Selected Notes on *HPJava*

Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox
Northeast Parallel Architectures Centre,
Syracuse University,
Syracuse, New York

December 16, 1996

Contents

1 Java and Distributed Simulation. *Geoffrey Fox, email dated 6 Aug 1996* **4**

2 HPJava: Work in progress. *Bryan Carpenter, Yuh-Jye Chang, August 9, 1996* **6**
- 2.1 Communication infra-structure: Channels . . . 6
  - 2.1.1 The `Port` class . . . 7
  - 2.1.2 The `RPC` class . . . 8
  - 2.1.3 The `Child` class . . . 9
  - 2.1.4 Stream interfaces . . . 10
  - 2.1.5 Non-deterministic communications . . . 10
  - 2.1.6 Limitations of the prototype . . . 11
- 2.2 Parallel arrays and data parallel computation . . . 12
- 2.3 Applications . . . 13

3 Some Remarks on Parallel Processing and Java. *Geoffrey Fox, email dated 6 Aug 1996* **19**

4 HPJava Suggestions. *Bryan Carpenter, email dated 5 Aug 1996* *22*

5 HPJava: Work in progress. *Bryan Carpenter, Yuh-Jye Chang, November 12, 1996* **29**
- 5.1 Parallel arrays and data parallel computation . . . 29
  - 5.1.1 Example data parallel Java program . . . 29
  - 5.1.2 Starting an HPJ program . . . 32
  - 5.1.3 Outlook on data-parallel programming in Java . . . 32
- 5.2 Channels . . . 32
  - 5.2.1 Overview of the channel model . . . 33
  - 5.2.2 Outlook on channel communication in Java . . . 33

6 "HPJava": Demo Description. *Bryan Carpenter, Yuh-Jye Chang, November 16, 1996* **34**
- 6.1 Conway's Game of Life . . . 34
- 6.2 2D FFT "Image Compression" . . . 36

7 MPI Java Wrapper Implementation. *Yuh-Jye Chang, December 16, 1996* **37**
- 7.1 MPI Java wrapper introduction . . . 37
- 7.2 MPI Java wrapper design . . . 37
- 7.3 Java classes for MPI . . . 38
- 7.4 Class methods for Java MPI . . . 38
- 7.5 Java native method . . . 39
- 7.6 Java datatypes . . . 41

7.7 Problems due to strong typing and no pointer 42
7.8 The polymorphism of Java Datatype class 42
7.9 Other problems 43
7.10 Conclusion 43
7.11 Test example 43
7.12 Execution result 45
7.13 List of detailed Java wrapper for MPI 45

1 Java and Distributed Simulation. *Geoffrey Fox, email dated 6 Aug 1996*

Distributed computing is a powerful tool in many high performance applications but it is particularly relevant when the underlying problem is itself distributed. The best known examples come from tactical military simulations as implemented in SIMNET and successor programs such as DSI—Distributed Simulation Internet. However there are many other distributed organizations and/or applications which can naturally use this paradigm—in the commercial arena, finance and interdisciplinary (inter-organization) manufacturing are two examples.

Distributed simulation is characterized by some key features—geographically distributed but relatively loosely coupled components. These components can be pure simulation modules but can also include "people in the loop" or active sensors and instruments ("Machines in the loop"). In the military case, one has linked simulated vehicles (say aircraft and tanks) with real vehicles in the field and with military commanders and pilots/tank drivers all in the same simulation.

We suggest that the Web and Java are very suitable for such applications. The Web itself naturally supports a distributed set of loosely coupled applications coordinated by linked web servers. The session management function of current collaborative tools needs to be generalized to support the event driven simulation paradigm underlying typical distributed simulation applications. VRML provides the natural client technology to support simulated virtual environments for "people in the loop"—here we use the latest VRML 2.0 enhancements to provide dynamic behavior. Web-linked databases and an overall Dataflow (WebFlow) computing model provide the necessary high level information and computing services.

Java is the natural implementation language at all levels of such a distributed computing environment. Some arguments for the use or Java are quite general and reflect my prejudice that Java will become a dominant general purpose language. The "web-ready" nature of Java and its excellent support for graphics output are some such general features. Looking at particular capabilities, event driven simulations are usually built as a set of objects interacting via messages. This appears to be naturally implemented using Java's object-oriented features. Java (as in MIT's Jigsaw and NCSA's Habanero) is getting used in a growing number of Web server and collaboration systems. This illustrates that Java will be excellent base control software for a distributed simulation environment. Note that we expect that Java will soon be supported in three distinct but coexisting modes with different performance-functionality tradeoffs. There is the current semi-interpreted (Universal bytecode) Applet model; JavaScript (being integrated with Java by Netscape) and the VRMLScript discussions illustrate that a fully interpreted version of Java will be supported. This is essential

in many applications. Finally there will be native Java compilers which will give Java (native) classes with performance competitive with C and FORTRAN code. Distributed simulation needs all three modes in different parts of its implementation.

Java and the Web automatically build in support for the coordination of loosely coupled simulation modules which is the major form of parallelism needed in most distributed simulations. However one can expect to need data parallel execution of some tightly coupled components. For example, these could be an airflow simulation component in a distributed manufacturing system or a military weather simulation or image processing application. We have finished a preliminary study of data parallelism in Java and believe that techniques developed in HPF and HPC++ can be adapted for Java. Interpreted Java front ends can invoke high performance Java wrapper classes which link to native FORTRAN (HPF) or compiled Java programs.

2 HPJava: Work in progress. *Bryan Carpenter, Yuh-Jye Chang, August 9, 1996*

Our present emphasis is on providing class libraries to facilitate communication and parallel computation within the framework of Java. For now this means communicating Java *applications*, but soon we will provide class libraries which allow applets downloaded from a common server to communicate (in the short term, by having their server transparently through-route communications).

We are implementing (and have already prototyped) a class library for channel communication layered on the Java socket interface.

It may also be useful to have a Java interface to MPI (by which we mean some class library that allows a Java program to communicate with a C or Fortran program which itself uses the standard MPI interface). This could be implemented by "native methods" wrappers around some pre-existing MPI implementation. In the view of one of us (DBC) this will only be particularly useful if the underlying MPI implementation supports dynamic spawning of new processes.

We are also considering various options for implementing parallel arrays and collective communication within Java. The naive first idea is to simply extend the C++ STL approach to provide parallel arrays. This now looks quite difficult because of the limitations of the Java language [Java jettisoned most of the interesting features of C++ that make STL work...]. We are presently considering a lower level approach, closer in spirit to the nascent PCRC interface for Fortran. This approach looks more practical to realize in Java, and may still be useful to the application programmer.

2.1 Communication infra-structure: Channels

We are developing a Java class library for channel communication. The semantics are modelled on the dynamic channels of Fortran M.

This model is particularly attractive in Java because

1. It is "connection-oriented" so ordinary data communications map directly onto the socket I/O primitives efficiently implemented for Java and the Web.

2. It directly supports highly dynamic situations where new remote processes (eg, applets) come into existence unpredictably, and may have unpredictable communication demands (eg, determined by requirements of a remote user).

3. It provides a layer of abstraction above internet addresses and socket port numbers, and associated resources to which access may be limited by Java security models.

The remainder of this section describes our prototype interface. There is a complete example at the end of this report.

2.1.1 The `Port` class

Access to logical channels is provided through a Java class called a `Port`. The `Port` dresses up and adds functionality to an internet port, but should not be confused with this lower-level idea. The `Port` interface does not provide direct access to port numbers or internet addresses.

In the existing prototype the public interface to the `Port` class is

```
public class Port {

  // Constructor.

  public Port() ;

  // Channel creation.

  public static synchronized void channel(Port P, Port Q) ;

  // Public members for communication.

  public synchronized void send(byte data []) ;
  public synchronized int recv(byte data []) ;
  public synchronized byte [] recv() ;

  public synchronized void sendInt(int data) ;
  public synchronized int recvInt() ;

  public synchronized void sendChan(Port S) ;
  public synchronized void recvChan(Port R) ;
}
```

The constructor `Port.Port()` creates an unconnected port.

The static member `Port.channel(Port, Port)` connects two current unconnected `Port`s. [These are `Port`s "inside the current program", so to speak. The `Port` itself is purely a local entity.] Logically, it creates a channel within the local processor, and binds the two ends of the channel to the two ports.

A channel supports bi-directional communication (with unbounded buffering). `send(byte [])` and `recv(byte [])` send and receive messages. A message is an atomic entity. If an `n`-byte vector is output on one end of the channel with `send`, the `recv` operation at the other end must be prepared to accept an `n`-byte vector. `revc()` is a variant of `recv(byte [])` which allocates space for the received message internally, and returns it to the caller.

`sendInt` and `recvInt` are provided for convenience. They are simply wrappers on `send(byte [])` and `recv(byte [])`.

The more interesting operations are `sendChan(Port)` and `recvChan(Port)`. The argument of `sendChan` should be a connected port—a port with a bound channel-end. The argument of `recvChan` should be an unconnected port. The effect of a `sendChan(P)`/`recvChan(Q)` communication is to transfer the channel-end, leaving `P` unconnected, and the channel-end bound to `Q`. This may involve transferring the channel-end between different processors, if the communication channel connects different processors.

