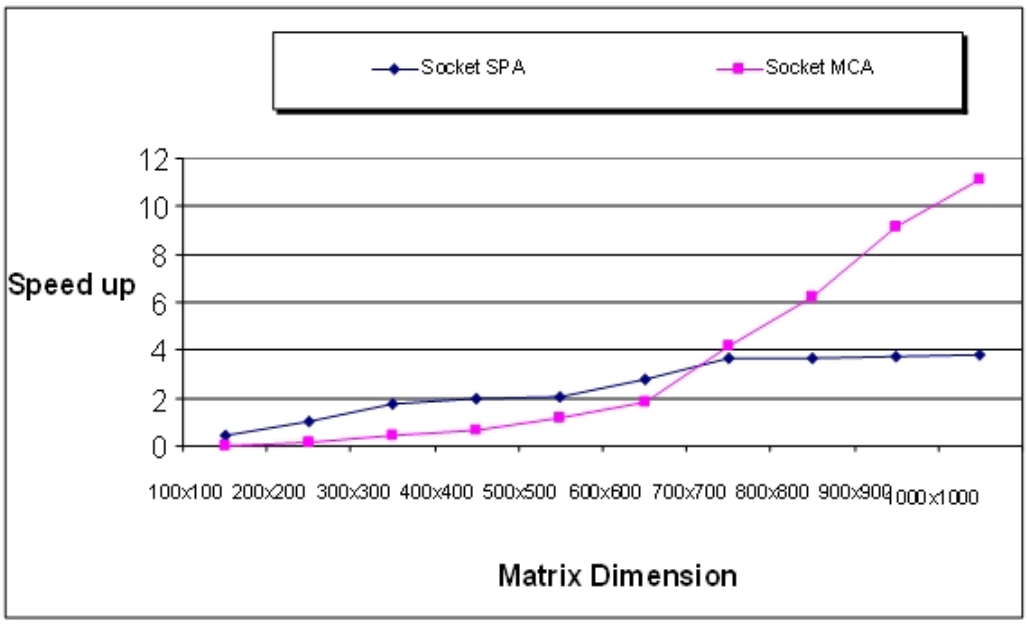

FIG. 4.7. *Computation Speedup*

even though faster sequential algorithms such as Strassens $\Theta(n^{2.807})$ exist. Going by the strict definition of computation speedup, the use of the generic algorithm is incorrect and raises a valid concern. However because the ultimate goal of this paper is to gauge the performance of four Java communication mechanisms, the speedup data is provided to the reader only as tool for understanding the potential advantages of a distributed algorithm.

For the simple parallel algorithm, the computation speedup approaches 4 as the size of the matrices increase. This observation is expected due to the fact there were 4 worker processes performing the calculations. Unfortunately, ideal speedup is rarely realized due to the following factors: data dependencies, input/output bottlenecks, synchronization overhead, and inter-processor communication overhead [5], [6].

For the modified Cannon's algorithm, the computation speedup actually exceeds the ideal case a phenomenon which is referred to as super linear speedup [14]. Recall the modified Cannon's algorithm had an interesting performance characteristic due to its more memory efficient design. At matrices sizes of 600×600 and smaller, all the SPA implementations outperform their corresponding MCA implementations. But at 700×700 and larger, the roles are reversed with the MCA implementations offering better performance. As the size of matrices approach and pass 700×700, it appears the SPA implementations start to suffer from memory related performance problems. Whether these potential memory problems are a result of paging faults, garbage collection, or some other problem is not clear.

Calculating the multiplicative constants for the data in Table 4.3, reveal little variation for the modified Cannon's algorithm. For the most part, the multiplicative constants for the simple parallel algorithm remain relatively close. However the multiplicative constants for the sequential algorithm increase dramatically for matrices larger than 600×600.

If indeed the sequential algorithm is suffering from decreased performance due to memory related problems such as garbage collection, or paging faults, then the workload of the sequential algorithm becomes larger than the parallel algorithm, leading to superlinear speedup. This result highlights another advantage of distributed computing: access to a larger amount of memory.

