Java Virtual Machine Support for Object Serialization

Fabian Breg
University of Illinois, Urbana-Champaign
breg@csrd.uiuc.edu

Constantine D. Polychronopoulos
University of Illinois, Urbana-Champaign
cdp@csrd.uiuc.edu

ABSTRACT

Distributed computing has become increasingly popular in the high performance community. Java's Remote Method Invocation (RMI) provides a simple, yet powerful method for implementing parallel algorithms in Java. The performance of RMI has been less than adequate, however, and object serialization is often identified as a major performance inhibitor. We believe that object serialization is best performed in the Java Virtual Machine (JVM), where information regarding object layout and hardware communication resources are readily available. We implement a subset of Java's object serialization protocol in native code, using the Java Native Interface (JNI) and JVM internals. Experiments show that our approach is up to eight times faster than Java's original object serialization protocol for array objects. Also for linked data structures, our approach obtains a moderate speedup and better scalability. Evaluation of our object serialization implementation in an RMI framework indicates that a higher throughput can be obtained. Parallel applications, written using RMI, obtain better speedups and scalability when this more efficient object serialization is used.

1. INTRODUCTION

The Java [6] programming language is increasingly becoming a language of choice for distributed computing. The portability offered by the Java Virtual Machine (JVM), its support for secure dynamic class downloading, and powerful networking API make it an easy-to-use, yet safe language for intranet and internet applications.

The Java binding to the operating system's socket API is convenient when complete control of interprocess communication is needed. For a large number of distributed applications, however, programming sockets is too complex. Therefore, on top of the socket API, the Java runtime environment provides a remote method invocation (RMI) facility that allows communication between Java objects in separate virtual machines. Java RMI hides low level communication issues from the programmer, allowing the programmer to focus on the distributed algorithm instead.

Java object serialization [9] provides persistence for regular Java objects. Object serialization captures the state of a Java object [15] and writes it to a byte stream. Object persistence is often used to store objects to file or database, but can also be used to pass objects between Java Virtual Machines. Java RMI uses object serialization to pass parameters and return values to and from remote objects by value. In its simplest form, serialization of an object just requires this object to be tagged as being serializable. No object specific code is needed to convert objects to and from a bytestream, although providing such code can potentially improve the performance of object serialization.

Java RMI and object serialization facilitate the development of distributed Java applications by abstracting many of the networking issues. In the end, however, deployment of distributed applications succeed or fail with their performance. The performance of Java RMI is generally found to be inadequate, and often, object serialization is considered to be an important performance inhibitor [13, 4, 7].

In this article, we introduce our implementation of object serialization in native code. Such an implementation has two advantages. First, if enough work is done within native code, without invoking Java code, the overhead of JNI should be relatively minimal, which means that the efficiency with which native code is executed can be exploited optimally.

A second advantage is that native code can exploit the actual layout of Java objects in memory. Instead of having to serialize each array element of primitive arrays separately, native code could exploit the fact that most JVMs would layout primitive arrays in a contiguous memory area. The object serialization implementation could directly copy the memory area, occupied by the array object, to the network.

Disadvantages of implementing object serialization in native code include a high overhead associated with the Java Native Interface (JNI) when serializing individual primitives, like bytes and integers. A hybrid implementation of object serialization could combine the best of both approaches.

Another disadvantage is the loss of portability, since native code needs to be compiled for every platform it is deployed on. To make the problem worse, the JNI does not provide us with all of the functionality we need from the JVM. Instead of calling back into Java code, we chose to directly call functions provided by the JVM, making our implementation JVM dependent. We believe, however, that an important platform for distributed high performance computing are homogeneous clusters, for which portability is not an issue. Although our implementation is based on Sun's Java Development Kit version 1.2, our approach seems to be applicable to other JVM implementations.

Our object serialization protocol shows an improvement factor of 8 over standard Java object serialization for serialization of double arrays. Reading doubles from a byte stream shows an improvement factor of 6 over Java respectively. Serialization and deserialization of linked structures show an improvement of 3 and 1.5 respectively. However, our experiments also show that our native code implementation is still slower than deploying custom serialization routines.

When used in an RMI framework, our object serialization protocol obtains a higher throughput in byte array ping-pong tests than Java RMI, but still performs worse than KaRMI. For double arrays, the throughput of LuchatRMI and native object serialization is a factor 3 higher than the other implementations. Performance evaluation of an RMI based parallel algorithm shows that standard Java RMI is not capable of obtaining any significant speedups over the sequential algorithm. Using LuchatRMI with our improved native object serialization protocol shows a speedup of 6 on 9 hosts.

This paper is organized as follows. Section 2 describes some problems with the standard object serialization, and some existing approaches to object serialization performance improvement. Section 3 describes our implementation of object serialization. Section 4 evaluates the performance of our approach, comparing it to other object serialization implementations. In Section 5, we present some conclusions.

2. RELATED WORK

The process of copying abstract data types has been described by Herlihy and Liskov [8]. Their approach differs from Java's object serialization approach in that they require each object to include marshaling and unmarshaling operations. Their approach for detecting multiple references to a single object and correctly handling circular structures is similar to our approach.

