Automatic Object Serialization in the mpiJava Interface to MPI

Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko and Sang Lim
NPAC at Syracuse University
Syracuse, NY 13244
{dbc,gcf,shko,slim}@npac.syr.edu

Abstract

The paper discusses use of the Java object serialization model for marshalling communication data in Java bindings of the Message Passing Interface standard, MPI. This approach is compared with a Java transcription of the derived datatype mechanism used for buffer description in C and Fortran bindings of MPI. We describe an implementation of the mpiJava *interface to MPI incorporating automatic object serialization. The programming model is attractive, though benchmark results show that the current JDK implementation of serialization is not fast enough for high performance applications. Strategies to improve this situation are discussed.*

1 Introduction

In the few years since the Fortran, C and C++ bindings of the Message Passing Interface, MPI, were defined [14], Java has emerged as a major language for distributed programming. There are various indications [8, 9, 10] that Java may also become an important language for scientific and parallel computing. Not surprisingly, therefore, several groups have independently developed Java bindings to MPI and Java implementations of MPI subsets. In the absence of a standarized Java API for MPI, everyone implements a different interface. In a move to remedy this situation, with support of several groups working in the area, the Java Grande Forum drafted an initial proposal for a common Java interface to MPI [4].

Three major issues arising in the design of a Java interface to MPI are: the class hierarchy, the detailed argument lists for methods, and mechanisms for describing message buffers. The proposal in [4] attempts to follow the standardized language bindings for MPI as closely as practical, while conforming to Java's object-oriented programming model and following programming conventions commonly adopted by Java programmers. MPI, even in its original C and Fortran instantiations, deliberately followed object-oriented design principles. The subsequent addition of a C++ binding for MPI [14] already defined a detailed class hierarchy. The Java binding suggested in [4] adopts that hierarchy as it stands. At the level of detailed argument lists there are many small differences between the Java interface and the established C or C++ interfaces. These differences are either enforced by restrictions of the Java language (for example the lack of call-by-reference arguments), or exploit obvious features of Java to simplify the interface (for example, explicit array-size arguments can often be omitted because they can be extracted from run-time information in a Java array). These details are quite important in practise, but they are relatively superficial. More fundamental questions occur at the level of buffer description.

A characteristic feature of MPI is its flexible method for describing message buffers consisting of mixed primitive fields scattered, possibly non-contiguously, over the local memory of a processor. These buffers are described through special objects called *derived datatypes*—run-time analogues of the user-defined types supported by languages like C. The standard MPI approach does not map very naturally into Java, because Java does not expose the the physical memory layout of user data, or necessarily implement a fixed, linear address space internally[1].

In [2, 3, 1] we suggested a Java-compatible restriction of the general MPI derived datatype mechanism, in which all primitive elements of a message buffer have the same type, and they are selected from the elements of a one-dimensional Java array passed as the buffer argument. This approach preserves some of the functionality of the original MPI mechanism—for example the ability to describe strided sections of a one dimen-

[1] For example, the garbage collector is allowed to relocate objects unpredicably at runtime, so the absolute and relative addresses of objects are effectively ill-defined.

sional buffer argument, and to represent a subset of elements selected from the buffer argument by an indirection vector. But it does not allow description of buffers containing elements of mixed primitive types.

The derived datatype mechanism is retained in the initial draft of [4], but its usefulness seems to be limited. In the context of Java, a more promising approach may be the addition a new basic datatype to MPI representing a serializable object. The buffer array passed to communication functions is still a one-dimensional array, but as well as allowing arrays with elements of primitive type, the element type is allowed to be `Object`. The serialization paradigm of Java can be adopted to transparently serialize buffer elements at source and unserialize them at destination. An immediate application is to multidimensional arrays. A Java multidimensional array is an array of arrays, and an array is an object. Therefore a multidimensional array is a one-dimensional array of objects and it can be passed directly as a buffer array. The options for representing sections of such an array are limited, but at least the one can communicate whole multidimensional arrays without explicitly copying them (of course there may well be copying inside the implementation).

1.1 Overview of this article.

This article discusses our current work on use of object serialization to marshal arguments of MPI communication operations. It builds on earlier work on the *mpiJava* interface to MPI [1], which is implemented as a set of JNI wrappers to native C MPI packages for various platforms. The original implementation of mpiJava supported MPI derived datatypes, but not object types.

