Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++

Fabian Breg, Shridhar Diwan, Juan Villacis,
Jayashree Balasubramanian, Esra Akman, Dennis Gannon
Department of Computer Science
Indiana University

Abstract

Java RMI provides an elegant and powerful model for invoking member functions on objects that exist in remote address spaces. Unfortunately, it is a Java-to-Java communication model, and in many of the scientific applications we would like Java objects to interact with modules written in C++ or Fortran. This paper explores the design of RMI and extracts a subset of the RMI object model that is compatible with C++ and HPC++ remote object semantics. This RMI subset has been implemented to run over the Nexus runtime system and is being used as part of the LSA distributed linear system analyzer project.

1 Introduction

Many high performance computing applications in the future will be designed and built as distributed systems based on an object component architecture. While some of the components in these systems will be built as Java clients and servers, many others will be built using Fortran and C++. In one such system, the Linear System Analyzer (LSA) [7], about 22% of the source is Java, 15% is C++ and 63 % is Fortran. Java is used to program the front end of the system which serves as the graphical environment for composing computation components. The components are Fortran modules that run on sequential servers or, with the help of MPI, on parallel systems. These modules are encapsulated with HPC++ [6], which is a library based extension to C++ to support multi-threaded computation and remote method calls on distributed objects. In the construction of the LSA we were presented with the following problem: what is the best way for our Java application to communicate with remote objects written with HPC++? There were three obvious choices:

1. Use a low level socket model or simple RPC mechanisms built on that.
2. Use the emerging Java CORBA[8] standard.
3. Find a way to allow the Java programmer to use a subset of the standard Java Remote Method Invocation (RMI) mechanism that can interoperate with the corresponding HPC++ remote object model.

Our first implementation followed the first solution, but that approach proved to be awkward. While we have had extensive experience and success with CORBA, many in our programming team found the Java RMI model to be richer and more appealing. Consequently, we undertook the task of implementing an RMI-to-C++ communication interface that would be consistent with the HPC++ remote object model being used for communication between the C++/Fortran components. This paper describes what we have learned from this experiment.

There are two primary contributions of this work.

1. We explore the problem of finding the maximal subset of the Java RMI object model that can be mapped into the remote object semantics of a C++ based language. Our goal is to find those properties that are both powerful and useful for high performance, scientific applications.

2. We illustrate, by means of a set of simple experiments, that Java RMI exhibits very uneven performance. While it is extremely fast in some cases, it is subject to non-linear anomalies in others. We demonstrate that our Java-to-HPC++ prototype is able to avoid these performance problems.

2 A Java - HPC++ RMI System

Java RMI [10] provides a mechanism for making method invocations on Java objects residing in different virtual machines. Java RMI is specifically designed to operate in the Java environment and so it seamlessly integrates a distributed object model into the Java language.

While other RMI systems can be adapted to handle method invocations on remote Java objects, these systems fall short of seamless integration with the Java system due to their interoperability requirement with other languages. For example, CORBA presumes a heterogeneous, multi-language environment and thus must have a language neutral object model. In contrast, the Java language's RMI system assumes the homogeneous environment of the Java virtual machine, and the system can therefore follow the Java object model whenever possible. For example, RMI has the ability to send linked structures as arguments in a remote invocation, and the ability to send an argument whose type is a subclass of the formal parameter type. These features are either missing in CORBA or are difficult to implement.

As our goal is to implement remote method invocations between Java and C++, from a Java programmer's point of view, we want to preserve as much of the Java language's object semantics as possible. By implementing our own version of Java RMI to HPC++ communication system, we are able to support an object model that, in terms of richness, lies between the object semantics of the Java language and that of CORBA.

As a first step towards implementing a Java RMI to HPC++ communication system, we implemented a Java to Java RMI system on top of Nexus Java [5], a Java interface to the Nexus communication library [4]. This system is called Nexus RMI. Next we extended the communication mechanism underlying HPC++ so that it became compatible with the Nexus RMI communication mechanism. This established the communication link between Nexus RMI and HPC++.

The rest of the section is organized as follows. We start with a brief overview of the Java RMI system, followed by a description of Nexus RMI. Next we give a brief description of HPC++ and related concepts that are relevant to communication with Java. Finally, we describe our implementation of the Java RMI to HPC++ communication system, and the issues of language semantics involved in the implementation.