One expensive aspect of Java object serialization involves the automatic discovery of the structure of the data to be serialized. Obtaining information about fields contained within objects is done in Java through Java's reflection facilities, which require expensive interaction with the virtual machine.

A general approach to improving Java object serialization performance, therefore, involves requiring the programmer to provide explicit object serialization routines, usually on a per class basis [4, 7]. Instead of using the JVM's reflection mechanism to obtain the internal state of an object, these routines directly write an object's internal state to the stream. This approach requires considerable effort from the programmer, although these routines could potentially be generated by a compiler.

Another aspect that potentially decreases performance is the fact that Java object serialization provides a single algorithm that can handle all kinds of objects. Some objects require a less complex, and therefore less expensive object serialization implementation. The object serialization framework of NexusRMI [4] allows implementing the object serialization protocol explicitly on a per class basis. If it is known that a particular object does not contain reference cycles or even reference sharing, a simple protocol that does not check for these cases can be written, allowing a more efficient serialization implementation.

To reduce the communication overhead of RMI, the size of the data streams generated by object serialization can be compressed, for instance by using gzip [5] or by recognizing class names that share a common prefix [12]. This common prefix is inserted only once in the bytestream. When needed, a reference is included for all other occurrences. Although these approaches reduce the amount of data to be transferred, it introduces some computation overhead.

The XStream object serialization implementation, which is described in [7] and [14], exploits the fact that, when used in distributed computing, the generated byte stream does not need to contain all the information needed for general object persistence. In addition, a more aggressive type caching scheme is employed. Finally, a more efficient buffering scheme is offered for use in explicit serialization routines.

Some of the optimizations described above are also exploited in our object serialization approach. Our implementation only generates necessary information for use in distributed computing. Instead of using the standard Java buffer streams, which would involve additional overhead by invoking Java methods from native code, we handle buffering inside our object serialization protocol, although in a very general way.

Manta [11] is another implementation of RMI. Manta compiles a Java RMI application to native code, which can be executed without the need of a Java Virtual Machine. In contrast, our approach could be used to extend the functionality of the JVM, without sacrificing portability of Java.

3. OBJECT SERIALIZATION IMPLEMENTATION

Like the standard object serialization implementation, our implementation of object serialization consists of the classes `edu.uiuc.csrd.ObjectOutputStream`, for writing objects to a stream, and `edu.uiuc.csrd.ObjectInputStream` class for reading objects from a stream. These classes provide most of the functionality provided by their Java counterparts in package `java.io`. In the remainder of this section, we will not include the package name when referring to our implementation. In addition we will refer to generic stream types instead of specifically denoting specific input and output streams.

The `ObjectOutputStream` and `ObjectInputStream` classes extend `java.io.OutputStream` and `java.io.InputStream`, respectively, like any other stream in Java. In addition, these classes implement the interfaces `java.io.ObjectOutput` and `java.io.ObjectInput`, respectively. However, because we do not provide full functionality of these interfaces, our implementation cannot be used in Java RMI. Instead, we have used our object serialization with LuchatRMI and present some results in Section 4. LuchatRMI is described in [3].

The constructor of `ObjectOutputStream` takes one argument of class `java.io.OutputStream`. However, we do require, that the stream passed to this constructor is actually an instance of class `java.io.FileOutputStream`. We do not handle more generic streams, because our implementation writes directly to the filedescriptor that is specific to file streams. For network communication, socket streams are used, which extend file streams. Similarly, the constructor of `ObjectInputStream` takes one argument of class `java.io.FileInputStream`.

Our implementation provides native methods for reading and writing primitive types and generic objects. In addition, we provide native implementations for `close()`, `flush()` and `reset()`. Our object serialization protocol follows the approach described in [8, 2]. The following sections describe our implementation in detail.

3.1 Wire Protocol

Each object in the bytestream consists of a zero byte, acting as a delimiter, followed by the internal state of that object. Alternatively, the delimiter may contain a one byte, indicating a reset token. Reset tokens are inserted in the stream by invoking `reset()` on `ObjectOutputStream`. We will describe the effect of a reset below. The internal state of a primitive object consists of the individual bytes making up the primitive object.

The internal state of a reference consists of a byte, identifying what kind of object is written, followed by the object's bytestream representation. A 'zero' byte indicates a null reference, which carries no additional information. A byte value of two indicates the object has previously been written to this stream after the last `reset()`. The bytestream representation for such objects consists of an integer as will be explained in Section 3.2. Other byte values identify specific array objects or a non array reference.

Serialization of primitive array types exploit the fact that these objects are laid out in a consecutive memory area in the JVM. The byte value indicates the primitive component type of the array and the bytestream representation is a direct copy of the memory area occupied. This approach is more efficient than the standard serialization of primitive arrays, but is inherently non portable.

The bytestream representation of arrays of references consists of the length of the array, followed by the name of the component class from the array declaration, followed by the internal state of all objects in the array. The bytestream representation of class names consists of the length, followed by the name as a C char pointer.

