Object Serialization for Marshalling Data in a Java Interface to MPI

Bryan Carpenter, Geoffrey Fox, Sung-Hoon Ko, and Sang Lim

www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html

Some issues in design of a Java API for MPI

- **Class hierarchy**. MPI is already object-based. "Standard" class hierarchy exists for C++.
- **Detailed argument lists for methods.** Properties of Java language imply various superficial changes from C/C++.
- ***Mechanisms for representing message buffers*.**

Representing Message Buffers

Two natural options:

- Follow the MPI standard route: **derived datatypes** describe buffers consisting of mixed primitive fields scattered in local memory.
- Follow the Java standard route: automatic marshalling of complex structures through **object serialization**.

Overview of this talk

- Review the **mpiJava** wrapper interface.
- Discuss incorporation of derived datatypes in the Java API, and limitations.
- Adding object serialization at the API level.
- Describe implementation using JDK serialization.
- Benchmarks for naïve implementation.
- Ongoing work: optimizing serialization.

Related projects

- **mpiJava** (Syracuse)
- **JavaMPI** (Getov et al, Westminster)
- **JMPI** (MPI Software Technology)
- **MPIJ** (Judd et al, Brigham Young)
- **jmpi** (Dincer et al)

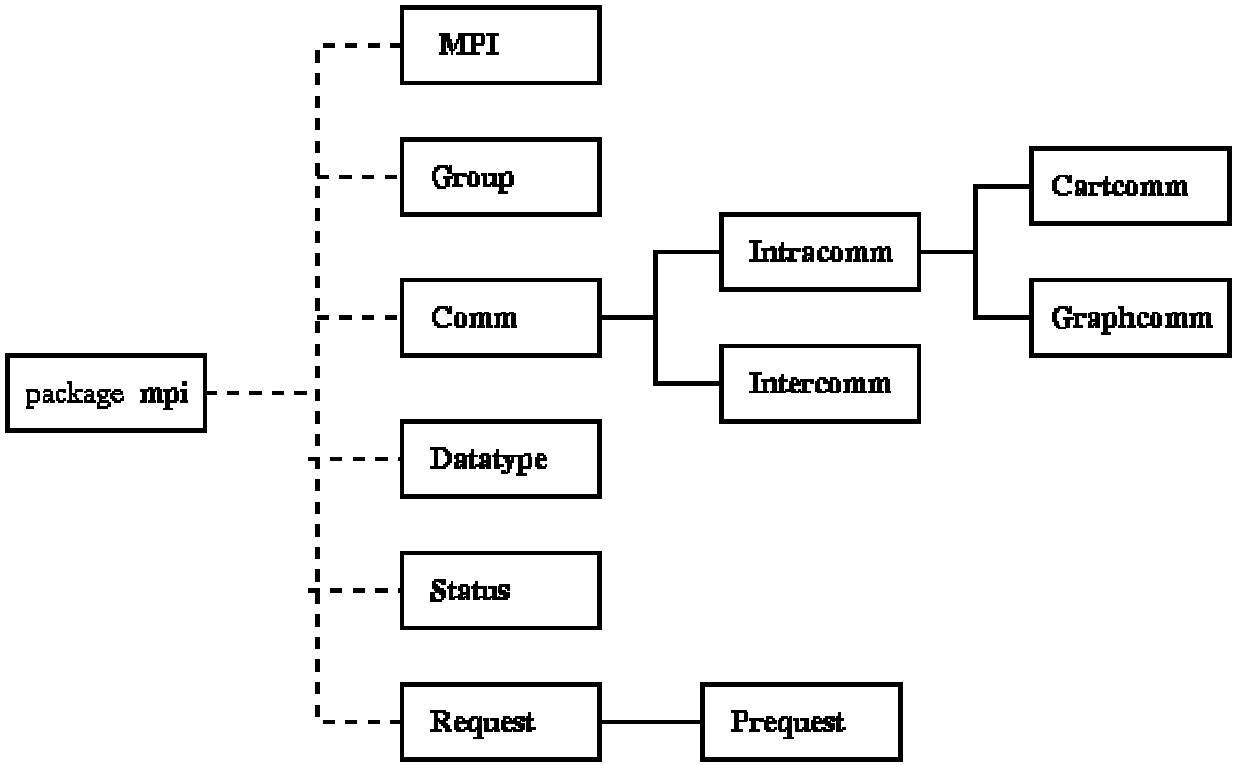
The mpiJava wrapper

- Implements a Java API for MPI suggested in late '97.
- Builds on work on Java wrappers for MPI started at NPAC about a year earlier.
- **People**: Bryan Carpenter, Yuh-Jye Chang, Xinying Li, Sung Hoon Ko, Guansong Zhang, Mark Baker, Sang Lim.

mpiJava features.

- Fully featured Java interface to MPI 1.1
- Object-oriented API based on MPI 2 standard C++ interface
- Initial implementation through JNI to native MPI
- Comprehensive test suite translated from IBM MPI suite
- Available for Solaris, Windows NT and other platforms

