Object Serialization for Marshalling Data in a Java 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

November 5, 1999

Abstract

Several Java bindings to Message Passing Interface (MPI) software have been developed recently. Message buffers have usually been restricted to arrays with elements of primitive type. We discuss adoption of the Java object serialization model for marshalling general communication data in MPI-like APIs. This approach is compared with a Java transcription of the standard MPI derived datatype mechanism. We describe an implementation of the *mpiJava* interface to MPI that incorporates automatic object serialization. Benchmark results confirm that current JDK implementations of serialization are not fast enough for high performance messaging applications. Means of solving this problem are discussed, and benchmarks for greatly improved schemes are presented.

1 Introduction

The Message Passing Interface standard, MPI [15], defines an interface for parallel programming that is portable across a wide range of supercomputers and workstation clusters. The MPI Forum defined bindings for Fortran, C and C++. Since those bindings were defined, Java has emerged as a major language for distributed programming, and there are reasons to believe that Java may rapidly become an important language for scientific and parallel computing [8, 9, 10]. Over the past two years several groups have independently developed Java bindings to MPI and Java implementations of MPI subsets. With support of several groups working in the area, the Java Grande Forum drafted an initial proposal for a common MPI-like API for Java [4].

A characteristic feature of MPI is its flexible method for describing message buffers containing 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 modern procedural languages. The standard MPI approach does not map very naturally into Java. 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 dimensional 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.

This version of the MPI derived datatype mechanism was retained in the initial draft of [4], but its value is not yet certain. 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 one can communicate whole multidimensional arrays without explicitly copying them (though there may 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 an implementation of automatic object serialization in mpiJava. In section 4 we discuss benchmarks for this initial implementation. The results confirm that naive use of existing Java serialization technology does not provide the performance needed for high performance message passing environments. Section 5 illustrates how various overheads of serialization can be eliminated by customizing the object serialization stream classes. The final section relates these results to other work, and draws some conclusion.

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 [17, 11]. A subset of MPI is implemented in the DOGMA system for Java-based parallel programming [13, 14]. A pure Java implementation of MPI built on top of JPVM has been described in [6] (JPVM is a pure Java implementation of the Parallel Virtual Machine message-passing environment [7]). So far these systems have not attempted to use object serialization for data marshalling.

For an extensive discussion of performance issues surrounding object serialization see section 3 of [12] and references therein. Work of the Karlsruhe group is also reported in [18]. The discussion there mainly relates to serialization in the context of fast RMI (Remote Method Invocation) implementations. As we may anticipate, the cost of serialization is an even more critical issue in MPI, because the message-passing paradigm usually has lower overheads.

2 Datatypes in an MPI-like API for Java

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 of MPI.

In our Java version a class `MPJ` with only static members acts as a module containing global services, such as initialization of the message-passing layer, and many global constants including a default communicator `COMM_WORLD`[1]. 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 1.

The methods corresponding to 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 compatible with the `datatype` argument. If the specified type corresponds to a primitive type, the buffer must be a one-dimensional array. Multidimensional arrays can be communicated directly if an object type

[1] 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.

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

Figure 1: Basic datatypes in proposed Java binding

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 (as we discuss at length in section 5) it could be some specialized serialization tuned for message-passing.

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. (But effects similar to these uses of `MPI_TYPE_STRUCT` can be achieved by using `MPJ.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 remaining 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.)

An MPI derived dataype constructor such as `MPI_TYPE_INDEXED`, which allows an arbitray indirection array, has a potentially useful role in Java. It allows to send (or receive) messages containing values scattered randomly in some one-dimensional array. The draft proposal incorporates versions of this and other type constructors from MPI including `MPI_TYPE_VECTOR` for strided sections.

3 Adding serialization to the API

In this section we will discuss the other option for representing complex data buffers in the Java API of [4]—introduction of an `MPJ.OBJECT` datatype.

It is natural to assume that the elements of buffers passed 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 naive use of serialization streams with more specialized code that uses platform-specific knowledge to communicate data fields efficiently. For example, one might modify the standard `writeObject` in such a way that a native method creates an MPI derived datatype structure describing the layout of data in the object, and this buffer descriptor could be passed 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 data is copied to them.

The first implementation scheme is more straightforward, and this approach will be considered in the remainder 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 that something like the second approach (or some suitable combination of the two) deserves serious consideration, hence section 5 describes one realization of this scheme.