Finally, the bytestream representation of a non array object consists of the class name of the object, followed by all non-transient instance fields. The fields of the most generic serializable superclass are written first, followed by the fields of the subclasses of this class, down to the original class. At the receiving side, the object is recreated by invoking the no argument constructor of the most specialized non serializable class in the hierarchy, after which all fields are copied from the bytestream to the object.

If, during serialization, a class is encountered that is not serializable, an abort token is written to the stream. Both sides will consequently report an error.

3.2 Shared References

The object serialization protocol recursively writes all objects directly or indirectly referenced by the object that is written. This assures that an exact copy of the object is created at the receiving side. For instance, when writing the root of a tree structure, the complete tree will be written and recreated at the receiving side. This causes a problem when two references are written that reference the same object. In a naive approach, two instances of the object instead of one will be created at the receiving side. This will also cause the serialization process to loop indefinitely when encountering cyclic data structures. A solution to this problem has been described in [8] and our implementation follows a similar approach, that we describe next.

The first time an object is serialized, the reference to the object is stored in a table together with an index number, where the first object serialized is assigned index 1, the next object is assigned index 2, and so on. Whenever an object to be serialized is found in this table, its assigned index number is written to the stream instead of its internal state. A reset causes this table to be emptied and the current index number to be reset to 1.

The receiving side encounters all objects in the same order and can therefore build a similar table. Whenever an index is encountered when an object is expected, the receiving side can use the index to assign a reference to a previously deserialized object. To ensure that the indices at the receiving side match those of the sending side, the receiving side has to store an object after it has been allocated, but before its internal state is deserialized. Since its internal state is not actually used while deserializing, storing an uninitialized object causes no problems.

At the sending side, we use a hashtable to store object references and their indices. Hashing on the object reference allows the protocol to efficiently determine if an object has been previously written. For efficiency, we implemented our own hashtable in native code. The receiving side adds objects to a dynamically expanding array. This allow fast retrieval of a previously seen object using its index number.

3.3 Buffering

Buffering is handled internally in our object streams. The `ObjectOutputStream` writes all data to a memory buffer. The contents are written to the filedescriptor when the buffer is full, the stream is closed, or the stream is flushed. The `ObjectInputStream` reads all available data on the filedescriptor in a memory buffer and consequently reads from this buffer. This buffering scheme drastically improves the performance of our object serialization.

3.4 Elimination of recursion

Serializing linked list structures involves a recursive function call for every consecutive element. The maximum length of a list that can be (de)serialized is therefore limited by available stack space. Recursion elimination could therefore potentially allow larger linked structures to be (de)serialized.

We eliminated recursion from both the `writeObject()` and `readObject()` methods by replacing the recursive function call with a jump to the beginning of the function. This approach potentially overwrites local variables that may be needed after calling the recursive function. We save these variables on a separate stack, located on the heap.

Experiments indicate that recursion elimination can increase the length of the longest path in a graph of objects by a factor of 6. Eventually, size of serializable objects will be limited by the amount of JNI local references available in the JVM.

3.5 Class introspection

Our implementation of object serialization relies heavily on class introspection facilities offered by the JVM. Introspection is used to obtain the superclass of serializable classes as well as to obtain read and write access to all fields of an object. Since not all of these features are offered by JNI, we either had to invoke Java methods from native code to obtain class meta information, or call functions exported by the JVM directly. For efficiency reasons, we chose the latter approach, sacrificing additional portability.

A disadvantage of using class introspection is its high overhead. As mentioned in Section 2, replacing class introspection by explicit marshaling routines can significantly improve object serialization performance. In our implementation, such custom serialization routines should be implemented in native code, which is too complex for programmers not familiar with C.

To relieve some of the overhead introduced by class introspection, we decided to store the result of two relative expensive functions in a hashtable. The first function returns a list of all fields of a certain class. The second function result that we store returns a JNI field identifier for a certain `java.lang.reflect.Field` object. Especially for homogeneous linked structures, this improved performance significantly.

4. PERFORMANCE COMPARISON

In this section, we evaluate and discuss the performance of our object serialization protocol. Section 4.1 writes objects to files and consequently reads the same objects from file. Section 4.2 evaluates its performance within LuchatRMI.

4.1 Object Serialization Performance

In this section, we evaluate the performance of our object serialization implementation, comparing it against both standard Java object serialization and the XStream [7, 14] serialization protocol. In these experiments, we serialize objects to file and consequently read these objects from file. To minimize overhead introduced by disk I/O, we use a filesystem on a ramdisk device. These experiments were run a Pentium III 500 MHz with 256 Mbytes RAM memory running Linux 2.2. We use Blackdown's Linux port (version 1.2.2FCS) of Sun's Java Development Kit 1.2 with the *sunwjit* JIT compiler.

All the performance graphs in this section use the label *java.io.ObjectOutputStream* for standard Java object serialization. The label *java.io.XObjectOutputStream* is used for the X-Stream object serialization protocol and the label *edu.uiuc.csrd.ObjectOutputStream* is used for our native object serialization implementation. When we use explicit marshaling routines, the label *java.io.Externalizable* is used for the performance of object serialization using the `java.io.Externalizable` interface, while custom X-Stream object serialization is labeled *java.io.XSerializable*. A similar labeling scheme is used for object deserialization.

