MPI programming in HPJava

Bryan Carpenter

September 22, 1997

The initial proposal for NPAC HPJava assumes that the translator is bundled with an interface to the collective communication library *Adlib*. Adlib directly provides transformational operations on HPJava distributed arrays, akin to the intrinsic functions of Fortran 90. To increase the flexibility of the system, and make it more attractive to programmers accustomed to the direct message-passing style, it has been suggested that a comparably seamless interface to MPI (or a Java-compatible subset of MPI) would be very desirable. In HPF, with its global-thread-of-control model, a proper interface to the underlying message-passing platform is only practical through the *extrinsic procedure* mechanism. In HPJava, which is explicitly based on the loosely-synchronous SPMD model, it should be possible to access an MPI interface much more directly.

1 Example Java MPI interface

A subset interface from Java to MPI has been described in [1]. Appendix A gives the complete text of a working MPI program using this interface. The example is an implementation of Conway's Life. It illustrates various characteristic features of MPI, including use of Cartesian communicators to represent the topology of a two-dimensional grid of processors, use of derived data types to describe the edges of the local blocks of the board, and deadlock-free implementation of cyclic permutations using the `Sendrecv` primitive.

A logical possibility is to accept this program directly as a valid HPJava program. However, this would conflict with the proposed *usage rules* for variables in HPJava. In HPJava, values stored in a variable which is not a distributed array are supposed to be identical in all processes—more precisely, in all processes of the *active process group* at the point of declaration of the variable. The program in appendix A does not satisfy this restriction. The variables in the program are not distributed arrays but, they are assigned different values according to their position in the process grid.

We therefore insist on one modification to this program to turn it into a well-formed HPJava program. Suppose `id` is some globally defined object of type `Range` which indexes all processes in the initial process group (the set of process on which the MPI program is initialized)[1]. The body of the program program in the appendix must appear inside an *over* construct:

```
static public void main(String [] args) {
  over(id)
    ... body of 'main' from example in appendix
}
```

According to the normal semantics of *over*, the active process group inside the construct is reduced by restricting to a single value of the parametric range. In the current case this restricts the active process group to a single process. Variables declared inside the construct hold values "replicated" over exactly one process—in other words there are no constraints about consistency in values across different processes.

2 Mixing the message-passing and data-parallel style

Figure 1 is a pure data parallel version of the program in the appendix. This is supposed to be a strict transcription of the original MPI program. For this reason we use `StepRange` distribution format rather than the HPF-like `BlockRange`[2]. Ghost regions are updated using a call to `writeHalo` (presumed extended to deal with cyclic boundary conditions). Output is handled using Adlib `aprintf` and `gprintf`.

Now (while it is difficult to for us to imagine a more compact and readable parallel implementation of this demo than the one in the figure!) suppose we wanted to combine features of this version with the original MPI version. In particular, suppose the `Adlib` communication interface was not available, but we still wanted to perform array initialization and updates using the convenient HPJava array syntax.

The program involves several phases of MPI communication, separated by phases of data parallel computation. In the original MPI program, preparations for the edge-exchange communications are made before entering the main loop. It is wasteful to recompute the parameters of the local block (the `blockSize` and `blockBase` values) and set up the MPI derived data types and source and destination in every iteration of the main loop. The *schedule* of communications should be computed once, at the beginning of the program. This means that the device of the previous section—declare all MPI specific variables as ordinary

[1]Assuming the initial process group is a one dimensional process array—as it is in the current Adlib implementation—`id` is the range of the single process dimension of the that group.

[2]`StepRange` represents a blockwise distribution format with a non-constant block size—there are some blocks of size $S + 1$ followed by some blocks of size S. This is an illustration of how, in HPJava, new distribution formats can often be added by extending the runtime library of `Range` subclasses.

```
class Life {
  int wlo [] = {1, 1}, whi [] = {1, 1} ;  // ghost widths for 'writeHalo'
  int mode [] = {CYCL, CYCL} ;            // boundary conds for   "

  void main(String args) {
    Procs2 p(2, 2) ;
    on(p) {
      Range x = new StepRange(N, p.dims(0), 1) ;  // ghost width 1.
      Range y = new StepRange(N, p.dims(1), 1) ;  //   "     "

      byte [[,]] state = new byte [[x, y]] ;

      // Define initial state of Life board

      over(x)
        over(y)
          if(x == N / 2 || y == N / 2)
            state [[x, y]] = 1 ;
          else
            state [[x, y]] = 0 ;

      // Main update loop.

      int sums [[,]] = new byte [[x, y]] ;

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

        // Update ghost regions using Adlib

        Adlib.writeHalo(state, wlo, whi, mode) ;

        // Calculate neighbour sums.

        over(x)
          over(y)
            sums [[x, y]] =
              state [[x << 1, y << 1]] + state [[x << 1, y]] + state [[x << 1, y >> 1]] +
              state [[x,      y << 1]] +                     + state [[x,      y >> 1]] +
              state [[x >> 1, y << 1]] + state [[x >> 1, y]] + state [[x >> 1, y >> 1]] ;

        // Update state of board values.

        over(x)
          over(y)
            switch (sums [[x, y]]) {
              case 2 : break;
              case 3 : state [[x, y]] = 1; break;
              default: state [[x, y]] = 0; break;
            }
      }

      Adlib.aprintf(" %d%*N", state, N)
      Adlib.gprintf("\n") ;
    }
  }
}
```

