Experiments with "HPJava"

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

February 18, 1997

Abstract

We consider the possible röle of Java as a language for High Performance Computing. After discussing reasons why Java may be a natural candidate for a portable parallel programming language, we describe several case studies. These cover Java socket programming, message-passing through a Java interface to MPI, and class libraries for data-parallel programming in Java.

Contents

1 Introduction

The explosion of interest in Java over the last year has been driven largely by its röle in bringing a new generation of interactive pages to the World Wide Web. Undoubtedly various features of the language—compactness, byte-code portability, security, and so on—make it particularly attractive as an implementation language for applets embedded in Web pages. But it is clear that the ambitions of the Java development team go well beyond enhancing the functionality of HTML documents. According to [20]

> "Java is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Paramount among these challenges is secure delivery of applications that consume the minimum of system resources, can run on any hardware and software platform, and can be extended dynamically."

Several of these concerns are mirrored in developments in the High Performance Computing world over a number of years. A decade ago the focus of interest in the parallel computing community was on parallel hardware. A parallel computer was typically built from specialized processors integrated through a proprietary high-performance communication switch. If the machine also had to be programmed in a proprietary language, that was an acceptable price for the benefits of using a supercomputer. This attitude was not sustainable as one parallel architecture gave way to another, and the cost of porting software became exorbitant. For several years now, portability across platforms had been a central concern in parallel computing [4, 5, 14, 13, 23].

More fundamentally, the assumption that high performance computing will be done primarily on specialized supercomputers is questioned increasingly. Rapid strides in performance and connectivity of ordinary workstations and PCs make it look equally possible that the future of parallel computing will be on local area networks, or even the Internet [19, 17].

With Java positioned to become a standard programming language on the Internet, and scientific parallel processing edging towards network-based computation, it is natural to ask how these two technologies will interact. How suitable is Java for scientific computing, and do lessons from research in parallel computing have implications for the development of Java?

Popular acclaim aside, there are some reasons to think that Java may be a good language for scientific and parallel programming.

- Java is a descendant of C++. C and C++ are used increasingly in scientific programming; they are already used almost universally by *implementers* of parallel libraries and compilers. In recent years numerous variations on the theme of C++ for parallel computing have appeared. See, for example [7, 28, 11, 2, 12, 21].

- Java omits various features of C and C++ that are considered "difficult"—notably, pointers. Poor compiler analysis has often been blamed on these features. The inference is that Java, like Fortran, may be a suitable source language for highly optimizing compilers (although direct evidence for this belief is still lacking).

- Java comes with builtin multithreading. Independent threads *may* be scheduled on different processors by a suitable runtime. In any case multithreading can be very convenient in explicit message-passing styles of parallel programming [24].

We will return to the question of whether parallel computing may have implications for the development of Java in section 5.

The acronym "HPJava" was coined in a draft white paper produced by members of the PCRC consortium in the first half of 1996 [10]. At NPAC we have been experimenting with some of the ideas put forward there.

1.1 Overview of this article.

Section 2 outlines various options for parallel programming in Java—possible ways to express parallelism, and ways to handle inter-process communication.

The main technical content of the paper is in sections 3 and 4. Section 3 contains some case studies in which we explore the message-passing style of programming in Java. We cover parallel programming using sockets directly, and describe our Java interface to MPI. In section 4 we discuss approaches to data-parallel programming in Java, and outline one of our demo programs.

In this article our emphasis is more on language bindings and interface issues, and less on performance. Java compilers are in an early stage of development, and we assume that current performance figures are not indicative of future potential.

2 Issues

2.1 Approaches to Parallelism in Java

Java already supports concurrency through the thread mechanism and monitor synchronization built into the language. In this article we are interested in truly *parallel* computation, involving multiple CPUs. Such parallelism could be introduced into Java in a number of ways.

It could be achieved through automatic parallelization of sequential code, but it is unclear why this would be more successful for Java than for other languages. Alternatively, the Java virtual machine for a shared memory multiprocessor can schedule the threads of a multi-threaded Java program on different processors. Some success with these approaches has already been demonstrated [3]. For computation on a network (or distributed memory computer) realistic options include language extensions or directives akin to HPF, or provision of libraries—class libraries—to support task parallelism or data parallelism.