Figure 1 shows the performance of serialization of primitive bytes. The performance graphs for byte primitives serialization mainly show the high overhead associated with a native method invocation in our object serialization implementation. When using Java object serialization, reading and writing a byte just involves copying it to the buffer, without any conversion overhead involved and is therefore very efficient.

Figure 2 shows the performance of double primitive serialization and deserialization. In standard object serialization, a double is written by using a native method to convert the floating point quantity into an eight byte representation. Each byte of this representation is than written separately to the destination bytestream. In our implementation a byte is written in a native method that obtains a pointer to the floating point value in the JVM and than directly copies the bytes to the buffer.

The performance of byte array and double array serialization is depicted in Figure 3 and 4 respectively. Java object serialization serializes both primitive and object arrays by serializing each element of the array separately. For each element, the corresponding serialization method is invoked. As described earlier, serialization of byte primitives is relatively cheap, whereas doubles need to be converted to eight separate bytes.

Our implementation of object serialization exploits the fact that arrays occupy a contiguous block of memory in the JVM that we are using. Array serialization is performed by obtaining a pointer to the array and directly copying this memory block to the serialization buffer. Although this particular method of array layout is not guaranteed to be used by different JVM implementations, it is the most obvious layout to use for arrays.

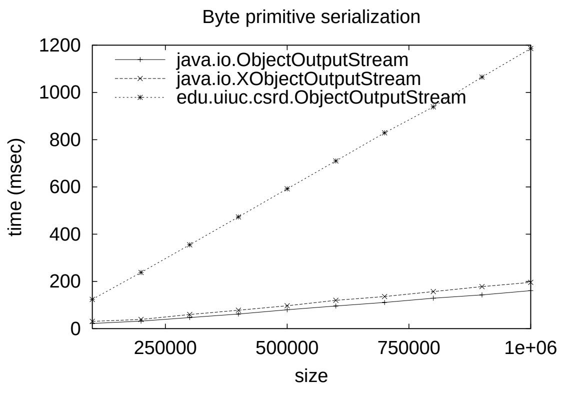
Object serialization of byte arrays in Java is done efficiently by copying the byte array directly to the serialization buffer. Our approach, therefore, does not obtain a significant speedup, and is even slower for writing byte arrays than the X-Stream object serialization. Our approach does obtain a factor of eight for double array serialization and deserialization. Distributed scientific applications using floating point matrices could benefit significantly from our object serialization protocol.

Figure 5 shows the performance of serialization of a binary tree object. Each node in the binary tree is a Java object containing a reference of type `java.lang.Object` and two references to binary tree nodes. The objects we store in the tree contain a field for each primitive type. These objects thus contain eight fields and have a total size of 30 bytes. The tree is serialized by invoking `writeObject()`, passing the root node as argument.

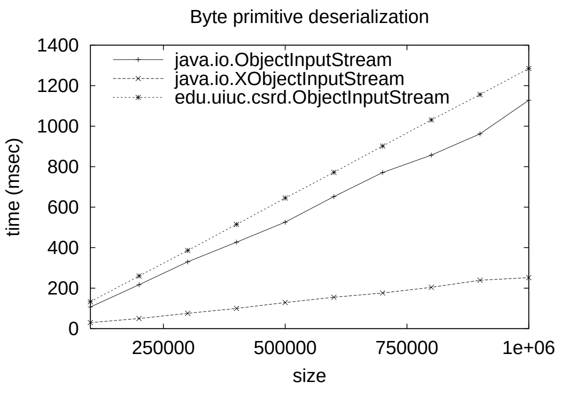
Since we have the source code for both the binary tree and the data objects, we included the serialization performance obtained by adding custom object serialization methods from the `java.io.Externalizable` interface. Since neither of these objects have serializable superclasses, all the `writeExternal()` and `readExternal()` methods do is write and read each field. In addition, we included the performance obtained with custom serialization methods added for the X-Stream serialization method.

In this particular test, explicit serialization routines using the `Externalizable` interface do not show a significant improvement over Java's default object serialization protocol. The explicit serialization routines that use the X-Stream serialization protocol, however, show a factor of four performance improvement for writing and a factor of more than two for reading, and outperforms native code object serialization. Without explicit serialization routines, native object serialization outperforms both standard Java object serialization and X-Stream object serialization by more than a factor of two for writing and a factor of 1.5 for reading.