Section 2 reviews the parts of the API of [4] relating to derived datatypes and object serialization. Section 3 describes our prototype implementation of automatic object serialiation in mpiJava. In section 4 we describe some benchmarks for this initial implementation. The results imply that naive use of existing Java serialization technology does not provide the performance needed for high performance message passing environments. Possible remedies for this situation are outlined briefly in the final discussion section.

1.2 Related work

Early work by the current authors on Java MPI bindings is reported in [2]. A comparable approach to creating full Java MPI interfaces has been taken by Getov and Mintchev [15, 11]. A subset of MPI is implemented in the DOGMA system for Java-based parallel programming [13]. MPI Software Technology, Inc have announced their intention to deliver a commercial Java interface to MPI called JMPI [5]. Java implementations of the related PVM message-passing environment have been reported in [16] and [7]. A pure Java implementation of MPI built on top of JPVM has been described in [6].

For an extensive discussion of performance issues surrounding object serialization see section 3 of [12] and references therein. The discussion there mainly relates to serialization in the context of fast RMI implementations. The cost of serialization is likely to be an even more critical issue in MPI, because the message-passing paradigm usually has lower overheads.

2 Review of an API for derived datatypes and Object datatypes

The MPI standard is explicitly object-based. The C++ binding specified in the MPI 2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The Java API proposed in [4] follows this model, and lifts its class hierarchy directly from the C++ binding. The major classes are illustrated in Figure 1.

The class MPI only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator `COMM_WORLD`[2]. The communicator class `Comm` is the single most important class in MPI. All communication functions are members of `Comm` or its subclasses. Another class that is relevant for the discussion below is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and other communication functions. Various basic datatypes are predefined in the package. These mainly correspond to the primitive types of Java, shown in figure 2.

The standard send and receive operations of MPI are members of `Comm` with interfaces

```
void send(Object buf, int offset, int count,
          Datatype datatype, int dst, int tag)

Status recv(Object buf, int offset, int count,
            Datatype datatype, int src, int tag)
```

In both cases the *actual* argument corresponding to `buf` must be a Java array with element type determined by the `datatype` argument. If the specified type

[2] It has been pointed out that if multiple MPI threads are allowed in the same Java VM, the default communicator cannot be obtained from a static variable. The final version of the API may change this convention.

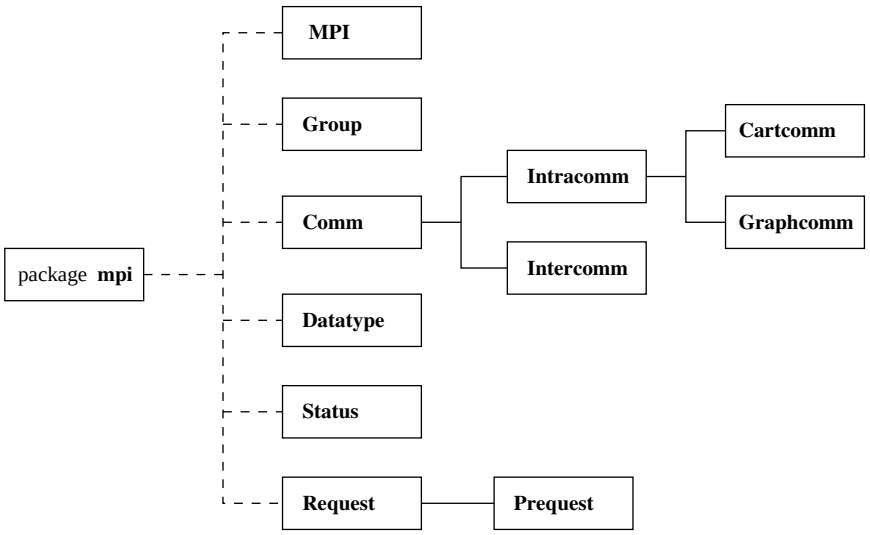

Figure 1. Principal classes of MPI

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.OBJECT	`Object`

Figure 2. Basic datatypes in proposed Java binding

corresponds to a primitive type, the buffer will be a one-dimensional array. Multidimensional arrays can be communicated directly if an object type is specified, because an individual array can be treated as an object. Communication of object types implies some form of serialization and unserialization. This could be the built-in serialization provided in current Java environments, or it could be some specialized serialization optimized for MPI (or some combination of the two).