Figure 1: HPJava data parallel version of the "life" example.

variables inside an `over(id)` construct—is inapplicable. Each phase of MPI communication must occur in a separate `over` construct, and of course variables declared inside one such construct have no persistence outside the construct.

There is a simple and elegant solution to this problem. The programmer defines a class `MyMPContext` (say) which contains all the local variables associated with message-passing communication. The `MyMPContext` class has a member function defined for each kind of communication phase. Now a distributed array of these objects is created with range `id`. The `over(id)` statement for each phase simply dispatches the appropriate member of the local element. The new main program is given in figure 2.

To define the class `MyMPContext` we need some way to get at the low-level, local components of the `state` distributed array.

The initial implementation strategy for HPJava is via a Java to Java converter. In the output Java, distributed arrays are replaced by descriptor objects plus sequential arrays of local array elements, while *over* constructs are replaced with simple loops. As a part of this strategy it will be necessary to provide Java interfaces to low-level inquiry functions—already part of the Adlib implementation—for extracting exactly the kind of information needed in the MPI phases of the current example. So the inquiry functions needed in `MyMPContext` essentially come for free—they are a necessary part of the overall translation system.

A possible binding for these low-level inquiries is illustrated in figure 3. The inquiry function `dat` returns a sequental Java array containing the locally held elements of the distributed array. The member `block` on `Range` initializes an object whose fields parametrize a local block of the range. In particular the `count` and `glb_bas` fields of this object define the extent of the block and the base value of the global subscript. The `volume` member of the range defines the extent of locally allocated segment. (In general this is not equal to `count`. In the current example, this is because there are ghost edges.)

A priori there is no relationship between the MPI Cartesian communicator, `p`, defined in the `MyMPContext` class and the HPJava process array, `p`, in the main program. For the program to work properly, these two objects must implement consistent parametrizations of the process grid. This is a detail which would have to be sorted out in a practical interface. Probably some sort of explicit conversion between HPJava groups and MPI groups is needed.

```
class Life {
  void main(String args) {
    Procs2 p(2, 2) ;

    Range x = new StepRange(N, p.dims(0), 1) ;  // ghost width 1.
    Range y = new StepRange(N, p.dims(1), 1) ;
    byte [[,]] state = new byte [[x, y]] on p ;

    MyMPContext [[]] comms = new MyMPContext [[id]] ;

    // Define initial state of Life board

    on(p)
      over(x)
        over(y)
          if(x == N / 2 || y == N / 2)
            state [[x, y]] = 1 ;
          else
            state [[x, y]] = 0 ;

    // Initialize local state for MPI portion of program

    over(id)
      comms [[id]] = new MyMPContext(state) ;

    // Main update loop.

    int sums [[,]] = new byte [[x, y]] on p ;

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

      // Update ghost regions using MPI

      over(id)
        comms [[id]].exchangeEdges() ;

      on(p) {
        // Calculate neighbour sums.

        over(x)
          over(y)
            sums [[x, y]] =
              state [[x << 1, y << 1]] + state [[x << 1, y]] + state [[x << 1, y >> 1]] +
              state [[x,      y << 1]] +                       + state [[x,      y >> 1]] +
              state [[x >> 1, y << 1]] + state [[x >> 1, y]] + state [[x >> 1, y >> 1]] ;

        // Update state of board values.

        over(x)
          over(y)
            switch (sums [[x, y]]) {
              case 2 : break;
              case 3 : state [[x, y]] = 1; break;
              default: state [[x, y]] = 0; break;
            }
      }

      over(id)
        comms [[id]].dumpBoard() ;
    }
  }
}
```