2.1 Java RMI

Java RMI is designed to simplify the communication between two objects in different virtual machines by allowing an object to invoke the methods of an object in another virtual machine, in the same way as methods on local objects are invoked. To invoke methods on a remote object, a remote reference to that object has to be obtained. Since the object resides in a different virtual machine and hence in a different name space, a registry is used to manage remote references. RMI servers can register their objects at the registry after which clients can obtain a reference to these remote objects. A more detailed description of the RMI architecture can be found in [10].

2.2 Nexus RMI

Nexus RMI is our implementation of Java RMI on top of Nexus Java, which is a Java interface to the Nexus communication library [2]. The Nexus communication library provides dynamic resource management, multi-threading and multiple methods for communication, allowing it to operate in a heterogeneous environment. Nexus Java currently implements a subset of Nexus in Java.

The interface to Nexus Java is organized around six basic abstractions. A *node* represents a physical processing resource. On a node, multiple *contexts* can be running, which can be considered to be equivalent to a JVM. In each context, multiple *threads* can be present. Communication is performed over a communication link which is created by binding a communication *startpoint* to a communication *endpoint*. To invoke methods on objects associated with an endpoint, a *remote service request* (RSR) can be issued on the startpoint connected to the endpoint. When the RSR is issued on a startpoint, a message containing a *message handler* *identifier* and a data buffer are sent over the communication link to the endpoint, after which the method specified by the handler identifier is invoked providing it with the buffer as a parameter. Nexus Java is compatible with the C/C++ language interface to the Nexus library. Thus, Nexus Java can be used to invoke RSRs on contexts that are either other JVMs or C/C++ contexts.

Nexus RMI is implemented on top of the basic RSR mechanism provided by Nexus Java. When a server binds a remote object into the registry, a startpoint referring to the remote object is also provided to the registry. When a client does a lookup for the remote object, the registry provides it with the startpoint for the remote object and this startpoint is stored inside the stub object. When the client invokes a method on the stub, the stub uses this startpoint and the Nexus Java RSR mechanism to forward the request over to the skeleton object in the server JVM.

The complete RMI system consists of the RMI communication protocol (which includes support for object serialization) and the stub and skeleton compiler. The classes in our implementation can be divided into classes that implement the client-server communication, classes that implement the registry, and the exception classes. The exception classes in our implementation are copied directly from the RMI specification.

In Nexus RMI, we have implemented the original RMI classes, thus implementing the RMI communication protocol. Nexus RMI also has support for object serialization of all object types except for exception objects. It provides a complete stub compiler implementation, which generates stubs and skeletons for remote objects. It also adds the serialization methods to serializable classes. Although Nexus RMI has support for remote exceptions, the implementation is not complete yet. Also, distributed garbage collection is not implemented yet.

2.3 HPC++

HPC++ is a C++ library and language extension framework that is being developed by the HPC++ consortium as a standard model for portable parallel C++ programming. The current HPC++ framework describes a C++ library along with compiler directives which support parallel C++ programming.

The standard architecture model supported by HPC++ is a system composed of a set of interconnected nodes. Each node is a shared-memory multiprocessor (SMP) and may have several contexts, or virtual address spaces.

The central problem associated with multi-context computation is the communication and synchronization of events between two contexts. HPC++ is based on the CC++ [3] global pointer concept and the library implements this with a $GlobalPtr < T >$ template as is done in the MPC++ Template Library [9].

A global pointer generalizes the C pointer type to support pointers to objects that exist in other address spaces. It is closely linked to the idea of a global reference which is an object that acts as proxy for a remote object. A global pointer can be used to access the remote object that it points to, or it can also be used to make method invocations on the remote object. So, a global pointer is similar to a stub object in Java RMI.

For a user defined class C with member function `foo`, the standard way to invoke `foo` through a pointer is as follows:

```
class C {
public:
  int foo(float, char);
};
C *p;
p->foo(3.14, 'x');
```

It is a bit more work to make the member function call through a global pointer. First, all members that will be called through global pointers need to be registered with the HPC++system as shown below:

```
int result;
hpcxx_id_t C_foo_id = hpcxx_register(C::foo, id_value);
HPCxx_GlobalPtr<C> P;
hpcxx_invoke(P, result, C_foo_id, 3.14, 'x');
```

To invoke the member function `foo`, the special function template `hpcxx_invoke` that calls `C::foo(3.14, 'x')` in the context which contains the object pointed to by `P`. The calling process waits until the function returns.

HPC++ is implemented on top of the Nexus communication library, making it compatible with the communication mechanism of Nexus RMI. A global pointer maintains a Nexus startpoint that points to the actual remote object. When `hpcxx_invoke` is called on a global pointer, the function uses the startpoint inside the global pointer to invoke an RSR on the remote object's context.

