Java for High Performance Computing

MPI-based Approaches for Java

http://www.hpjava.org/courses/arl

Instructor: Bryan Carpenter

Pervasive Technology Labs

Indiana University

MPI: The Message Passing Interface

- RMI originated in the Java world. Efforts like *JavaParty* and *Manta* aimed to bring RMI into the HPC world, by improving its performance.
- MPI is a technology from the HPC world, which various people have worked on importing into Java.
  - MPI is the HPC *Message Passing Interface* standardized in the early 1990s by the *MPI Forum*—a substantial consortium of vendors and researchers.
  - It is an API for communication between nodes of a distributed memory parallel computer (typically, now, a workstation cluster).
  - The original standard defines bindings to C and Fortran (later C++).
  - The low-level parts of API are oriented to: fast transfer of data from user program to network; supporting multiple modes of message synchronization available on HPC platforms; etc.
  - Higher level parts of the API are concerned with organization of process groups and providing the kind of collective communications seen in typical parallel applications.

Features of MPI

- MPI (**http://www-unix.mcs.anl.gov/mpi**) is an API for sending and receiving messages. But it goes further than this.
  - It is essentially a general platform for *Single Program Multiple Data* (SPMD) parallel computing on distributed memory architectures.
  - In this respect it is directly comparable with the *PVM* (Parallel Virtual Machine) environment that was one of its precursors.
- It introduced the important abstraction of a *communicator*, which is an object something like an N-way communication channel, connecting all members of a group of cooperating processes.
  - This was introduced partly to support using multiple parallel libraries without interference.
- It also introduced a novel concept of *datatypes*, used to describe the contents of communication buffers.
  - Introduced partly to support "zero-copying" message transfer.

MPI for Java: Early History

- When Java first appeared there was immediate interest in its possible uses for parallel computing, and there was a little explosion of MPI and PVM "bindings" for Java, e.g.:
  - **JavaMPI**
    - University of Westminster
  - **mpiJava**
    - Syracuse University/Florida State University/Indiana University
  - **DOGMA MPIJ**
    - Brigham Young University
  - **JMPI**
    - MPI Software Technology
  - **JavaPVM (jPVM)**
    - Georgia Tech.
  - **JPVM**
    - University of Virginia

More Recent Developments

- Most of those early projects are no longer active. **mpiJava** is still used and supported. We will describe it in detail later.
- Other more contemporary projects include
  - **MPJ**
    - An API specification by the "Message-passing Working Group" of the ***Java Grande Forum***. Published 2000.
  - **CCJ**
    - An MPI-like API from some members of the Manta team. Published 2002.
  - **MPJava**
    - A high performance Java message-passing framework using **java.nio**. Published 2003.
  - **JMPI**
    - An implementation of the MPJ spec from University of Massachusetts. Published 2002.
  - **JOPI**
    - Another Java Object-Passing Interface from U. Nebraska-Lincoln. Published 2002.

MPJ

- A specification by the "Message-Passing Working Group" of the ***Java Grande Forum***, published in:

  MPJ: MPI-like message passing for Java

  B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox

  Concurrency Practice and Experience, 12(11), 2000.

  This spec aimed to fill the gap left by the absence of any "official" binding for Java from the MPI forum.

  – The working group had nominal representation from all teams responsible for the Java MPI systems on the "Early History" slide.
- **mpiJava** was voted the "reference implementation" of MPJ (but note that the **mpiJava** API is only similar to, *not* identical to, the MPJ spec…)
- The MPJ spec was cited regularly in the literature as if actual software, though nobody was implementing it. Recently some people at University of Massachusetts have announced an implementation:

  JMPI: Implementing the Message Passing Standard in Java

  S. Morin, I. Koren, and C.M. Krishna, IPDPS 2002 Workshop.

  and some people at U. of A Coruna (Spain) have benchmarked it (2003).

CCJ

- CCJ, which comes from the Manta team (**http://www.cs.vu.nl/manta**) is an API that includes various features of MPI, notably
  - collective communication operations modeled on those in MPI,
  - a new thread group concept playing a role similar to MPI's communicators, and
  - a few point-to-point communication methods.
- Unlike MPJ, CCJ doesn't try to follow the MPI spec in detail. It deliberately diverges from MPI where other approaches are considered more "natural" to Java, e.g. in its integration with Java threads, and emphasis on general objects, rather than arrays, as communication buffers.
- CCJ is implemented on top of Java RMI: for performance, it relies on the Manta compilation system and Manta RMI (otherwise the overheads of Java object serialization, for example, would be a major problem).
- CCJ was presented with a thorough benchmark analysis in:

  CCJ: Object-based Message Passing and Collective Communication in Java

  A. Nelisse, J. Maasen, T. Kielmann, H. Bal

  ACM Java Grande/ISCOPE 2001

MPJava

- MPJava is ongoing work from U. Maryland. It emphasizes exploitation of the **java.nio** package ("new I/O") to produce *pure Java* implementations of HPC message passing, competitive with those (like **mpiJava**) based on calls to a native MPI.
- Presented in

  MPJava: High-Performance Message Passing in Java using java.nio

  W. Pugh and J. Spacco, MASPLAS 2003 (also LCPC '03)
- Unclear if software (yet) released.

JOPI

- Another object-oriented “pure Java” API for message-passing parallel programming.
- Goals reminiscent of CCJ, but emphasis more on “ease of use”, and supporting heterogeneous networks (less on hard performance?)
- Presented in:

  JOPI: A Java Object-Passing Interface

  J. Al-Jaroodi, N. Mohamed, H. Jiang and D. Swanson,

  ACM Java Grande/ISCOPE 2002

Summary

- A number of groups worked actively in the general area of "MPI for Java"—starting around 1996/1997 when Java first emerged as an important language.
- New APIs and approaches are still being published, half a dozen years later.
- The rest of this unit will be strongly biased toward **mpiJava**. Be aware there are other systems out there…
  - As a related special topic we also describe our *HPJava* system.
- In the next section we describe **mpiJava** in detail.

Introduction to **mpiJava**

mpiJava

- **mpiJava** is a software package that provides Java wrappers to a native MPI, through the Java Native Interface.
  - It also comes with a test suite and several demo applications.
- Implements a Java API for MPI suggested in late ’97 by Carpenter et al.
  - Builds on work on Java wrappers for MPI started at Syracuse about a year earlier by Yuh-Jye Chang.
- First “official” release in 1998. Implementation still evolving.
  - Conceptually simple, but it took *several years* to iron out the problems. There are many practical issues interfacing Java to existing native software.
- Contributors:
  - Bryan Carpenter, Yuh-Jye Chang, Xinying Li, Sung Hoon Ko, Guansong Zhang, Mark Baker, Sang Boem Lim.

Defining the mpiJava API

- In MPI 1, the MPI Forum defined bindings for the C and Fortran language.
  - Although not initially bound to object-oriented languages, the specification was always "object-centric", using C or Fortran handles to "opaque objects" of various types.
  - Some early proposals for Java APIs mimicked the C API directly (e.g. JavaMPI from Westminster).
- In 1997 the MPI 2 standard was published. Along with many other additions, it defined C++ bindings.
  - It then seemed natural to base a Java API on the C++ version, as far as possible, because the languages have (quite) similar syntax.
- A document called a "A Draft Java Binding for MPI" was handed out at SC '97, and a graduate student (Xinying Li) was given the task of implementing the API, through JNI wrappers.
  - Although the *syntax* is modeled on MPI 2 C++, the mpiJava API only covers the *functionality* in **MPI 1.1** (because there are few implementations of the rest).

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() ;
   }
}
```

Installing mpiJava

- The current release (1.2.5) has been tested on Linux, Solaris and AIX (SP-series machines).
  - Windows is also possible, but currently not well supported.
- First you *must have a native MPI installed*. In general this can be MPICH, LAM, SunHPC or IBM MPI (SP*X*).
  - Other platforms are possible but will need work to port.
- Download the mpiJava sources from

  http://www.hpjava.org/mpiJava.html
- Extract the archive, then go through the conventional GNU-like installation process.
  - **./configure**, **make**, …
  - You will need to specify a suitable option to the configure script for MPI implementations other than MPICH.
- Put the **mpiJava/src/scripts/** directory on your **PATH**, and put **mpiJava/lib/classes/** directory on your **CLASSPATH**.

Running mpiJava Programs

- The mpiJava release bundle includes a test suite translated from IBM MPI test suite. The first thing you should probably try is running this by:

  $ make check

- Assuming this completes successfully (you can live with a few warning messages), try running some examples in the **mpiJava/examples/** directory.
  - e.g, Go into **mpiJava/examples/simple** and try:

    $ javac Hello.java

    $ prunjava 2 Hello
- The script **prunjava** is a convenience script mostly designed for test purposes.
  - It takes just two arguments: the number of processes to run in, and the class name.
  - Most likely these two processes will be on the local host, unless you take steps to specify otherwise.
- For production runs you will probably have to use a procedure dependent on your MPI implementation to start parallel programs. See the file **mpiJava/README** for additional information.

Programming Model

- In mpiJava, every MPI process or node corresponds to a single JVM, running on some host computer.
  - mpiJava is *not* integrated with Java threads: it is not possible to use MPI operations to communicate between Java threads, and in general it is not safe for more than one thread in a single JVM to perform MPI operations (concurrently).
  - This is because mpiJava is implemented on top of a native MPI, and most native implementations of MPI are not thread-safe.
  - It *should*, however, be OK to have multiple Java threads in your mpiJava program, provided only one (per JVM) does MPI communication, *or* provided inter-thread synchronization is used to ensure only one thread is doing MPI communication at any given time (I'm not aware the latter has been tested).
- Generally all processes in an mpiJava program must be started simultaneously by a command like **mpirun** or equivalent, depending on your underlying MPI.

Setting up the Environment

- An mpiJava program is defined as a Java *application*. In other words the program is implemented by a public static **main()** method of some class defined by the programmer.
- All the classes of the mpiJava library belong to the package **mpi**. Either import these classes at the top of your source files, or use the fully-qualified names (**mpi.Comm**, etc) throughout your program.
- MPI is initialized by calling **mpi.MPI.Init()**.
  - The **MPI** *class* has a few static methods for administrative things, and numerous static fields holding global constants.
- You should forward the arguments parameter of the **main()** method to the **Init()** method.
  - This array is not necessarily identical to the command line arguments passed when starting the program (depends on the native MPI). If you need the user-specified command line arguments, use the result of the **Init()** method, e.g.:

    ```
    String [] realArgs = MPI.Init() ;
    ```
- Call **mpi.MPI.Finalize()** to shut down MPI before the **main()** method terminates.
  - Failing to do this may cause your executable to not terminate properly.

The **Comm** class

- The **Comm** class represents an *MPI communicator*, which makes it probably the most important class in the API. All communication operations ultimately go through instances of the **Comm** class.
- In MPI a communicator defines two things:
  - it defines a group of processes—the participants in some kind of parallel task or subtask, and
  - it defines a *communication context*.
- The idea of a communication context is slightly subtle. The idea is that *the same group* of processes might be involved in more than one kind of "ongoing activity" (for example the different kinds of activity might be parallel operations from libraries defined by different third parties).
- You don't want these distinct "activities" to interfere with one another.
  - For example you don't want messages that are sent in the context of one activity to be accidentally received in the context of another. This would be a kind of race condition.
- So you give each activity a different communication context. You do this by giving them different instances of the **Comm** class.
  - Messages sent on one communicator can never be received on another.

Rank and Size

- A process group in MPI is defined as a *fixed set of processes*, which never changes in the lifetime of the group.
- The number of processes in the group associated with a communicator can be found by the **Size()** method of the **Comm** class. It returns an **int**.
- Each process in a group has a unique *rank* within the group, an **int** value between **0** and **Size() – 1**. This value is returned by the **Rank()** method.
- Note that the methods of mpiJava usually have the same names as in the C binding, but omit prefixes like "**MPI_Comm_**" that redundantly identify the class.
  - This follows the MPI C++ binding, but it has the result that method names start with upper-case letters. This is contrary to normal (good) Java practice (and it was changed in the MPJ spec).

The World Communicator

- In MPI, the “initial” communicator is a communicator spanning all the processes in which the SPMD program was started.
- In mpiJava, this communicator is accessed as a static field of the MPI class:

  MPI.COMM_WORLD

  - Pretty clearly this ought to be a static *final* field of the MPI class. For weird historic reasons it is not declared **final** in the mpiJava API. This should probably be changed.
- Simple mpiJava programs may only ever need to use the world communicator.

Simple send and receive

- Not surprisingly, the basic point-to-point communication methods are members of the **Comm** class.
- 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) ;
```