Finally, Figure 6 shows the performance of the serialization of `java.util.Vector` objects. The internal state of a Vector consists mainly of a array of Object references. Serialization of a Vector does not rely on any explicit serialization code, like for instance `java.util.Hashtable` does,

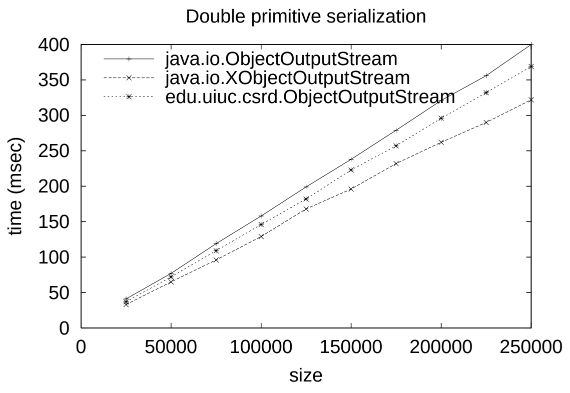


(a) ObjectOutput

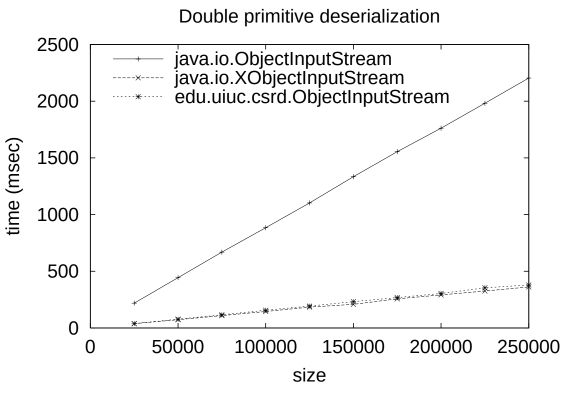


(b) ObjectInput

Figure 1: Serialization and deserialization of byte primitives

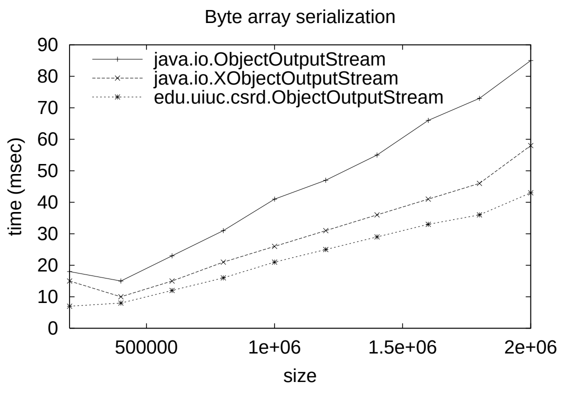


(a) ObjectOutput

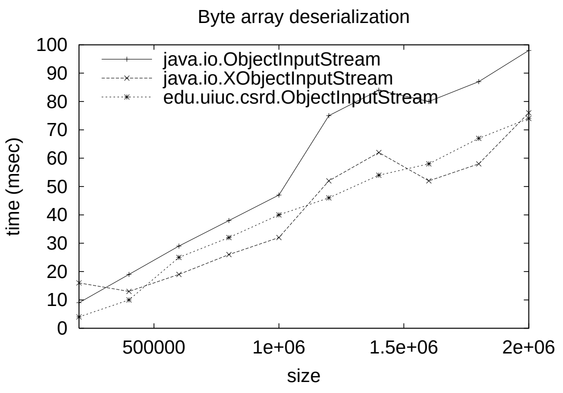


(b) ObjectInput

Figure 2: Serialization and deserialization of double primitives

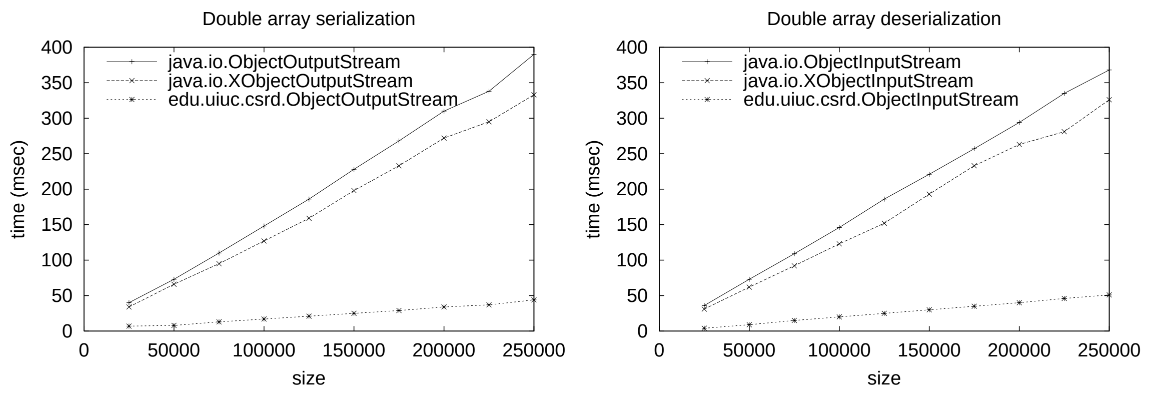


(a) ObjectOutput



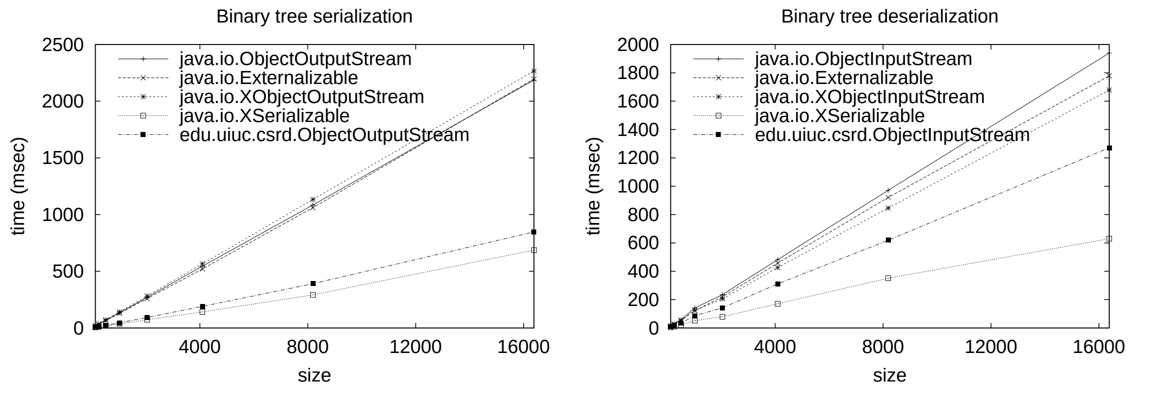
(b) ObjectInput

Figure 3: Serialization and deserialization of byte arrays



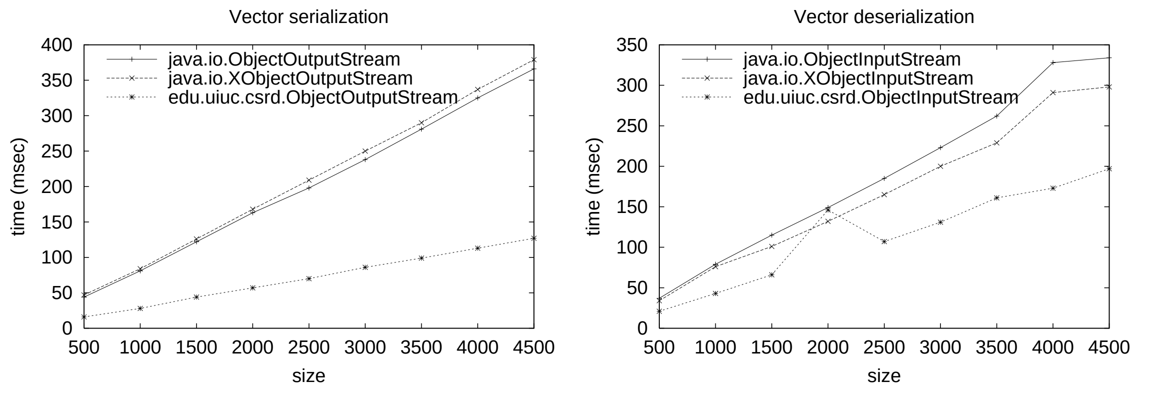
(a) ObjectOutput (b) ObjectInput

Figure 4: Serialization and deserialization of double arrays



(a) ObjectOutput (b) ObjectInput

Figure 5: Serialization and deserialization of binary trees



(a) ObjectOutput (b) ObjectInput

Figure 6: Serialization and deserialization of Vectors

RMI version	latency (ms)
JavaRMI / ObjectOutput	17
LuchatRMI / ObjectOutput	25
LuchatRMI / native object serialization	21
KaRMI / X-Stream serialization	8

Table 1: RMI latency

and can therefore be directly serialized using our implementation.

Since the individual objects in an object array do not typically occupy a contiguous block of memory, our implementation, too, has to iterate over all objects, serializing each one separately. Still, native object serialization shows a factor of 2.5 performance improvement for writing vectors and a factor of more than 1.5 for reading vectors.

4.2 RMI performance

Encouraged by results obtained in the previous section, we now evaluate the performance of our object serialization protocol within an RMI framework. We will use our own implementation of RMI, called LuchatRMI. The performance of LuchatRMI with native object serialization is compared with both Java RMI, using standard Java object serialization and LuchatRMI using standard Java object serialization.

The experiments in this section were run on the Distributed ASCI Supercomputer [1]. In our experiments we only use the DAS cluster located in Leiden. Every node in this cluster consists of a Pentium Pro 200MHz with 128 Mbytes internal RAM, running Linux 2.2, and the Linux port (version 1.2.2, release candidate 4) of the Sun JDK 1.2.2 from Blackdown with the *sunwjit* JIT compiler. Since we do not have access to a TCP/IP implementation on top of Myrinet, our experiments use the 100Mbps Fast Ethernet interconnect.