In the prototype implementation every Port has an associated `ServerSocket` and an associated input handler thread. The use of a server socket restricts use of `Port`s to Java *applications*. The implementation is fairly straightforward, but requires some care in the protocol for notifying a Port when its peer Port (the Port connected to the other end of its channel) is in motion, and in ensuring that input data currently buffered in a port is physically communicated at the same time as the channel-end is logically transferred.

It should be possible to provide alternate (probably less efficient) implementations that can be used within applets.

2.1.2 The `RPC` class

The `RPC` class supports remote procedure calls. Its current interface is

```
public class RPC {
  public static void call(String childClass, String remoteHost,
                          Port toChild) ;

  public static void spawn(String childClass, String remoteHost,
                           Port toChild) ;
}
```

The `call` operation invokes a Java application on a remote processor, and sends a channel-end to it. `childClass` should be the name of a class derived from `Child`, with a `main` method. `remoteHost` should be the name of a host. `toChild` should be a port with a bound channel end. The `call` operation terminates when the remote call to `main` terminates. When the call completes, `toChild` is unconnected (the remote application is responsible for disposal of the channel-end it was initially sent).

`spawn` is similar, but invokes `call` in a new thread, so it returns immediately.

Typical usage would be

```
// Create output channel

Port R = new Port() ;
Port S = new Port() ;

Port.channel(R, S) ;

// Spawn remote 'Child' application
```

```
RPC.spawn("EgChild", "koum", S) ;

... communicate with child through 'R'.
```

The current implementation of `RPC.call` uses the Java `exec` mechanism to run the command

```
rsh remoteHost java childClass
```

The remote host is a processor on which

1. A remote shell can be executed
2. The Java interpretter `java` is installed
3. The byte-code for `childClass` is accessible through the CLASSPATH environment variable.

`call` transfers the channel-end through negotiations performed over the standard input and output streams of the remote process. Subsequent output from the child process is forwarded to the current process' standard output stream.

2.1.3 The `Child` class

A new application spawned by the `RPC` class should be derived from the class `Child`, which has interface

```
class Child {
  static Port initial ;
}
```

Typical usage would be

```
public class EgChild extends Child {
  public static void main(String args[]) {

    ... communicate with parent through 'initial'.
  }
}
```

The current implementation of the `Child` includes static initialization code which receives a channel-end from the `RPC` parent and binds it to `initial`. The negotiations involved are performed through the standard input and output streams of the child thread.

2.1.4 Stream interfaces

The classes

```
public class PortInputStream extends InputStream {

  // Constructor.

  public PortInputStream(Port P) ;

  // Methods.

  public synchronized int read() ;
  public synchronized int read(byte b [], int off, int len) ;

  [etc...]
}
```

and

```
public class PortOutputStream extends OutputStream {
  Port P ;
  ByteArrayOutputStream strm ;

  // Constructor.

  public PortOutputStream(Port P) ;

  // Methods.

  public synchronized void write(int b) ;
  public synchronized void write(byte b [], int off, int len) ;

  public synchronized void flush() ;
}
```

provide a convenient interface for Java I/O on ports. `DataInputStream` and `DataOutputStream` wrappers can be created around these streams, making the standard Java read and write operations available.

A `PortOutputStream` buffers data written to it until a `flush` operation is performed. At this point a `send` operation on the Port is executed and the buffered data is output as a single message.

A `PortInputStream` contains a buffer which is replenished by executing a `recv` on the Port whenever the buffer is empty and a a read operation is performed on the stream.

2.1.5 Non-deterministic communications

A functionality similar to the "merge" in Fortran-M is provided through the "merge pool" class, `Merge`. The public interface is

```
public class Merge {
  public Merge() ;

  public void add(Port P) ;
  public void rem(Port P) ;

  public Port select() ;
}
```

When initially created, a merge pool is empty. Channel ends are added to or removed from the pool by passing the associated ports to the `add` and `rem` members.

The only other operation on a merge pool is `select`. This returns a port from the pool which presently has input data ready. If no port has a message ready when the call is executed `select` blocks until a message arrives.

Merge pools enable non-deterministic patterns of communication.

Example usage

```
Merge pool = new Merge() ;

for(int i = 0 ; i < NNODES ; i++) {

  // ... create a slave process which returns data on port 'U'

  pool.add(U) ;
}

while(nodesActive) {
  Port V = pool.select() ;

  DataInputStream fromNode = new DataInputStream(new PortInputStream(V)) ;

  // ... read data from from slave
}
```

Note that the port returned by `select` remains in the pool. Ports can only be removed from the pool by using `rem`.

In the present implementation a channel end can belong to at most one merge pool at any given time.

2.1.6 Limitations of the prototype

1. Only Java *applications* can use Ports because a Port incorporates a `ServerSocket`. It should be possible to devise a (less efficient) implementation which works for applets, sending messages indirectly via the applet's server.

2.2 Parallel arrays and data parallel computation

After some preliminary investigation of possible interfaces for parallel container classes in Java, we have a rather cautious view about the opportunities offered by this approach.

The standard Java library provides a few container classes (`Vector`, `Dictionary`, `Hashtable`...). Because Java doesn't provide templates, these are implemented as containers for the `Object` base class. This involves forsaking the safety of compile-time checking, and incurring the inefficiency of run-time checking of type-casts.

In the majority of HP applications the arrays required will have elements of primitive types (int, float, double,...). So these will have to wrapped up as objects. This introduces the further inefficiency and inconvenience of allocating the wrapper objects for all array elements, and accessing the data through methods on the wrappers.

Of course it is not possible to provide iterator classes in the STL sense for Java container classes, due to the lack of pointers and operator overloading.

For these reasons, in the present state of development of Java, it may not be productive to devote effort to designing parallel container classes [maybe eventually Java will evolve into C++, and this will become a better option...]. The interface is likely to be so clumsy and inefficient that it would be unattractive to application programmers, or as a target for compilation.

Presently we are focussing on providing a set of lower level "helper" classes, to facilitate writing data parallel programs. Ultimately these classes could be incorporated as part of the implementation of container classes, if somebody comes up with a more viable framework for implementing those.

The kind of helper classes we have in mind describe, for example, process grids and distributed index ranges.

Our interface is in its early stages, but we will illustrate the general approach by an example. Consider the HPF program

```
!HPF$ PROCESSORS p(4, 4)
      REAL a(0 : 99, 0 : 99)
!HPF$ DISTRIBUTE a(BLOCK, BLOCK) ONTO p


      FORALL(i = 0 : 99, j = 0 : 99)
        a(i, j) = i + j
```

A possible Java rendition is

```
Procs p = new Procs(4, 4) ;

Range x = new Range(100, BLOCK, p, 1) ;
Range y = new Range(100, BLOCK, p, 2) ;

float [] [] a = new float [x.seg()] [y.seg()] ;
```

```
for(x.forall() ; x.next() ; x.test())
  for(y.forall() ; y.next() ; y.test())
    a [x.sub()] [y.sub()] = x.idx() + y.idx() ;
```

Here `Procs` is a class describing a process grid, and `Range` is a class for describing an index range distributed over a particular grid dimension. The `Range` members are

```
int seg() ;
```

which returns the size of a local array segment (25 in this example),

```
void forall() ;
boolean test() ;
void next() ;
```

which enumerate the local segment of the index range,

```
int sub() ;
```

which returns the local subscript for the current iteration and

```
int idx() ;
```

which returns the global index for the current iteration.

This example is for illustration only. The concrete interface to the classes is bound to be changed and extended.

Once the general scheme for representing and accessing distributed arrays is in place, we can define the interfaces to high level collective communications (shift, transpose, broadcast, reduce, ...). These could, for example, be implemented on top of the channels interface described in the previous section.

2.3 Applications

We need some applications. Don Leskiw has suggested a target-tracking application which combines data and task parallelism with opportunities for applet front ends. Other numerical applications of Java, suitable for parallelism, are needed.