2.4 The Java RMI - HPC++ Communication system

2.4.1 Basic Communication Infrastructure

The communication endpoint in Java RMI is the stub object that acts as a surrogate for the remote object in another JVM. If Java RMI is to communicate with HPC++, the stub object must act as a surrogate for some remote object located in an HPC++ context. By having the startpoint in the Nexus RMI stub object point to an HPC++ object rather than a Java remote object, method calls made on the Java object are forwarded to the HPC++ object.

The first step in implementing this communication mechanism is to use a common registry between HPC++ and Nexus RMI. We used the Nexus RMI registry as the common registry. Nexus RMI already has a `Naming` class implemented that can be used to lookup and bind remote objects. So on the HPC++ side, we implemented a similar `Naming` class for registering HPC++ objects in the Nexus RMI registry. The `bind` function of the `Naming` class takes a global pointer to an HPC++ object and binds it in the registry. The global pointer is bound with an appropriate user provided name. As a result of the `bind` function call, the startpoint pointing to the HPC++ object is stored in the registry. When a Java client invokes `lookup` with the object's name, the registry provides the startpoint that points to the HPC++ object. This startpoint is kept in the client stub object thus letting the stub communicate with the HPC++ object. The `Naming` class for HPC++ also contains a `lookup` method, so that HPC++ clients can lookup Java remote objects (or HPC++ objects for that matter) and invoke methods on them.

2.4.2 Issues in Java RMI to HPC++ communication

Although being able to forward method requests from Java to HPC++ objects is a good first step, the whole process involves deeper semantics issues because of the differences in object models of Java and C++. This subsection discusses the problems we faced in implementing the Java RMI to HPC++ communication link. It also describes some as yet unresolved problems, and our proposed solutions for them.

Exception Handling. HPC++ has an exception model based on the CORBA exception model. There are two kinds of exceptions:

- system exceptions: these correspond to standard runtime errors which may occur during the execution of a request.
- user exceptions: user defined exceptions resulting from executing user code in the implementation of a method.

The system exceptions are predefined for an HPC++ implementation and are compiled into the HPC++ library. The user exceptions must be available to the HPC++ system at compile time.

On the other hand, a Java function can throw exceptions that may be dynamically loaded and hence not known at compile time. So when a Java remote object sends back an exception to the Java client, the client may have to dynamically load the exception class before throwing the exception to the client code. This is implemented in Nexus RMI by requiring the remote object to marshal the fully qualified name of the exception class along with the exception object while sending it to the client.

These differences in the exception model lead to different issues when a Java client invokes a method on an HPC++ object, and vice-versa:

- Java to HPC++ method call: As all HPC++ exceptions are available at compile time, they can be mapped into appropriate Java RMI exceptions. The skeleton code on the HPC++ side maps the exception thrown by the HPC++ object method into an appropriate RMI exception before sending it back to the Java side.
- HPC++ to Java method call: When the HPC++ client receives an exception from a Java object, it receives the fully qualified name of the exception class first. The HPC++ system maps this name into an appropriate HPC++ exception and then throws this exception to the client code. We have implemented a mapping from standard RMI exceptions to HPC++ exceptions. If the remote Java object throws an exception unknown to the HPC++ system, it just thrown an HPC++ exception called `UnknownException`. If the client were a Java client, it would have been able to load the new exception class and throw the proper exception. This limitation will probably be faced by any system that allows communication between Java and C++.

Sending an actual parameter that is a subclass of the formal parameter in a remote method request. Consider the following piece of Java code:

```
public class D implements Serializable { ... }   // parameter class
public Class E extends D { ... }                 // parameter class

public interface CIface extends Remote {         // a remote interface
  int fun(D param1, D param2) throws RemoteException;
}

class C extends UnicastRemoteObject implements CIface { // implementation class
  public int fun(D param1, D param2) { ... }
}
```

Referring to the code above, the client can invoke a method as follows:

```
CIface cref;
cref = (CIface)Naming.lookup(...); // lookup for a remote object implementing CIface

D arg1 = new D;
E arg2 = new E;
int ret = cref.fun(arg1, arg2); // arg2 is of type E
```

Here, an RMI request can send an argument whose type is a subclass of the formal parameter class. However, this will not work if the remote object is an HPC++ object. Since standard C++ cannot load classes dynamically (without depending on system specific dynamic loading functionality), HPC++ is limited to only the classes known to it at compile time. This is a major restriction when invoking methods from Java to C++. Furthermore, the Java side cannot send an argument that is a subclass of the parameter class. This can be enforced by defining the parameter class `final`, so that the Java client cannot create subclasses of the parameter class.