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 exploiting standard JDK object serialization and a native MPI package, uniform support for the `MPJ.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. We *also* support MPI-like derived datatypes as described in the previous section. On grounds of uniformity, these should be definable with base type `OBJECT`, just as for primitive elements. The message is then some subset of the array of objects passed in the buffer argument, selected according to the displacement sequence of the derived datatype. This case must be dealt with in the the Java wrapper, because a native `MPI_Datatype` entity cannot be constructed to directly represent Jave objects. Thus when the base type is `OBJECT` the Java-side `Datatype` class requires additional fields; it 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[2]. 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, and ends up requiring extra fields in the Java `Request` class. 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.

These changes were made throughout the mpiJava API, and will be included in the next release of the software.

4 Benchmark results for multidimensional arrays

For the sake of concrete discussion we will make an assumption that, in the kind of *Grande* applications where MPI is likely to be used, some of the most pressing performance issues concern arrays and multidimensional arrays of small objects—especially arrays of primitive elements such as `int`s and `float`s. For benchmarks we therefore 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[3].

[2] A better protocol would be to eagerly send data for short messages in the header, assuming some fixed-size buffer is preallocated at the receiving end. The two-message protocol would be reserved for long messages. This marginally complicates the implementation but does not essentially change the rest of the discussion, or the benchmark results presented below, since the latter concentrate on the asymptotic case. We are grateful to one of the referees for raising this point.

[3] We note that there some debate about whether the Java model of multidimensional arrays is the most appropriate one for high performance computing. There are various proposals for optimized HPC array class libraries [16]. See section 6 for some further discussion.

N^2 **float vector**

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

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

$N \times N$ **float array**

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

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

$1 \times N^2$ **float array**

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

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

Figure 2: Send and receive operations for various array shapes.

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 case we also timed communication of a 1 by N^2 array 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 for sending and receiving the various array shapes is given schematically in Figure 2.

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

$$
\begin{aligned}
t^{\mathtt{T}\,\mathtt{[}N^2\mathtt{]}} &= c + t^{\mathrm{T}}_{\mathrm{com}} N^2 && (1)\\
t^{\mathtt{T}\,\mathtt{[1]}\,\mathtt{[}N^2\mathtt{]}} &= c' + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 && (2)\\
t^{\mathtt{T}\,\mathtt{[}N\mathtt{]}\,\mathtt{[}N\mathtt{]}} &= c'' + (t^{\mathrm{vec}}_{\mathrm{ser}} + t^{\mathrm{vec}}_{\mathrm{unser}}) N + \\
&\quad (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 && (3)
\end{aligned}
$$

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.

$t^{\text{byte}}_{\text{ser}}$	=	0.043μs	$t^{\text{float}}_{\text{ser}}$	=	2.1μs	$t^{\text{vec}}_{\text{ser}}$	=	100μs
$t^{\text{byte}}_{\text{unser}}$	=	0.027μs	$t^{\text{float}}_{\text{unser}}$	=	1.4μs	$t^{\text{vec}}_{\text{unser}}$	=	53μs
$t^{\text{byte}}_{\text{com}}$	=	$0.062\mu\text{s}^{\dagger}$	$t^{\text{float}}_{\text{com}}$	=	$0.25\mu\text{s}^{\dagger}$			
$t^{\text{byte}}_{\text{com}}$	=	$0.008\mu\text{s}^{\S}$	$t^{\text{float}}_{\text{com}}$	=	$0.038\mu\text{s}^{\S}$			

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

All measurements were performed on a cluster of 2-processor, 200 Mhz Ultra-Sparc 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 $t^{\text{vec}}_{\text{ser}}$ and $t^{\text{vec}}_{\text{unser}}$ were estimated by plotting the difference between serialization and unserialization times for $\texttt{T[1][}N^2\texttt{]}$ and $\texttt{T[}N\texttt{][}N\texttt{]}$[4]. The raw communication speed was estimated from ping-pong results for $t^{\texttt{T[}N^2\texttt{]}}$. Table 1 contains the resulting estimates of the various parameters for `byte` and `float` elements.

Figure 3 plots actual measured times from ping-pong benchmarks for the mpiJava sends and receives of arrays with `byte` and `float` elements. In the plots the array extent, N, ranges between 128 and 1024. The measured times for $t^{\texttt{T[}N^2\texttt{]}}$, $t^{\texttt{T[1][}N^2\texttt{]}}$ and $t^{\texttt{T[}N\texttt{][}N\texttt{]}}$ 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 communication 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.