- The arguments **buf**, **offset**, **count**, **type** describe the data buffer—the storage of the data that is sent or received. They will be discussed on the next slide.
- **dst** is the rank of the destination process relative to **this** communicator. Similarly in **Recv()**, **src** is the rank of the source process.
- An arbitrarily chosen **tag** value can be used in **Recv()** to select between several incoming messages: the call will wait until a message sent with a matching tag value arrives.
- The **Recv()** method returns a **Status** value, discussed later.

Communication Buffers

- Most of the communication operations take a sequence of parameters like

  Object buf, int offset, int count, Datatype type

- In the actual arguments passed to these methods, **buf** must be an array (or a run-time exception will occur).
  - The reason for not *declaring* it as an array was that one would then need to overload with about 9 versions of most methods, e.g.

    void Send(int [] buf, …)
    void Send(long [] buf, …)
    …

    and about 81 versions some odd operations that involve two buffers, possibly of different type. Declaring **Object buf** allows any kind of array in one signature.
- **offset** is the element in the **buf** array where message starts. **count** is the number of items to send. **type** describes the type of these items.

Buffer Element Type

- If the **type** argument is one of these basic datatype, the **buf** argument must be an array of elements of the corresponding Java type.
  - e.g. if the **type** argument is **MPI.BYTE** the **buf** argument must have type **byte []**.
- In these cases the **type** argument is slightly redundant in Java, because the element type could be determined from the **buf** argument by reflection.
- But the reason for retaining the MPI-like **type** argument was that we wanted to support MPI-like *derived datatypes* (see later).

ANY_SOURCE and ANY_TAG

- A **recv()** operation can explicitly specify which process within the communicator group it wants to accept a message from, through the **src** parameter.
- It can also explicitly specify what *message tag* the message should have been sent with, through the **tag** parameter.
- The **recv()** operation will block until a message meeting both these criteria arrives.
  - If other messages arrive at this node in the meantime, this call to **recv()** ignores them (which may or may not cause the senders of those other messages to wait, until they *are* accepted).
- If you want the **recv()** operation to accept a message from *any* source, or with *any* tag, you may specify the values **MPI.ANY_SOURCE** or **MPI.ANY_TAG** for the respective arguments.

Status values

- The **recv()** method returns an instance of the **Status** class.
- This object provides access to several useful pieces about the message that arrived. Below we assume the **Status** object is saved to a variable called **status**:
  - **int** field **status.source** holds the rank of the process that sent the message (particularly useful if the message was received with **MPI.ANY_SOURCE**).
  - **int** field **status.tag** holds the message tag specified by the sender of the message (particularly useful if the message was received with **MPI.ANY_TAG**).
  - **int** method **status.Get_count(type)** returns number of items received in the message.
  - **int** method **status.Get_elements(type)** returns number of basic elements received in the message.
  - **int** field **status.index** is set by methods like **Request.Waitany()**, described later.

MPIException

- Nearly all methods of mpiJava are *declared* to throw the **MPIException**.
- This is *supposed* to report the error status of a failed method, closely following the failure modes documented in the MPI standard.
  - Actually this mechanism has never been implemented in mpiJava, and instead failed MPI methods normally abort the whole program.
  - Also be warned that the current mpiJava wrappers lack most of the safety checks you might expect in the Java libraries—erroneous programs may cause the JVM to crash with very un-Java-like error messages.

Communication Modes

- Following MPI, several communication modes are supported through a family of send methods. They differ mostly in their approaches to buffering and synchronization.
  - **Send()** implements MPI's *standard mode* semantics. The message *may* be buffered by the system, allowing **Send()** to return before a matching **Recv()** has been posted, but the implementation does not guarantee this.
  - **Bsend()** the system will attempt to buffer messages so that **Bsend()** method can return immediately. But it is the programmer's responsibility to tell the system how much buffer will be needed through **MPI.Buffer_attach()**.
  - **Ssend()** is guaranteed to block until the matching **Recv()** is posted.
  - **Rsend()** is obscure—see the MPI standard.
- I always use *standard mode sends*, and program defensively to guard against deadlocks (i.e. assume that the **Send()** method *may* block if the receiver is not ready).
  - **Send()** *may* behave like **Bsend()**, or it *may* behave like **Ssend()**.
  - Avoiding deadlocks may require use of the non-blocking versions of the communication operations…

Non-blocking Communication Operations

- Sometimes—for efficiency or correctness—you need to be able to do something else while you are waiting for a particular communication operation to complete.
  - This is particularly true for receive operations, or for send modes that may block until the receiver is ready.
- MPI and mpiJava provide *non-blocking variants* of **Send()**, **Bsend()**, **Ssend()**, **Rsend()**, and **Recv()**. These are called **Isend()**, **Ibsend()**, **Issend()**, **Irsend()**, and **Irecv()**.
- The parameter lists are the same as for the blocking versions, but each of them immediately returns a **Request** object.
- Later, separate methods are applied to the **Request** object, to wait for (or detect) completion of the originally-requested operation.
  - For each non-blocking variant, local completion is defined in the same way as for the blocking versions.

Simple completions

- The simplest way of waiting for completion of a single non-blocking operation is to use the instance method **Wait()** in the **Request** class, e.g:

```
// Post a receive operation
Request request =
        Irecv(intBuf, 0, n, MPI.INT, MPI.ANY_SOURCE,  0) ;
// Do some work while the receive is in progress
...
// Finished that work, now make sure the message has arrived
Status status = request.wait() ;
// Do something with data received in intBuf
...
```

- The **Wait()** operation is declared to return a **Status** object. In the case of a non-blocking receive operation, this object has the same interpretation as the **Status** object returned by a blocking **Recv()** operation.
  - For a non-blocking *send*, the status object is not particularly interesting and there isn't usually much point saving it.

Void requests

- In **mpiJava** we say a request object is “void” after the communication has completed and its **Wait()** operation has returned (this is slightly different from the terminology used in the MPI standard).
- A void request object has no further use. It will eventually be removed from the system by the garbage collector.

Overlapping Communications

- One useful application of non-blocking communication is to break cycles of dependence that would otherwise result in deadlock.
- For example, if **P** processes in a group attempt to cyclically shift some data amongst themselves by

```
int me = comm.Rank() ;
comm.Send(srcData, 0, N, type, (me + 1) % P, 0) ;
comm.Recv(dstData, 0, N, type, (me + P - 1) % P, 0) ;
```

  this may (or may not) deadlock (depending on how much buffering the system provides), because initially everybody is doing a **Send()**. All these operations may block, because initially nobody is doing a **Recv()**.
- A safe version is

```
Request sreq = comm.Isend(srcData, 0, N, type, (me + 1) % P, 0) ;
Request rreq = comm.Irecv(dstData, 0, N, type, (me + P - 1) % P, 0) ;
rreq.Wait() ;
sreq.Wait()
```

Sendrecv

- Since it is fairly common to want to simultaneously send one message while receiving another (as illustrated on the previous) slide, MPI provides a more specialized operation for this.
- In **mpiJava** the corresponding method of **Comm** has the complicated signature:

```
Status Sendrecv(Object sendBuf, int sendOffset, int sendCount,
                Datatype sendType, int dst, int sendTag,
                Object recvBuf, int recvOffset, int recvCount,
                Datatype recvType, int src, int recvTag) ;
```

  - This can be more efficient that doing separate sends and receives, and it can be used to avoid deadlock conditions (if no process is involved in more than one cycle of dependency).
  - There is also a variant called **Sendrecv_replace()** which only specifies a single buffer: the original data is sent from this buffer, then overwritten with incoming data.

Multiple Completions

- Suppose several non-blocking sends and/or receives have been posted, and you want to wait just for one to complete—you are willing to handle whatever happens first. In mpiJava this is done with a static method of **Request**:

  static Status Waitany(Request [] requests)

  - Here **requests** is an array of request objects we are interested in (it may contain some void requests: they are ignored).
  - The returned status object has the same interpretation as the one returned by simple **Wait()**, except that the field **index** in this status object will have been defined. The request that completed (and is now void) was **requests [status.index]**.
- If you need to wait for *all* requests in an array to terminate, use the method **Waitall()**, which has a similar signature but returns an array of **Status** objects, one for each input request.
- As in MPI, there is also **Waitsome()**, and various "test" methods, but in general they are less useful.

Groups

- Every communicator has a process group associated with it.
- There is a separate class that describes just process groups, stripped of any communication context: the **Group** class.
- MPI provides a set of functions for manipulating and comparing groups, and mpiJava makes these functions available as methods. Some of the most important are:

Group group()

» An instance method of **Comm**. It returns the process group associated with this communicator

Group Incl(int[] ranks)

» An instance method of **Group**. Create a subset group including specified processes.

int[] Translate_ranks(Group group1, int[] ranks1, Group group2)

» A static method of **Group**. Translate ranks relative to one group to ranks relative to another.

The **Free()** method

- The C and Fortran bindings of MPI have many "**MPI_*Type*_Free**" methods used to free opaque objects that have been allocated.
- Advice of some authorities notwithstanding, the mpiJava API drops most of these methods from the API, and defines native **finalize()** methods on its classes, which the garbage collector calls to free the associated native objects.
- The **Comm** class is an exception. If you call any of the methods described here to create a temporary communicator, you should explicitly call the **Free()** method of the communicator if it needs to be deallocated.
  - This is because **MPI_Comm_Free()** is a *collective operation*.

Collective Communications

- A popular feature of MPI is its family of collective communication operations. Of course these all have **mpiJava** bindings.
- All processes in the communicator group must engage in a collective operation "at the same time"
  - i.e. all processes must invoke collective operations in the same sequence, all passing consistent arguments.
- Can argue that collective communications (and other similar collective functions built on them) are a defining feature of the SPMD programming model.
- The simplest interesting example is the broadcast operation: all processes invoke the operation, all agreeing one root process. Data is broadcast from that root.