As a trivial example of the use of our channels interface, I have ported one of the examples from the Cornell meta-computing seminar to Java. This outputs its results to files which can be viewed with xv. It should be given an applet interface. The source is included below.

```
--------------------------- Life.java ------------------------------

import java.io.* ;
```

```
public class Life {
  static final int N      = 64 ;
  static final int NITER  = 50 ;

  static final String [] hosts = {"koum", "naos"} ;
  static final int NNODES = hosts.length ;

  public static void main(String args[]) {
    try {
      DataInputStream fromNode [] = new DataInputStream [NNODES] ;

      // Create a ring of channels

      Port R [] = new Port [NNODES] ;
      Port S [] = new Port [NNODES] ;

      for(int i = 0 ; i < NNODES ; i++) {
        R [i] = new Port() ;
        S [i] = new Port() ;
      }

      for(int i = 0 ; i < NNODES ; i++)
        Port.channel(R [i], S [(i + 1) % NNODES]) ;

      // Create the node processes.

      for(int i = 0 ; i < NNODES ; i++) {
        Port U = new Port() ;
        Port V = new Port() ;
        Port.channel(U, V) ;

        RPC.spawn("LifeWorker", hosts [i], V) ;

        // Send parameters to node

        U.sendInt(N) ;
        U.sendInt(NITER) ;

        U.sendInt(NNODES) ;
        U.sendInt(i) ;

        U.sendChan(R [i]) ;
        U.sendChan(S [i]) ;

        fromNode [i] = new DataInputStream(new PortInputStream(U)) ;
      }
```

```
      /*  Handle output.  */

      for(int iter = 0 ; iter <= NITER ; iter++) {

        /*  Copy current state of board from nodes to a `.pgm' file.  */

        String fname = "life" + iter / 10 + iter % 10 + ".pgm" ;
        PrintStream out = new PrintStream(new FileOutputStream(fname)) ;

        out.println("P2") ;
        out.println("" + N + "  " + N) ;
        out.println("" + 1) ;

        for(int node_id = 0 ; node_id < NNODES ; node_id++) {
          DataInputStream in = fromNode [node_id] ;

          int blockLen = in.readInt() ;

          for(int i = 0 ; i < blockLen ; i++) {
            out.print("" + in.readInt() + "  ") ;
            if(i % 20 == 19) out.println("") ;
          }
        }
      }
    }
    catch(Exception e) {
      ;
    }
  }
}
```

---------- LifeWorker.java ----------

```
import java.io.* ;


public class LifeWorker extends Child {
  public static void main(String args[]) {
    try {
      Port R = new Port() ;
      Port S = new Port() ;

      // Get parameters from host

      int N       = initial.recvInt() ;
```

```
int NITER   = initial.recvInt() ;

int NNODES  = initial.recvInt() ;
int node_id = initial.recvInt() ;

initial.recvChan(R) ;
initial.recvChan(S) ;

// Create port streams

DataOutputStream toHost =
    new DataOutputStream(new PortOutputStream(initial)) ;

DataOutputStream toNodePrev =
    new DataOutputStream(new PortOutputStream(R)) ;
DataInputStream fromNodePrev =
    new DataInputStream(new PortInputStream(R)) ;

DataOutputStream  toNodeNext =
    new DataOutputStream(new PortOutputStream(S)) ;
DataInputStream fromNodeNext =
    new DataInputStream(new PortInputStream(S)) ;

// Define block

int blockSizeMax = (N + NNODES - 1) / NNODES ;
int blockBase    = blockSizeMax * node_id ;

int blockSize ;
if(blockBase + blockSizeMax > N)
  blockSize = N - blockBase ;
else
  blockSize = blockSizeMax ;

// 'block' has 'blockSize + 2' columns.  This allows for ghost cells.

int block [] [] = new int [blockSize + 2] [N] ;

for(int i = 0 ; i < blockSize ; i++) {
  int ib = i + 1 ;
  for(int y = 0 ; y < N ; y++) {
    int x = blockBase + i ;
    if(x == N / 2 || y == N / 2)
      block [ib] [y] = 1 ;
    else
      block [ib] [y] = 0 ;
  }
```

```
}

// Dump initial state of board to host

toHost.writeInt(blockSize * N) ;
for(int i = 0 ; i < blockSize ; i++) {
  int ib = i + 1 ;
  for(int y = 0 ; y < N ; y++)
    toHost.writeInt(block [ib] [y]) ;
}
toHost.flush() ;

// Main update loop.

int neighbours [] [] = new int [blockSize] [N] ;

for(int iter = 0 ; iter < NITER ; iter++) {

  // Shift this block's upper edge into next neighbour's lower ghost edge

  for(int y = 0 ; y < N ; y++)
    toNodeNext.writeInt(block [blockSize] [y]) ;
  toNodeNext.flush() ;

  for(int y = 0 ; y < N ; y++)
    block [0] [y] = fromNodePrev.readInt() ;

  // Shift this block's lower edge into prev neighbour's upper ghost edge

  for(int y = 0 ; y < N ; y++)
    toNodePrev.writeInt(block [1] [y]) ;
  toNodePrev.flush() ;

  for(int y = 0 ; y < N ; y++)
    block [blockSize + 1] [y] = fromNodeNext.readInt() ;

  /*  Calculate a block of neighbour sums.  */

  for(int i = 0 ; i < blockSize ; i++) {
    int ib = i + 1 ;
    for(int y = 0 ; y < N ; y++) {
      int y_n = (y - 1 + N) % N ;
      int y_p = (y + 1) % N ;

      neighbours [i] [y] =
        block [ib - 1] [y_n] + block [ib - 1] [y] + block [ib - 1] [y_p] +
        block [ib]     [y_n] +                      block [ib]     [y_p] +
```

```
            block [ib + 1] [y_n] + block [ib + 1] [y] + block [ib + 1] [y_p] ;
          }
        }

        /*  Update block of board values.  */

        for(int i = 0 ; i < blockSize ; i++) {
          int ib = i + 1 ;
          for(int y = 0 ; y < N ; y++) {
            int neighbour = neighbours [i] [y] ;

            if(neighbour < 2 || neighbour > 3)
              block [ib] [y] = 0 ;
            if(neighbour == 3)
              block [ib] [y] = 1 ;
          }
        }

        /*  Dump current state of board to host */

        toHost.writeInt(blockSize * N) ;
        for(int i = 0 ; i < blockSize ; i++) {
          int ib = i + 1 ;
          for(int y = 0 ; y < N ; y++)
            toHost.writeInt(block [ib] [y]) ;
        }
        toHost.flush() ;
      }
    }
    catch(Exception e) {
      ;
    }
  }
}
```

3 Some Remarks on Parallel Processing and Java. *Geoffrey Fox, email dated 6 Aug 1996*

We can distinguish (at least) three forms of parallelism (concurrency) in Java of which the first two are reasonably uncontroversial.

a) Fine grain functional parallelism as exhibited by the built-in threads of Java. These could be very helpful in latency hiding by allowing several concurrent processes on a single node but do not naturally implement large scale parallelism.

b) Coarse grain functional or task parallelism or what the Linda group and Jim Browne would call coordination. This is roughly what is implemented in the Applet and network connection mechanisms of Java. This capability is the basis of WebFlow our proposed dataflow mechanism on the Web. Note that threads are shared memory but Applet mechanism is distributed memory parallelism.

c) Data parallelism is less clear for both technical and emotional reasons (Is it in the "spirit" of Java!). Let us discuss this in more detail.

In general, it seem plausible that data parallelism in Java should build on the corresponding discussions in FORTRAN and C++ (HPF and HPC++). Most relevant Java features are seen in one or both of these languages. In the following we list some considerations to be borne in mind in considering data parallelism in Java.

1. Data parallel FORTRAN or C++ typically compiles down to FORTRAN or C plus message passing. We note that the Java plus message passing (data parallel) model is uncontroversial. Thus there is no problem in defining the target implementation of a data parallel Java application. Further the Java equivalents of Fortran-M and C++ can be naturally defined.

2. The "Java plus message passing model" includes the case where "Java" immediately invokes a native class which could be an existing compiled C, Fortran or C++ and even an optimized Java code compiled directly for the native machine. Some argue that use of such (non-portable) native classes violates the philosophy of the Web or of Java. I disagree. I at least download C code very often from the Web and current versions of Netscape illustrate how one is happy to download either the browser or plugins. We propose that users will be willing to download once and for all, a set of high performance Engineering and Science native classes. This implies that PCRC (Parallel Compiler Runtime Consortium) compiler runtime should be included in such a library and I expect this to be a critical part of any high performance Java environment.

3. The most powerful model assumes a WebServer (as opposed to a client) attached to each process in our "Java/Native Classes + Message Passing" model. This approach allows natural integration of Web computing as we have demonstrated in Kivanc Dincer's "HPF on the Web" prototype supporting Pablo performance and scientific result visualization from data passed by Java process to associated WebServer. Note that our standard "WebWindows" philosophy implies such a linked se of servers to coordinate computation.

4. Any efficient implementation must use "simple types" and not "objects" for distributed arrays as objects come with too much overhead. However we can make use of objects as a wrapper which stores at a high level overall information about array and links via intrinsic "methods" to the high performance native classes. Such a wrapper class does more than support data parallelism. It allows a general and convenient Java interface to existing C and Fortran data structures. This will allow easier development of Java based interfaces to existing simulations.

5. We suggest implementing data parallel Java using the HPF Interpreter approach we explored with Arpa funding and demonstrated (the work of Furmanski) in Supercomputing 93. The essential idea between the HPF Interpreter is simple. Take any Fortran90/HPF instruction such as:

   ```
   A = MATMUL(B,C)
   ```

   This can be executed in interpreted fashion without significant overhead as we are only concerned with cases that `A`, `B`, `C` are large arrays and time to interpret the single coarse grain array statement is small compared to its execution time even when interpreter invokes optimized parallel execution. Note that for MIMD parallelism, we imply large grain size in each process. Furmanski's HPF Interpreter was successful but we left it as a prototype as we did not have the resources necessary to complete a full blown system. Now Java and the Web have given us a more natural and powerful implementation and further our PCRC HPF infrastructure is much better.