Buffer array arguments are followed by offsets that specify the element in the array where the message actually starts.

Besides object types the draft Java binding proposal retains a model of MPI derived datatypes. In C or Fortran bindings of MPI, derived datatypes have two roles. One is to allow messages to contain mixed types. The other is to allow noncontiguous data to be transmitted.

The first role involves using the `MPI_TYPE_STRUCT` derived data constructor, which allows one to describe the physical layout of, say, a C *struct* containing mixed types. This will not work in Java, because Java does not expose the low-level layout of its objects. In C or Fortran `MPI_TYPE_STRUCT` also allows one to incorporate displacements computed as differences between absolute addresses, so that parts of a single message can come from separately declared arrays and other variables. Again there is no very natural way to do this in Java. Effects similar to of these uses of `MPI_TYPE_STRUCT` can be achieved by using `MPI.OBJECT` as the buffer type, and relying on object serialization.

We conclude that in the Java binding the first role of derived dataypes should probably be abandoned—derived types can only include elements of a single basic type. This leaves description of noncontiguous buffers as the essential role for derived data types.

Every derived data type constructable in the Java binding has a uniquely defined *base type*. This is one of the 9 basic types enumerated above. A *derived datatype* is an object that specifies two things: a base type and a sequence of integer displacements. In contrast to the C and Fortran bindings the displacements can be interpreted in terms of subscripts in the buffer array argument, rather than as byte displacements.

For example the type constructor `indexed` is a member of `Datatype` with interface

```
Datatype indexed(int [] arrayOfBlocklengths,
                 int [] arrayOfDisplacements)
```

This is a binding of the standard MPI operation `MPI_TYPE_INDEXED`. It constructs a new datatype representing replication of the original datatype (to which

the method is applied) into a sequence of blocks. Each block can contain a different number of copies and have a different displacement. The base type of the new datatype will be the same as the base type of the original type. If the displacement sequence of the original type was

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent[3] ex, and `B` is `arrayOfBlocklengths` argument and `D` is `arrayOfDisplacements` argument, the new datatype will have displacement sequence

$$\begin{aligned}
\{\ & disp_0 + \mathtt{D}[0] \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[0] \cdot ex, \\
& disp_0 + (\mathtt{D}[0] + 1) \cdot ex, \ldots, disp_{n-1} + (\mathtt{D}[0] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \ldots, \\
& \qquad disp_{n-1} + (\mathtt{D}[0] + \mathtt{B}[0] - 1) \cdot ex, \\
& \ldots, \\
& disp_0 + \mathtt{D}[c-1] \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[c-1] \cdot ex, \\
& disp_0 + (\mathtt{D}[c-1] + 1) \cdot ex, \ldots, \\
& \qquad disp_{n-1} + (\mathtt{D}[c-1] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\mathtt{D}[c-1] + \mathtt{B}[c-1] - 1) \cdot ex, \ldots, \\
& \qquad disp_{n-1} + (\mathtt{D}[c-1] + \mathtt{B}[c-1] - 1) \cdot ex\ \}
\end{aligned}$$

Here, c is the number of blocks.

In Java the derived dataype constructed by `indexed` has a potentially useful role. It allows to send (or receive) messages containing values scattered randomly in some one-dimensional array. The draft proposal incorporates versions of other type constructors from MPI including `MPI_TYPE_VECTOR` for strided sections. We note, though, that the value of providing strided sections is reduced because Java has no natural mapping between elements of its multidimensional arrays and elements of equivalent one-dimensional arrays. This thwarts one common use of strided sections, for representing portions of multidimensional arrays.

3 Implementation issues for Object datatypes

As described in the previous section the proposal of [4] includes a restricted, Java-compatible version of the general datatype mechanism of MPI. The proposal retains much of the complexity of the standard MPI mechanism, but its value is apparently reduced in Java. In this section we will discuss the other option for representing complex data buffers in the Java binding—introduction of an `MPI.OBJECT` datatype.

[3] The *extent* of a dataype is a measure of the distance between its smallest and largest displacement.

It is natural to assume that the elements of arrays passed as buffer arguments to `send` and other output operations are objects whose classes implement the `Serializable` interface. There are at least two ways one may consider communicating object types in the MPI interface

1. Use the standard `ObjectOutputStream` to convert the object buffers to byte vectors, and communicate these byte vectors using the same method as for primitive byte buffers (for example, this might involve a native method call to C MPI functions). At the destination, use the standard `ObjectInputStream` to rebuild the objects.