Class hierarchy



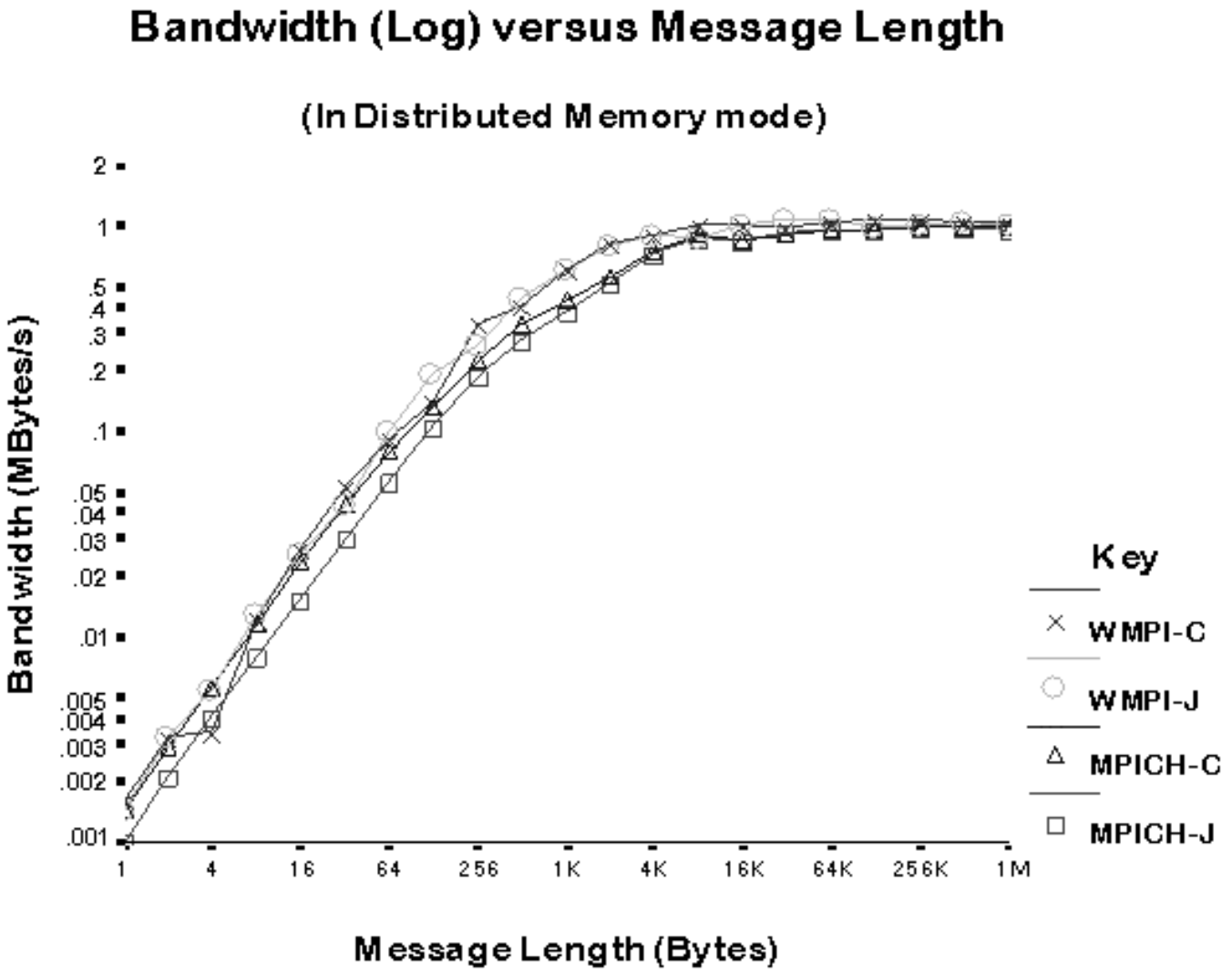
Minimal mpiJava program

```
import mpi.*

class Hello {
   static public void main(String[] args) {
      MPI.Init(args) ;

      int myrank = MPI.COMM_WORLD.Rank() ;
      if(myrank == 0) {
         char[] message = “Hello, there”.toCharArray() ;
         MPI.COMM_WORLD.Send(message, 0, message.length, MPI.CHAR, 1, 99) ;
      }
      else {
         char[] message = new char [20] ;
         MPI.COMM_WORLD.Recv(message, 0, 20, MPI.CHAR, 0, 99) ;
         System.out.println(“received:” + new String(message) + “:”) ;
      }

       MPI.Finalize() ;
   }
}
```

mpiJava implementation issues

- Interfacing Java to MPI not always trivial, eg, see low-level conflicts between the Java runtime and interrupts in MPI.
- Situation improving as JDK matures.
- Now reliable on Solaris MPI (SunHPC, MPICH), shared memory, NT (WMPI).
- Other ports in progress.

mpiJava performance



MPI datatypes

- Send and receive members of **Comm**:

void send(Object buf, int offset, int count,
Datatype type, int dst, int tag) ;

Status recv(Object buf, int offset, int count,
Datatype type, int src, int tag) ;

- **buf** must be an array. **offset** is the element where message starts. **Datatype** class describes type of elements.

Basic Datatypes

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

Derived datatypes

MPI derived datatypes have two roles:

- Non-contiguous data can be transmitted in one message.
- **MPI_TYPE_STRUCT** allows mixed primitive types in one message.

Java binding doesn't support second role. All data come from a homogeneous array of elements (no **MPI_Address**).

Restricted model

A derived datatype consists of

- **A base type**. One of the 9 basic types.
- **A displacement sequence.** A relocatable pattern of integer displacements in the buffer array:

$$\{disp_0, disp_1, \ldots, disp_{n-1}\}$$

Limitations

- Can't mix primitive types or fields from different objects.
- Displacements only operate within 1d arrays. Can't use **MPI_TYPE_VECTOR** to describe sections of Java multidimensional arrays.

Object datatypes

- If **type** argument is **MPI.OBJECT**, **buf** should be an array of objects.
- Allows to send fields of mixed primitive types, and fields from different objects, in one message.
- Allows to send multidimensional arrays, because they are arrays of arrays (and arrays are effectively objects).