6. We can implement the proposed data parallel Java as a main (host) class interpreting coarse grain statements linked to a set of child (native) distributed processes. This looks pictorially like:

   ```
   Main (host)                  Interpreted HPJava statements
   Java Class                   manipulating
                                Wrapper HPVector classes

   Set of Child                 Web Server running
   (Native) distributed         a Java Interpreter invoking a
   Processes                    Highly efficient ''node'' code
   ```

```
which is compiled Java, C and Fortran
using PCRC and MPI libraries etc.
```

7. We are suggesting this new `HPVector` class which is a data parallel array (and similarly for other parallel data structures). The `HPVector` class (of which `A`, `B`, `C` in 5åre instances) does not necessarily store array elements but rather user accesses elements through methods such as `A.grabelement(i1,i2)` to return `A(i1)` through `A(i2)`. We view `HPVector` class as a wrapper which links Java to an array in any relevant code including Java itself, F77, HPF, HPC++, F77 + Message Passing etc.

8. Wrapper `HPVector` methods will include `A.distribute()` and `A.align()` to implement HPF directives as calls to methods.

9. forall statements are very popular and powerful in HPF but are not so trivially implemented in our formalism as they involve array elements and not arrays. One possibility is to view a forall as implementing a new HPF array function in a flexible way and treat forall statement as a script which implements this new function. Thus something like:

```
forall(I=1 to 100)
        a(I)=b(I)*b(I+1)/c(I)
```

   could be written as:

```
A=HPVector.forall("forall(I=1 to 100);
                   a(I)=b(I)*b(I+1)/c(I)",B,C);
```

10. The implementation of independent DO loops is also not so clear as really these reflect control and not data parallelism. Perhaps these should be implemented through task parallel (coordination) mechanism in Java.

11. Interesting features of this approach include the fact that no new language extensions are required (although you could add forall to language); it allows a (slow) pure Java sequential version as well as optimized parallel versions. It allows one to build both Java wrappers to existing applications and new parallel Java applications in the same formalism.

12. The main (host) class is naturally fully interpreted and the use of something like JavaScript (when it has been integrated with Java) is particularly natural.

4 HPJava Suggestions. *Bryan Carpenter, email dated 5 Aug 1996*

For debate, some suggestions for a Java syntax (ie, class library interface).

I'm trying to incorporate the basic idea of everything being collective operations on array objects. I haven't defined FORALL "scripts", but the expression trees suggested here could be regarded as a sort of pre-parsed script. The advantage is that we don't need parsers and symbol tables in the run-time. The disadvantage is that the syntax is quite awkward. This is really a peculiarity of Java (see comments below).

One intention with this interface is that it is sufficiently high-level that it can be implemented in terms of a SIMD-style Java controller for an HPC back-end, or of a pure SPMD parallel Java model.

I'm using my "Range" structures because they allow a simple definition of "conformance", and they are useful in indexed array expressions, which go some way to replacing FORALL.

Unlike HPF, the notation defined here forces communication operations to be explicitly specified. In that sense it is lower level than HPF—closer to a run-time library.

The syntax is quite awkward in practise. A big problem is still that Java doesn't allow library-defined operator overloading, so expressions quickly become difficult to read.

Index ranges

- An index range is specified by a member of the 'Range' class.
- A "primitive range" is either collapsed (sequential), or distributed over a particular dimension of a particular process array.

  ```
  Range x = new Range(100) ;
  Range y = new Range(50, BLK, p.dim(0)) ;
  Range z = new Range(50, BLK, p.dim(1)) ;
  ```

  `x` is collapsed. `y` is distributed blockwise over the first dimension of process array `p`, `z` over the second.
- Two distributed primitive ranges are "orthogonal" if they are distributed over different dimensions of the same process array. A collapsed primitive range is orthogonal to *any* primitive range except itself.

Arrays

- The shape and mapping of an array are defined by an ordered list of orthogonal index ranges.

- The types of array elements are restricted to one of the Java primitive types (cf, F77). There is a separate (sub)class of array for each primitive type.

```
ArrayInt a = new ArrayInt(x, y) ;
ArrayFloat b = new ArrayFloat(y, x) ;
ArrayDouble c = new ArrayDouble(y) ;
ArrayDouble d = new ArrayDouble(z) ;
```

  `a` and `b` are rank 2. `c` is rank 1.

Expressions

- An "expression" is an array expression.

  An expression node may correspond to a complete temporary array constructed at run-time, or through some process of interpretation (akin to interpretation of the mooted of forall scripts) it may simply correspond to a scalar temporary inside a loop in some "node code".

  The classes for expressions parallel the classes for arrays:

```
ExprInt, ExprFloat, ExprDouble, ...
```

- The shape and mapping of an expression are defined by an *unordered set* of orthogonal index ranges.

- An array is an expression. It's shape and mapping are defined by the set of ranges of the array.

- A range is an integer expression. It's shape and mapping are defined by the singleton set containing the range itself.

- Members on the expression class provide arithmetic, eg

```
b.plus(c)
```

  corresponds to the array expression

```
b + c
```

  The shape of the result is the union of the shape of the addend and the argument. *The operation is illegal if this union contains any non-orthogonal range pairs.*

  Certain arithmetic coercions can be defined by overloading `plus`, eg

```
ExprInt   ExprInt :: plus(ExprInt) ;

ExprFloat ExprInt :: plus(ExprFloat) ;

ExprFloat ExprFloat :: plus(ExprFloat) ;

ExprFloat ExprFloat :: plus(ExprInt) ;
```

Other analogous members (`times`, `if`, and so on) are provided.

Simple operations on arrays

- An expression "conforms" with an array if its index range set is a subset of the index range list of the array.
- Assignment is through through suitable members, eg

```
void ArrayInt    :: assign(ExprInt) ;
void ArrayFloat  :: assign(ExprFloat) ;
void ArrayDouble :: assign(ExprDouble) ;
...
```

  Eg,

```
a.assign(b) ;    // a = b
b.assign(c) ;    // b = c
```

  The argument of `assign` must conform with the array being assigned.

```
c.assign(d) ;    // ILLEGAL
```

- The fact that a range is an expression provides some of the power of a "forall" statement...

```
a.assign(x.plus(y)) ;   // forall (x, y) a(x, y) = x + y ;
```

- A masked assignment operation 'where' could also be provided.

Communication

- The conformance rules for the previous operations mean no communication is needed in their implementation.

- The array member `remap` takes an argument of the same *shape* as the array (which does not, however, conform with the array) and copies its value into the array.

```
void ArrayInt    :: remap(ArrayInt) ;
void ArrayFloat  :: remap(ArrayFloat) ;
void ArrayDouble :: remap(ArrayDouble) ;
...
```

  Eg

```
c.remap(d) ;  // c = d
```

  This operation clearly *does* require communication.

- `shift` is similar to `remap`, except that the argument *does* conform with the array, but the copy is a shifted copy.

  Again, communication is required.

- Other analogous members are provided.

Subranges and Array sections

- A "subrange" is a triplet subrange of some primitive range, created by a suitable member function, eg

```
x.sub(lb, ub, stride) ;
```

  The signature of `sub` would be

```
Range Range :: sub(int, int, int) ;
```

  Subranges inherit the orthogonality properties of their parent primitive range.

- A section of an array is created by passing subranges of the array's range set to a suitable member, eg

```
a.sect(x.sub(0, 99, 2), y) ;  // a(0 : 99 : 2, :)
```

  The signatures of 'sect' would be

```
ArrayInt ArrayInt :: sect(Range x) ;
ArrayInt ArrayInt :: sect(Range x, Range y) ;
...
```

Example

The HPF/F90 program...

```
  integer n
  parameter(n = 8)

  real u(n, n)

  !hpf$ distribute u (block, block) onto p
  !hpf$ processors p (2, 2)

  integer i, j
  integer iter

  do i = 1, n
    do j = 1, n
      if(i == 1 .or. i == n .or. j == 1 .or. j == n) then
        u(i, j) = 1.0
      else
        u(i, j) = 0.0
      endif
    enddo
  enddo

  do iter = 1, 5
    u(2 : n - 1, 2 : n - 1) = &
        0.25 * (u(2 : n - 1, 1 : n - 2) + u(2 : n - 1, 3 : n) + &
                u(1 : n - 2, 2 : n - 1) + u(3 : n, 2 : n - 1))
  enddo

end
```

could be translated as...

```
  final integer n = 8 ;

  Procs p = new Procs(2, 2) ;

  Range x = new Range(n, BLK, p.dim(0)) ;
  Range y = new Range(n, BLK, p.dim(1)) ;

  ArrayFloat  u = new ArrayFloat(x, y) ;
  ArrayInt mask = new ArrayInt(x, y) ;
```

```
mask = x.gt(0).and(x.lt(n - 1)).and(y.gt(0)).and(y.lt(n - 1)) ;

u.where(mask, 0.0, 1.0) ;

ArrayFloat tmp1 = new ArrayFloat(x, y) ;
ArrayFloat tmp2 = new ArrayFloat(x, y) ;
ArrayFloat tmp3 = new ArrayFloat(x, y) ;
ArrayFloat tmp4 = new ArrayFloat(x, y) ;

for(int iter = 1 ; iter <= 5 ; iter++) {
  tmp1.shift(u, -1, 0) ;
  tmp2.shift(u,  1, 0) ;
  tmp4.shift(u, -1, 1) ;
  tmp3.shift(u,  1, 1) ;

  u.where(mask, tmp1.plus(tmp2).plus(tmp3).plus(tmp4).
                     times(0.25)) ;

      // where(mask) u = 0.25 * (tmp1 + tmp2 + tmp3 + tmp4)
}
```

This assumes that the arithmetic members on arrays include boolean arithmetic, and that two variants of masked assignment (`where`) are provided (one with two arguments corresponds to F90 simple WHERE; one with three arguments corresponds to F90 WHERE/ELSEWHERE).