2. Replace calls to the `writeObject`, `readObject` methods on the standard streams with specialized functions that use platform specific knowledge to communicate data fields more efficiently. For example, one might replace `writeObject` with a native method that creates an MPI derived datatype structure describing the layout of data in the object, and passes this buffer descriptor to a native `MPI_Send` function.

In the second case our implementation is responsible for prepending a suitable type descriptor to the message, so that objects can be reconstructed at the receiving end before the data fields are copied to them. This complexity is hidden in the first approach.

Evidently the first implementation scheme is more straightforward, and only this approach will be considered in the rest of this section. We discuss an implementation based on the *mpiJava* wrappers, combining standard JDK object serialization methods with a JNI interface to native MPI. Benchmark results presented in the next section suggest, however, that something like the second approach (or some suitable combination of the two) deserves serious consideration.

The original version of mpiJava was a direct Java wrapper for standard MPI. Apart from adopting an object-oriented framework, it added only a modest amount of code to the underlying C implementation of MPI. Derived datatype constructors, for example, simply called the datatype constructors of the underlying implementation and returned a Java object containing a representation of the C handle. A `send` operation or a `wait` operation, say, dispatched a single C MPI call. Even using standard JDK object serialization and a native MPI package, uniform support for the `MPI.OBECT` basic type complicates the wrapper code significantly.

In the new version of the wrapper, every send, receive, or collective communication operation tests if the base type of the datatype argument describing a buffer

is OBJECT. If not—if the buffer element type is a primitive type—the native MPI operation is called directly, as in the old version. If the buffer is an array of objects, special actions must be taken in the wrapper. If the buffer is a send buffer, the objects must be serialized. To support MPI derived datatypes as described in the previous section, we must also take account of the possibility that the message is actually some subset of the of array of objects passed in the buffer argument, selected according to the displacement sequence of the derived datatype[4]. In the implementation, a method

```
byte [] Object_Serialize(Object   buf,
                         int      offset,
                         int      count,
                         Datatype type)
```

takes the send buffer and descriptor, and returns a byte vector containing the serialized data. At the receiving end a corresponding Object_deserialize method is called. Making the Java wrapper responsible for handling derived data types when the base type is OBJECT requires additional state in the Java-side Datatype class. In particular the Java object explicitly maintains the displacement sequence as an array of integers.

A further set of changes to the implementation arises because the size of the serialized data is not known in advance, and cannot be computed at the receiving end from type information available there. Before the serialized data is sent, the size of the data must be communicated to the receiver, so that a byte receive buffer can be allocated. We send two physical messages—a header containing size information, followed by the data[5]. This, in turn, complicates the implementation of the various wait and test methods on communication request objects, and the start methods on persistent communication requests. A wait operation, for example, can no longer simply dispatch a native MPI_WAIT call. Instead it must check what kind of non-blocking communication the request object was created for. If it was a receive operation involving objects, the wait method makes an MPI_WAIT call to wait for the header, followed by a blocking MPI_RECV call to accept the data. This implies the introduction of extra state to the Java Request class, storing the argument list of the operation that created the request object.

Finally, comparable changes are needed in the collective communication wrappers. A gather operation, for example, involving object types is implemented as an MPI_GATHER operation to collect all message lengths, followed by an MPI_GATHERV to collect possibly different-sized data vectors.

[4]To clarify: at this point we are supporting *both* MPI derived datatypes (allowing non-contiguous buffers) *and* objects as basic MPI types. Uniformity requires support, in particular, of derived types whose base type is Object. When only buffers of primitive type were supported, the wrapper code never dealt with displacement sets in derived types directly—they were handled inside the native MPI. But of course the native MPI cannot deal with derived types that have objects as their basic components.

[5]In simple cases it would be possible to use MPI_PROBE to find out the physical message size at the receiving end before posting the receive, but this approach does not work well for all non-blocking communication modes (or for collective communication).

4 Benchmark results for multidimensional arrays

We assume that in the kind of *Grande* applications where MPI is most likely to be used, some of the most pressing performance issues about buffer description and object communication will concern arrays and multidimensional arrays of small objects—most especially arrays with primitive elements such as ints and floats. For initial benchmarks we concentrated on the overheads introduced by object serialization when the objects contain many arrays of primitive elements. Specifically we concentrated on communication of two-dimensional arrays with primitive elements.