One way to implement this functionality for HPC++ is to use the dynamic loading facility provided by the underlying operating system to load objects, classes and functions at runtime. For example, on Solaris, the calls `dlopen` and `dlsym` can be used to dynamically load and access shared objects.

Another way to implement a restricted form of the above functionality is to require the user to enumerate all the possible types that may be used as actual parameters in the place of a formal parameter. A stub compiler can then be used to generate appropriate code that handles the case when the received parameters' type is a subclass of the type of the formal parameter.

An argument in a method call that is a reference to another argument. Referring back to the Java code above, consider the following method invocation:

```
D arg1 = new D;
D arg2 = arg1;
int ret1 = cref.fun(arg1, arg2);
```

Here, both `arg1` and `arg2` refer to the same object. In order to preserve the semantic equivalence of local requests and remote requests, the RMI system sends only one copy of the argument to the remote object. When the method `fun` is invoked on the remote object, `arg1` and `arg2` both are references to the same object. It is difficult to maintain this semantic equivalence if the remote object is an HPC++ object rather than a Java object. If the method `fun` on the HPC++ side has the signature:

```
int fun(D arg1, D arg2);
```

then `arg1` and `arg2` cannot be references to the same object. However, if instead `arg1`, `arg2` are made references to `D`, then we may be able to invoke the method on the remote C++ object with `arg1` and `arg2` both pointing to the same object. But implementing Java references as C++ references may not always work due to the restrictive nature of C++ references. For example, if the parameter sent is an array with some of the array elements referring to the same object, this solution fails since C++ cannot create an array of references.

So the only way this problem can be solved is by implementing Java references as C++ *pointers*. This strategy works even for an array type parameter because on the Java side an array of *references* to objects of type `D` is essentially an array of *pointers* to objects of type `D`.

Nexus RMI already has a mechanism to deal with this problem of object reference aliasing. On the sender side, on every remote method invocation, a table is constructed out of objects as and when they are encountered during the marshaling procedure. Whenever an object is to be marshaled into the buffer, a check is made to see whether the object is already in the table. If it is not, it is marshaled into the buffer and also inserted into the table. If it is in the table, this means that the object was previously marshaled, and so only the object's index in the table is marshaled into the buffer. Since buffers in Nexus Java have a first in - first out property for their contents, exactly the same table is reconstructed at the receiving side. While unmarshaling the buffer, if an object is found, it is inserted into the table and the receiving object reference is set to this unmarshaled object. On the other hand, if an index is found, the receiving object reference is set to the object at the table entry at the index received.

Our current implementation of Java RMI - HPC++ communication system does not address the problem of multiple arguments referring to the same object. It requires the Java side to invoke methods such that the arguments do not reference the same object, if the receiving side is HPC++. A possible solution is to implement the above Nexus RMI scheme on the HPC++ side.

Sending linked structures as arguments. In Java RMI, it is possible to send a linked list of objects just by sending one element as an argument; the semantics dictates that the entire linked list must be recreated on the receiving side.

Again, it is difficult to maintain this semantics if the receiving side is a C++ object. For example, CORBA does not support this semantics mainly because in CORBA, an object used as a parameter cannot contain a pointer to another object.

We currently do not have support for this feature in our implementation of Java RMI - HPC++ communication system. The way we can implement this mechanism on the HPC++ side is by implementing a smarter unmarshaling algorithm that can unmarshal objects (containing pointers) recursively.

Member Classes. Both in Java and C++, a class can have another class as a member class. There is a slight difference in the semantics of member classes for the two languages. In C++, a member function of the inner class cannot directly access the data members of the outer class; while in Java such an access is permitted. We feel that this difference in semantics will not impose any restrictions on the object model that our communication system supports.

3 Experimental Analysis

3.1 Overview of Experiments

We set up a simple $ping - pong$ test in which identical random data of a given size was passed between a client and server program through a remote method call. Using the measured value of the round-trip delay we calculated throughput as,

$$TP = (totalBits)/(totalRoundTripTime)$$

TP was measured for variable-size arrays of primitive and object types. The primitive data type was taken to be an 8-byte `double`. The object data type had the form,

```
public class MyDouble {
  double value;
}
```

The size of this structure was assumed to be the same as for a `double` (i.e., 8 bytes).