[4] 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.

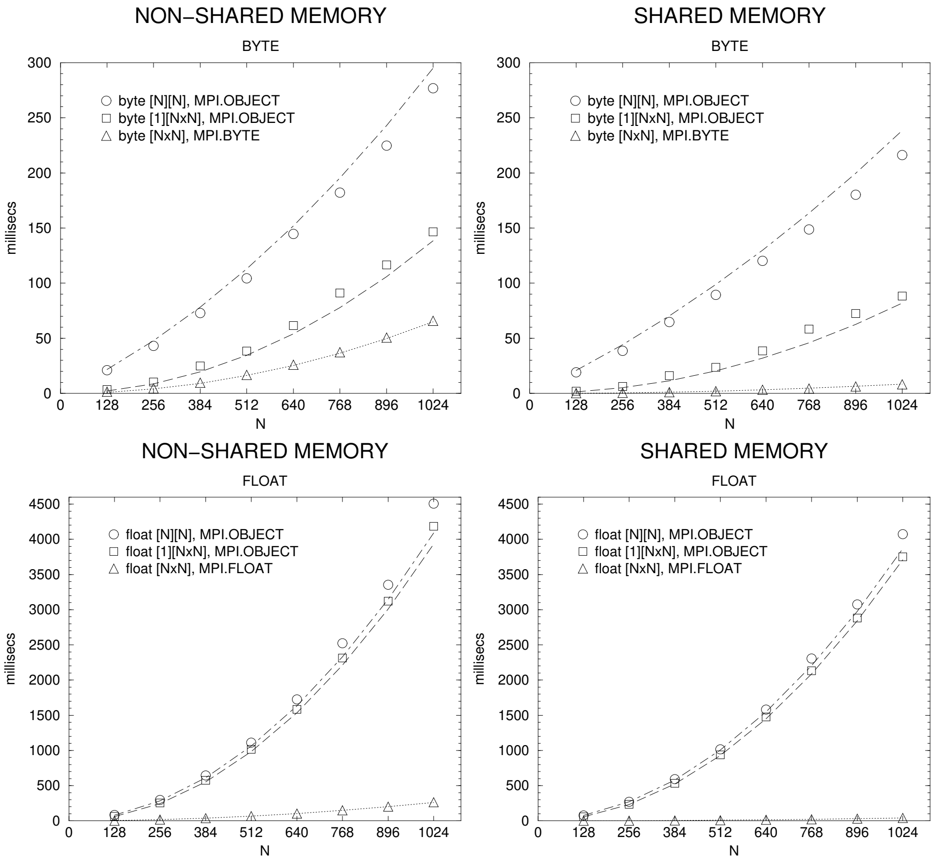

Figure 3: Communication times from ping-pong benchmark in non-shared-memory and shared-memory cases. The lines represent the model defined by Equations 1 to 3 in the text, with parameters from Table 1.

Serialization costs for `int` elements are intermediate.

The constant overheads for serializing each subarray, characterized by the parameters $t^{\rm vec}_{\rm ser}$ and $t^{\rm vec}_{\rm unser}$ 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 is relatively fast.

5 Reducing serialization overheads for arrays

The work of [18] and others has established that there is considerable scope to optimize the JDK serialization software. Here we pursue an alternative that is interesting from the point of view of ultimate efficiency in messaging APIs, namely to replace calls to the `writeObject`, `readObject` methods with special-