Variation: the `where` operation in the loop could be replaced by use of sections...

```
Range i = x.sub(1, n - 2) ;  // subrange 1 : n - 2
Range j = y.sub(1, n - 2) ;  // subrange 1 : n - 2

  ...

  u.sect(i, j).assign(tmp1.sect(i, j).plus(tmp2.sect(i, j)).
                plus(tmp3.sect(i, j)).plus(tmp4.sect(i, j)).
                                              times(0.25)) ;

      // u(2 : n - 1, 2 : n - 1) = 0.25 *
      //      (tmp1(2 : n - 1, 2 : n - 1) +
      //       tmp2(2 : n - 1, 2 : n - 1) +
      //       tmp3(2 : n - 1, 2 : n - 1) +
      //       tmp4(2 : n - 1, 2 : n - 1))
```

Variation: the `shift` operations could be replaced by `remap` operations...

```
ArrayFloat tmp1 = new ArrayFloat(i, j) ;
ArrayFloat tmp2 = new ArrayFloat(i, j) ;
ArrayFloat tmp3 = new ArrayFloat(i, j) ;
ArrayFloat tmp4 = new ArrayFloat(i, j) ;

...

  tmp1.remap(u.sect(x.sub(2, n - 1), j)) ;
  tmp2.remap(u.sect(x.sub(0, n - 3), j)) ;
  tmp3.remap(u.sect(i, y.sub(2, n - 1))) ;
  tmp4.remap(u.sect(i, y.sub(0, n - 3))) ;

  // tmp3 = u(1 : n - 2, 2 : n - 1)
  // tmp2 = u(0 : n - 3, 1 : n - 2)
  // tmp1 = u(2 : n - 1, 1 : n - 2)
  // tmp4 = u(1 : n - 2, 0 : n - 3)

  u.sect(i, j).assign(tmp1.plus(tmp2).plus(tmp3).plus(tmp4).
                      times(0.25)) ;

      // u(2 : n - 1, 2 : n - 1) = 0.25 *
      //                           (tmp1 + tmp2 + tmp3 + tmp4)
```

This corresponds more directly to the original Fortran which uses arithmetic on array sections, rather than shift operations.

5 HPJava: Work in progress. *Bryan Carpenter, Yuh-Jye Chang, November 12, 1996*

Emphasis to date has been on providing class libraries to facilitate communication and parallel computation within the framework of Java. For now this means communicating Java *applications*, but eventually it should be possible provide class libraries which allow Java applets downloaded from a common server to communicate (for example, by having their server transparently through-route communications).

We have prototyped a class library for Fortran-M-like *channel communication* to support a message-passinge style or programming. Presently we are experimenting with class library interfaces to directly support the *data parallel* programming style.

The existing software for both these approaches has been layered directly on the Java socket interface. For the next stage of development we plan to produce a Java interface to a run-time library being developed in the PCRC project. By interfacing to an existing body of software through Java *native methods* we will avoid reimplementing the complex collective operations needed in data parallel applications. We also expect to obtain better efficiency.

5.1 Parallel arrays and data parallel computation

Our experimental implementation of parallel arrays for Java is similar to the array model of the C++ library, Adlib, developed by one us. (A similar model is also being used in the NPAC PCRC library—it is based on the HPF distributed data model.)

In Java the distribution of an array is parametrized by a member of the `Array` class (as it turns out the Java `Array` object is more akin to an HPF *template* than a data array). An `Array` object is defined in terms a target group of processes and a set of distributed index ranges, one per array dimension. The group of processes is some multi-dimensional process grid represented by an object from the `Procs` class. Each distributed index range is represented by an object from the `Range` class.

Any parallel Java application is written as a a class extending the library class `Node`. The `Node` class maintains some global information and provides various collective operations on arrays. The code for the "main program" goes in the `run` member of the application class.

5.1.1 Example data parallel Java program

A simplified version of the code for the "Life" demo is given in figure 1.

The object `p` represents a 2 by 2 process grid. In this simplified example we assume that the program executes on exactly four processors. More generally the library provides a member function on `Procs` to determine whether the local

```
public class Life extends Node implements Runnable {
  ...

  public void run() {
    Procs p = new Procs(this, 2, 2) ;

    Range x = new Range(N, p, 0) ;
    Range y = new Range(N, p, 1) ;

    Array r = new Array(p, x, y) ;

    int xys = r.seg();
    byte[] w = new byte[xys];

    byte[] cn_ = new byte[xys];
    byte[] cp_ = new byte[xys];
    ... etc, create arrays for 8 neighbours

    // Initialize the Life board

    for(r.forall(); r.test(); r.next())
      w[r.sub()] = fun(x.idx(), y.idx()) ;

    // Main loop

    for (int k=0; k<NITER; k++) {

      // Get neighbours

      shift(cn_, w, r, 0,  1, CYCLIC);
      shift(cp_, w, r, 0, -1, CYCLIC);
      ... etc, copy arrays for 8 neighbours

      // Life update rule

      for(int i=0; i<w.length; i++) {
        switch (cn_[i] + cp_[i] + c_n[i] + c_p[i] +
                cnn[i] + cnp[i] + cpn[i] + cpp[i]) {
          case 2 : break;
          case 3 : w[i] = 1; break;
          default: w[i] = 0; break;
        }
      }
    }
  }
}
```

Figure 1: Simplified code of the Life demo program.

process holds any member of the virtual process grid. The `Procs` constructor takes the current `Node` object as an argument, from which it obtains information on the available physical processes.

The objects `x` and `y` represent index ranges of size `N` (the global index is in the range `0, ..., N - 1`) distributed over the first and second dimensions of the grid `p`. The default distribution format is blockwise. Cyclic distribution format can also be specified by using a range object of class `CRange`, which is derived from `Range` (the pilot implementation does not provide any more general distribution or alignment options).

The object `r` represents the shape and distribution of a two dimensional array. Note that this "template" can be shared by several actual arrays because it does not contain a data vector. The limited polymorphism of Java makes it awkward to create true container classes for primitive data types. The data vectors that hold the local segments of arrays are created separately using the inquiry function `seg` which returns the number of locally held elements. In the example the elements of the main data array are held in `w`. The extra arrays `cn_, cp_, ..., cnn, ...` will be used to hold arrays on neighbour sites.

The "forall loop" initializing `w` should be read as something like

```
forall(i in range x, j in range y)
  w(i, j) = fun(i, j)
```

where `fun` is some function of the global indices defining the initial state of the life board. The members `forall`, `test`, `next` update the state of `r` and the range structures contained in it so that `r.sub()` returns the local subscript for the current iteration, and `x.idx()` and `y.idx()` return the global index values for the current iteration

The main loop uses cyclic `shift` operations to obtain neighbours, communicating data where necessary. The `shift` operation is a member of the `Node` class. Eventually it will be overloaded to accept data vectors of any primitive type—here the array elements are `byte`s.

Finally `w` is implemented in terms of its neighbours. This could have been done using a "forall loop", but since global index values are not needed here the loop has been optimised for a simple for loop over the local segment. This performance-critical inner loop is coded at least as efficiently as a typical sequential program.

Note some characteristic features of the data-parallel style of programming:

- The distribution format of the arrays can be changed just by altering a few parameters at the start of the program—the main program is insensitive to these details
- low level message-passing is abstracted into high-level collective operations on distributed array structures.

5.1.2 Starting an HPJ program

A `Node` instance can be started on any host running a suitable daemon. A master server process records the pool of slave daemons currently running. When a new daemon is started, it registers with this central server. Clients may request a list of hosts running daemons from the main server, then ask for a particular Java object file to be run on any subset of those hosts. Typically the client is an applet running in a browser, and the main HPJ server runs on the same host as an HTTP server from which the applet was downloaded.

5.1.3 Outlook on data-parallel programming in Java

Although Java is quite a restrictive language, our experiments suggest that it should be possible develop a useful class library interface for data parallel programming, without necessarily extending the basic language.

Java's strict typing regime (the absence, for example, of castable pointers) makes it difficult to write interfaces to communication libraries that efficiently accept data of multiple types (polymorphism). The I/O and socket communication classes in the standard API resort to expensive format conversions whenever an object has to be transfered. Benchmarking the example presented earlier indicates that while parallelism gives genuine performance gains there is an obvious communication penalty even with large problem sizes. For example, the following timings were obtained on a grid size of 512 by 512

number of processors	time per iteration/ms
1	1502
2	1011
4	784

The volume of data that has to be communicated in an iteration is on the order of 1% of the total data set, but speedup is still a long way from ideal. Further benchmark results are available on the demo Web page.