A popular approach in C++ has been to defer language extensions and concentrate on class library support for parallel programming. The similarities between the two languages suggest this may be a fruitful avenue in Java too. The success of this analogy is by no means automatic, however. Features such as *templates* and user-defined *operator overloading* make the C++ language inherently more customizable than Java. In C++ library-defined types can be used on an identical footing to primitive types—*inline* methods mean they can be almost as efficient as primitive types. Less importantly, but conveniently, new control constructs can often be simulated in C or C++ through use of macros[1]. On grounds, presumably, of simplicity and transparency many of these features have been omitted from Java.

Such caveats notwithstanding, this article will concentrate on class libraries rather than language extensions. We will be working with class libraries implemented in the standard Java development environment.

Another open question is how multiple interacting Java tasks are to be initiated. Conventionally in network computing, instructions to execute a particular user task are sent to a specialized daemon or secure server running on the target host. Alternatively standard operating system daemons (*rshd*, *rexecd*, ...) can be used to the same effect. Java brings some options of its own. One is to use standard or enhanced Web servers to execute Java byte-codes. This approach is predicated on the existence of cooperative servers, probably running suitable CGI scripts. Another option is to exploit Java-enabled browsers. Applets downloaded from a particular server can perform computations and return the results to that server. In any case the substantive improvement over conventional network computing is that the byte-codes can be downloaded on the fly to target hosts, even in heterogenous networks. Conventionally, it was often necessary to install and compile an application on each target host before starting the task remotely.

2.2 Communication in Java

The standard Java API provides a simplified interface to Internet sockets. This interface hides much of the ugly detail involved in socket-programming at the at the traditional C/UNIX level. The *java.net* interface provides less flexibility than using the system calls directly. On the other hand, Java's built-in support for threads adds some flexibility in scheduling communications that is missing from raw C.

We will give an example of socket programming in section 3.1, but traditionally this has not been a popular paradigm in the parallel processing world.

[1] Of course we could run a preprocessor over Java, but this is not a natural procedure in existing Java environments

Scientific programmers have expected to program inter-process communication at a higher level, if at all. More successful schemes include

- Message-passing through language-level support [24, 15] or higher-level library interfaces [14].
- Data parallelism. We restrict the definition of "data parallelism" to cover algorithms that achieve parallelism through operations on distributed arrays. Synchronization is usually limited to bulk synchronization occuring naturally through collective operations.
- Communication through shared memory or shared objects. Some more or less intricate mechanism for inter-process synchronization is usually provided.

The case studies in the rest of this article restrict themselves to message-passing and data parallelism. As observed in the previous section, communication through true shared memory is already implicit in the Java thread model. Communication through remote objects is undoubtedly a natural and important paradigm in Java, especially for access to remote services [27, 22, 21], but we will not discuss it further here.

An orthogonal issue is whether the high-level libraries used to implement these paradigms (presuming class-library implementations) should be written in Java on top of the standard API, or whether they should be implemented as direct interfaces to native code. We will have more to say about this later.

3 Message-passing case studies

Message-passing remains one of the most effective and widely used communication paradigms in parallel computing. In this section we compare two approaches to message-passing in Java, in the context of a scientific application. The first approach is to use the socket interface in the standard Java API. The second is to work through a Java interface to the message-passing standard, MPI [14].

To minimize distracting details, our application will be elementary: Conway's Life automaton.

3.1 Java sockets

The UNIX socket model is most suitable for programming client-server applications. Typical scientific parallel programs do not fit directly into this model. Before a SPMD program can start two conditions must obtain: a pool of symmetric peer processes must have been created, and each peer must be able to address a message to any other. This situation is typically bootstrapped as follows. The program is invoked on one host. This host creates a listening socket.

It sends instructions to servers on some other hosts (either explicitly or through a command such as *rsh*) to start remote invocations of the program (somehow sending the number of the port on which it is listening in the initiation message). Each new process creates a listening socket and sends its port number to the original process. The original process broadcasts these address to all its slaves. At this point each process knows the port number on which each of its peers is listening. Either they establish all-to-all connections now, before entering the SPMD main program, or the main program starts immediately, and connections are established dynamically when a message needs to be sent.