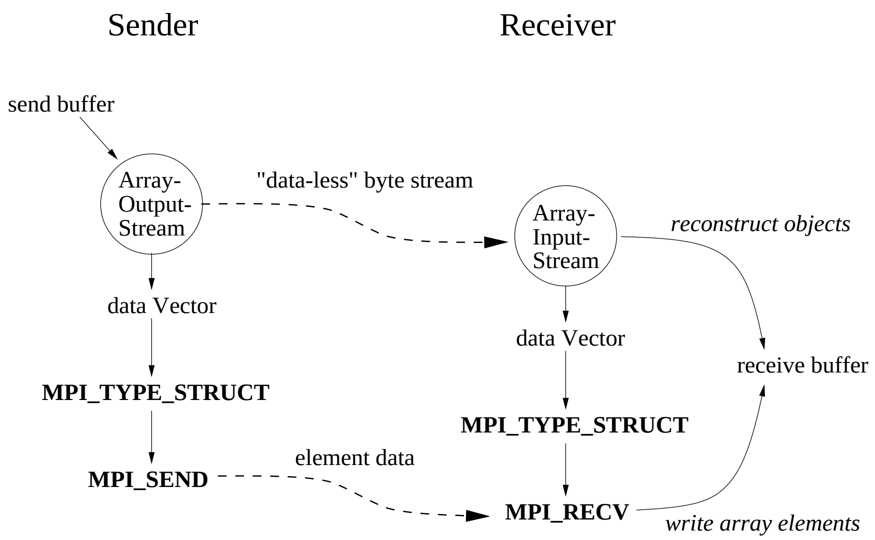

Figure 4: Improved protocol for handling arrays of primitive elements.

ized, MPI-specific, functions. A call to standard `writeObject`, for example, might be replaced with a native method that creates a native MPI derived datatype structure describing the layout of data in the object. This would provide the conceptually straightforward object serialization model at the user level, while retaining the option of fast ("zero-copy") communication strategies inside the implementation.

Implementing this general scheme for every kind of Java object is difficult or impractical because the JVM hides the internal representation of most objects. Less ambitiously, we can attempt to eliminate the serialization and copy overheads for arrays of primitive elements embedded in the serialization stream. The general idea is to produce specialized versions of `ObjectOutputStream` and `ObjectInputStream` that yield byte streams identical to the standard version *except* that array data is omitted from those streams. The "data-less" byte stream is sent as a header. This allows the objects to be reconstructed at the receiving end. The array data is then sent separately using, say, suitable native `MPI_TYPE_STRUCT` types to send all the array data in one logical communication. In this way the serialization overhead parameters measured in the benchmarks of the previous section can be drastically reduced or eliminated. An implementation of this protocol is illustrated in Figure 4.

A customized version of `ObjectOutputStream` called `ArrayOutputStream` behaves in exactly the same way as the original stream except when it encounters an array. When an array is encountered a small object of type `ArrayProxy` is placed in the stream. This encodes the type and size of the array. The array reference itself is placed in a separate container called the "data vector". When serialization is complete, the data-less byte stream is sent to the receiver. A piece of native code unravels the data vector and sets up a native derived type, then

```
class ArrayOutputStream extends ObjectOutputStream {
    Vector dataVector ;

    public Object replaceObject(Object obj) {
        if(obj instanceof int []) {
            dataVector.addElement(obj)
            return new ArrayIntProxy(((int []) obj).length) ;
        }
        ... deal with other primitive array types ...
        else
            return obj
    }
}

class ArrayInputStream extends ObjectInputStream {
    Vector dataVector ;

    public Object resolveObject(Object obj) {
        if(obj instanceof ArrayIntProxy) {
            int dat = new int [((ArrayIntProxy) obj).length] ;
            dataVector.addElement(dat)
            return dat ;
        }
        ... deal with other array proxy types ...
        else
            return obj
    }
}
```

Figure 5: Pseudocode for `ArrayOutputStream` and `ArrayInputStream`

the array data is sent. At the receiving end a customized `ArrayInputStream` behaves exactly like an `ObjectInputStream`, except that when it encounters an `ArrayProxy` it allocates an array of the appropriate type and length and places a handle to this array in the reconstructed object graph *and* in a data vector container. When this phase is completed we have an object graph containing uninitialized array elements and a data vector, created as a side effect of unserialization. A native derived data type is constructed from the data vector in the same way as at the sending end, and the data is received into the reconstructed object in a single MPI operation.

Our implementation of `ArrayOutputStream` and `ArrayInputStream` is straightforward. The standard `ObjectOutputStream` provides a method, `replace-Object`, which can be overridden in subclasses. `ObjectInputStream` provides a corresponding `resolveObject` method. Implementation of the customized streams is sketched in Figure 5.

Figure 6 shows the effect this change of protocol has on the original timings. As expected, eliminating the overheads of element serialization dramatically speeds communication of float arrays (for example) treated as objects, bringing bandwidth close to the raw performance available with `MPJ.FLOAT`.