Example Collective Operations

- All the following are instance methods of **Intracom**:

  void Bcast(Object buf, int offset, int count, Datatype type, int root)

  - Broadcast a message from the process with rank **root** to all processes of the group.

  void Barrier()

  - Blocks the caller until all processes in the group have called it.

  void Gather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

  - Each process sends the contents of its send buffer to the **root** process.

  void Scatter(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)

  - Inverse of the operation **Gather**.

  void Reduce(Object sendbuf, int sendoffset, Object recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)

  - Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the **root** process.

Topologies

- MPI includes support for “topologies”, a mechanism to define some structured “neighborhood” relations amongst the processes in the communicator.
- In mpiJava there are two subclasses of **Intracom** that provide this added structure:
  - Instances of **Cartcom** represent Cartesian communicators, whose processes are organized in the multidimensional grid.
  - Instances of **Graphcom** represent communicators with neighbor relation defined by a general graph.

Cartcom Example

```
int dims [] = new int [] {2, 2} ;
boolean periods [] = new boolean [] {true, true} ;
Cartcomm p = MPI.COMM_WORLD.Create_cart(dims, periods, false) ;

// Create local `block‘ array, allowing for ghost cells.
int sX = blockSizeX + 2 ;
int sY = blockSizeY + 2 ;
block = new byte [sX * sY] ;
...

ShiftParms spx=p.Shift(0,  1);
...

// Shift this block's upper x edge into next neighbour's lower ghost edge.
p.Sendrecv(block, blockSizeX * sY, 1, edgeXType, spx.rank_dest, 0,
                  block,  0, 1,                  edgeXType, spx.rank_source, 0) ;
```

Remarks

- Code adapted from **mpiJava/examples/Life.java** example code in the mpiJava release.
- Creates a Cartesian communicator **p**, representing a 2 by 2 grid with wraparound at the edges (defined by **dims** and **periods** arrays).
- The **Shift()** method of **Cartcom** defines the source and destination ranks required to define a cyclic shift pattern of communication.
- In this example, **edgeXType** is a derived datatype…

Derived Datatypes

- In addition to the basic datatypes enumerated earlier, an MPI datatype can be a *derived datatype*.
- A derived datatype represents some "pattern" of elements in the buffer. In MPI these patterns are characterized by the *types* of the component elements, and their *memory offset* relative to the start of the item in the buffer.
- **mpiJava** implements a restricted form of MPI derived datatypes: the component elements of the datatype can have arbitrary offsets in the *index space* of the buffer array, but they must all have *the same* primitive type.
- Although derived datatypes don't have such a natural role in mpiJava as in MPI, they can still be useful for representing message data that may be laid out non-contiguously in the buffer.

Derived Datatype Example

```
// Create local `block‘ array, allowing for ghost cells.
int sX = blockSizeX + 2 ;
int sY = blockSizeY + 2 ;
block = new byte [sX * sY] ;
...

// Define derived datatypes representing element pattern of x and y edges.
Datatype edgeXType = Datatype.Contiguous(sY, MPI.BYTE);
edgeXType.Commit() ;

Datatype edgeYType = Datatype.Vector(sX, 1, sY, MPI.BYTE);
edgeYType.Commit() ;
...

// Shift this block's upper x edge into next neighbour's lower ghost edge.
p.Sendrecv(block, blockSizeX * sY, 1, edgeXType, spx.rank_dest, 0,
                  block,  0, 1,                  edgeXType, spx.rank_source, 0) ;
```

Remarks

- Again adapted from **mpiJava/examples/Life.java** example code in the mpiJava release.
- Illustrates two of the factory methods for derived datatypes

  static Datatype Contiguous(int count, Datatype oldtype)
  - » Static method of **Datatype** class. Construct a new datatype representing replication of old datatype into contiguous locations.

  static Datatype **Vector**(int count, int blocklength, int stride, Datatype oldtype)
  - » Static method of **Datatype** class. Construct new datatype representing replication of old datatype into locations that consist of equally spaced blocks.

  The API includes several more similar factory methods, e.g. there is one that takes an index vector, allowing any gather or scatter pattern to be implemented.
- In buffer description parameters like:

  Object buf, int offset, int count, Datatype type

  if **type** is a derived type, **count** is the number of composite "items". But **offset** is still measured in individual elements of **buf**, i.e. the starting position is not generally required to be a multiple of the item size.

The **mpiJava** Implementation. Lessons Learned.

mpiJava Implementation Issues

- mpiJava was originally conceived as a set of simple wrappers to a native, high-performance MPI implementation.
- That was how it started out, but over time various problems emerged, and functionality has gradually moved into the "wrappers" themselves.
- Notable issues include:
  1. There are non-trivial problems interfacing complex systems like MPI and the JVM, due to interactions at the level of OS signals (and sometimes also thread safety of OS calls).
  2. Copying of buffers imposed by JNI implementations can affect the semantics of MPI calls.
  3. Communicating the **MPI.OBJECT** basic type requires a new layer of protocol.

1. Signal Handlers, etc

- Problems can arise interfacing packages that make extensive use of OS system calls to the JVM, because the package and the JVM implementation may simply make system calls in incompatible ways.
- One problem in particular that afflicted early versions of mpiJava was use of signal handlers.
- JVM implementations often set signal handlers for OS signals like **SIGSEGV**, etc:
  - e.g. the JVM implementer might allow segmentation violations to occur if a call stack overflows in the normal course of events, then take corrective action in a signal handler.
- MPI systems also typically set signal handlers.
  - e.g. if the user causes a segmentation violation on one process, MPI catches this in a signal handler, and tries to shut down all processes in an orderly way.
- In mpiJava, the JVM is started (the JVM's signal handlers are installed), *then*, later **MPI_INIT()** is called (MPI's signal handlers are installed).
  - By default the MPI handler for **SIGSEGV** (say) overrides the JVM handler.
  - So at random points in the program, where the JVM handler *should* have been invoked, the MPI handler gets invoked instead. All processes are shut down in an orderly way…

Signal Chaining

- This kind of problem was recognized by Sun and resolved when they introduced specific support for "signal chaining" in J2SE 1.4.
  - Check out

    http://java.sun.com/j2se/1.4/docs/guide/vm/signal-chaining.html

    if you think you may be experiencing this problem with your own native libraries
- By "preloading" the library called **libjsig**, the behavior of the standard library methods for installing system handlers are changed, so that signal handlers get chained together.
  - Now the JVM handler gets called first and execution resumes, as nature intended.
- This and similar solutions get automatically "configured in" to mpiJava 1.2.5 and later.

Thread Safety Issues

- In the past we have also seen some intermittent failures of mpiJava codes which we attribute to the MPI software not using OS system calls in a thread-safe way.
  - For example they may use the global variable errno. By default this is may not be *thread local*, hence using the value set by system calls isn't thread-safe.
- On Solaris, for example, we now recommend that MPICH or LAM should be built specifying the cc compiler flag:

  –D_REENTRANT

  (the corresponding flag on AIX is **–D_THREAD_SAFE**).
- See the README file in mpiJava 1.2.5 for further details.

2. JNI and Copying of Buffers

- Inside the implementation of a native method, an obvious way to get the C array of primitive elements corresponding to an mpiJava **buf** parameter, is by something along the lines…

```
… function-name( …,  JNIEnv *env,  jobject buf,  int baseType, …) {
    …
    switch(baseType) {
        case BYTE {
            jbyte* els = (*env)->GetByteArrayElements(env, buf, &isCopy) ;
            … do something with els pointer …
        }
        case INT {
            jint* els = (*env)->GetIntArrayElements(env, buf, &isCopy) ;
            … do something with els pointer …
        }
        …etc …
    }
}
```

- In the mpiJava sources, you will find code like this in the function **getBufPtr()** in **mpiJava/src/C/mpi_Comm.c**.

Copying of JNI Array Arguments

- The **els** pointer can be passed directly to MPI functions like MPI_SEND, etc, as the C-style reference to the message buffer.
  - With early implementations of the JVM (e.g. the so-called "Classic" JVM), this approach worked fine: **els** was indeed a reference to the physical memory used by the Java **buf** array.
- In most more recent JVMs from Sun ("Hotspot", etc) this isn't the case.
  - With these JVMs **isCopy** flag is set true by **Get*Type*ArrayElements()**, etc. This indicates that the pointer els is a a reference to a C-language copy of the physical memory used by the JVM to store the array elements.
  - This has two ramifications: first, the extra memory-to-memory copying implied by this is a substantial performance overhead.
  - Second, it may change the semantics of MPI calls, if we aren't careful.

Why Does JNI Copy Arrays?

- In general a given Java object or array is *not* allocated a fixed storage place in physical memory.
  - At any given time, of course, it is stored in some physical location.
  - But as new objects are created and old ones are deleted, the Garbage Collector will copy surviving objects around in memory, to optimize memory usage.
- Some garbage collectors support an operation of ***pinning***, which allows one to tell the GC *not* to relocate a particular object or array until it is explicitly ***unpinned***.
- If the GC supports pinning, JNI will typically exploit this, and **Get*Type*ArrayElements()**, etc, will return a real reference to the pinned array. Otherwise they return a copy.
- The Classic JVM *did* support pinning. So do subsequent IBM derivatives (e.g. the JVMs in IBM JDK for Linux, AIX). Most recent JVMs from Sun unfortunately *do not*.

Buffer Copying and MPI Semantics

- In a JVM where **Get*Type*ArrayElements()** creates a copy of the array elements, the subsequent call to **Release*Type*ArrayElements()** will by default copy the (possibly modified) C elements back to the Java array.
  - Should guarantee correct semantics of an MPI call, even if inefficient?
- Problems arise if more than one MPI call is concurrently modifying *different portions* of the same Java array. For example suppose the fragment from the Life code given earlier was written with non-blocking communications:

```
Request sreq =
        p.Isend(block, blockSizeX * sY, edgeXType, spx.rank_dest, 0) ;
Request rreq =
        p.Irecv(block, 0, 1, edgeXType, spx.rank_source, 0) ;
rreq.Wait() ;
sreq.Wait() ;
```

  The **Isend()** , **Irecv()** calls both copy the *whole* of the **block** array. The **Wait()** calls would presumably call **Release*Type*ArrayElements()** to overwrite the *whole* JVM array. In this example the unmodified **Isend()** copy is copied back last, and the changes **Irecv()** made are lost.
- You can probably find workarounds, but they are probably complex.

Aside: Completion of Non-Blocking Comms

- The example on the previous slide raises an interesting side-issue:
  - Even if the JVM *does* support pinning, so there is no overwriting, and the example on the previous slide works correctly, it assumes that buffers are *pinned* in the call to **Isend()**, etc, and *unpinned* in calls to **Wait()** (or **Waitany()**, or whatever other operation completes the request).
  - This already implies extra fields in the request object, beyond a simple reference to a native **MPI_Request** (so that the JNI implementation of the **Wait()** operation, etc, knows which Java buffer to release).
  - Incidentally, the problems with MPI *persistent requests* (which we did not discuss earlier) are even worse. In fact the mpiJava **Prequest** class does not use native persistent requests at all—it has a very naïve implementation and is probably best avoided.

mpiJava Strategy for Buffers

- Since version 1.2.3, mpiJava has taken a two-pronged approach to handling MPI buffers.
- First the **configure** script tests if the Java implementation appears to support pinning.
- If it *does* a macro **GC_DOES_PINNING** is defined that causes the C code to use the JNI functions **Get*Type*ArrayElements()** and **Release*Type*ArrayElements()** to access buffers, as described in the previous slides. This is efficient because it avoids all unnecessary copying, and it preserves the correct semantics.
- If it *does not*, a different, more complex approach is compiled into the JNI methods.