Figures 1, 2 give a schematic outline of a distributed Life program using *java.net*. The fairly intricate code sketched above for initialization and establishment of socket connections has been absorbed into the definition of an auxilliary class *hpj*. Its interface is

```
class hpj {
  hpj(String[] args) ;

  int my_id() ;
  int num_processor() ;

  DataInputStream Input(int id) ;
  DataOutputStream Output(int id) ;
  ...
}
```

The members *Input* and *Output* return streams associated with sockets connected to peer processes. In the example an N by N Life board is divided blockwise in one dimension, each processor holding a local block of width *blockSize*.

We note

- Initialization is a complex procedure and clearly it should not be coded anew for each application program.
- In this example the messages were contiguous byte vectors that could be transmitted efficiently through the *read* and *write* methods of the Java socket API. In general the messages will have more complex types and the data may not be contiguous in memory. Using the typed primitives of the standard API may then incur extra costs of copying and type-conversion.

For reasons such as these we suspect that direct socket programming will remain unattractive to scientific parallel programmers, even with the simplified Java socket API.

```
class life_java {
  static public void main(String[] args) throws Exception {
    hpj HPJava = new hpj(args);

    int np = HPJava.num_processor();
    int id = HPJava.my_id();

    ... compute local ‘blockSize’, ‘blockBase’ (avoiding empty blocks).

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

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

    ... initialize local block with some pattern

    // Main update loop.

    int next = (id + 1) % np;
    int prev = (id + np - 1) % np;

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

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

      HPJava.Output(next).write(block[blockSize]);
      HPJava.Output(next).flush();
      HPJava.Input(prev).read(block[0]);

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

      HPJava.Output(prev).write(block[1]);
      HPJava.Output(prev).flush();
      HPJava.Input(next).read(block[blockSize+1]);

      ... Calculate a block of neighbour sums.
      ... Update block of board values.
    }
  }
}
```

Figure 1: Skeleton of socket-based Life program.

```
int sums[][] = new int[blockSize][N];

....

  //  Calculate block of neighbour sums.

  for(i = 0, ib = 1 ; i < blockSize ; i++, ib++)
    for(y = 0 ; y < N ; y++) {
      int y_n = (y+N-1) % N ;
      int y_p = (y+1) % N ;
      sums[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(i = 0, ib = 1 ; i < blockSize ; i++, ib++)
    for(y = 0 ; y < N ; y++)
      switch (sums[i][y]) {
        case 2 : break;
        case 3 : block[ib][y] = 1; break;
        default: block[ib][y] = 0; break;
      }
```

Figure 2: Socket-based Life program (detail).

3.2 MPI Interface

We have produced a Java interface to an existing MPI implementation [25] using Java *native methods*[2]

The interface has been tested on a cluster of UltraSparc workstations running Solaris[3]. Our interface is modelled on the proposed C++ bindings of MPI. For example, many of the most basic functions of the library are members of the communicator class, *Comm*:

```
public class Comm {
  public int Size();
  public int Rank();

  void Send(Object buf, int offset, int count,
            Datatype datatype, int dest, int tag) ;

  Status Recv(Object buf, int offset, int count,
              Datatype datatype, int dest, int tag) ;
  ...
}
```

Figure 3 is a straightforward transcription of the socket-based program in the last section[4]. Our MPI inteface uses a slightly different model for accessing global resources—static members on an *MPI* class:

```
class MPI {
  static Init() ;
  static Finalize() ;

  public static Comm WORLD ;

  public static Datatype BYTE ;
  public static Datatype INT ;
  ...
}
```

rather than dynamic members of a *jpi* class—but this difference is not particularly significant (yet another approach will be taken in the example of section 4.1). Otherwise the correspondence between this code and the socket code is

[2]For
earlier work on interfacing Java to PVM see http://www.isye.gatech.edu/chmsr/JavaPVM/ and http://www.cs.virginia.edu/ãjfzj/jpvm.html

[3]Interfacing Java to MPICH/P4 was not straightforward, due to unpleasant interactions between the Java run-time and the underlying P4 implementation. For example, standard implementations of both use the UNIX *SIGALRM* signal. We found it necessary to patch the MPICH 1.0.13 release to work round these incompatibilities. The necessary patches are available from us on request.