Automatic serialization

- Send **buf** should be an array of objects implementing **Serializable**.
- Receive **buf** should be an array of compatible reference types (may be null).

Java serialization paradigm applied. Output objects (and objects referenced through them) converted to a byte stream. Object graph reconstructed at the receiving end.

Implementation issues for Object datatypes

- Initial implementation in mpiJava used **ObjectOutputStream** and **ObjectInputStream** classes from JDK.
- Data serialized and sent as a byte vector, using MPI.
- Length of byte data not known in advance. Encoded in a separate header so space can be allocated dynamically in receiver.

Modifications to mpiJava

- All mpiJava communications, including non-blocking modes and collective operations, now allow objects as base types.
- Header + data decomposition complicates, eg, **wait** and **test** family.
- Derived datatypes complicated.
- Collective comms involve two phases if base type is **OBJECT.**

Benchmarking mpiJava with serialization

- Assume in "Grande" applications, critical case is arrays of primitive element.
- Consider N x N arrays:

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

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

Timing model

- For comparision time **float [NxN]** (no serialization),
- **float [1] [NxN]** (1d serialization), and
- **byte** and **int** versions.

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

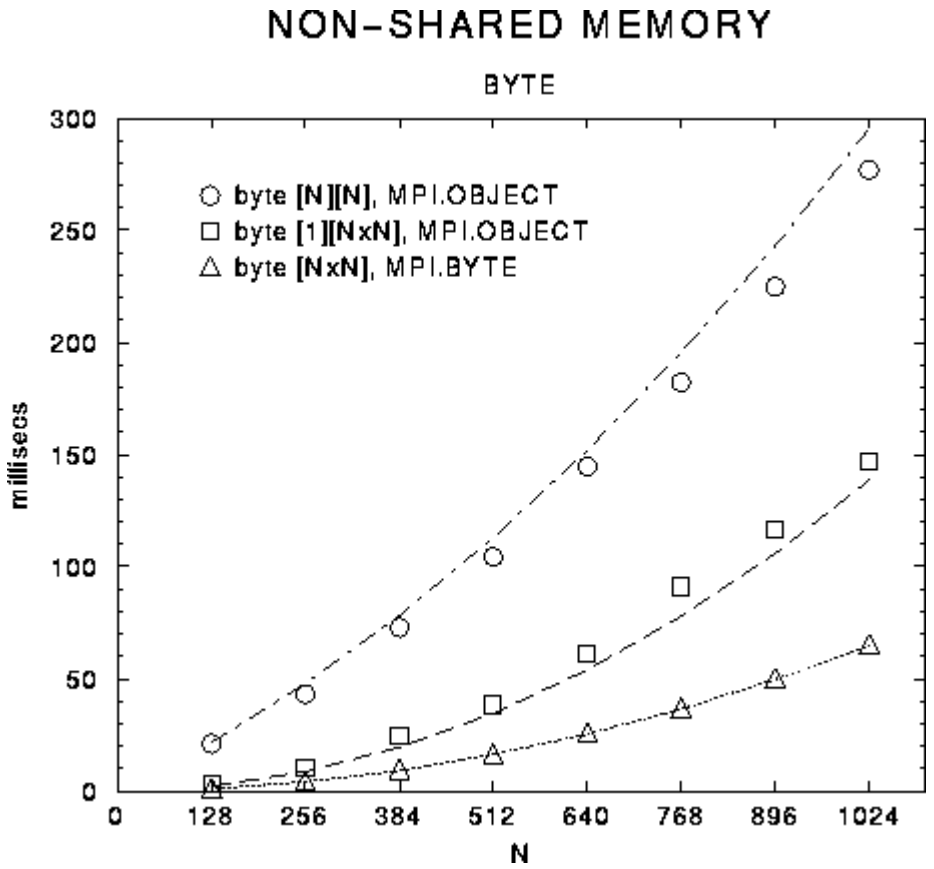
$$t^{\mathrm{T}[1][N^2]} = c' + (t^{\mathrm{T}}_{\mathrm{ser}} + t^{\mathrm{T}}_{\mathrm{com}} + t^{\mathrm{T}}_{\mathrm{unser}}) N^2 \quad (2)$$