Using **Get/ReleasePrimitiveArrayCritical**

- If the garbage collector does not support pinning, copying is unavoidable, but we endeavor to copy just the data that needs to be communicated, and nothing more.
- We use the JNI methods **GetPrimitiveArrayCritical()** and **ReleasePrimitiveArrayCritical()** to get a *temporary* reference to the physical address where the JVM stores elements.
  - The "contract" for these operations does not allow one to do general operations between two these calls: only operations that won't deschedule the thread are allowed.
- Memory-to-memory copying is OK in the "critical region", so we use the native **MPI_Pack()** to copy exactly the data required for a send into a temporary array allocated in the C code. This is sent as **MPI_BYTE** data.
- At the receiving end we again use **GetPrimitiveArrayCritical()** and **ReleasePrimitiveArrayCritical()**, and in the critical region we use **MPI_Unpack()** to copy contents of the received byte buffer directly into JVM memory.

3. Supporting **MPI.OBJECT**

- Since version 1.2, mpiJava has supported **MPI.OBJECT** as one of its basic message types.
- This is very natural in Java, and it allows various things to be done conveniently.
  - Besides sending *arrays of true objects*, it allows you to send *Java multidimensional arrays* directly.
- Unfortunately this comes with substantial performance overheads because Java object serialization is very *slow*, by the standards of HPC.
  - Some groups have developed faster serialization frameworks (notably the JavaParty and Manta groups). For best results these frameworks either require a special compiler, or require the programmer to add special methods to serializable classes.
  - At Syracuse University we did some work on reducing the serialization overheads in the specific context of MPI. We had some success, but the approach did not make it into the released version of mpiJava.
- In the end mpiJava just uses *standard Java object serialization*. Be aware that it is slow, and avoid using **MPI.OBJECT** in performance-critical parts of your code.

Extending the Communication Protocol

- One fundamental difference between sending and receiving objects, and sending normal MPI data, is that the receiver cannot put a bound on how much serialized data will be received, based only on locally available information.
- For example the receiver might do

```
MyClass[] message = new MyClass [20] ;
comm.Recv(message, 0, 20, MPI.OBJECT, src, 0) ;
```

  Even if there *was* a simple way for **Recv()** to compute how much space 20 serialized objects of class **MyClass** take (and actually there isn't), the sender might send *20 objects of a* **MyClass** *subclass*, which could have extra fields (and thus, presumably, a larger serialized representation).
- So it's impossible for the receiver to allocate a buffer and be sure it will be large enough for the serialized data, unless the sender first tells the receiver how much data to expect.
  - In general sending a single **MPI.OBJECT** message may require multiple low-level MPI message exchanges. In other words, it needs a higher level protocol.

Extended Protocol

- The extended protocol for MPI.OBJECT is simple: the sender first serializes the data, then sends a header containing:
  - *size* of the serialized data, in bytes
  - count of the number of *object elements* in the message.
- On receiving the header, a temporary buffer is allocated to hold the bytes of serialized data.
- On receiving the subsequent data into that buffer, it is *deserialized* into the user-specified buffer.
  - One could do slightly better by always allocating a minimum size buffer, and *only* sending *two* messages if the data was too big for that.
- Simple as it is, this requires many changes to the implementation of the wrappers.

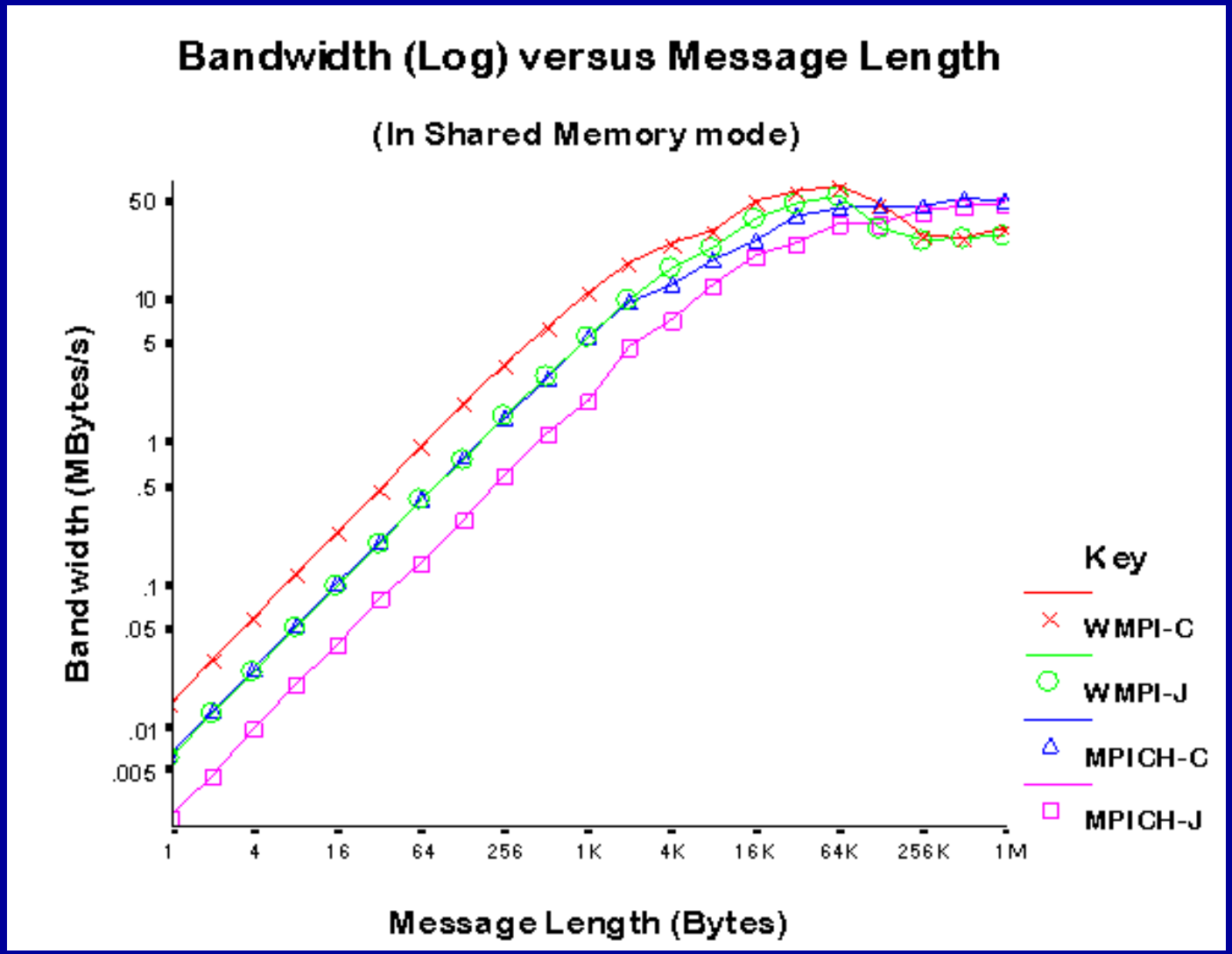
Changes to Support Objects

- All support of **MPI.OBJECT** is done on the Java side of the wrappers (we don't want to make the C code any more complex). Most important changes are:
  - Blocking send and receives are modified in an "obvious" way, using standard Java serialization utilites.
  - Non-blocking sends and receives are modified in a less obvious way:
    - Completion of a send recognized after the non-blocking *data send request* has completed.
    - Ready for completion of a receive when the non-blocking *header receive request* has completed. Then do a blocking receive of the data.
    - Further elaboration of the mpiJava **Request** class to support this.
    - To avoid deadlocks in strange cases, data packets should be sent on a different communicator—each mpiJava communicator acquires a native *shadow communicator* for sending object data.
  - Collective communications are implemented from scratch in the Java wrappers for the object case (not very efficiently).
  - Derived datatypes with **MPI.OBJECT** as base type are implemented from scratch in the Java wrappers.

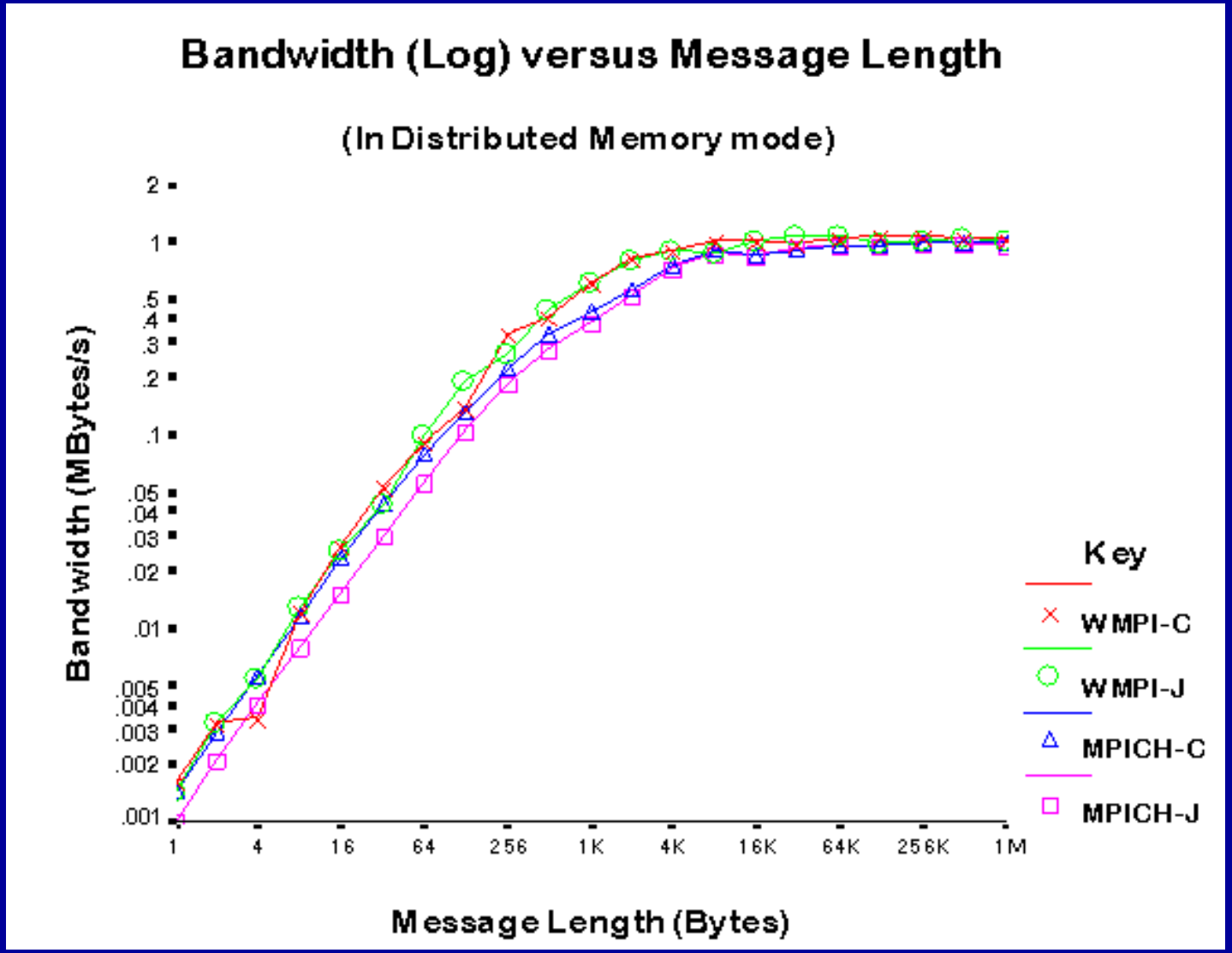
Lessons for the Future

- mpiJava grew out of a project that was supposed to be an interim solution. The result was more popular than expected. But the implementation isn't very elegant.
- We have learnt various lessons from it—many technical issues—but the most important is probably:
  - Trying to form JNI wrappers to a large API is probably not a good approach. Over time more and more of our code migrated into the wrappers. It would have been better to design a *small, base API* that could be implemented natively, then from the outset code the bulk of the *application-level functionality* on the Java side. The smaller base API would isolate the technical difficulties with associated with JNI, and be easier to maintain and port.
- This was the approach we took in **mpjdev**, mentioned later.