Figure 2: Data parallel version of the “life” example with MPI communications.

```
class MyMPContext {

  Cart p ;

  int blockSizeX, blockSizeY, blockBaseX, blockBaseY ;

  int sX, sY ;  // Segment sizes, including ghost regions.
  byte block [] ;

  Datatype edgeXType, edgeYType ;

  int[][] srcX, dstX, srcY, dstY ;

  void MyMPContext(byte [[,] state) {
    int dims [] = new int [2] ;
    boolean periods [] = new boolean [2] ;
    dims [0] = 2 ;
    dims [1] = 1 ;
    periods [0] = true ;
    periods [1] = true ;
    p = new Cart(MPI.COMM_WORLD, 2, dims, periods, false) ;

    Range x = state.rng(0) ;
    Range y = state.rng(0) ;

    block = state.dat() ;

    // Compute local 'blockSizeX', 'blockBaseX', 'blockSizeY', 'blockBaseY'.

    int coords = new int [2] ;
    p.Get(2, dims, periods, coords) ;

    Block blockX = new Block(), blockY = new Block() ;

    x.block(blockX, coords [0]) ;
    blockSizeX = blockX.count ;
    blockBaseX = blockX.glb_bas ;

    y.block(blockY, coords [1]) ;
    blockSizeY = blockY.count ;
    blockBaseY = blockY.glb_bas ;

    sX = x.volume() ;
    sY = y.volume() ;

    // Precompute parameters of shift communications

    ... identical to corresponding code in appendix
  }

  void exchangeEdges() {
    ... identical to code for shifts in appendix
  }

  void dumpBoard() {
    ... identical to 'dumpBoard' code in appendix
  }
}
```

Figure 3: Schematic definition `MyMPContext` class.

3 Recommendations

1. The HPJava language should support distributed arrays of *objects*, not just distributed arrays of primitive type elements. (There is no immediate necessity for the Adlib communication library to deal with these generalized arrays.)

2. Implementation of the Java MPI interface described in [1] should be completed. It will be callable from HPJava essentially as it stands.

3. Some work has to be done on specifying a correspondence between HPJava groups and MPI groups.

4. It would be convenient to have the predefined range, `id`, available as some sort of global variable.

References

[1] Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Donald Leskiw, Xiaoming Li, "Experiments with *HPJava*"