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

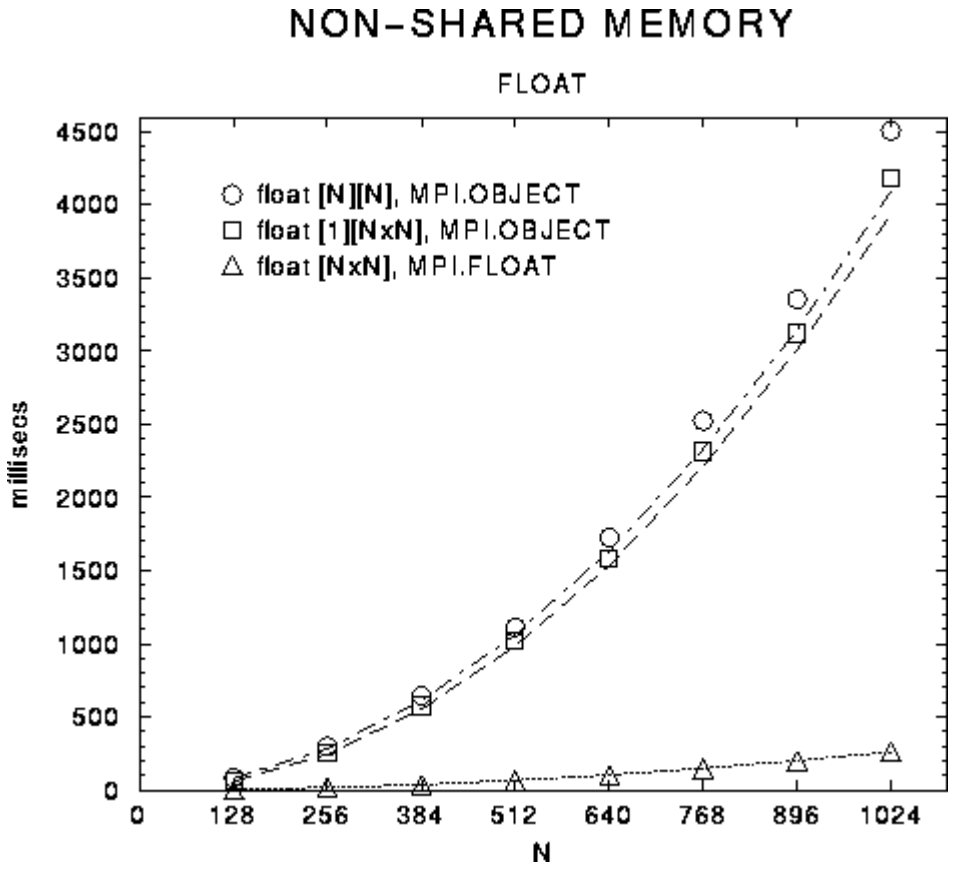
Platform

- Cluster of 2-processor, 200 Mhz Ultrasparc nodes
- SunATM-155/MMF network
- Sun MPI 3.0
- “non-shared memory” = inter-node comms
- “shared memory” = intra-node comms