We hope that the efficiency problems can be overcome by taking a different approach to implementing the interface. Instead of implementing the library within Java on top of the standard API we will call the existing PCRC runtime library through a *native methods* interface. In any case this approach will be easier than recoding collective communication operations, some much more complicated the `shift`, within Java.

5.2 Channels

We have prototyped a Java class library for channel communication. The semantics are similar (but not identical) to the dynamic channels of Fortran M.

The model is quite attractive in Java because

1. It is "connection-oriented" so ordinary data communications map directly onto the socket I/O primitives efficiently implemented for Java and the Web.

2. It directly supports highly dynamic situations where new remote processes (eg, applets) come into existence unpredictably, and may have unpredictable communication demands (eg, determined by requirements of a remote user).

3. It provides a layer of abstraction above internet addresses and socket port numbers, and associated resources to which access may be limited by Java security models.

5.2.1 Overview of the channel model

In our model a channel supports bi-directional communication of data (with unbounded buffering). Access to logical channels is provided through a Java class called a `Port`[1]. Networks of Java applications connected by channels are created through the following operations

- A pair of locally defined ports can be *connected*, creating a local channel between a pair of connected *channel ends*.

- A *remote procedure call* mechanism allows initiation of a Java application on a remote processor, and creates an initial channel between the parent and the remote application.

- Channel ends can be communicated over other channels, on much the same footing as data.

These primitives are sufficient to create sets of processes with arbitrary channel connectivity.

A functionality similar to the "merge" in Fortran-M is provided through the "merge pool" class, `Merge`. Channel ends can be added to or removed from the pool by passing their ports to suitable member functions. A blocking `select` operation returns a handle to a port in the pool which presently has input data ready. Merge pools enable non-deterministic patterns of communication.

5.2.2 Outlook on channel communication in Java

This seems like a intersting approach to Java communication, which could have applications beyond scientific parallel programming. For now this line of investigation is being pursued less actively than the data parallel approach.

[1] The `Port` dresses up and adds functionality to an internet port, but should not be confused with this lower-level idea. The `Port` interface does not provide direct access to port numbers or internet addresses.

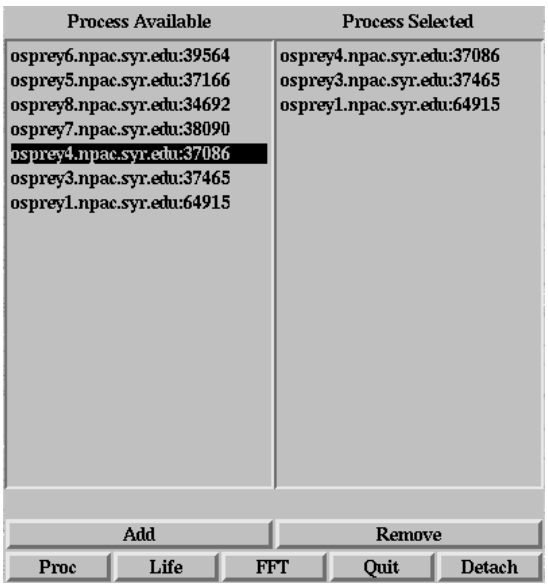

Figure 2: Menu for selecting processors.

6 "HPJava": Demo Description. *Bryan Carpenter, Yuh-Jye Chang, November 16, 1996*

The Web server for the demo runs on a processor from a cluster of high performance workstations located at NPAC. Various other hosts in this cluster run daemons that support remote execution of the Java class files for the demos.

When the demo starts, a menu of available hosts is presented. A subset of these hosts can be selected and added to the set subsequently used for running the examples.

6.1 Conway's Game of Life

"Life" is a familiar *cellular automaton*, in which cells on a two-dimensional board are updated according to the current state of their eight neighbours. It provides a simple illustration of the the archetypal nearest-neighbour update, which recurs in various numerical simulations, and lends itself well to parallel computation.

The Java implementation uses an experimental version of a class library intended to facilitate *data-parallel* computations on regular arrays. Characteristics of this approach, similar to *High Performance Fortran*, include

- The distribution format of the arrays can be changed just by altering a few

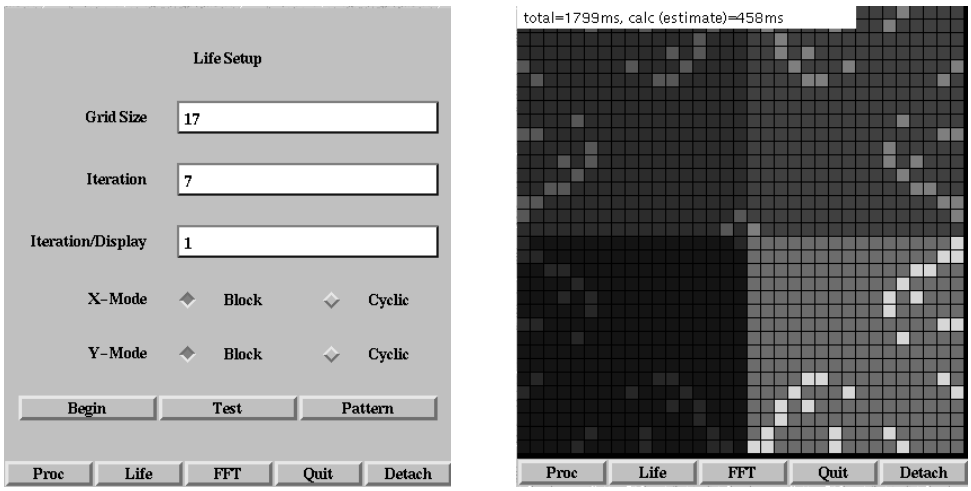

Figure 3: Screens of the "Life" demo.

parameters at the start of the program—the main program is insensitive to these details

- low level message-passing (in the Java case, through Internet sockets) is abstracted into high-level collective operations on distributed array structures.

Clicking on the "Life" button brings up a menu of parameters which can be changed. These include the size of the grid ("board"), the number of iterations, and the frequency with which the state of the board is displayed. Displaying the board generally takes much longer than updating it: for benchmarking purposes the display should happen infrequently (for demonstration purposes, frequently). The options also allow changing the *distribution format* of the board—the way the board is split between processors. Like HPF, block or cyclic distributions are allowed.

Clicking on the "Begin" button causes the simulation to start. The initial state of the board is currently a fixed "cross" of live cells. This will evolve into more interesting patterns as the simulation proceeeds. The processor "owning" a portion of the board is coded through the colour of the cells, providing a clear visualization of the various distribution formats.

The processor set can be changed by clicking the "Proc" button. The parameters or distribution format can be changed by clicking the "Life" button again.

6.2 2D FFT "Image Compression"

This demo is a Java port of one of the HPFA (*High Performance Fortran Applications*) kernels. As explained in the description of the original code:

> The Fast Fourier Transform (FFT) is the most widely known example of the Spectral method for computational problems. In all such methods, one performs a linear transformation of the stated problem into another physical domain where it is hoped will be more tractable. In Fourier transformations, the mapping is from the time-domain to the frequency-domain. The FFT is widely used in the field of image processing, where one commonly describe an image in terms of intensity values in a two-dimensional matrix. The codes contained here performs a two-dimensional FFT over an example matrix in the following manner:
>
> 1. Set up the input 2-D matrix.
> 2. Perform FFT across the leading dimension of the matrix.
> 3. Transpose the matrix, using an F90/HPF intrinsic function.
> 4. Perform FFT again, across the leading dimension.

In our case "transpose" is a new operation in the class library, on a par with the "shift" operation used in nearest neighbour updates. As explained above, sequential FFTs are performed on all columns of the image data in parallel, then on all rows of the image in parallel. The transpose operation sits between these two phases.

After selecting the set of processors click on the "FFT" button. In the demo the image data is a photograph of a wolf. One of several image files (of different resolution) can be selected. In order of increasing size, the options presently are

```
wolf_4.pgm
wolf_2.pgm
wolf.pgm
```

The mask width defines the size of a square centered in the Fourier space representation of the picture. You can choose to delete modes inside or outside this boundary. On clicking the "Begin" button, the original image is displayed then the FFT is performed. The result of the Fourier transform is also displayed as an image, mapping Fourier components into pixels. According to the masking selection, some of the components will be deleted. Finally the FFT is reversed, and the modified ("uncompressed") version of the original image is displayed.

Once again the processor "owning" a particular pixel or Fourier component is coded through the colour of the cells.