The immediate goal, therefore, was to understand the overheads in communicating multidimensional Java arrays using generic object serialization technology. The "ping-pong" method was used to time point-to-point communication of an N by N array of primitive elements treated as a one dimensional array of objects, and compare it with communication of an N^2 array without using serialization. As an intermediate step we also timed communication of a 1 by N^2 arrey treated as a one-dimensional (size 1) array of objects. This allows us to extract an estimate of the overhead to "serialize" an individual primitive element.

The code to send the N^2 float vector, for example, looks schematically like

```
float [] buf = new float [N * N] ;
MPI.COMM_WORLD.send(buf, 0, N * N, MPI.FLOAT,
                    dst, tag) ;
```

and the code to receive it looks like

```
float [] buf = new float [N * N] ;
MPI.COMM_WORLD.recv(buf, 0, N * N, MPI.FLOAT,
                    src, tag) ;
```

For the $N \times N$ float arrays we have

```
float [] [] buf = new float [N] [N] ;
MPI.COMM_WORLD.send(buf, 0, N, MPI.OBJECT,
                    dst, tag) ;
```

and

```
float [] [] buf = new float [N] [] ;
MPI.COMM_WORLD.recv(buf, 0, N, MPI.OBJECT,
                    src, tag) ;
```

For the $1 \times N^2$ float arrays we have

```
float [] [] buf = new float [1] [N * N] ;
MPI.COMM_WORLD.send(buf, 0, 1, MPI.OBJECT,
                    dst, tag) ;
```

and

```
float [] [] buf = new float [1] [] ;
MPI.COMM_WORLD.recv(buf, 0, 1, MPI.OBJECT,
                    src, tag) ;
```

As a crude timing model for these benchmarks, one can assume that there is a cost $t^{\mathrm{T}}_{\mathrm{ser}}$ to serialize each primitive element of type `T`, an additional cost $t^{\mathrm{vec}}_{\mathrm{ser}}$ to serialize each subarray, similar constants $t^{\mathrm{T}}_{\mathrm{unser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ for unserialization, and a cost $t^{\mathrm{T}}_{\mathrm{com}}$ to physically tranfser each element of data. Then the total time for benchmarked communications should be

$$t^{\mathtt{T[}N^2\mathtt{]}} = c + t^{\mathrm{T}}_{\mathrm{com}} N^2 \quad (1)$$

$$t^{\mathtt{T[1][}N^2\mathtt{]}} = c' + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 \quad (2)$$

$$t^{\mathtt{T[}N\mathtt{][}N\mathtt{]}} = c'' + (t^{\mathrm{vec}}_{\mathrm{ser}} + t^{\mathrm{vec}}_{\mathrm{unser}}) N + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 \quad (3)$$

These formulae do not attempt to explain the constant initial overhead, don't take into account the extra bytes for type description that serialization introduces into the stream, and ignore possible non-linear costs associated with analysing object graphs, etc. Empirically these effects are small for the range of N we consider.

All measurements were performed on a cluster of 2-processor, 200 Mhz UltraSparc nodes connected through a SunATM-155/MMF network. The underlying MPI implementation was Sun MPI 3.0 (part of the Sun HPC package). The JDK was jdk1.2beta4. Shared memory results quoted are obtained by running two processes on the processors of a single node. Non-shared-memory results are obtained by running peer processes in different nodes.

In a series of measurements, element serialization and unserialization timing parameters were estimated by independent benchmarks of the serialization code. The parameters $t^{\mathrm{vec}}_{\mathrm{ser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ were estimated by plotting the difference between serialization and unserialization times for `T[1][`N^2`]` and `T[`N`][`N`]`. The raw communication speed was estimated from ping-pong results for $t^{\mathtt{T[}N^2\mathtt{]}}$. Table 1 contains the resulting estimates of the various parameters for `byte`, `int` and `float` elements.