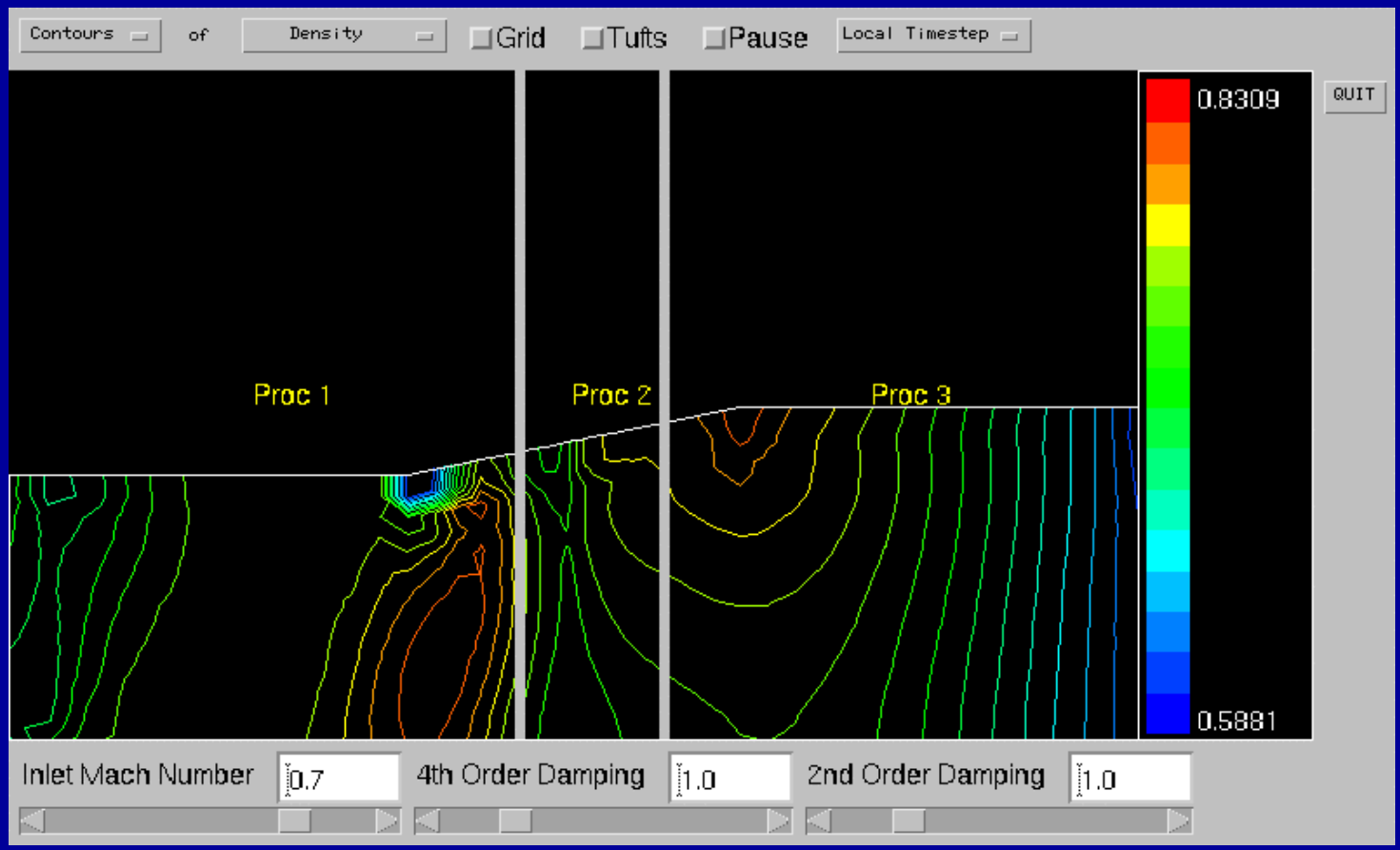
mpiJava performance 1. Shared memory mode



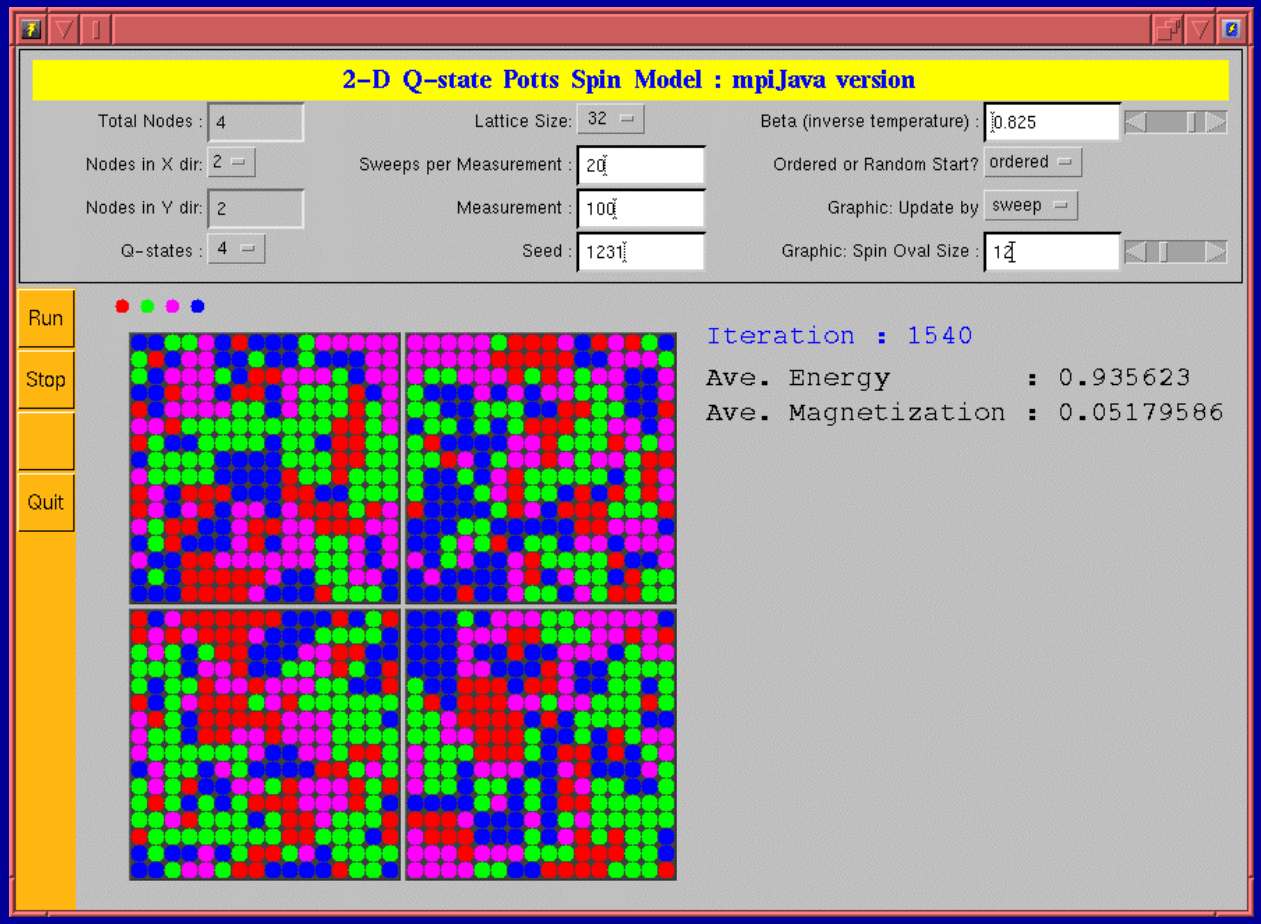
mpiJava performance 2. Distributed memory



mpiJava demos 1. CFD: inviscid flow



mpiJava demos 2. Q-state Potts model



Special Topic: HPJava

mpiJava and HPJava

- mpiJava was always supposed to be one component of a larger project called *HPJava*.
- HPJava is based on a set of Java *language extensions* to support library-based, parallel programming in a style somewhere in between the classical HPC standards of MPI and *High Performance Fortran* (HPF).
- Our HPJava development kit software, *hpjdk*, was released earlier this year.
  - It contains about an order of magnitude more code than the mpiJava release bundle, and took several years to develop (started the same time as mpiJava).
- HPJava is probably more controversial the mpiJava, because it involves language extensions, and because it has an unfamiliar programming model.
  - But it can be very neat.

Summary

- HPJava is a language for parallel computing.
- It extends Java with features from languages like Fortran.
- New features include true *multidimensional arrays* and parallel data structures (*distributed arrays*).
- It introduces a parallel computing model we call the *HPspmd* programming model.

HPF Background

- By early 90s, value of portable, standardized languages universally acknowledged.
- Goal of *HPF Forum*—a single language for High Performance programming. Effective across architectures—vector, SIMD, MIMD, though SPMD a focus.
- HPF—an extension of Fortran 90 to support the data parallel programming model on distributed memory parallel computers
- Supported by Cray, DEC, Fujitsu, HP, IBM, Intel, Maspar, Meiko, nCube, Sun, and Thinking Machines

Motivations 1:HPspmd

- SPMD (Single Program, Multiple Data) programming has been very successful for parallel computing.
  - Many higher-level programming environments and libraries assume the SPMD style as their basic model—ScaLAPACK, DAGH, Kelp, Global Array Toolkit,…
- But the library-based SPMD approach to data-parallel programming lacks the uniformity and elegance of HPF.
  - Compared with HPF, creating distributed arrays and accessing their local and remote elements is clumsy and error-prone.
- Because the arrays are managed entirely in libraries, the compiler offers little support and no safety net of compile-time or compiler-generated-run-time checking.
- These observations motivated our suggestion of the *HPspmd model*—direct SPMD programming supported by additional syntax for HPF-like distributed arrays.