Each one-dimensional array in the stream needs some separate processing here (associated with calls to `replaceObject`, `resolveObject`, and setting up

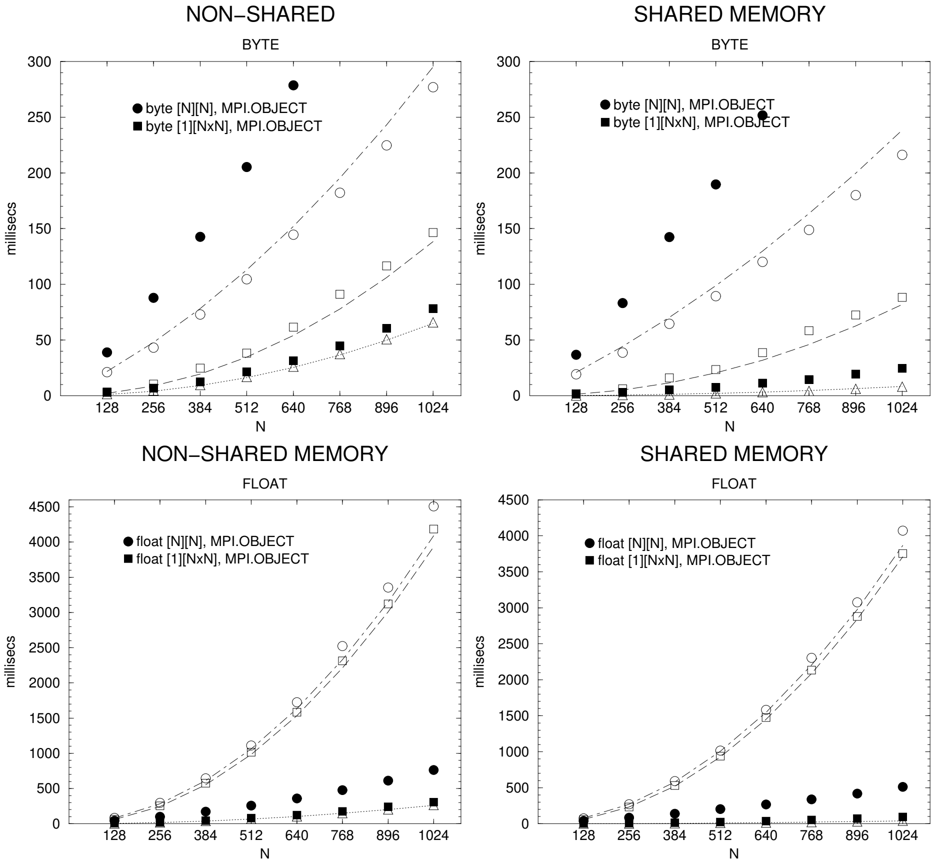

Figure 6: Ping-pong timings with primitive array data sent separately (solid points), compared with the unoptimized results from Figure 3 (open points). Recall that the goal is to bring times for "object-oriented" sends of arrays down to the "native" send times, most closely approximated by the triangular points.

the native MPI_TYPE_STRUCT). Our fairly simple-minded prototype happened to increase the constant overhead of communicating each subarray (parametrized by $t^{\mathrm{vec}}_{\mathrm{ser}}$ and $t^{\mathrm{vec}}_{\mathrm{unser}}$ in the previous section). As mentioned at the end of section 4, this overhead typically dominates the time for communicating two-dimensional byte arrays (where the element serialization cost is less extreme), so performance there actually ends up being worse. A more highly tuned implementation could probably reduce this problem. Alternatively we can go a step further with our protocol, and have the serialization stream object directly replace *two-dimensional* arrays of primitive elements[5]. The benefits of this approach are shown in Figure 7.

This process could continue almost indefinitely—adding special cases for arrays and other structures considered critical to Grande applications. Currently we do not envisage pushing this approach any further than two-dimensional array proxies. Of course three-dimensional arrays and higher will automaticall benefit from the optimization of their lower-dimensional component arrays. Recognizing a rectangular two-dimensional arrays already adds some unwanted complexity to the serialization process[6].

6 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 Java multidimensional arrays. Such arrays are likely to be common in scientific programming.

Our initial attempt to add automatic object serialization to our MPI-like API for Java was impaired by poor performance of the serialization code in the current Java Development Kit. Buffers were serialized using standard technology from the JDK. The benchmark results from section 4 showed that this implementation 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]. 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.

In our benchmarks and tests the organization of primitive elements—their byte-order, in particular—was the same in sender and receiver. This is commonly the case in MPI applications, which are often run on homogenous clusters

[5]Defined to be arrays of objects, each element being an array of primitive type of the same type and length.

