HPJava: Work in progress

Bryan Carpenter, Yuh-Jye Chang

August 9, 1996

Contents

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.

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.

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.

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.

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.

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.

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 holds a buffered message. If no port has a message when the call is executed `select` blocks until a message arrives.

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`.

Merge pools enable non-deterministic patterns of communication.

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 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.

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) {
      ;
    }
  }
}
```