3.2 Implementation Environment

Our experiments tested two implementations of remote method invocation (Java RMI using a 1.1 based JVM and our Nexus RMI) versus a purely HPC++ implementation. Although we performed experiments on several architectures, time and space considerations allow us to present only three. The experimental data presented here was obtained by measuring client/server throughput between two similarly configured machines and for three types of network connections: 10-Mbit ethernet, 155-Mbit ATM, and loopback. In the ethernet case, we used two identical IBM RS/6000 117MHz 604-based powerpc quad-processor machines equipped with 256MB RAM and running AIX4.2 . We used IBM's JIT-optimized JDK1.1.2 compiler to run our Java programs, and compiled our C/C++ programs using `xlC` with `-O` optimization. In the ATM case, we used two dual-processor ultrasparcs (one operating at 200MHz with 256MB RAM, the other at 300MHz with 512MB RAM) and each running Solaris2.6 . We used Sun's JIT-optimized JDK1.1.3 compiler to run our Java programs, and compiled our C/C++ programs using the SunPro `cc` compiler with `-O` optimization. In the loopback case, we used the 200MHz dual-processor ultrasparc with the same OS and compilation parameters as in the ATM case.

Due to the wide ranges in our data, we present graphs of throughput versus array size in *loglog* format for each experiment described below.

3.3 Experiments

In this section we describe each of the four experiments in some detail. The first experiment involved using strictly Java to Java communication via Java RMI. The second experiment also involved Java to Java communication but used Nexus RMI protocol instead. The third experiment involved a combination of Nexus RMI and HPC++ objects. The fourth and final experiment measured a purely HPC++ communicating object system.

Java RMI. For the Java RMI experiment, we used Java's `System.currentTimeMillis()` to measure time. Although the time resolution was in milliseconds, our tests involved network latencies larger than a millisecond, and so Java's timing method was sufficient for our purposes.

In addition to measuring overall round-trip time, we also inserted timings within the stub and skeleton sources (as generated by Sun's `rmic` tool) to measure time spent marshaling and unmarshaling the data. Since our remote method simply returned its argument, we expected the total time to be,

$$Time = CMT + SUT + SMT + CUT \tag{1}$$

where

$$
\begin{aligned}
Time &= \text{total roundtrip time} \\
CMT &= \text{client marshal time} \\
SUT &= \text{server unmarshal time} \\
SMT &= \text{server marshal time} \\
CUT &= \text{client unmarshal time}
\end{aligned}
$$

To our surprise, we found that marshaling and unmarshaling times overlapped, i.e., data was being *pipelined* rather than passed serially. This meant that total time in Java RMI was effectively,

$$Time = max(CMT, SUT) + max(SMT, CUT) \tag{2}$$

Nexus RMI. In the Nexus RMI experiment, we use our own prototype RMI compiler to generate stub and skeleton Java code that uses the Nexus Java from Argonne. Again, we inserted timings within the generated stub/skeleton codes in order to measure marshaling/unmarshaling times. The timings we measured followed the equation given in (1). This was expected since Nexus explicitly blocks until data is either ready to send or ready to read from its internal buffers.

Our first experiments with the array of objects demonstrated the complexity of Java object serialization. The algorithm used to marshal and unmarshal the data structures in our compiler had *quadratic* complexity in the number of objects marshaled. Consequently, performance fell very rapidly in the case of an array of objects. After modifying the algorithm, the drop in performance was eliminated.

Nexus RMI/HPC++. In this experiment, we used a Java client to invoke the remote method (via Nexus RMI) on an HPC++ server following the method described in the previous subsection.

HPC++. In this experiment, we had an HPC++ client invoke a remote method (via Nexus) on an HPC++ server. The timings were obtained using Unix `getrusage` system call.

3.4 Performance Analysis

Using the data in the first graph of Figure 1, we compared the relative performance of each method on the array of primitives over ethernet. We see that Java RMI outperforms HPC++, Nexus RMI/HPC++, and Nexus RMI for sufficiently large data arrays. Since data is pipelined in Java RMI, but not in Nexus RMI, this result is reasonable. What is more surprising is that when the JIT is enabled, Java RMI is able to outperform HPC++.