Figures 3 to 4 plot actual measured times from ping-pong benchmarks for the mpiJava sends and receives of

$t^{\mathrm{byte}}_{\mathrm{ser}}$	$= 0.043$	$t^{\mathrm{int}}_{\mathrm{ser}}$	$= 0.74$
$t^{\mathrm{byte}}_{\mathrm{unser}}$	$= 0.027$	$t^{\mathrm{int}}_{\mathrm{unser}}$	$= 0.42$
$t^{\mathrm{byte}}_{\mathrm{com}}$	$= 0.062^{\dagger}$	$t^{\mathrm{int}}_{\mathrm{com}}$	$= 0.25^{\dagger}$
$t^{\mathrm{byte}}_{\mathrm{com}}$	$= 0.008^{\S}$	$t^{\mathrm{int}}_{\mathrm{com}}$	$= 0.038^{\S}$
$t^{\mathrm{float}}_{\mathrm{ser}}$	$= 2.1$	$t^{\mathrm{vec}}_{\mathrm{ser}}$	$= 100$
$t^{\mathrm{float}}_{\mathrm{unser}}$	$= 1.4$	$t^{\mathrm{vec}}_{\mathrm{unser}}$	$= 53$
$t^{\mathrm{float}}_{\mathrm{com}}$	$= 0.25^{\dagger}$		
$t^{\mathrm{float}}_{\mathrm{com}}$	$= 0.038^{\S}$		

Table 1. Estimated parameters in serialization and communication timing model. The $t^{\mathrm{T}}_{\mathrm{com}}$ values are respectively for non-shared memory (†) and shared memory (§) implementations of the underlying communication. All timings are in microseconds.

arrays with `byte`, `int` and `float` elements. In the plots the array extent, N, ranges between 128 and 1024. The measured times for $t^{\mathtt{T[}N^2\mathtt{]}}$, $t^{\mathtt{T[1][}N^2\mathtt{]}}$ and $t^{\mathtt{T[}N\mathtt{][}N\mathtt{]}}$ are compared with the formulae given above (setting the c constants to zero). The agreement is good, so our parametrization is assumed to be realistic in the regime considered.

According to table 1 the overhead of Java serialization nearly always dominates other communiation costs. In the worst case—floating point numbers—it takes around 2 microseconds to serialize each number and a smaller but comparable time to unserialize. But it only takes a few *hundredths* of a microsecond to communicate the word through shared memory. Serialization slows communication by nearly two orders of magnitude! When the underlying communication is over a fast network rather than through shared memory the raw communication time is still only a fraction of a microsecond, and serialization still dominates that time by about one order of magnitude. For `byte` elements serialization costs are smaller, but still larger than the communication costs in the fast network and still much larger than the communication cost through shared memory. Serialization costs for `int` elements are intermediate.

The constant overheads for serializing each subarray, characterized by the parameters $t^{\mathrm{vec}}_{\mathrm{ser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ are also quite large, although, for the array sizes considered here they only make a dominant contribution for the `byte` arrays, where individual element serialization