[6]It can also introduce some unexpected behaviour. Our version subtly alters the semantics of serialization, because it does not detect aliasing of rows (either with other rows of the same two-dimensional array, or with one-dimensional primitive arrays elsewhere in the stream). Hence the reconstructed object graph at the receiving end will not reproduce such aliasing. Whether this is a serious problem is unclear.

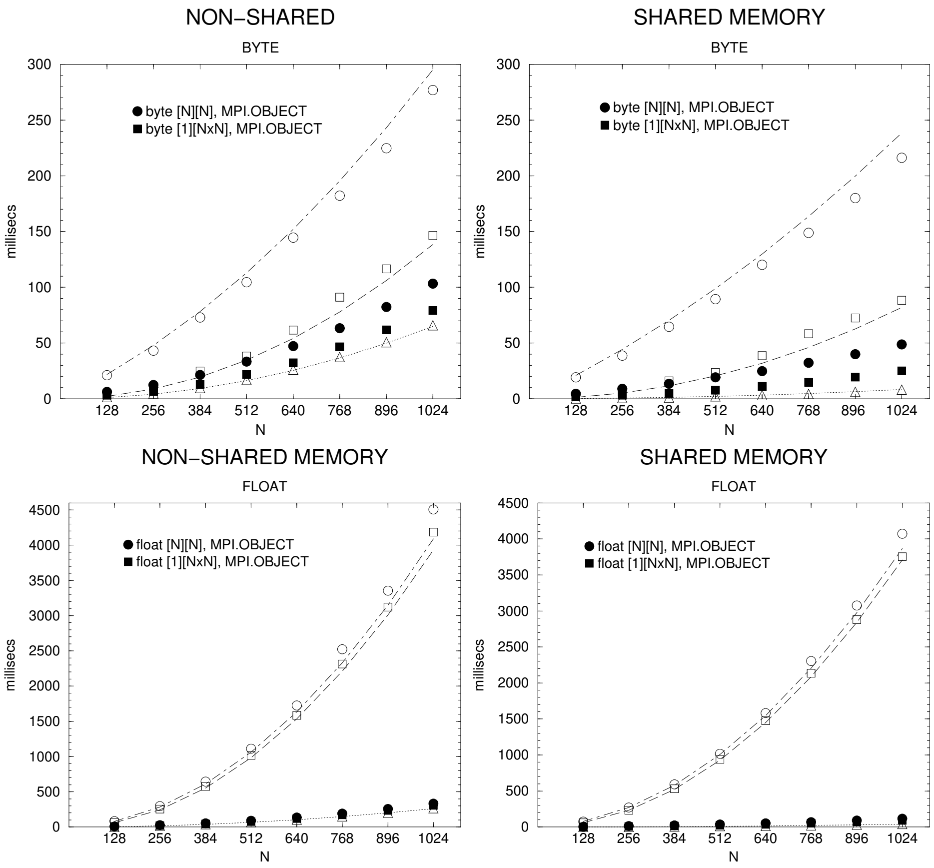

Figure 7: Timings allowing *two-dimensional* array proxies in the object stream (solid points), compared with the unoptimized results from Figure 3 (open points). Sends of two-dimensional Java arrays (solid circles) are now much closer to the native bandwidth (of which the triangular points are representative).

of computers. Hence it should be possible to send the bytes with no format conversion at all. More generally an MPI-like package can be assumed to know in advance if sender and receiver have different layouts, and need only convert to an external representation in the case that they do. Presuming we are building on an underlying native MPI in the first place, then, a reasonable assumption is that the conversions necessary for, say, communication of float arrays between little-endian and big-endian machines in a heterogenous cluster are dealt with inside the native MPI. This may degrade the effective native bandwidth to a greater or lesser extent, but should not impact the Java wrapper code. In any case, to exploit these features in the native library, we need a way to marshal Java arrays that avoids performing conversions inefficiently in the Java layer.

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 standard Java framework for serialization does not provide a direct way to handle arrays, but in section 5 we customized the object streams themselves by suitably defining the `replaceObject`, `resolveObject` methods. Primitive array data was removed from the serialization stream and sent separately using *native* derived datatype mechanisms of the underlying MPI, without explicit conversion or explicit copying. This dramatically reduced the overheads of treating Java arrays uniformly as objects at the API level. Order of magnitude degradations in bandwidth were typically replaced by fractional overheads.