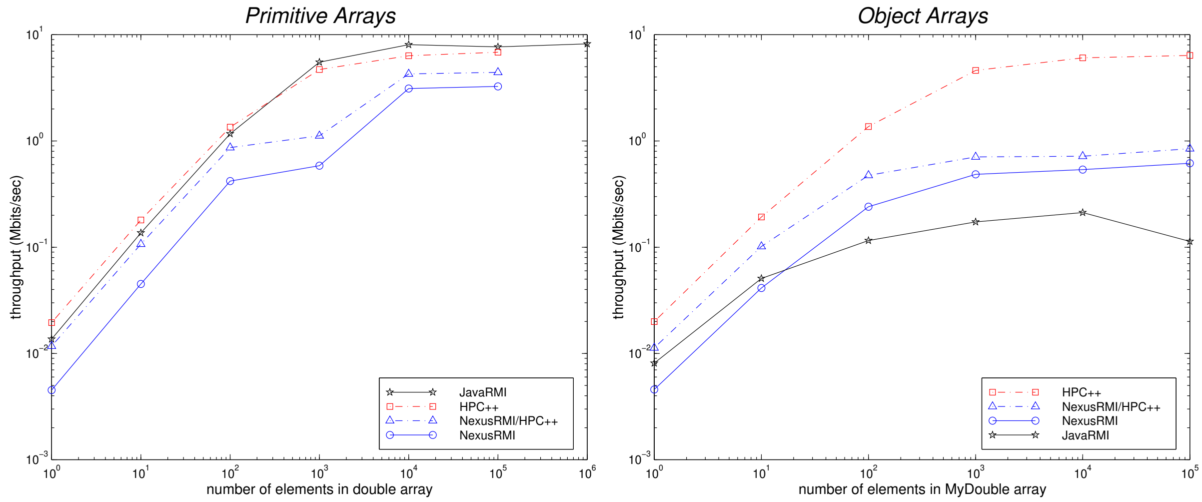

Figure 1: Ethernet TP performance for arrays of primitive type `double`, and object type `MyDouble`.

The second graph of Figure 1 illustrates the relative performance for all methods on the array of objects over ethernet. Here, Java RMI *under-performs* all the others. One reason might be that marshaling/unmarshaling *objects* is a very costly operation since RMI must maintain Java object reference semantics across different JVMs (see Section 2.4.2).

In comparing the TPs of both graphs in Figure 1, we see that the performance of Java RMI w/objects is several orders of magnitude worse than Java RMI w/primitives. Also, Java RMI w/objects exhibits a strange downturn in performance as array sizes increase.

We performed a few more experiments to narrow down the possible causes for the poor performance and anomalous behavior in Java RMI w/objects.

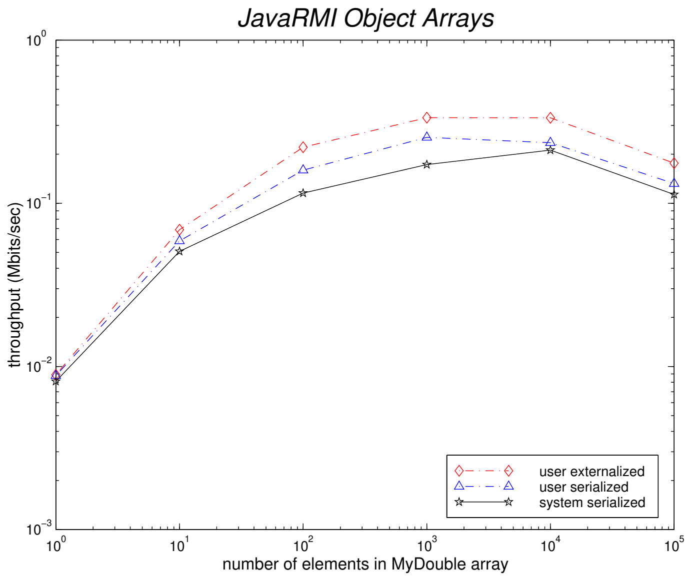

Figure 2: Java RMI object array performance with different serialization methods.

In the first experiment, we tested whether inefficient default serialization methods were to blame. For example, since we did not specify *how* to serialize `MyDouble`, the system `defaultReadObject()` and `defaultWriteObject()` methods were used for object serialization. In this experiment, we varied the degree of control a user has in the serialization process. First, we allowed for some user control by adding the `readObject()` and `writeObject()` private methods to `MyDouble`. Next, we enforced maximum user control in the serialization process by adding the `readExternal()` and `writeExternal()` public methods to `MyDouble`. In both instances, the added methods had empty bodies so that we could see the minimum effect they might have on the TP. The results of this experiment are shown in Figure 2. We discovered that although there was some improvement in the TP, the increase was less than a factor of 2. Furthermore, the *shape* of each graph was the same as the original Java RMI w/objects.