HPspmd Features

- Proposed by Fox, Carpenter, Xiaoming Li around 1998.
- Independent processes executing same program, sharing elements of *distributed arrays* described by *special syntax*.
- Processes operate directly on locally owned elements. Explicit communication needed in program to permit access to elements owned by other processes.
- Envisaged bindings for base languages like Fortran, C, Java, etc.
  - HPJava is the only version that was actually realized.
- Claimed benefits:
  - Translators are *much* easier to implement than HPF compilers. No compiler magic needed
  - Attractive framework for library development, avoiding inconsistent representations of distributed array arguments
  - Can directly call MPI-like functions from within an HPspmd program
  - Better prospects for handling irregular problems (easier to fall back on specialized libraries as required)?

Motivations 2: Multidimensional Arrays

- Java is an attractive language, but arguably could be improved for large computational tasks
- As many people have observed, the lack of *true multidimensional arrays* is an issue.
  - Java provides *array of arrays*
  - But compiler analysis of their use—thus compiler optimization—is harder than for true multidimensional arrays.
  - They don't support Fortran 90-like array-sections, which are useful in scientific algorithms, and especially for calling scientific libraries.
- See

  "A Comparison of Three Approaches to Language, Compiler, and Library Support for Multidimensional Arrays in Java"

  J. Moreira, S. Midkiff, M. Gupta

  and references therein, for a relatively recent discussion.
- Java Grande Forum Numerics Working Group made a series of proposals for adding *multiarrays* to Java:

  http://math.nist.gov/javanumerics

Multiarray Syntax

- One feature of HPJava is that it provides *multiarrays*. These multiarrays admit *regular sections*.
  - This was a by-product of the large goal of providing *distributed arrays*, but it seems to be a useful feature in its own right.
- Examples of HPJava syntax for multiarrays:

```
int [[ * , * ]] a = new int [[ 5 , 5 ]] ;    // Type signature, creation expression
for (int i=0; i<4; i++)
    a [ i , i+1 ] = 19 ;                     // Subscripting
foo ( a[[ : , 0 ]] ) ;                       // Section

int [[ * ]] b = new int [[ 100 ]] ;          // 1d Multiarray
int [ ] c = new int [ 100 ] ;                // Ordinary Java array
                // (Note difference between b and c).
```

- Syntax compatible with proposals within the Java Grande Forum.

Parallel HPJava Syntax

```
Procs p = new Procs2(2, 2);
on(p) {
    Range x = new ExtBlockRange(M, p.dim(0), 1),
          y = new ExtBlockRange(N, p.dim(1), 1);
    float [[-,-]] a = new float [[x, y]] ;
    . . . Initialize edge values in 'a' (boundary conditions) …
    float [[-,-]] b = new float [[x,y]],  r = new float [[x,y]];  // r = residuals
    do {
        Adlib.writeHalo(a);
        overall (i = x for 1 : N – 2)
            overall (j = y for 1 : N – 2) {
                float newA = 0.25 * (a[i - 1, j] + a[i + 1, j] + a[i, j - 1] + a[i, j + 1] );
                r [i, j] = Math.abs(newA – a [i, j]);
                b [i, j] = newA;
            }
            HPutil.copy(a, b);  // Jacobi relaxation.
     } while(Adlib.maxval(r) > EPS);
}
```

Remarks

- The details will become clearer shortly.
- Points to note are we have three new pieces of *syntax* here:
  - The ***on*** *statement*
  - *Distributed array* type signature and creation expression
  - The ***overall*** *statement*

Distributed Arrays in HPJava

- Many differences between distributed arrays and ordinary arrays of Java. New kind of container type with special syntax.
- Type signatures, creation expressions, use *double brackets* to *emphasize* distinction:

```
Procs2 p = new Procs2(2, 3);
on(p){
    Range x = new BlockRange(M, p.dim(0));
    Range y = new BlockRange(N, p.dim(1));
    float [[-,-]] a = new float [[x, y]] ;
    . . .
}
```

  - *Distributed dimension* distinguished from an ordinary, sequential (non-distributed) dimension of a multiarray by hyphen, –, instead of asterisk, *, in type signature.
  - In corresponding slot of *distributed array creation expression*, must use an instance of a **Range** class. This defines the extent, target grid dimension for distribution, and distribution format.

The **overall** construct

- ***overall***—a distributed, parallel loop
- General form, parameterized by index triplet:

  overall (i = x for l : u : s) { . . . }

  i = *distributed index*,

  x = a range object,

  l = lower bound, **u** = upper bound, **s** = step.

  – An overall construct scopes the distributed index symbol, i, which stands for a location in the range **x**.
- In general a subscript used in a distributed dimension of an array *must be a distributed index* in the corresponding range of the array.

  – It can be a *shifted index*, **i ± *expression***, if and only if the array dimension has suitable ghost regions.

Example Revisited

```
Procs p = new Procs2(2, 2);
on(p) {
    Range x = new ExtBlockRange(M, p.dim(0), 1),
          y = new ExtBlockRange(N, p.dim(1), 1);
    float [[-,-]] a = new float [[x, y]] ;
    . . . Initialize edge values in 'a' (boundary conditions) …
    float [[-,-]] b = new float [[x,y]],  r = new float [[x,y]];  // r = residuals
    do {
        Adlib.writeHalo(a);
        overall (i = x for 1 : N – 2)
            overall (j = y for 1 : N – 2) {
                float newA = 0.25 * (a[i - 1, j] + a[i + 1, j] + a[i, j - 1] + a[i, j + 1] );
                r [i, j] = Math.abs(newA – a [i, j]);
                b [i, j] = newA;
            }
            HPutil.copy(a, b);  // Jacobi relaxation.
     } while(Adlib.maxval(r) > EPS);
}
```

Visualization of Ghost Regions

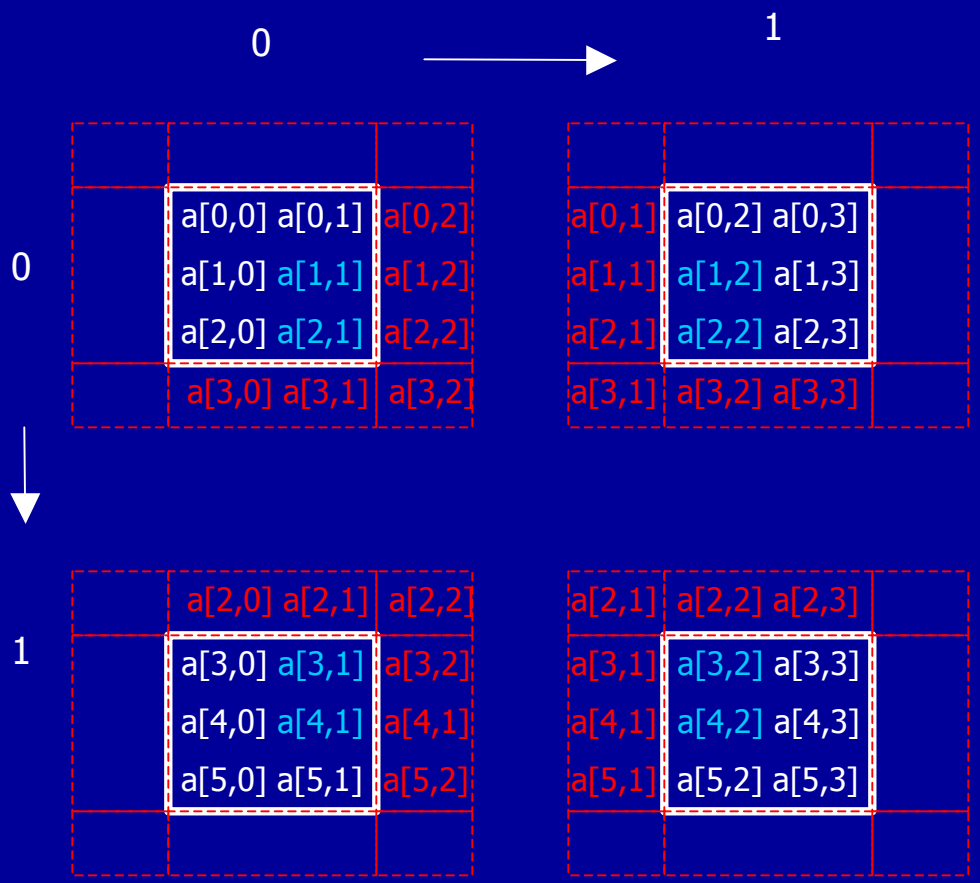
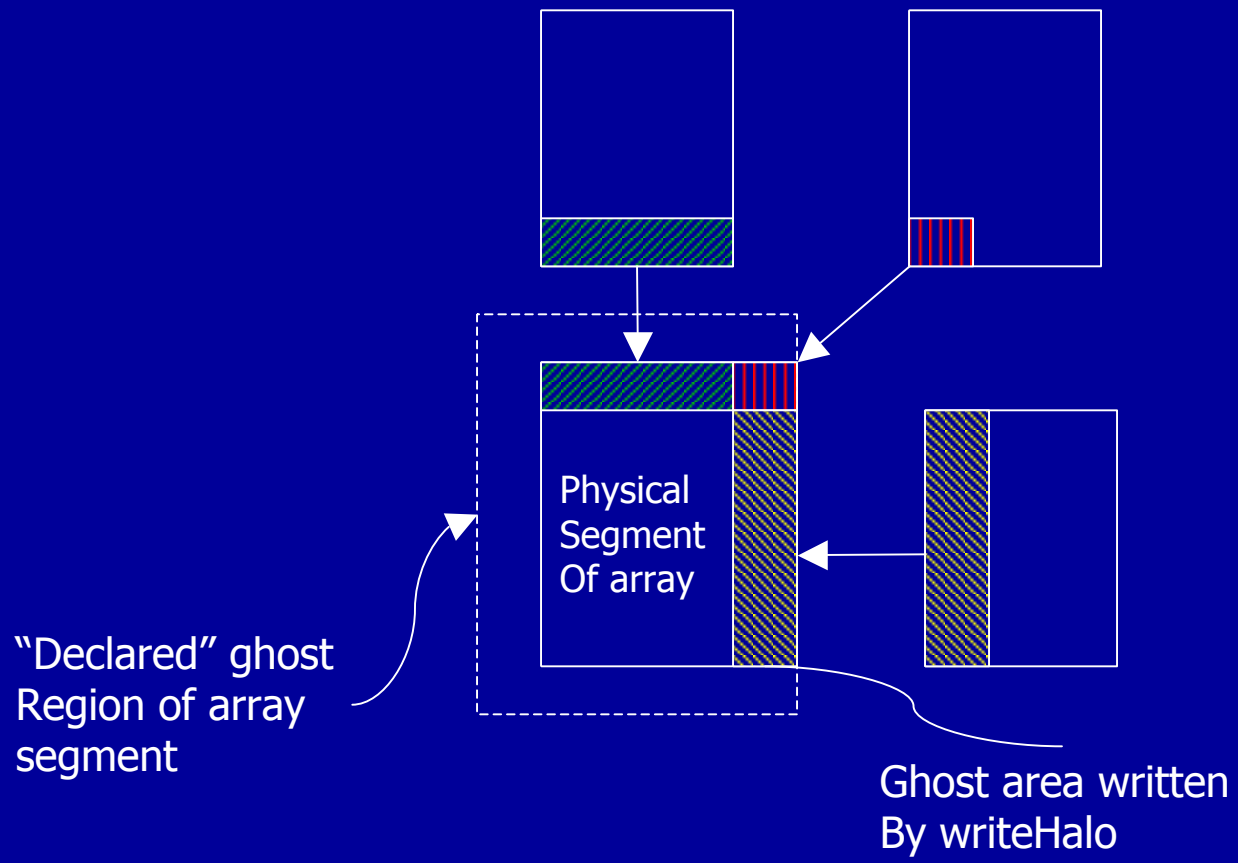


Illustration of the effect the **writeHalo()** function



Remarks

- Ghost regions are used for algorithms with some kind of local *stencil* update, involving neighbor elements in an array.
  - This certainly isn't the *only* kind of application of HPJava, but it is an important one.
- By using **ExtBlockRange** distribution format, arrays are allocated with extensions at the edge of the local block.
- One must *explicitly* call **Adlib.writeHalo()** to update these extra cells with corresponding values from neighbor processes, whenever those have been changed.
  - By default **Adlib.writeHalo()** fills *entire* ghost regions defined by ranges; previous slide illustrates a more general case: variants of writeHalo() fill a subset.
- The example also illustrates a quite different **Adlib** communication function called **Adlib.maxval()**.