[4]MPI afficionados will note that the use of standard mode *send* is "unsafe", and could deadlock if the system does not provide enough buffering. The same remark could be made about the socket-based version

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

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

    ... compute local ‘blockSize’, ‘blockBase’ (avoiding empty blocks).

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

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

    ... initialize local block with some pattern

    // Main update loop.

    int next = (id + 1) % np;
    int prev = (id + np - 1) % np;

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

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

      MPI.WORLD.Send(block[blockSize], N, MPI.BYTE, next, 0);
      MPI.WORLD.Recv(block[0],         N, MPI.BYTE, prev, 0);

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

      MPI.WORLD.Send(block[1],             N, MPI.BYTE, prev, 0);
      MPI.WORLD.Recv(block[blockSize + 1], N, MPI.BYTE, next, 0);

      ... Calculate block of neighbour sums.
      ... Update block of board values.
    }

    MPI.Finalize();
  }
}
```

Figure 3: Simple MPI Life program.

direct. In the next section we illustrate some of the added value that an MPI interface brings.

We have broken with the usual MPI convention of returning an error status from every function. This practice is inconvenient in Java because arguments cannot be passed by reference and directly modified. This makes the return value precious, and using it up on an error value that everybody ignores is a waste. Java has a well established exception mechanism for ignoring error information.

3.3 Derived data types and higher-level MPI features

Description of the data buffers passed to communication operations presents some special problems in Java. Existing MPI bindings depend on a linear memory model and explicit or implicit use of pointers. Java does not have a linear memory model. Even behind the scenes a Java array has no uniquely defined address in memory, because the garbage collector is allowed to relocate objects unpredicatably during program execution to avoid fragmentation of its workspace. Our Java interface tries to retain as much of the MPI derived datatype mechanism as practical, but some functionality has been sacrificed. The buffer argument passed to a send or receive operation must be a one-dimensional array of primitive type. Any offset specified in a derived type argument then refers to a displacement within this one-dimensional array, never a displacement in memory.

All MPI derived types expressible through our interface have a uniquely defined *base type*—a Java primitive type. Interfaces to MPI_TYPE_HVECTOR and MPI_TYPE_HINDEXED are provided, but the strides and displacements are in units of the base type, not bytes. An interface to MPI_TYPE_STRUCT is provided, but all component types in the "struct" must have the same base type.

In the concrete Java binding of the *send* function, for example,

```
void Send(Object buf, int offset, int count,
          Datatype datatype, int dest, int tag) ;
```

the formal *buf* argument is presented as a generic Java *Object*. As explained above, the actual argument must be a linear array. The second argument is the offset in this array of the first element of the message[5]. The remaining arguments correspond directly to arguments of MPI_Send. The base type of the *datatype* argument must be the type of the elements of *buf*.

Figures 4, 5 sketch a version of the Life program illustrating several of these features. As well as derived types, this program uses the Cartesian topologies of MPI. The *Cart* class is derived from *Comm*. In the example, the topology

[5] This offset is in units of the *buf* array element—or the base type of *datatype*—not of any compound type. The *Object* + *offset* presentation is reminiscent of the interface of the *arrayCopy* utility in the standard Java API.

```
class Life {
  void main(String args) {
    MPI.Init(args) ;

    int dims [] = new int [2] ;
    ... Set `dims', etc
    Cart p = new Cart(MPI.WORLD, dims, periods, false) ;

    int coords = new int [2] ;
    p.Get(dims, periods, coords) ;
    ... Compute local `blockSizeX', `blockBaseX', `blockSizeY', `blockBaseY'.

    // Create `block', allowing for ghost cells.

    int sX = blockSizeX + 2 ;
    int sY = blockSizeY + 2 ;
    block = new byte [sX * sY] ;
    ... Define initial state of Life board

    // Precompute parameters of shift communications.

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

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

    CartShift pX = p.shift(0,  1) ;
    CartShift nX = p.shift(0, -1) ;
    CartShift pY = p.shift(1,  1) ;
    CartShift nY = p.shift(1, -1) ;

    // Main update loop.