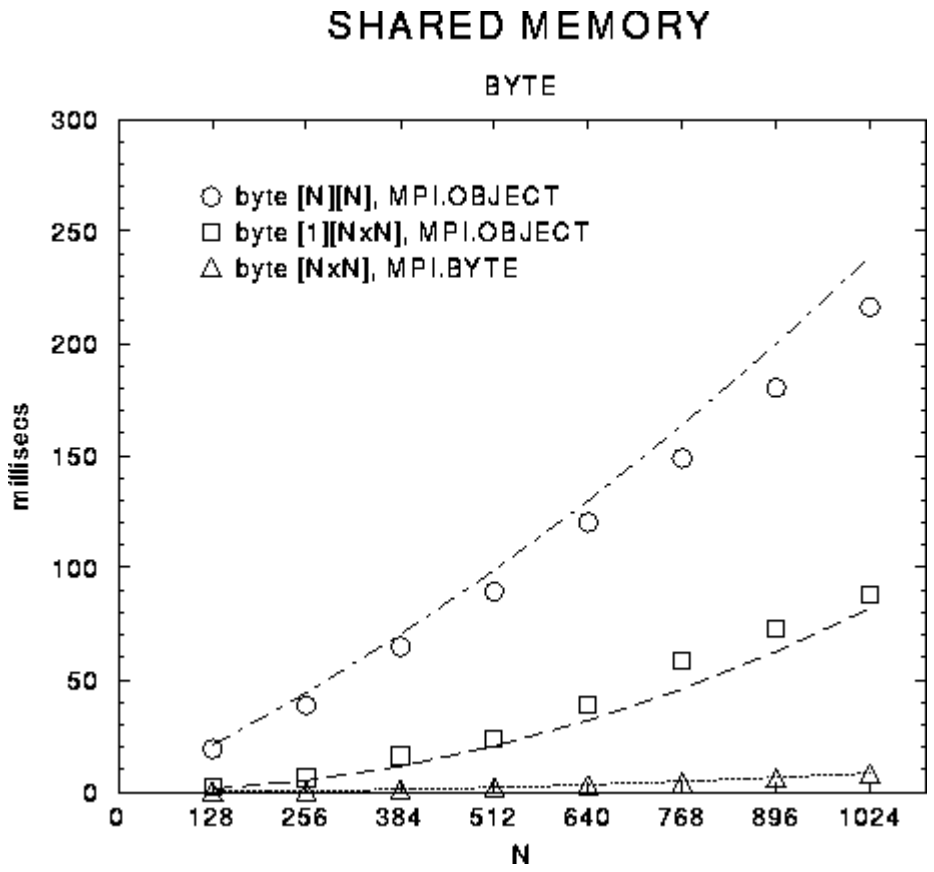
Non-shared memory: byte



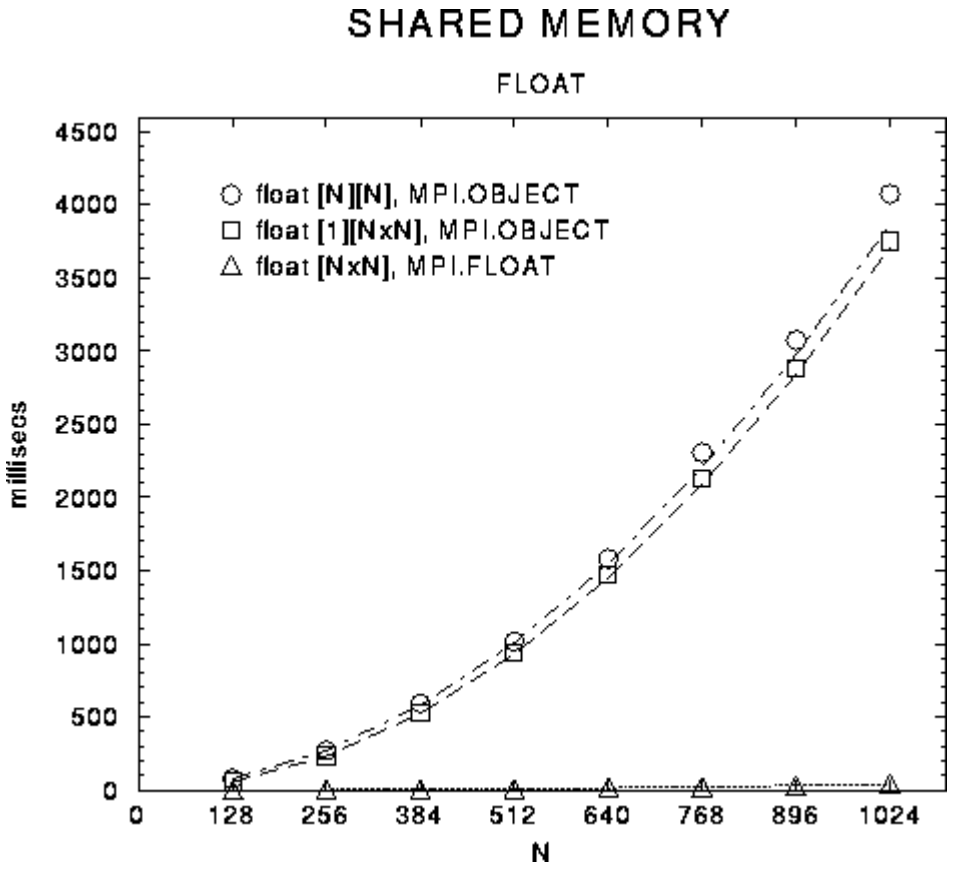
Non-shared memory: float



Shared memory: byte



Shared memory: float



Parameters in timing model (microseconds)

$t_{ser}^{byte} = 0.043$ $t_{ser}^{float} = 2.1$

$t_{unser}^{byte} = 0.027$ $t_{unser}^{float} = 1.4$

$t_{com}^{byte} = 0.062$ $t_{com}^{float} = 0.25$ (non-shared)

$t_{com}^{byte} = 0.008$ $t_{com}^{float} = 0.038$ (shared)

Benchmark lessons

- Cost of serializing and unserializing an individual **float** one to two *orders of magnitude* greater than communication!
- Serializing subarrays also expensive:

$$t_{ser}^{vec} = 100 \qquad t_{unser}^{vec} = 53$$

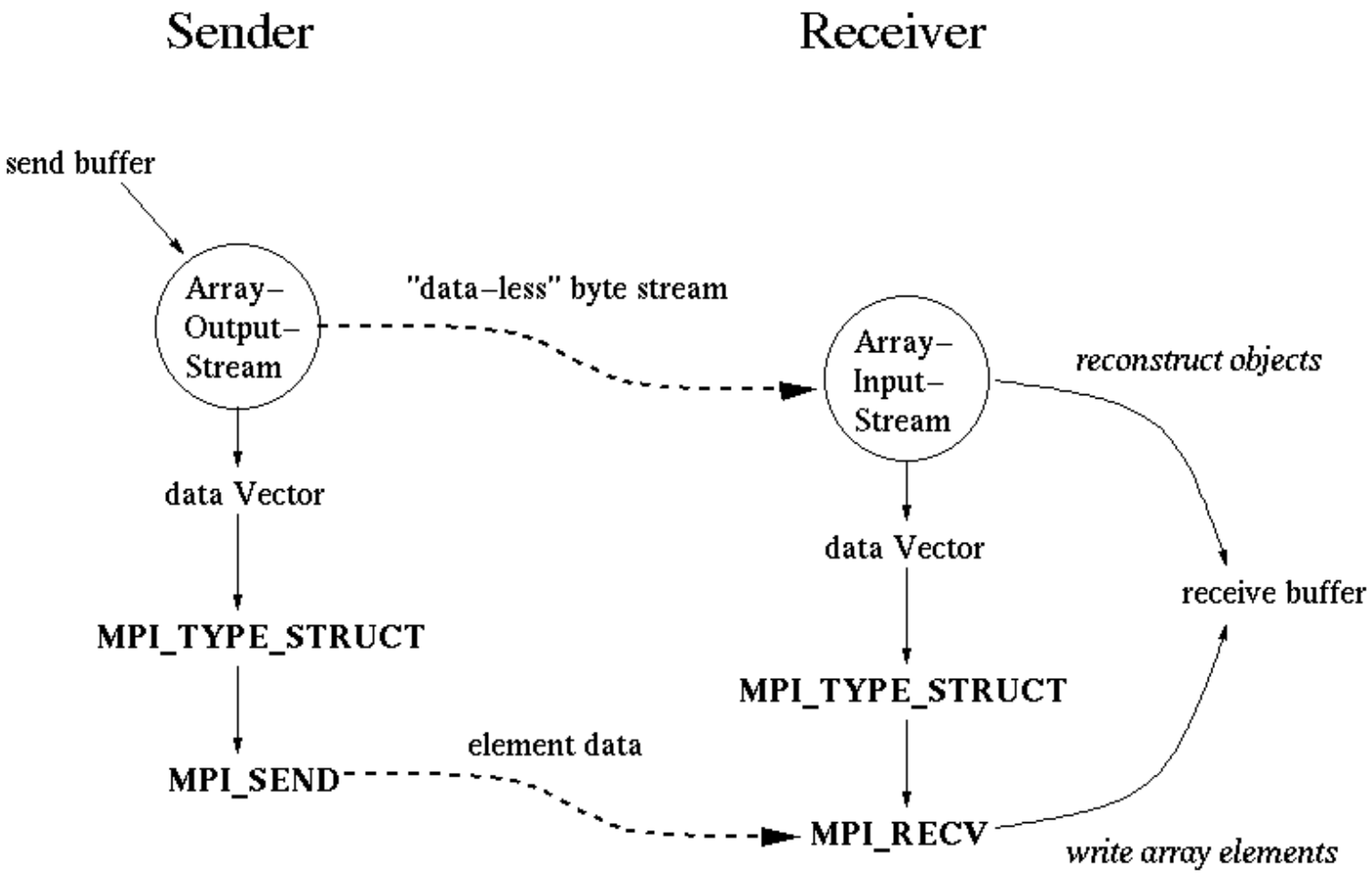
Improving serialization

- Sources of **ObjectOutputStream**, **ObjectInputStream** are available, and format of serialized stream is documented.
- By overriding performance-critical methods in classes, and modifying critical aspects of the stream format, can hope to solve immediate problems.

Eliminating overheads of element serialization

- Customized **ObjectOutputStream** replaces primitive arrays with short **ArrayProxy** object. Separate **Vector** holding the Java arrays is produced.
- "Data-less" byte stream sent as header.
- New **ObjectInputStream** yields **Vector** of allocated arrays, not writing elements.
- Elements then sent in one comm using **MPI_TYPE_STRUCT** from vector info.

Improved protocol



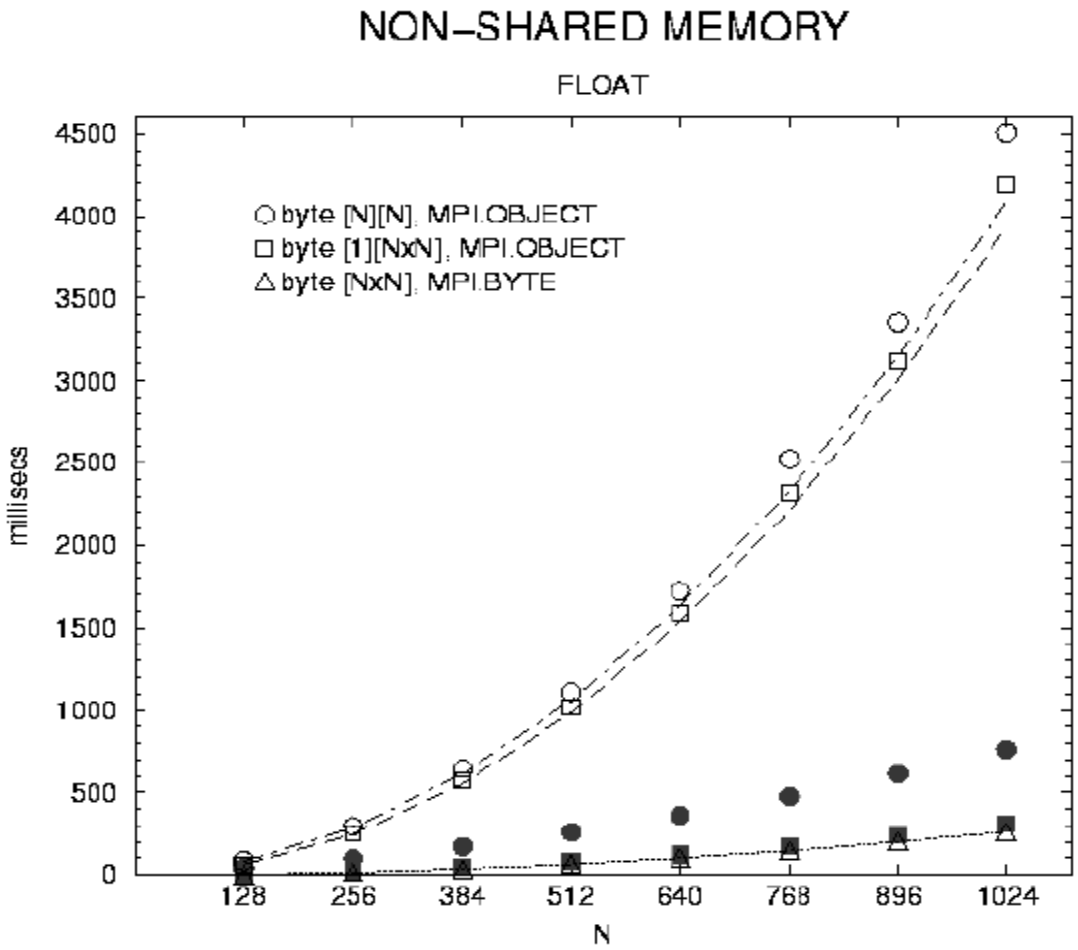
Customized output stream class

Class **ArrayOutputStream** extends **ObjectOutputStream** {
    Vector **dataVector** ;
    public Object **replaceObject**(Object obj) {
      if **obj** is a **primitive array**, then
        if **obj** is (eg) an **int[]**, then
            int len = ((int []) obj).length ;
            dataVector.**addElement**(new ArrayInfo(INT, len, obj)) ;
            return new ArrayProxy(INT, len) ;
        etc (deal with other primitive types).
      else, if not a primitive array, then
        return obj ;
    }
}