Adlib

- Adlib is an application-level library for collective communications involving distributed arrays.
  - Something like a higher-level version of the collective communication functions in MPI.
- Current version exploits the features of HPJava, and is written in Java.
  - The HPJava language was specifically designed to allow this kind of "clean" design for libraries operating on distributed data.
- Historically:
  - Library called Adlib was completed at Syracuse University, based on earlier work at Southampton University.
  - Original version used C++ as an implementation language.
  - Initial emphasis was supporting translation of High Performance Fortran (HPF).
  - Used by two experimental HPF translators (SHPF, and "PCRC" HPF).

mpjdev I

- Meant for library developer.
- Application level communication libraries like Java version of Adlib (or potentially MPJ) can be implemented on top of mpjdev.
- API for mpjdev is small compared to MPI (only includes point-to-point communications)
  - Blocking mode (like **MPI_SEND**, **MPI_RECV**)
  - Non-blocking mode (like **MPI_ISEND**, **MPI_IRECV**)
- The sophisticated data types of MPI are omitted.
- Provide a flexible suit of operations for copying data to and from the buffer. (like gather- and scatter-style operations.)
  - Buffer handling has similarity to JDK 1.4 new I/O.

mpjdev II

- **mpjdev** could be implemented on top of Java sockets in a portable network implementation, or—on HPC platforms—through a JNI interface to a subset of MPI.
- Currently there are three different implementations.
  - The initial version was targeted to HPC platforms, through a JNI interface to a subset of MPI.
  - For SMPs, and for debugging on a single processor, we implemented a pure-Java, multithreaded version.
  - We also developed a more system-specific mpjdev built on the IBM SP system using LAPI.
- A Java sockets version which will provide a more portable network implementation and will be added in the future.

Overview of HPJava execution

- Download **hpjdk** from **www.hpjava.org**.
  - For complete installation instructions and language reference see **"Parallel Programming in HPJava"** from same location.
  - For full documentation of the *Adlib API* see the appendix of

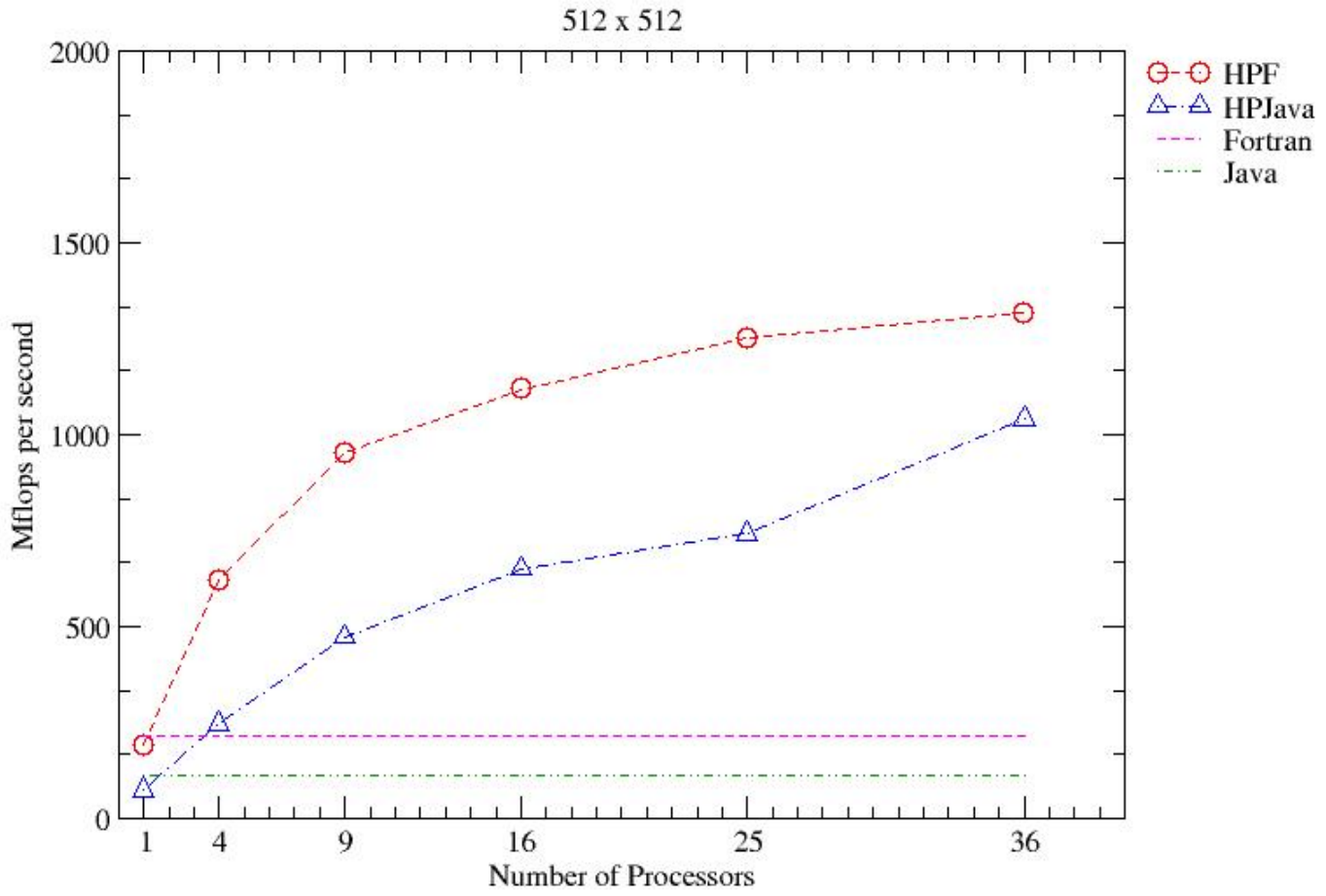
    "Platforms for HPJava: Runtime Support for Scalable Programming in Java", Sang Boem Lim, doctoral dissertation, http://grids.ucs.indiana.edu/ptliupages/publications
- Works by source-to-source translation from HPJava to standard Java, then compile to Java bytecode. To do both use the script **hpjavac**, e.g.

  $ hpjavac MyClass.hpj
- Can run parallel programs in a "multithreaded mode" directly by e.g.

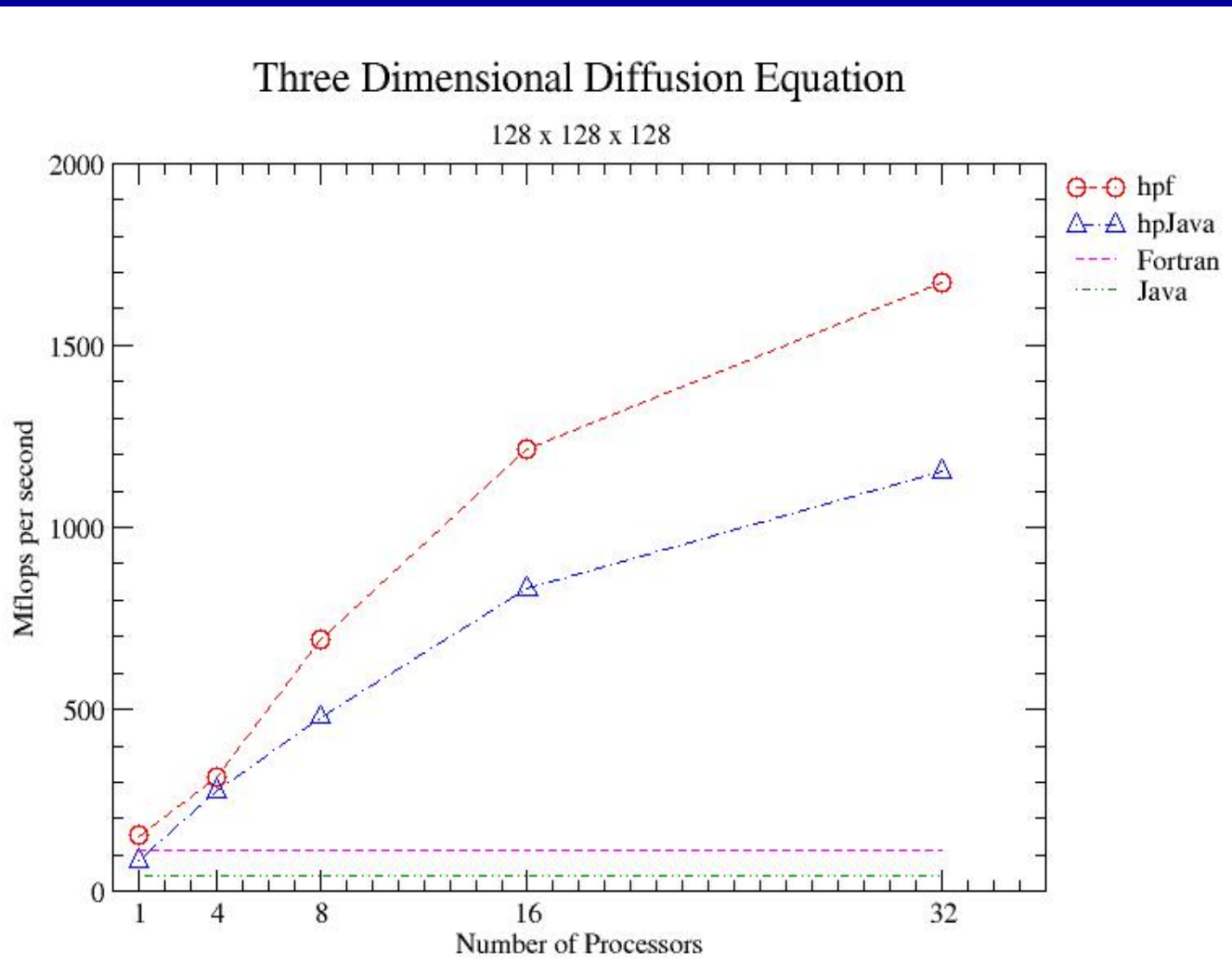
  $ java –Dhpjava.numthreads=4 MyClass
- To run bytecode on distributed collection of JVMs, first install a suitable version of mpiJava, then use e.g. **prunjava**, to start program.