5. Conclusion. Distributing computation among several processors has become an important technique in the high performance community. Distributed computing represents a viable solution to the problem of finding more powerful architectures, capable of harnessing the power of multiple machines. Among the first design issues a Java programmer must face when creating a distributed application, is which connection mechanism to use. In this paper, a performance and programming complexity analysis of all four Java API network connection mechanisms was presented. The four connection mechanisms are: RMI using the *Serializable* interface, the *Externalizable* interface, the socket, and datagram socket.

The socket implementation provided the best overall performance followed closely by Java RMI using the *Externalizable* interface. Because Java RMI is a high level communication mechanism, it allows programmers to focus more on

```
public class MultiplyDataNode implements Serializable {
                        public int subMat[][];
                        public int mainMatrix[][];
                        public transient int rowSize;
                        public transient int colSize;
                        public transient int mainColSize;
                        public int rowOffset;

                        public MultiplyDataNode() { }

        }
```

FIG. 5.1. *MultiplyDataNode.java Using Serialization*

the distributed algorithm and less on the low level communication concerns. It is up to the programmer to decide which is more important for a given application, coding complexity or program performance.

Due to their respective overheads the implementations using, Java RMI with the *Serializable* interface and the datagram socket performed comparably slower. With regard to serialization, the overhead exists in the generic marshalling and demarshalling algorithm. However, an advantage of this generic algorithm is that it makes the design and development of distributed applications much easier than the other communication mechanisms.

With the datagram socket implementation, the overhead occurs in the stop-and-wait packet synchronization method. In the stop-and-wait method, each packet must travel all the way to the receiver and an acknowledgment must be received before the next packet may be transmitted. This ensures that all packets will arrive in proper order, but also results in a tremendous waste of bandwidth because each packet is alone on the transmission line. The poor performance and complex programming required by the datagram socket effectively eliminate it from consideration when using the stop-and-wait method.

REFERENCES

[1] MAASSEN, J., NIEUWPOORT, R. V., VELDEMA, R., BAL, H., PLAAT, A., *An Efficient Implementation of Java's Remote Method Invocation*, Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Atlanta, GA, (1999), pp. 173-182.

[2] BREG, F. AND POLYCHRONOPOULOS, C. D., *Java Virtual Machine Support for Object Serialization*, Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande, Palo Alto, CA, (2001), pp. 173-180.

[3] SILBERSCHATZ, A., GALVIN, P. B., *Operating System Concepts (Fifth Edition)*, Addison Wesley Longman,Inc., Berkeley, CA, (1998), pp. 14-20.

[4] HENNESSY, L. AND PATTERSON, D. A., *Computer Architecture: A Quantitative Approach (Second Edition)*, Morgan KaufmannPublishers, Inc., San Francisco, CA, (1996), pp. 635-644.

[5] ZARGHAM, M. R., *Computer Architecture Single and Parallel Systems*, Prentice Hall, Upper Saddle River, NJ, (1995), pp. 300-303.

[6] SHEIL, H., *Distributed Scientific Computing in Java: Observations and Recommendations*, Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, Kilkenny City, Ireland, (2003), pp 219-222.

[7] SUTHERLAND, D., *RMI and Java Distributed Computing*, http://java.sun.com/features/1997/nov/rmi.html, retrieved November 13, (2003).

[8] MAASSEN, J., NIEUWPOORT, R. V., VELDEMA, R., BAL, H., KIELMANN, T, JACOBS, C., AND HOFMAN, R., *Efficient Java RMI for Parallel Programming*, ACM Transactions on Programming Languages and Systems, New York, NY, (2001), pp. 747 775.

[9] GROSSO, W., *Java RMI*, O'Reilly and Associates, Sebastopol, CA, (2002), pp. 179-198.

[10] CURRY, A., *Unix Systems Programming for SVR4*, OReilly and Associates, Sebastopol, CA, (1996), pp. 391-396.

[11] FOROUZAN, B. A.., *Data Communications and Networking (SecondEdition)*, McGraw-Hill, New York, NY, (2001), pp. 441-447.