A somewhat different approach was taken by the authors of [18]. Their remote method invocation software, KaRMI, incorporates an extensive reimplemention of the JDK serialization code, to better support their optimized RMI. Their ideas for optimizing serialization can certainly benefit message-based APIs as well, and KaRMI does also reduce copying compared with standard RMI. But we believe they do not immediately support the "zero-copy" strategy we strive for here, whereby large arrays are removed from the serialization stream and dealt with separately by platform-specific software[7]. In our case the platform-specific software was a native MPI binding, but similar strategies could apply to other devices, such as a binding to the new industry standard Virtual Interface Architecture, VIA[8].

Given that the efficiency of object serialization can be improved dramatically—although probably it will always introduce a non-zero overhead—a reasonable question is whether an MPI-like API for Java needs to retain anything like the

[7] Our use of the phrase "zero-copy" has been criticized on the basis that a number of existing JVMs *always* copy arrays that are passed through the JNI interface, in which case there is always at least one copy. To our knowledge, there is nothing in the JVM specification that requires such behaviour, and other existing JVMs pin the storage inside the JVM and return a pointer to the actual storage to the native method, rather than copying. But it is true that the phrase zero-copy must be understood modulo the behaviour JNI implementation associated with the JVM and garbage collector that one is using.

[8] We should add that KaRMI can also use specific communication hardware such as VIA for its transport layer, and in principle could even plug in native MPI-routines in this layer. We believe it would nevertheless serialize data first.

old derived datatype mechanism of MPI at all?

The MPI mechanism still allows non-contiguous sections of a buffer array to be sent directly. Although implementations of MPI derived types, even in the C domain, have often had disappointing performance in the past, we note that VIA provides some low-level support for communicating non-contiguous buffers, and recently there has been interest in producing Java bindings of VIA [5, 19]. So perhaps in the future it will become possible to support derived types quite efficiently in Java. We have emphasized the use of object serialization as a way of dealing with communication of Java multidimensional arrays. Assuming the Java model of multidimensional arrays (as arrays of arrays), we suspect serialization is the most natural way of communicating them. On the other hand there is an active discussion (especially in Numerics Working Group of the Java Grande Forum) about how Fortran-like multidimensional rectangular arrays could best be supported into Java. A reasonable guess is that multidimensional array sections would be represented as strided sections of some standard one-dimensional Java array. In this case the best choice for communicating array sections may come back to using MPI-like derived datatypes similar to `MPI_TYPE_VECTOR`.

In any case—whether or not a version of MPI derived data types survive in Java—the need to support object serialization in a message-passing API seems relatively clear.

References

[1] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[2] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, and Xiaoming Li. Experiments with HPJava. *Concurrency: Practice and Experience*, 9(6):633, 1997.

[3] Bryan Carpenter, Geoffrey Fox, Guansong Zhang, and Xinying Li. A draft Java binding for MPI., November 1997. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[4] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, and Geoffrey Fox. MPI for Java: Position document and draft API specification. Technical Report JGF-TR-3, Java Grande Forum, November 1998. http://www.javagrande.org/.

[5] Chi-Chao Chang and Thorsten von Eiken. Interfacing Java to the Virtual Interface Architecture. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[6] Kivanc 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, September 1998. http://www.cs.cf.ac.uk/hpjworkshop/.

[7] Adam J. Ferrari. JPVM: Network parallel computing in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

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

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

[10] Geoffrey C. Fox, editor. *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

[11] Vladimir Getov, Susan Flynn-Hummel, and Sava 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*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

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

[13] Glenn Judd, Mark Clement, and Quinn Snell. DOGMA: Distributed object group management architecture. In *ACM 1998 Workshop on Java for High-Performance Network Computing. Palo Alto, February 1998*, volume 10(11-13) of *Concurrency: Practice and Experience*, 1998.

[14] Glenn Judd, Mark Clement, and Quinn Snell. Design issues for efficient implementation of MPI in Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

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

[16] S.P. Midkiff, J.E. Moreira, and M. Snir. Optimizing array reference checking in Java programs. *IBM Systems Journal*, 37(3):409, 1998.

[17] Sava Mintchev and Vladimir 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.

[18] Christian Nester, Michael Philippsen, and Bernhard Haumacher. A more efficient RMI for Java. In *ACM 1999 Java Grande Conference*. ACM Press, June 1999.

[19] Matt Welsh. Using Java to make servers scream. Invited talk at ACM 1999 Java Grande Conference, San Francisco, CA, June, 1999.