In the second experiment, we tested how the TP scaled with increasing bandwidth. We first ran each object system over a high bandwidth ATM network. The results of this experiment are shown in Figure 3. We found that although each object system took advantage of the increased bandwidth, Java RMI w/objects didn't improve much over the ethernet case. Next, we eliminated the effect the network might have on TP by executing both client and server on the same machine. In this case, the client/server communication went through the machine's loopback interface. The results of this experiment are shown in Figure 4. We found that even when communication costs are factored out, the overhead due to object serialization still limits the maximum TP that can be achieved.

From this we have concluded that an end user cannot significantly improve the performance of Java RMI w/objects, and that the performance problems in Java RMI are inherent to the serialization process itself.

It should be noted that in all cases involving Java, tests with array sizes over 10^5 could not be consistently performed due to either memory limitations in the JVM (e.g., could not allocate sufficient heap size), or the entire array would not fit in RAM (thus causing too much variability in timing due to swapping).

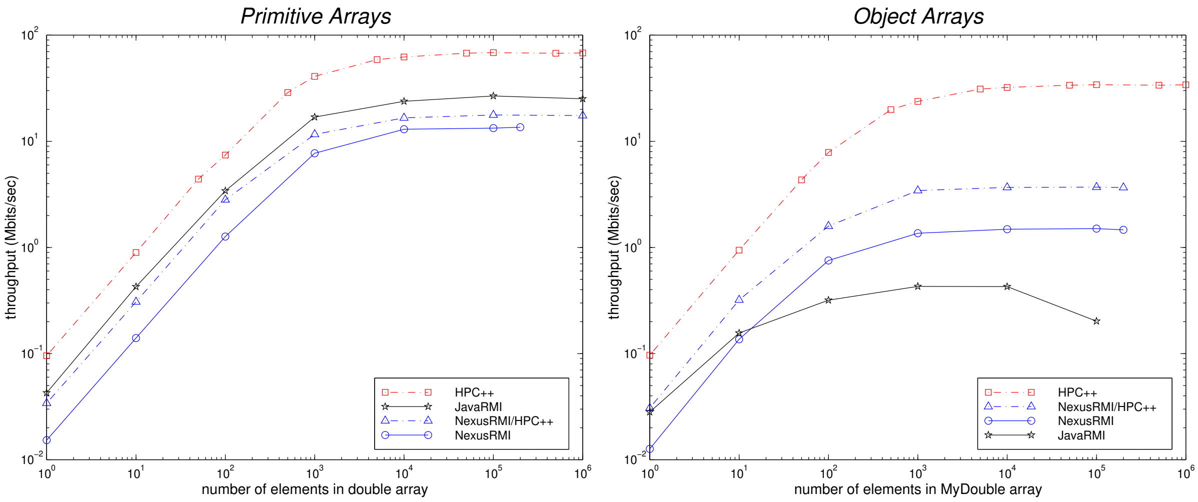

Figure 3: ATM TP performance for arrays of primitive type `double`, and object type `MyDouble`.

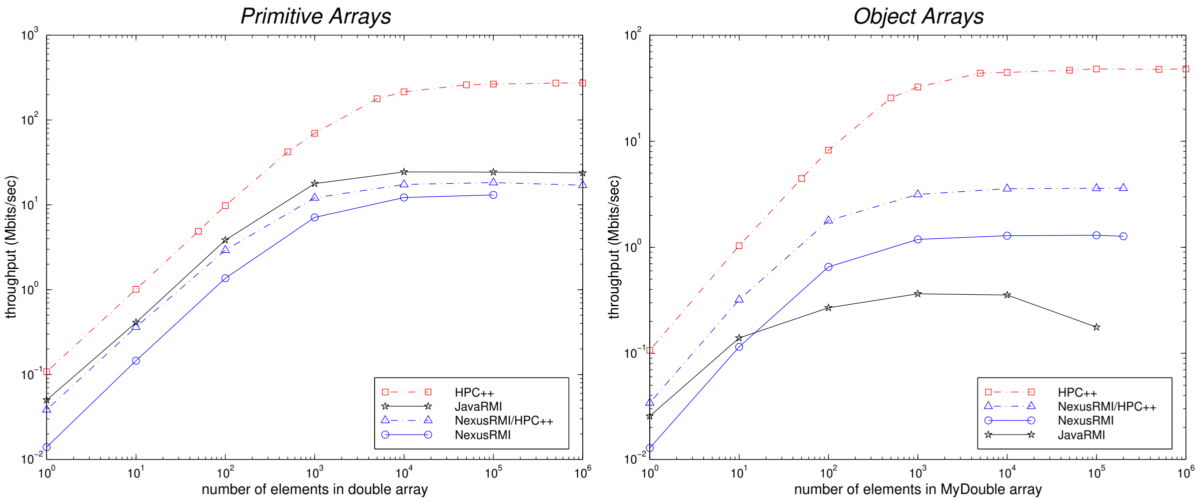

Figure 4: Loopback TP performance for arrays of primitive type `double`, and object type `MyDouble`.