[12] HALL, J. D., CARR, N. A., AND HART, J. C., *GPU Algorithms for Radiosity and Subsurface Scattering*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, San Diego, CA, (2003), pp. 51-59.

[13] MAASSEN, J., NIEUWPOORT, R. V., VELDEMA, R., BAL, H., AND KIELMANN, T., *Wide-Area Parallel Computing in Java*, Proceedings of the ACM 1999 Conference on Java Grande, San Francisco, CA,(1999), pp. 8-14.

[14] GRAMA, A., GUPTA, A., KARYPIS, G., AND KUMAR, V., *Introduction to Parallel Computing (SecondEdition)*, Pearson EducationLimited, Harlow, England, (2003), pp. 345-349.

[15] FLANAGAN, D., *Java in a Nutshell (Third Edition)*,O'Reilly and Associates Inc., Sebastopol, CA, (1999), pp. 74-93.

Appendix. *RMI Code Listings: Contrasting the Serializable and Externalizable interfaces* Below is some actual code to help emphasize the role the methods *readObject()*, *writeObject()*, *readExternal()*, and *writeExternal()* play in the Serializable and Externalizable interfaces. The two write methods are responsible for writing a remote object to a

```
public class MultiplyDataNode implements Externalizable  {
        public int subMat[][];
        public int mainMatrix[][];
        public transient int rowSize;
        public transient int colSize;
        public transient int mainColSize;
        public int rowOffset;

        public MultiplyDataNode()    { }

    public void writeExternal(ObjectOutput out) throws IOException   {
         int row, col;

         out.writeInt(rowSize);
         out.writeInt(colSize);
         out.writeInt(mainColSize);
         out.writeInt(rowOffset);

         for (row = 0; row < rowSize; row++)    {
             for (col = 0; col < colSize; col++)   {
                      out.writeInt(subMat[row][col]);
             }
         }

        for (row = 0; row < colSize; row++)    {
             for (col = 0; col < mainColSize; col++)   {
                      out.writeInt(mainMatrix[row][col]);
             }
        }
    }

    public void readExternal(ObjectInput in
             throws IOException  ClassNotFoundException    {
         int row, col;

         rowSize = in.readInt();
         colSize = in.readInt();
         mainColSize = in.readInt();
         rowOffset = in.readInt();
         subMat = new int[rowSize][colSize];
         mainMatrix = new int[colSize][mainColSize];

       for (row = 0; row < rowSize; row++) {
             for (col = 0; col < colSize; col++) {
                    subMat[row][col] = in.readInt();
             }
        }

       for (row = 0; row < colSize; row++) {
             for (col = 0; col < mainColSize; col++) {
                    mainMatrix[row][col] = in.readInt();
             }
        }
    }
}
```

FIG. 5.2. *MultiplyDataNode.java Using Externalization*

stream, while the read methods are responsible for reading a remote object from a stream. Fig. 5.1 shows the complete implementation of the MultiplyDataNode class, a class that implements the Serializable interface.

This class is composed of two integer matrices, four integers, and a zero argument constructor. The transient keyword tells the serialization mechanism to ignore a particular variable. In Java, the size of a matrix is embedded within the matrix object itself. Notice the *readObject()* and *writeObject()* methods are not implemented. This is because all the locally defined variables are either primitive data types or serializable objects. Therefore the serialization mechanism will work without any further effort by the programmer. This makes distributed programming much easier but less efficient.

Now observe the same class, this time implementing the Externalizable interface as shown in Fig. 5.2. As before, this class (Fig. 5.2) is composed of two integer matrices, four integers, and a zero argument constructor. Notice the *readExternal()*, and *writeExternal()* methods are implemented, a requirement when dealing with externalization. The method, *writeExternal()*, systematically writes each object to a stream while the corresponding readExternal() method reads in each object in the exact order as it was written. This time, the transient keyword is not present because in this case of externalization, it is the programmer's responsibility to forward the size of the matrices. This makes distributed programming using the externalizable interface more difficult but hopefully the increased efficiency will be worth it.

Edited by: Marcin Paprzycki
Received: December 14, 2005
Accepted: October 24, 2006