First, we will use a simple ping-pong application to determine the latency and throughput of LuchatRMI. The latency is determined by measuring the invocation time for a parameterless remote method, that returns no value. Although no parameters and return values are transmitted, object serialization is still used to communicate RMI internal data, consisting of a few primitive typed data. Table 1 lists the latencies measured. KaRMI obtains the lowest latency. The latency for LuchatRMI, in general, is higher than the latency of JavaRMI. The latency of LuchatRMI with native object serialization is slightly lower than the latency of LuchatRMI with default Java object serialization. This is surprising, since we found earlier that native code serialization of primitive types is as expensive as or even more expensive than serializing those types with the standard object serialization. We attribute the difference to the fact that setting up the serialization streams in our implementation does not involve communication, while in standard object serialization, a header is communicated.

The throughput for the different RMI implementations is determined using a similar ping-pong test, which passes different sized arrays back and forth. The results are shown in Figure 7. KaRMI obtains the highest throughput when transferring byte arrays. LuchatRMI with native object serialization still performs better than the default object serialization protocol. When transferring double arrays, our native object serialization protocol obtains an improvement of factor 3 over the other implementations.

Finally, we evaluate the performance of a parallel matrix multiply, implemented using RMI. The distributed version of matrix multiply uses 1 master process and a number of worker processes on different hosts. Each of the worker processes executes part of the iteration space of the outer loop of a standard matrix multiply algorithm.

When multiplying matrices A and B to produce matrix C, matrices A and C need to be distributed among all worker processes according to the iterations this worker is assigned to. Matrix B, however, needs to be sent in its entirety to all worker processes. In this experiment, we use matrices with elements of primitive type double.

The first step of our algorithm involves sending matrix B to all workers, using a remote method that takes the double matrix and its size as an integer and returns no value. Next, a worker is started by invoking a remote method, passing parts of A and B as double matrices and a set of integers, indicating the worker's loop iteration space. An integer job id is returned by this method. The master waits for each worker by invoking a blocking remote method, passing the job id as parameter. The result matrix is returned by invoking a remote method, passing the job id. Finally, the job is deleted from the worker by invoking a remote method that takes the integer job id as parameter. This method returns no value.

We measured the performance of our distributed matrix multiply algorithm with 1, 2, 4, and 8 worker hosts for different matrix sizes. The client was run on a separate host, thus the total number of hosts used is one more than the X-axis shows. We calculated the speedup of the distributed algorithm relative to the original sequential algorithm. The speedup results for matrix sizes 250×250 and 500×500 are shown in Figure 8.

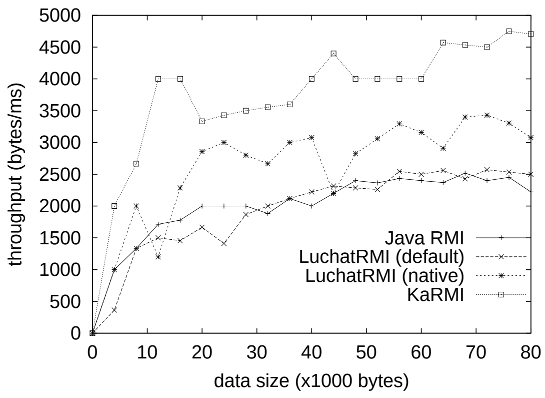
The speedup obtained with Java RMI, KaRMI, and with LuchatRMI with default object serialization is just below 2 when using 4 worker hosts and drops when using 8 worker hosts. When using native object serialization, a speedup of almost 4.5 on 8 worker hosts is obtained. When calculating the product of arrays of size 500×500 elements our object serialization implementation obtains a speedup of 6 with 8 worker hosts. These results show that object serialization is a significant factor in RMI performance. Improving the performance of object serialization can significantly improve the performance of RMI in parallel applications.

5. CONCLUSION

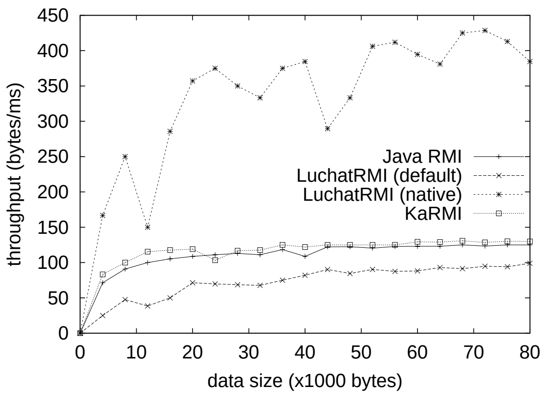
In this paper, we have shown that Java RMI is not efficient enough to be used effectively in distributed parallel programming. We have described problems with object serialization that cause the performance of Remote Method Invocation to be poor.

We have discussed a native code implementation of the object serialization protocol, that is especially targeted for distributed computing on homogeneous cluster architectures. Performance improvements on such architectures are obtained by exploiting the specific layout of certain data structures in the JVM. Additional performance is obtained by providing less information in the generated bytestreams, instead relying on this information to be present on all nodes beforehand.