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

      ... Calculate block of neighbour sums.
      ... Update block of board values.
    }

    MPI.Finalize();
  }
  ...
}
```

Figure 4: Life program using full MPI.

```
// Execute shifts...

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

p.Sendrecv(block, blockSizeX * sY, 1, edgeXType, pX.dst, 0,
           block, 0,              1, edgeXType, pX.src, 0) ;

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

p.Sendrecv(block, sY,                    1, edgeXType, nX.dst, 0,
           block, (blockSizeX + 1) * sY, 1, edgeXType, nX.src, 0) ;

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

p.Sendrecv(block, blockSizeY, 1, edgeYType, pY.dst, 0,
           block, 0,          1, edgeYType, pY.src, 0) ;

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

p.Sendrecv(block, 1,              1, edgeYType, nY.dst, 0,
           block, blockSizeY + 1, 1, edgeYType, nY.src, 0) ;
```

Figure 5: Full MPI Life program (detail).

p represents a two dimensional periodic grid of processes. The *Get* member returns the coordinates of the local process. From these the parameters of the local array block are computed.

The values sX, sY represent the sides of the locally held array segment, including ghost regions. This segment is created as a one-dimensional Java array, *block*. The derived type *edgeXType* describes the structure of ghost area on the upper or lower x sides: contiguous regions of the *block* array of extent sY. The type *edgeYType* describes the y-side ghost areas: non-contiguous regions of count sX, regular stride sY.

The *shift* member of *Cart* corresponds to the MPI function MPI_CART_SHIFT: it returns the source and destination processors for a cyclic shift. The Java binding returns these values in an object of class *CartShift* which just contains two integers. Finally, in the main loop, the shifts are executed by using the *Comm* member *Sendrecv*, which corresponds to the standard MPI function MPI_Sendrecv. This performs a send and a receive concurrently (avoiding a potential deadlock in the implementations given in the previous sections).

4 Data parallellism in Java

The most comprehensive statement of the data parallel model of computation is the High Performance Fortran standard [13, 23]. That document is supposed to embody much of the collective experience of the scientific parallel programming community. Presumably, then, any attempt to incorporate data parallelism into Java should build on the HPF model wherever possible.

The HPF definition consists of a large set of directives that can be used to annotate a standard Fortran program, a small handful of language extensions, and a library of new functions for operating on arrays. An initial data-parallel Java may well be implemented through a class-library. This library would assume the röles of the directives and language extensions in HPF as well as the HPF library.

We will loosely distinguish two different levels at which a library implementation of the HPF semantics can operate.

- The first is the level of the so-called *run-time libraries* [1, 8, 9, 6]. This kind of library provides functions for scheduling and executing specific patterns of collective communication already identified by a compiler (in the HPF case) or else by an application programmer using the library directly. Such a library may also provide functions for translating between global subscripts and local, node-level subscripts—ie, for computing the mapping of a distributed array into the address spaces of individual processors.

- Alternatively, a library can operate at a higher level that conceals all aspects of data localization and transfer from the user. The only responsibility of the user is to specify the distribution format of arrays when they are declared. Subsequently the user just tells the library to do particular operations on particular distributed arrays. It is left to the library to work out whether or not a communication is implied. In effect the library is operating at the same level as the HPF language. An example of such a library is A++/P++ [26].

In either case a *class library* version is likely to include classes to describe the elements of the HPF data model, such as processor arrangements and the distributed arrays themselves.

4.1 Parallel arrays and collective communication

At the run-time level, a class library implementation of the HPF model is likely to include

- Classes to describe process arrays and distributed data arrays.

- Classes or functions to simplify access to locally held elements of a distributed array (including parallel iteration).

- Functions for collective communication through operations on distributed arrays: regular "copying" operations including shifts and transposes, arithmetic reduction operations, irregular gather/scatter operations, and so on.

Our first experiments with a Java binding only touch the surface of the full HPF semantics, but they provide some hints about a general framework. The interface given here borrows from the C++ class library, Adlib, developed by one of us [6].

A distributed array is parametrized by a member of the *Array* class. In C++ *Array* would naturally be a *template* for a *container class*. In Java, generic container classes are problematic. Without the template mechanism, the obvious options are that a container holds items of type *Object*, the base class for all non-primitive types, or that a separate container class is provided for each allowed type of element. The first option doesn't allow for array elements of primitive type, and prevents compile-time type-checking (reminiscent of using *void** in C). The second approach presumably involves restricting elements to the finite set of primitive types (*int*, *float*, ...)[6]. For now we have side-stepped the issue by leaving the data elements out of the *Array* class. *Array* defines the shape and distribution of an array, but space for elements is allocated in a separately declared vector of the appropriate type[7].

The constructor for an *Array* defines its shape and distribution format. This is expressed through two auxilliary classes: the *Procs* and *Range* classes. The *Procs* class corresponds directly to the HPF *processor arrangement*. It maps the set of physical processes on which the program is executing to a multi-dimensional grid. A *Range* describes a single dimension of an HPF array. It embodies an array extent (the size of the array in the dimension concerned), and a mapping of the subscript range to a dimension of a *Procs* grid.

In our pilot implementation any parallel Java application is written as a class extending the library class *Node*. The *Node* class maintains some global information and provides various collective operations on arrays as member functions. The code for the "main program" goes in the *run* member of the application class[8].

A simplified version of the code for the "Life" demo is given in figure 6. The object p represents a 2 by 2 process grid. The *Procs* constructor takes the current *Node* object as an argument, from which it obtains information on the available physical processes. In this simplified example we assume that the program executes on exactly four processors.

The objects x and y represent index ranges of size N 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).

[6]Perhaps a good compromise is to provide one container class for each primitive type and one for *Object*.

[7]Confusingly enough, this makes our *Array* more akin to an HPF *template* than an HPF array. Needless to say, there is no connection between C++ templates and HPF templates.

[8]This approach is modelled on the *Thread* and *Applet* classes in the standard Java API. Other approaches to providing library-wide resources were illustrated in earlier sections.