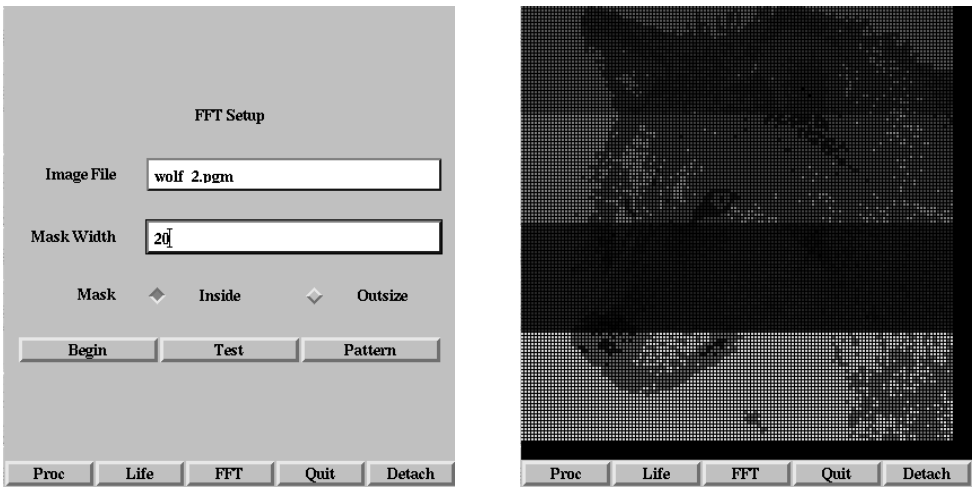

Figure 4: Screens of the "FFT" demo.

7 MPI Java Wrapper Implementation. *Yuh-Jye Chang, December 16, 1996*

7.1 MPI Java wrapper introduction

This draft presents a Java language interface for MPI. There are some issues specific to Java that must considered in the design of this interface that go beyond the simple description of language bindings. In particular, in Java, we must be concerned with the design of objects, their methods, the feature of Java native methods, rather than just the design of a language-specific functional interface to MPI. Fortunately, the original design of MPI was based around the notion of objects, so a natural set of classes is already part of MPI.

7.2 MPI Java wrapper design

The Java wrapper for MPI is designed according to the following criteria:

- The Java wrapper for MPI consists of a small set of classes with a lightweight functional interface to MPI. The classes are based upon the fundamental MPI object types (e.g. communicator, group, etc.).
- The Java wrapper language bindings provide a semantically correct interface to MPI.
- There is a one-to-one mapping between MPI functions and their Java wrapper bindings.

- To the greatest extent possible, the Java wrapper for MPI functions are methods functions of MPI classes.

7.3 Java classes for MPI

All MPI classes, constants, and methods are declared within the scope of an `MPI` package. Thus, by import the `MPI` package or using the `MPI.xxx` prefix, we can reference the MPI Java wrapper. The classes of the `MPI` package are those classes corresponding to objects implicitly used by MPI. An abbreviated definition of the MPI package and its member classes is as follows:

```
package MPI;

public class MPI;
public class Comm;
public class Group;
public class Datatype;
public class Op;
public class Status;
public class Request;
public class Errhandler;
```

7.4 Class methods for Java MPI

All methods (except for constructors and destructors) of MPI classes are public native. Which means in Java program the methods identifier and arguments are defined without further implementation.

Example 1. Example showing a simple Java MPI wrapper usage.

```
Example1.java :
import MPI.*;
public class Example1 {

  static public void main(String[] args) {
    MPI JMPI = new MPI(args);
    int[] myid = new int[1];
    int[] numprocs = new int[1];

    JMPI.COMM_WORLD.Size(numprocs);
    JMPI.COMM_WORLD.Rank(myid);

    System.out.println("Process "+myid[0]+"/"+numprocs[0]+
                       " on "+JMPI.Get_processor_name());

    JMPI.finalize();
  }
}
```

```
MPI JMPI = new MPI(args);
```

This statement will create a MPI class instance called `JMPI`. The `MPI` classes constructor will transform the `String[]` arguments into C style string array reference, call `MPI_Init`, create communicator `COMM_WORLD`, create default MPI reduce operators `MIN`, `MAX`, `SUM`, etc.

```
int[] myid = new int[1];
int[] numprocs = new int[1];
```

Here, two variable `myid` and `numprocs` are declared. In Java, the simple type like `int`, `byte`, `double`, ...will only pass by value. So we can never reflect the value change back to the caller. To work around the fact that Java don*t have pass by reference and pointer, we can use simple type array instead.

```
JMPI.COMM_WORLD.Size(numprocs);
JMPI.COMM_WORLD.Rank(myid);
```

Now, two native methods belong to `COMM_WORLD` were called. The `COMM_WORLD` is a `Comm` class instance which initiated in `MPI()` constructor. The `COMM_WORLD` communicator provide all the MPI communication binding that use the `MPI_COMM_WORLD` communicator. As we had mention before, the `Size()` and `Rank()` methods will take a `int[]` argument, so the result can be received.

```
System.out.println("Process "+myid[0]+"/"+numprocs[0]+
                   " on "+JMPI.Get_processor_name());
```

Output the `myid`, `numberprocs`, and `processor_name` into standard output.

```
JMPI.finalize();
```

The last step that conclude the MPI usage is calling `finalize()` method.

7.5 Java native method

The Java native method is a great way to gain and merge the power of C or C++ programming into Java. To use Java as a scientific and high performance language, when efficient native Java compilers are not fully implemented, use native method can boost the performance to at least the speed of C compiled code.

Example 2. Example showing how Java native method works.

```
JMPI.java :
public class JMPI {
  public native int Init(String[] args);
  public native int Finalize();
  static {
    System.loadLibrary("JMPI");
```

```
  }
}

JMPI.h : (created by javah and JMPI.class)
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <native.h>
/* Header for class JMPI */

#ifndef _Included_JMPI
#define _Included_JMPI

typedef struct ClassJMPI {
    char PAD; /* ANSI C requires structures to have a least one member
*/
} ClassJMPI;
HandleTo(JMPI);

#ifdef __cplusplus
extern "C" {
#endif
struct Hjava_lang_String;
extern long JMPI_Init(struct HJMPI *,HArrayOfString *);
extern long JMPI_Finalize(struct HJMPI *);
#ifdef __cplusplus
}
#endif
#endif

JMPI.c : (created by javah -stub and JMPI.class)
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <StubPreamble.h>

/* Stubs for class JMPI */
/* SYMBOL: "JMPI/Init([Ljava/lang/String;)I", Java_JMPI_Init_stub */
stack_item *Java_JMPI_Init_stub(stack_item *_P_,struct execenv *_EE_) {
extern long JMPI_Init(void *,void *);
_P_[0].i = JMPI_Init(_P_[0].p,((_P_[1].p)));
return _P_ + 1;
}
/* SYMBOL: "JMPI/Finalize()I", Java_JMPI_Finalize_stub */
stack_item *Java_JMPI_Finalize_stub(stack_item *_P_,struct execenv
*_EE_) {
extern long JMPI_Finalize(void *);
_P_[0].i = JMPI_Finalize(_P_[0].p);
return _P_ + 1;
}
```

```
JMPINative.c :
#include "mpi.h"
#include "JMPI.h"
#include "stdlib.h"

long JMPI_Init(struct HJMPI *this, HArrayOfString *args) {
  int i, result, len;
  char** sargs;
  HString **data = unhand(args)->body;
  len = obj_length(args);
  sargs = (char**)calloc(len, sizeof(char*));
  for (i=0; i<len; i++) {
    sargs[i] = allocCString(data[i]);
  }
  result = MPI_Init(&len, &sargs);
  for (i=0; i<len; i++) free(sargs[i]);
  free(sargs);
  return result;
}

long JMPI_Finalize(struct HJMPI *this) {
  return MPI_Finalize();
}
```

The only programs user created are `JMPI.java` and `JMPINative.c`. The `JMPI.h` and `JMPI.c` are generated by `javah` and compiled `JMPI.class` files. Compile the `JMPI.c` and `JMPINative.c` into `libJMPI.so` (in UNIX) or `JMPI.dll` (in Microsoft Windows) and you are done.

7.6 Java datatypes

The following table lists all of the Java basic simple type and their corresponding C/C++ and MPI datatype.

```
Java datatype C/C++ datatype MPI datatype
byte signed char MPI_CHAR
char signed short int MPI_SHORT
short signed short int MPI_SHORT
boolean signed long int MPI_LONG
int signed long int MPI_LONG
long signed long long int MPI_LONG_LONG_INT
float float MPI_FLOAT
double double MPI_DOUBLE
```

Because Java is platform independent, the size of simple type will be the same in all platforms. So in order to fit into some system that has 64bits pointer, we use the long in Java to store the MPI object handle or pointer reference.

7.7 Problems due to strong typing and no pointer

All MPI functions with choice arguments associate actual arguments of different datatypes with the same dummy argument. This is not allowed by Java. In C, the `void*` formal arguments avoid these problems.

The following code fragment is technically illegal and may generate a compile-time error.

```
float f = new float[10];
double r = new double[10];

MPI.COMM_WORLD.Send(f, *);
MPI.COMM_WORLD.Send(r, *);
```

Technically, we will have to use methods overload with different argument datatype or methods with different identifier.

The methods overload implementation in native method will cause problem. Because the methods that has the same name will have the same sub initialization function generated by javah.

The methods with different identifier will implement as following.

```
MPI.COMM_WORLD.SendFloat(f, *);
MPI.COMM_WORLD.SendDouble(r, *);
```

But, there are many MPI communication functions, eg. `MPI_Send`, `MPI_Bsend`, `MPI_Ssend`, `MPI_Rsend`, etc. If we use this approach, than we are going to have tons of native methods for each functions and datatypes. Which we believe is quite a waste. So we introduce a Java Datatype class which perform the polymorphism between different Java datatypes.