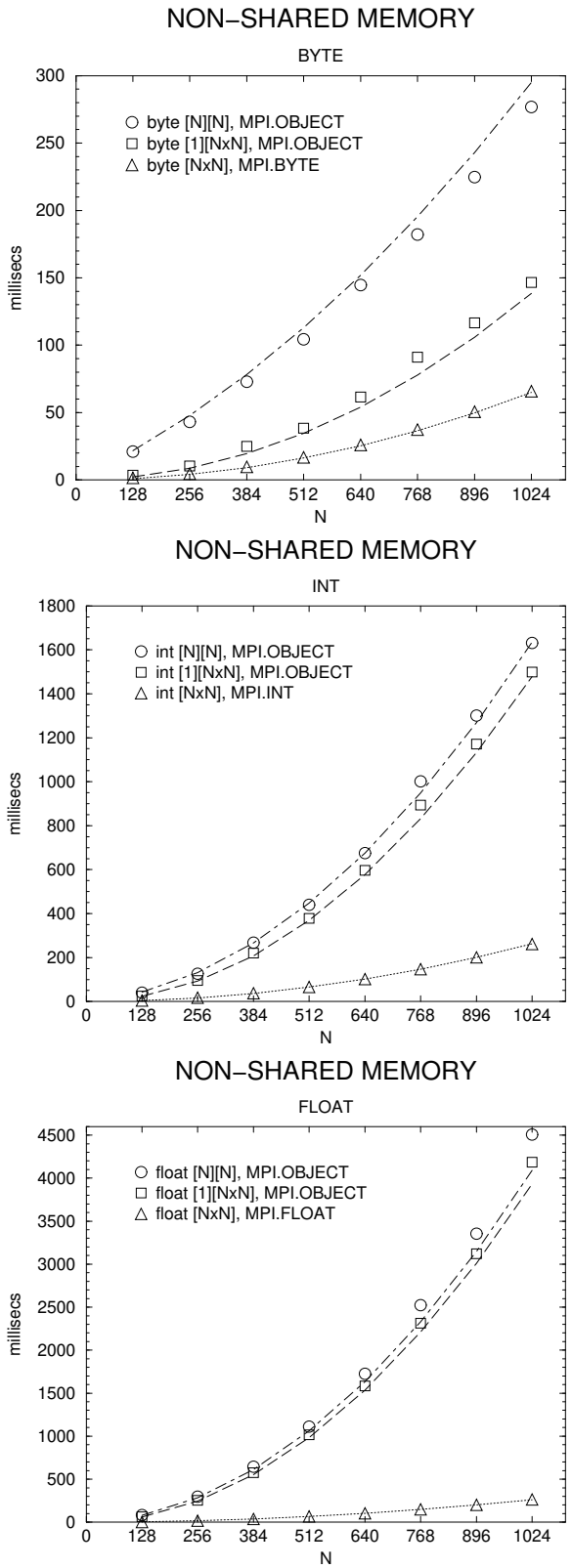

Figure 3. Communication times from Pingpong benchmark in non-shared-memory case, compared with model defined by Equations 1 to 3 and Table 1.

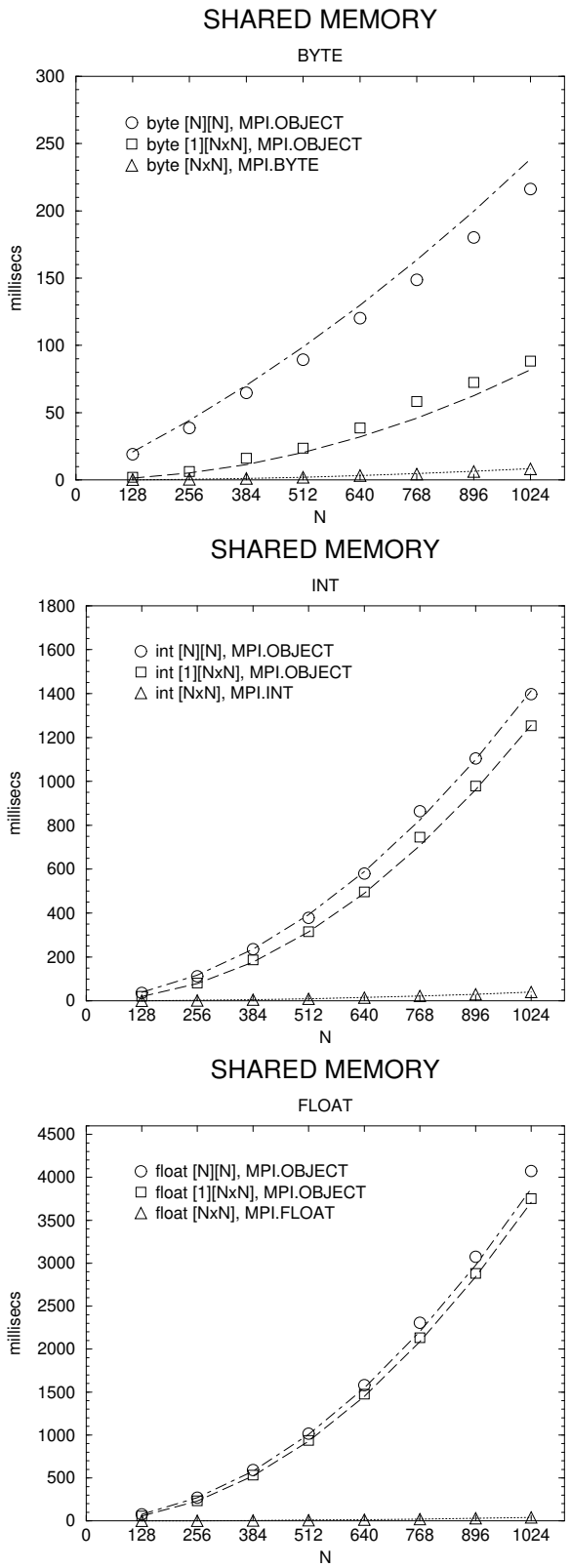

Figure 4. Communication times in shared-memory case, compared with model.

is relatively fast[6].