4 Conclusion

In this paper we have illustrated how a subset of the Java RMI object model can be mapped onto the HPC++ remote object system. Global references in Java map to global pointers in HPC++lib and `final` object classes map to C++ classes with little ambiguity. However, the semantic mapping between these object systems has limitations. For example, parts of Java object serialization, exception handling and certain types of aliased references cannot be duplicated in HPC++. The most serious problems relate to the inability of standard C++ to dynamically load classes or to use reflection. In future work we plan to provide a more formal mapping by using an extended CORBA IDL to define the interface and semantics of the RMI subset more concretely.

The performance experiments presented here illustrate several important points. First, we demonstrate that Java RMI is very good at managing modest size arrays of primitive types on *commodity* networks such as Ethernet. Due to the pipelining mechanism implemented by RMI, Java was able to attain a better TP than HPC++ which is designed as a substrate for high performance scientific applications. This further suggests that communication libraries and runtime systems (such as Nexus) designed for high performance applications can use RMI's pipelining strategy to make a substantial gain in TP. Although problems were encountered with very large arrays, we do not consider this a serious matter. However, there are major concerns regarding the performance of RMI with arrays of object and on high performance networks such as ATM. The first problem is that RMI shows a non-linear behavior while transferring large arrays of objects. Because of the more limited semantics of our RMI subset, we were able to obtain superior performance with our implementation. On the other hand, we feel that there is nothing about the RMI semantics that requires a non-linear algorithm for serialization and we can expect to see improvements in future versions. The other problem is that RMI is unable to exploit the high bandwidth afforded by high performance networks. As demonstrated by our experiment, either for the primitive or for the object array case, RMI performs quite poorly as compared to HPC++. Our experiments also suggest that the bottleneck lies in the object serialization part. We feel that RMI is currently unsuitable for high performance applications due to its poor performance on high bandwidth networks.

In the future, we plan to repeat these tests over a wider range of high bandwidth networks and we will do a deeper analysis of the actual protocols. Our goal is to be able to allow high performance components of a distributed application to communicate with low latency and high bandwidth. It is our feeling that the performance of the current generation of Java and CORBA distributed object systems lags far behind the requirements of scientific and engineering application. However, we are confident that Java based solutions will eventually emerge as the solution.

Acknowledgments

We would like to thank Aart J.C. Bik who is the author of the `javar` compiler [1] and various routines that read bytecode representations, which we used as a basis for our Nexus RMI compiler

References

[1] Aart J. C. Bik, J. E. Villacis, and D. B. Gannon. *The JAVAR manual*, 1997. see `http://www.extreme.indiana.edu/hpjava/`.

[2] Fabian Breg. *Compiler Support for an RMI Implementation using NexusJava.* Technical report, Computer Science Dept., Indiana University, September 1997.

[3] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation, 1993. In *Research Directions in Concurrent Object Oriented Programming*, MIT Press.

[4] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *J. Parallel and Distributed Computing*, 37:70–82, 1996.

[5] I. Foster and S. Tuecke. Enabling technologies for web-based ubiquitous supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, 1997.

[6] D. Gannon, P. Beckman, E. Johnson, and T. Green. *Compilation Issues on Distributed Memory Systems*, chapter 3 HPC++ and the HPC++Lib Toolkit. Springer-Verlag, 1997.

[7] D. Gannon, R. Bramley, T. Stuckey, J. Villacis, J. Balasubramanian, E. Akman, F. Breg, S. Diwan, and M. Govindaraju. *Component Architectures for Distributed Scientific Problem Solving.* Technical report, Indiana University, September 1997.

[8] Object Management Group. The Common Object Request Broker: Architecture and specification, July 1995. Revision 2.0.

[9] Yutaka Ishikawa. *Multiple Threads Template Library.* Technical Report TR-96-012, Real World Computing Partnership, September 1996.

[10] JavaSoft. Java Remote Method Invocation (RMI), 1997. in *The JDK 1.1 RMI Specification* see `http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/index.html`.