A MPI "life" program in standard Java

```
import MPI.*;

public class Life {

  static final int N = 11 ;
  static final int NITER = 3 ;

  static int blockSizeX, blockSizeY, blockBaseX, blockBaseY ;
  static int sX, sY ;  // Segment sizes, including ghost regions.
  static byte block [] ;

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

    int dims [] = new int [2] ;
    boolean periods [] = new boolean [2] ;
    dims [0] = 2 ;
    dims [1] = 1 ;
    periods [0] = true ;
    periods [1] = true ;
    Cart p = new Cart(MPI.COMM_WORLD, 2, dims, periods, false) ;

    /* Compute local ‘blockSizeX’, ‘blockBaseX’, ‘blockSizeY’, ‘blockBaseY’. */

    {
      int coords [] = new int [2] ;
      int b, stepCoord ;

      p.Get(2, dims, periods, coords) ;

      b         = N / dims [0] ;
      stepCoord = N % dims [0] ;
      if(coords [0] < stepCoord) {
        blockSizeX = b + 1 ;
        blockBaseX = blockSizeX * coords [0] ;
      }
      else {
        blockSizeX = b ;
        blockBaseX = b * coords [0] + stepCoord ;
      }

      b         = N / dims [1] ;
      stepCoord = N % dims [1] ;
      if(coords [1] < stepCoord) {
        blockSizeY = b + 1 ;
        blockBaseY = blockSizeY * coords [1] ;
      }
      else {
        blockSizeY = b ;
        blockBaseY = b * coords [1] + stepCoord ;
      }
    }


    /* Create ‘block’, allowing for ghost cells. */

    sX = blockSizeX + 2 ;
    sY = blockSizeY + 2 ;
    block = new byte [sX * sY] ;


    /* Define initial state of Life board */
```

```
for(int i = 0, ib = 1 ; i < blockSizeX ; i++, ib++)
  for(int j = 0, jb = 1 ; j < blockSizeY ; j++, jb++) {
    int x = blockBaseX + i, y = blockBaseY + j ;
    if(x == N / 2 || y == N / 2)
      block[ib * sY + jb] = 1 ;
    else
      block[ib * sY + jb] = 0 ;
  }

dumpBoard() ;


/* Precompute parameters of shift communications */

Datatype edgeXType = MPI.BYTE.Contiguous(sY) ;
edgeXType.Commit() ;

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

int[][] srcX = new int[2][1];
int[][] dstX = new int[2][1];
int[][] srcY = new int[2][1];
int[][] dstY = new int[2][1];

p.Shift(0,  1, srcX[0], dstX[0]) ;
p.Shift(0, -1, srcX[1], dstX[1]) ;
p.Shift(1,  1, srcY[0], dstY[0]) ;
p.Shift(1, -1, srcY[1], dstY[1]) ;


/* Main update loop. */

int sums [] = new int [blockSizeX * blockSizeY] ;

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

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

  p.Sendrecv(block, blockSizeX * sY, 1, edgeXType, dstX[0][0], 0,
             block, 0,               1, edgeXType, srcX[0][0], 0) ;

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

  p.Sendrecv(block, sY,                     1, edgeXType, dstX[1][0], 0,
             block, (blockSizeX + 1) * sY, 1, edgeXType, srcX[1][0], 0);

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

  p.Sendrecv(block, blockSizeY, 1, edgeYType, dstY[0][0], 0,
             block, 0,          1, edgeYType, srcY[0][0], 0) ;

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

  p.Sendrecv(block, 1,              1, edgeYType, dstY[1][0], 0,
             block, blockSizeY + 1, 1, edgeYType, srcY[1][0], 0) ;


  /*  Calculate block of neighbour sums. */

  for(int i = 0, ib = 1 ; i < blockSizeX ; i++, ib++)
    for(int j = 0, jb = 1 ; j < blockSizeY ; j++, jb++) {
      int ibns = (ib - 1) * sY, ibs = ib * sY, ibps = (ib + 1) * sY ;
      int jbn = jb - 1, jbp = jb + 1 ;
      sums[i * blockSizeY + j] =
```

```
            block[ibns + jbn] + block[ibns + jb] + block[ibns + jbp] +
            block[ibs  + jbn] +                    block[ibs  + jbp] +
            block[ibps + jbn] + block[ibps + jb] + block[ibps + jbp] ;
      }

    /* Update block of board values. */

    for(int i = 0, ib = 1 ; i < blockSizeX ; i++, ib++)
      for(int j = 0, jb = 1 ; j < blockSizeY ; j++, jb++) {
        int ibs = ib * sY ;
        switch (sums[i * blockSizeY + j]) {
          case 2 : break;
          case 3 : block[ibs + jb] = 1; break;
          default: block[ibs + jb] = 0; break;
        }
      }

    dumpBoard() ;
  }

  MPI.Finalize();
}


static void dumpBoard() {

  /* Output current state of board */

  int np = MPI.COMM_WORLD.Size() ;
  int id = MPI.COMM_WORLD.Rank() ;

  if(id == 0) {
    byte board [] = new byte [N * N] ;

    for(int i = 0, ib = 1 ; i < blockSizeX ; i++, ib++)
      for(int j = 0, jb = 1 ; j < blockSizeY ; j++, jb++) {
        int x = blockBaseX + i, y = blockBaseY + j ;
        board[N * x + y] = block[ib * sY + jb] ;
      }

    for(int src = 1 ; src < np ; src++) {
      int params [] = new int [4] ;

      MPI.COMM_WORLD.Recv(params, 0, 4, MPI.INT, src, 0) ;
      int sizeX = params [0] ;
      int sizeY = params [1] ;
      int baseX = params [2] ;
      int baseY = params [3] ;

      /* Get local block from slave. */

      Datatype blockType = MPI.BYTE.Vector(sizeX, sizeY, N) ;
      blockType.Commit() ;

      MPI.COMM_WORLD.Recv(board, N * baseX + baseY, 1, blockType, src, 0) ;
    }

    for(int x = 0 ; x < N ; x++) {
      for(int y = 0 ; y < N ; y++)
        System.out.print(" " + board[N * x + y]) ;
      System.out.println() ;
    }
    System.out.println() ;
  }
  else {
```

```
      int params [] = new int [4] ;
      params [0] = blockSizeX ;
      params [1] = blockSizeY ;
      params [2] = blockBaseX ;
      params [3] = blockBaseY ;

      MPI.COMM_WORLD.Send(params, 0, 4, MPI.INT, 0, 0) ;

      /* Send local block (excluding ghost areas) to master */

      Datatype blockType = MPI.BYTE.Vector(blockSizeX, blockSizeY, sY) ;
      blockType.Commit() ;

      MPI.COMM_WORLD.Send(block, sY + 1, 1, blockType, 0, 0) ;
    }
  }
}
```