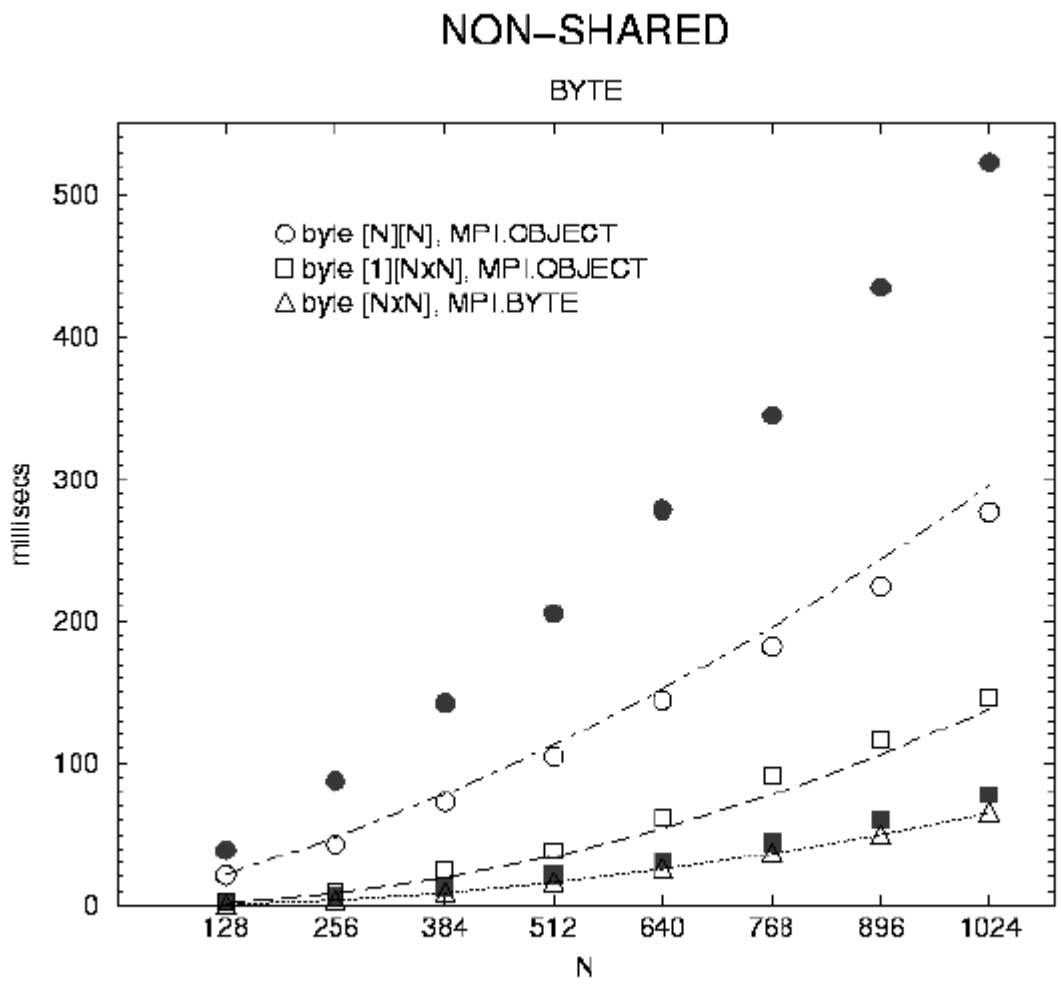
Customized input stream class

```
Class ArrayInputStream extends ObjectInputStream {
    Vector dataVector ;
    public Object resolveObject(Object obj) {
      if (obj instanceof ArrayProxy) then
          ArrayProxy proxy = (ArrayProxy) obj ;
          switch (proxy.type) {
          case INT :
              int [] dat = new int [proxy.length] ;
              dataVector.addElement(new ArrayInfo(INT, dat.length, dat)) ;
              return dat ;
           etc (deal with other primitive types).
       else  return obj ;
    }
}
```

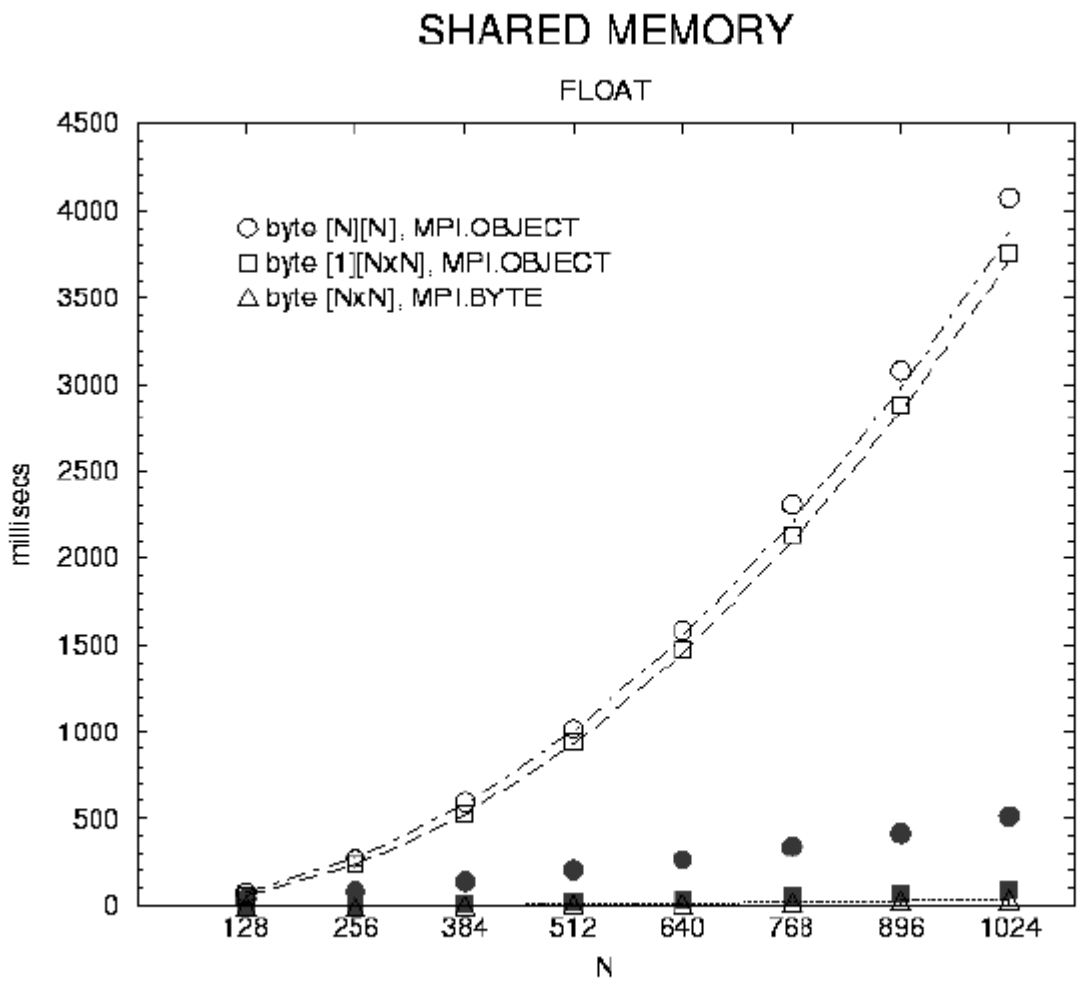
Non-shared memory: float (optimized in red)