Performing object serialization at the JVM level poten-

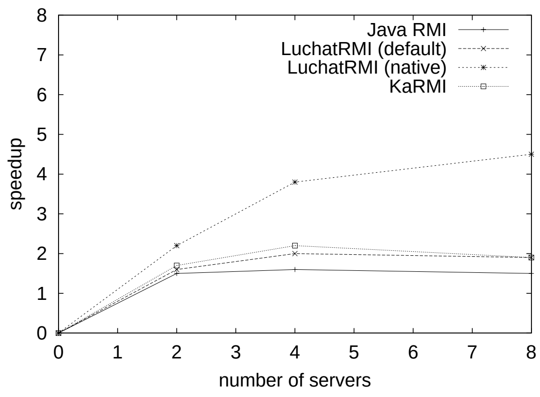


(a) byte array

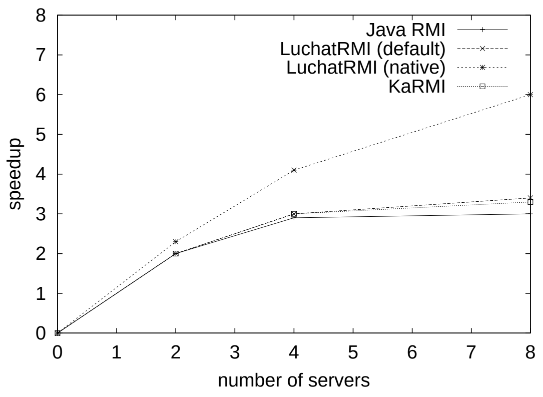


(b) double array

Figure 7: RMI throughput on the DAS



(a) matrix size = 250 x 250



(b) matrix size = 500 x 500

Figure 8: Distributed matrix multiply speedup

tially allows more efficient implementations for, for instance, serialization of arrays of primitive types. Instead of serializing each primitive separately, we copy the contiguous memory area containing the array directly to the filedescriptor associated with the output stream. Current research into multidimensional arrays for Java [10] holds promises for very efficient serialization of such data structures using our framework.

Our experiments showed a factor of almost 8 performance improvement for reading and writing double arrays. For simpler primitive types, there is still a performance improvement, although a moderate one. Reading and writing linked structures like binary trees show a factor of 1.5 and 2, respectively. Using our object serialization protocol in an RMI framework shows a higher throughput for byte arrays, and better speedup and scalability for a distributed matrix multiply of double matrices.

Future research in this project consists of comparing our object serialization implementation with implementations from more recent Java Development Kit versions. Initial tests that we did run seem to suggest that the object serialization performance in the JDK 1.3.1 from Sun has significantly improved. Although our native implementation also obtains a higher throughput in later JDK versions, the performance improvement over Sun's object serialization implementation is significantly smaller.

One possible approach to improving our object serialization implementation would be to exploit even more knowledge of the actual layout of data structures in the Java Virtual Machine. The development of our own Java Virtual Machine implementation will provide an excellent framework to experiment with this.

6. REFERENCES

[1] H.E. Bal et al. The distributed ASCI supercomputer project. *ACM SIGOPS Operating Systems Review*, 34(4):76–96, oct 2000.

[2] A. Birrell, G. Nelson, S.S. Owicki, and E. Wobber. Network Objects. *Software: Practice and Experience*, 25(S4):87–130, Dec 1995.

[3] F. Breg. *Java and High Performance Computing.* PhD thesis, Leiden University, November 2001. in progress.

[4] F. Breg and D Gannon. A Customizable Implementation of RMI for High Performance Computing. In *Proc. of Workshop on Java for Parallel and Distributed Computing of IPPS/SPDP99*, pages 733–747, Apr 1999.

[5] S. Campadello, O. Koskimies, K. Raatikainen, and H. Helin. Wireless Java RMI. In *The 4th International Enterprise Distributed Object Computing Conference*, pages 114–123, Makuhari, JAPAN, Sep 2000. IEEE Computer Society.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley Developers Press, 1996.

[7] B. Haumacher and M. Philippsen. More efficient object serialization. In *International Workshop on Java for Parallel and Distributed Computing*, Apr 1999.

[8] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, Oct 1982.

[9] Sun Microsystems Inc. Java^{tm} Object Serialization Specification, Nov 1998. revision 1.43.

[10] J.E. Moreira, S.P. Midkiff, and M. Gupta. A Comparison of Three Approaches to Language, Compiler, and Library Support for Multidimensional Arrays in Java. In *Proc. of the Joint ACM Java Grande - ISCOPE 2001 Conference*, pages 116–125, Palo Alto, CA, June 2001.

[11] R. van Nieuwpoort, J. Maassen, H.E. Bal, T. Kielman, and R. Veldema. Wide-area parallel computing in java. In *ACM 1999 Java Grande Conference*, jun 1999.

[12] L. Opyrchal and A Prakash. Efficient Object Serialization in Java. In *ICDCS 99 Workshop on Middleware*, jun 1999.

[13] M. Philippsen and B. Haumacher. Bandwidth, Latency, and other Problems of RMI and Serialization. JavaGrande report, May 1998.

[14] M. Philippsen and B. Haumacher. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[15] R. Riggs, J. Waldo, and A. Wollrath. Pickling State in the Java(tm) System. In *USENIX 1996 Conference on Object-Oriented Technologies*, Toronto, Ontario, Canada, Jun 1996.