Applications and Performance

- **System**: IBM SP3 supercomputing system with AIX 4.3.3 operating system and 42 nodes.
- **CPU**: A node has four processors (Power3 375 MHZ) and 2 gigabytes of shared memory.
- **Network MPI Setting**: Shared “css0” adapter with User Space (US) communication mode.
- **Java VM**: IBM’s JIT
- **Java Compiler**: IBM J2RE 1.3.1 with “-O” option.
- **HPF Compiler**: IBM xlhpf95 with “-qhot” and “-O3” options.
- **Fortran 95 Compiler**: IBM xlf95 with “-O5” option.

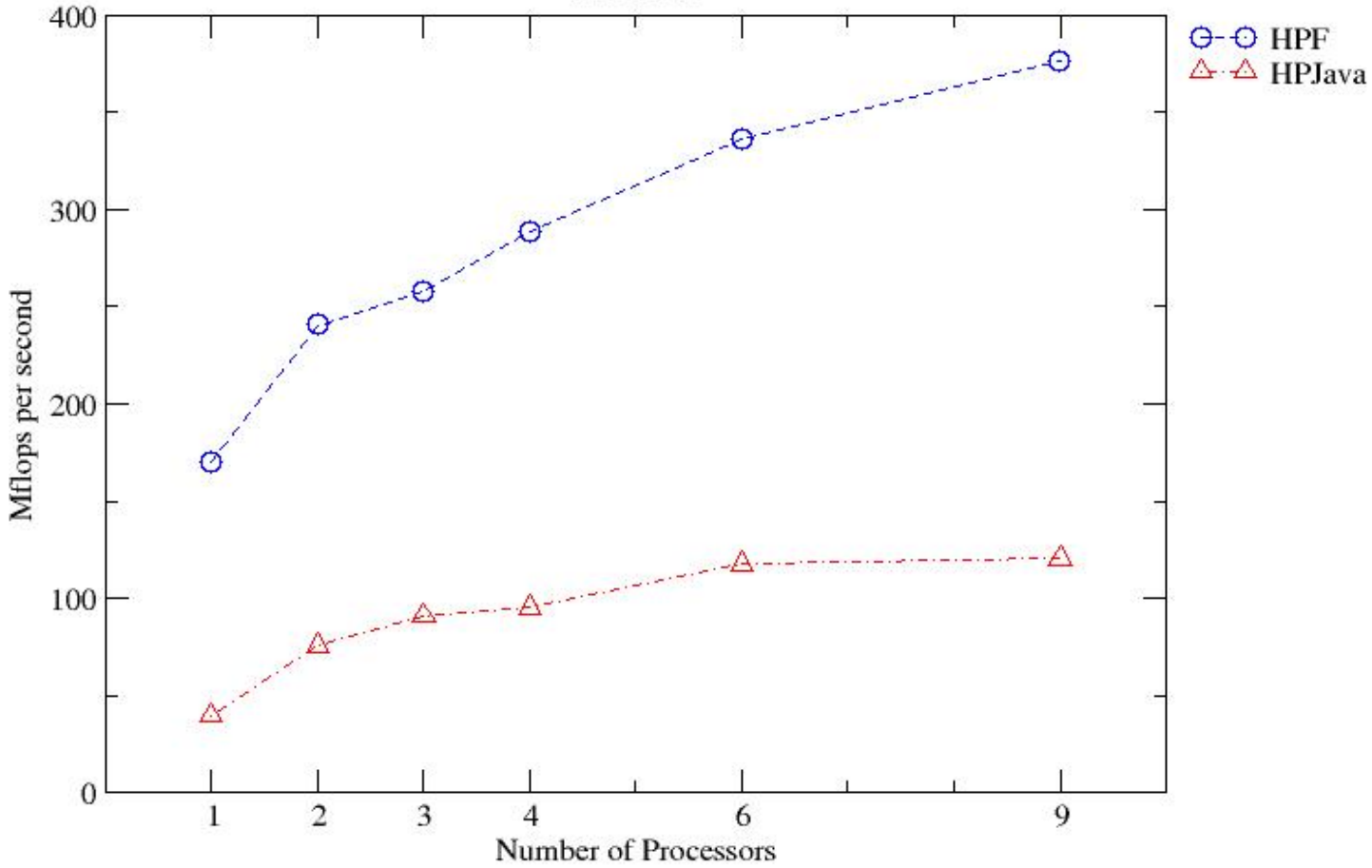


- HPJava can out-perform sequential Java by up to 17 times.
- On 36 processors HPJava can get about 79% of the performance of HPF.


Three Dimensional Diffusion Equation
128 x 128 x 128
2000
1500
1000
500
0
Mflops per second
1
4
8
16
32
Number of Processors
hpf
hpJava
Fortran
Java

Multigrid

- The multigrids method is a fast algorithm for solution of linear and nonlinear problems. It uses hierarchy grids with *restrict* and *interpolate* operations between current grids (*fine grid*) and restricted grids (*coarse grid*).
- General stratagem is:
  1. make the error smooth by performing a relaxation method.
  2. restricting a smoothed version of the error term to a coarse grid, computing a correction term on the coarse grid, then interpolating this correction back to the original fine grid.
  3. Perform some step of the relaxation method again to improve the original approximation to the solution.

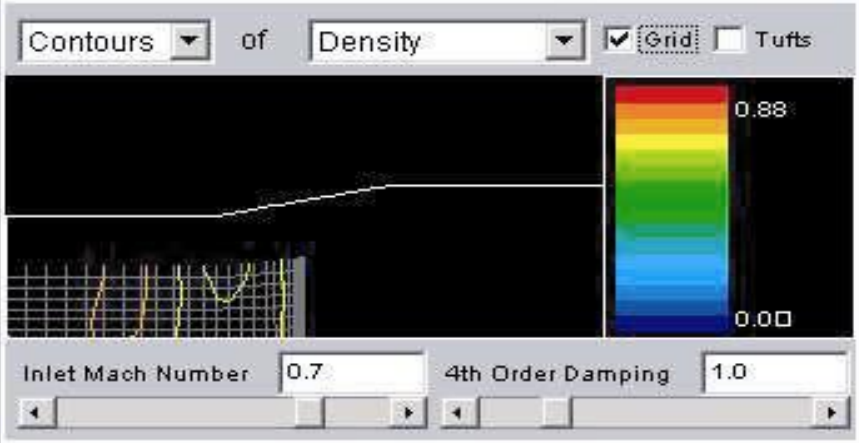


- Speedup is relatively modest. This seems to be due to the complex pattern of communication in this algorithm.

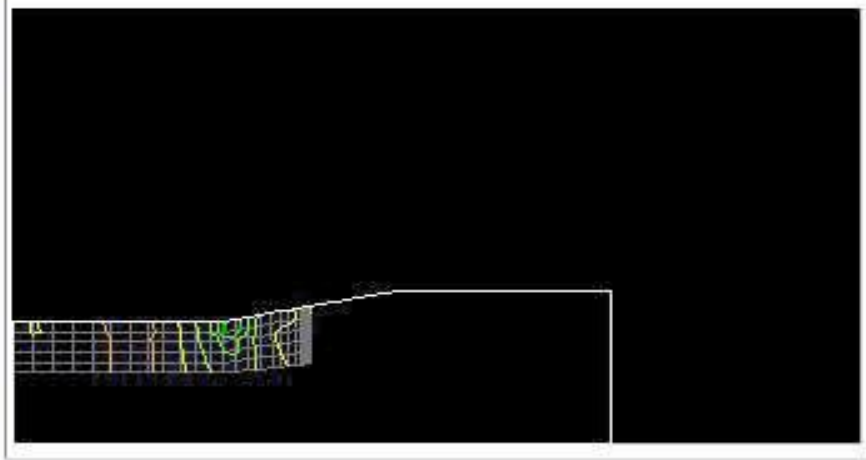
HPJava with GUI

- Illustrate how our HPJava can be used with a Java graphical user interface.
- The Java multithreaded implementation of mpjdev makes it possible for HPJava to cooperate with Java AWT.
- For test and demonstration of multithreaded version of mpjdev, We implemented computational fluid dynamics (CFD) code using HPJava.
- Illustrates usage of Java object in our communication library.
- You can view this demonstration and source code at **http://www.hpjava.org/demo.html**

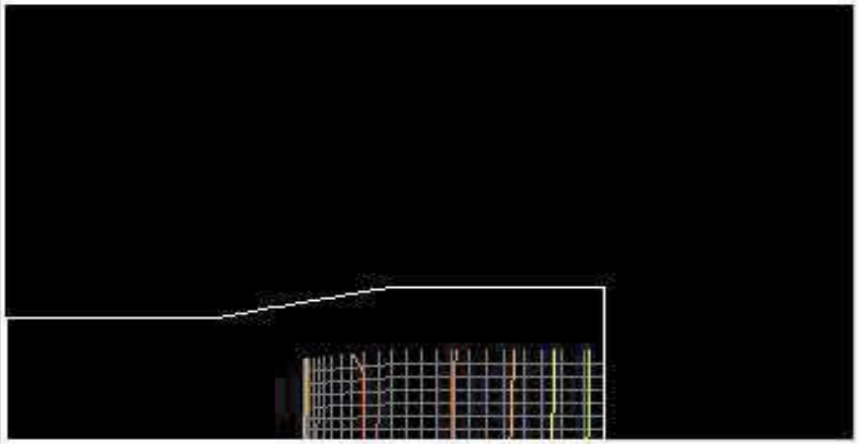
Applet 1



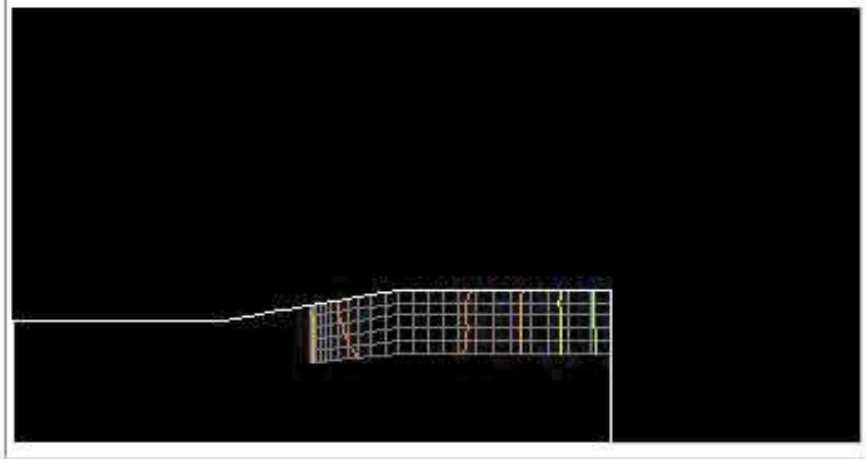
Applet 2



Applet 3



Applet 4



Related Systems

- Co-Array Fortran – Extension to Fortran95 for SPMD parallel processing
- ZPL – Array programming language
- Jade – Parallel object programming in Java
- Timber – Java-based programming language for array-parallel programming
- Titanium – Java-based language for parallel computing
- HPJava – Pure Java implementation, data parallel language and explicit SPMD programming