7.8 The polymorphism of Java Datatype class

The Datatype class listed partly as following.

```
Datatype.java
package MPI ;
public class Datatype {
  public Datatype() { handle = type = 0;}
  public Datatype(byte[] data) { SetupByte(data);}
  public Datatype(char[] data) { SetupChar(data);}
  ...
  private native void SetupByte(byte[] data);
  private native void SetupChar(char[] data);
  ...
  private long handle, type;
  private int size;
}
```

There are constructors for each java datatype array. In each constructor, it will invoke a native method with different identifier that store the memory address into 64 bits *handle* variable, store the corresponding `MPI_Datatype` object (eg. `MPI_CHAR`, `MPI_SHORT`, ...) into 64 bits *type* variable, and the buffer size into *size* variable.

The original MPI call in C/C++

```
MPI_Send(void*, int size, MPI_Datatype, int dest, int tag, MPI_Comm);
```

would become much simpler in Java

```
MPI.Comm.Send(MPI.Datatype, int dest, int tag);
```

7.9 Other problems

1. The creation of `MPI_Op` and `MPI_Request` will involve the pointer to function, which is not allowed in Java also. But we think this problem could be resolve by invoke a Java class and invoke its method automatically.

2. The current implementation of MPI conflict with Java seriously. When number of processor `> 1`, use `mpirun` to invoke Java interpreter will core dump or even hang the processes. We already reflect this problem to MPI implementation authors. Hopefully they will solve this problem soon. Currently, Bryan Carpenter modify part of the MPI source code and it works quite good (but will still core dump when `np > 3`). Thanks to his patch, we can have further progress in this implementaion.

7.10 Conclusion

To propose the Java as the scientific and high performance language, we believe that the Java MPI wrapper is a very useful and important step. Which shows the Java versatility and flexibility. We also believe that Java will play a very important role in scientific and high performance world.

7.11 Test example

```
test.java :
import MPI.*;

class test {
  static public void main(String[] args) {
    MPI JMPI = new MPI(args);
    byte[] buf = new byte[1024];
    int i, done = 0;
    int[] n = new int[1];
    int[] myid = new int[1];
    int[] numprocs = new int[1];
```

```
Datatype N = new Datatype(n);
double PI25DT = 3.14159265358979323846264 3;
double[] mypi = new double[1];
double[] pi = new double[1];
Datatype Mypi = new Datatype(mypi);
Datatype Pi = new Datatype(pi);
double h, sum, x;
double startwtime = 0.0;
double endwtime;
String processor_name;
Status stat = new Status();

JMPI.COMM_WORLD.Size(numprocs);
JMPI.COMM_WORLD.Rank(myid);
System.out.println("Process "+myid[0]+"/"+numprocs[0]+
  " on "+JMPI.Get_processor_name());
n[0] = 0;
while (done == 0) {
  if (myid[0] == 0) {
    n[0] = (n[0]==0) ? 100 : 0;
    startwtime = JMPI.Wtime();
  }
  JMPI.COMM_WORLD.Bcast(N, 0);
  if (n[0] != 0) {
    h = 1.0 / (double)n[0];
    h = 1.0 / (double)n[0];
    sum = 0.0;
    for (i = myid[0] + 1; i <= n[0]; i += numprocs[0]) {
      x = h * ((double)i - 0.5);
      sum += (4.0 / (1.0 + x*x));
    }
    mypi[0] = h * sum;
    JMPI.COMM_WORLD.Reduce(Mypi, Pi, JMPI.SUM, 0);
    if (myid[0] == 0) {
      System.out.println("pi is approximately "+pi[0]+
        ", Error is "+Math.abs(pi[0] - PI25DT));
      endwtime = JMPI.Wtime();
      System.out.println("wall clock time = " +
        (endwtime-startwtime));
    }
  } else done = 1;
}
n[0] = 99;
System.out.println("Send self n: "+n[0]);
JMPI.COMM_WORLD.Send(N, myid[0], 23);
JMPI.COMM_WORLD.Recv(N, myid[0], 23, stat);
System.out.println("Recv self n: "+n[0]);
```

```
    System.out.println("Status: cound: "+stat.count+", SOURCE: "+
      stat.SOURCE+", TAG: "+stat.TAG+", ERROR:"+stat.ERROR);
    JMPI.Buffer_attach(buf);
    n[0] = 98;
    System.out.println("Buffer Send self n: "+n[0]);
    JMPI.COMM_WORLD.Bsend(N, myid[0], 22);
    JMPI.COMM_WORLD.Recv(N, myid[0], 22, stat);
    System.out.println("Recv self n: "+n[0]);
    JMPI.Buffer_detach(buf);
    System.out.println("Status: cound: "+stat.count+", SOURCE: "+
      stat.SOURCE+", TAG: "+stat.TAG+", ERROR:"+stat.ERROR);
    JMPI.finalize();
  }
}
```

7.12 Execution result

```
mpirun -np 2 run
Process 0/2 on osprey1.npac.syr.edu
Process 1/2 on osprey2.npac.syr.edu
pi is approximately 3.1416, Error is 8.33333e-06
wall clock time = 0.040595
Send self n: 99
Recv self n: 99
Status: cound: 1, SOURCE: 0, TAG: 23, ERROR:0
Buffer Send self n: 98
Recv self n: 98
Status: cound: 1, SOURCE: 0, TAG: 22, ERROR:0
Send self n: 99
Recv self n: 99
Status: cound: 1, SOURCE: 1, TAG: 23, ERROR:0
Buffer Send self n: 98
Recv self n: 98
Status: cound: 1, SOURCE: 1, TAG: 22, ERROR:0
```

7.13 List of detailed Java wrapper for MPI

```
MPI.java :
package MPI;
public class MPI {
  public Comm COMM_WORLD;
  public Op MAX, MIN, SUM, PROD, LAND, BAND,
    LOR, BOR, LXOR, BXOR, MINLOC, MAXLOC;

  public MPI(String[] args);
  public void finalize();
  public native double Wtime();
```

```
  public native double Wtick();
  public native String Get_processor_name();
  public native int Buffer_attach(byte[] buf);
  public native int Buffer_detach(byte[] buf);
  *
  static {
    System.loadLibrary("MPI");
  }
}
```

Comm.java :

```
package MPI;
public class Comm {
  public final static int NULL  = 0;
  public final static int SELF  = 1;
  public final static int WORLD = 2;

  public Comm() { handle = 0;}
  public Comm(int Type) { Setup(Type);}
  private native void Setup(int Type);

  public native int Barrier();
  public native int Size(int[] size);
  public native int Rank(int[] rank);

  public native int Send(Datatype buf, int dest, int tag);
  public native int Bsend(Datatype buf, int dest, int tag);
  public native int Ssend(Datatype buf, int dest, int tag);
  public native int Rsend(Datatype buf, int dest, int tag);
  public native int Recv(Datatype buf, int source, int tag,
    Status stat);
  *
  public native int Bcast(Datatype buf, int root);
  public native int Gather(Datatype sbuf, Datatype rbuf, int root);
  public native int Scatter(Datatype sbuf, Datatype rbuf, int root);
  public native int Allgather(Datatype sbuf, Datatype rbuf);
  public native int Alltoall(Datatype sbuf, Datatype rbuf);
  public native int Reduce(Datatype sbuf, Datatype rbuf, Op op,
    int root);
  *
  private long handle ;
}
```

Datatype.java :

```
package MPI ;
public class Datatype {
  public Datatype() { handle = type = 0;}
```

```
  public Datatype(byte[] data) { SetupByte(data);}
  public Datatype(char[] data) { SetupChar(data);}
  public Datatype(short[] data) { SetupShort(data);}
  public Datatype(boolean[] data) { SetupBoolean(data);}
  public Datatype(int[] data) { SetupInt(data);}
  public Datatype(long[] data) { SetupLong(data);}
  public Datatype(float[] data) { SetupFloat(data);}
  public Datatype(double[] data) { SetupDouble(data);}
  *
  private native void SetupByte(byte[] data);
  private native void SetupChar(char[] data);
  private native void SetupShort(short[] data);
  private native void SetupBoolean(boolean[] data);
  private native void SetupInt(int[] data);
  private native void SetupLong(long[] data);
  private native void SetupFloat(float[] data);
  private native void SetupDouble(double[] data);
  *
  private long handle, type;
  private int size;
}
```

Op.java :

```
package MPI ;
public class Op {
  public final static int NULL  = 0;
  public final static int MAX   = 1;
  public final static int MIN   = 2;
  public final static int SUM   = 3;
  public final static int PROD  = 4;
  public final static int LAND  = 5;
  public final static int BAND  = 6;
  public final static int LOR   = 7;
  public final static int BOR   = 8;
  public final static int LXOR  = 9;
  public final static int BXOR  =10;
  public final static int MINLOC=11;
  public final static int MAXLOC=12;

  public Op() { handle = 0;}
  public Op(int Type) { Setup(Type);}
  private native void Setup(int Type);

  private long handle ;
}
```

Status.java :

```
package MPI ;
public class Status {
  public int count;
  public int SOURCE;
  public int TAG;
  public int ERROR;
}
```

Request.java :

```
package MPI ;
public class Request {
  private long handle;
}
```