5 Discussion

In Java, the object serialization model for data marshalling has various advantages over the MPI derived type mechanism. It provides much (though not all) of the flexibility of derived types, and is presumably simpler to use. Object serialization provides a natural way to deal with multidimensional arrays. Such arrays are, of course, very common in scientific programming. The Java mapping of derived datatypes, on the other hand, is problematic—a significant part of the flexibility in the original C/Fortran specification is lost in the transcription to Java. It is at least arguable that in Java the MPI derived datatypes mechanism should be abandoned altogether in favour of using object serialization.

Our initial implementation of automatic object serialization in the context of MPI is somewhat impaired by performance of the serialization code in the current Java Development Kit. In our implementation buffers were serialized using standard technology from the JDK. The benchmark results from section 4 show that this implementation of serialization introduces very large overheads relative to underlying communication speeds on fast networks and symmetric multiprocessors. Similar problems were reported in the context of RMI implementations in [12]. We find that in the context of fast message-passing environments (not surprisingly) the issue is even more critical. Overall communication performance can easily be downgraded by an order of magnitude or more.

The standard Java serialization framework allows the programmer to provide optimized serialization and unserialization methods for particular classes, but in scientific programming we are often more concerned with the speed of operations on arrays, and especially arrays of primitive types. The documented parts of the standard Java framework for serialization do not to our knowledge allow a way to customize handling of arrays. However the available source code for `ObjectOutputStream` and `ObjectInputStream` classes includes the methods for serializing arrays, and we are optimistic that by tuning these classes it should be possible to greatly improve performance for cases that concern us here. In any case the message is clear: we need much faster implementations of object serialization, better attuned to the needs of scientific computation. Especially, arrays of primitive elements need to be handled much more carefully.

While we expect that there is considerable scope to optimize the JDK serialization software, an interesting alternative from the point of view of ultimate efficiency is to replace calls to the `writeObject`, `readObject` methods with specialized, MPI-specific, functions. A call to `writeObject`, for example, could be replaced with a native method that creates an MPI derived datatype structure describing the layout of data in the object. This would allow one to provide the conceptually straightforward object serialization model at the user level, while retaining the option of fast, "zero-copy" communication strategies (enabled by MPI derived datatypes) inside the implementation.

We are investigating the practicality of both these approaches to optimization.

[6] Our timing model assumed the values of these parameters is independent of the element type. This is only approximately true, and the values quoted in the table and used in the plotted curves are averages. Separately measured values for `byte` arrays were smaller than these averages, and for `int` and `float` arrays they were larger.

References

[1] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, Sept. 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[2] B. Carpenter, Y.-J. Chang, G. Fox, D. Leskiw, and X. Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.

[3] B. Carpenter, G. Fox, G. Zhang, and X. Li. A draft Java binding for MPI., Nov. 1997. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[4] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, Nov. 1998. http://www.javagrande.org/.

[5] G. Crawford III, Y. Dandass, and A. Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users. http://www.mpi-softtech.com/publications.

[6] K. Dincer. jmpi and a performance instrumentation analysis and visualization tool for jmpi. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, Sept. 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[7] A. J. Ferrari. JPVM: Network parallel computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.

[8] G. C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling*, volume 9(6) of *Concurrency: Practice and Experience*, June 1997.

[9] G. C. Fox, editor. *Java for Computational Science and Engineering—Simulation and Modelling II*, volume 9(11) of *Concurrency: Practice and Experience*, Nov. 1997.

[10] G. C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear. http://www.cs.ucsb.edu/conferences/java98.

[11] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.

[12] Java Grande Forum. Java Grande Forum report: Making Java work for high-end computing. Technical Report JGF-TR-1, Java Grande Forum, Nov. 1998. http://www.javagrande.org/.

[13] G. Judd, M. Clement, and Q. Snell. DOGMA: Distributed object group management architecture. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* University of Tenessee, Knoxville, TN, June 1995. http://www.mcs.anl.gov/mpi.

[15] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. Technical Report TR-CSPE-07, University of Westminster, School of Computer Science, Harrow Campus, July 1997.

[16] N. Yalamanchilli and W. Cohen. Communication performance of Java based Parallel Virtual Machines. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, Concurrency: Practice and Experience, 1998. To appear.