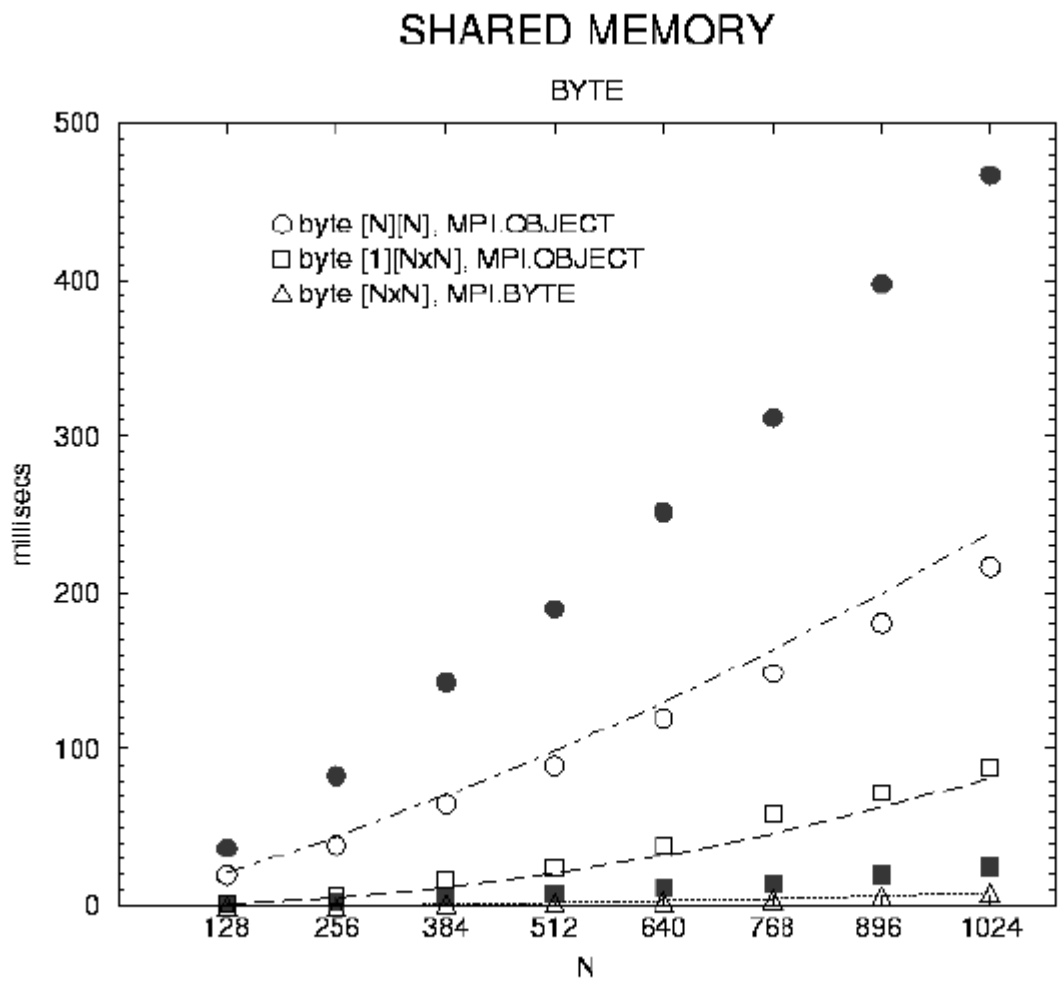
Non-shared memory: byte (optimized in red)



Shared memory: float (optimized in red)



Shared memory: byte (optimized in red)



Comments

- Relatively easy to get dramatic improvements.
- Have only truly optimized one dimensional arrays embedded in stream.
- Next look at direct optimizations for rectangular multi-dimensional arrays---replace wholesale in stream?

Conclusions

- Derived datatypes workable for Java, but slightly limited.
- Object basic types attractive on grounds of simplicity and generality.
- Naïve implementation too slow for bulk data transfer.
- Optimizations should bring asymptotic performance in line with C/Fortran MPI.

www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html