```
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 s = r.seg();
    byte[] w = new byte[s];

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

    // Initialize the Life board

    for(r.forall(); r.test(); r.next())
      w[r.sub()] = fun(r.idx(0), r.idx(1)) ;

    // 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(r.forall(); r.test(); r.next()) {
        int i = r.sub() ;
        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 6: Simplified code of the Life demo program.

The object r represents the shape and distribution of a two dimensional array. This "template" is be shared by several distributed arrays—it does not contain a data vector. 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 of neighbour values[9].

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 internal state of r so that *r.sub()* returns the local subscript for the current iteration, and *r.idx(0)* and *r.idx(1)* return the global index values for the current iteration. We are using r as an *iterator* class[10].

The main loop uses cyclic *shift* operations to obtain neighbours, communicating data where necessary. The *shift* operation is a member of the *Node* class. It should be overloaded to accept data vectors of any primitive type—here the array elements are *bytes*.

Finally w is updated in terms of its neighbours.

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.

It may be unclear that this framework has the same power as HPF. If the distributed array model is extended to the full HPF model, as it can be, and the *shift* operation is augmented by some more powerful collective operations including gather and scatter operations parametrized by subscript arrays, we claim that it does. The proof lies in the observation that this is, as advertised, a simplified model of the kind of run-time library that various HPF translators target. This is not to say that programming in this style is always as straightforward as the example given here, or as comprehensible as the corresponding HPF program (if it was, there would be no need for HPF).

[9] Here we will use whole arrays of neighbours and a *shift* operation. This is arguably the more conventional approach in a data-parallel setting, but the the ghost-edge mechanism can also be fitted into this framework.

[10] Our *Array* class is perched somewhere between STL container and iterator classes. This is a slightly awkward position, and it may be more satifactory to separate these functions into different classes.

4.2 “Array syntax” in Java.

The higher level approach would make *Array* classes look like true container classes (for a restricted set of types) and *all* operations on arrays collective, something like:

```
ArrayFloat a = new ArrayFloat(p, x, y) ;
ArrayFloat b = new ArrayFloat(p, x, y) ;
ArrayInt c = new ArrayInt(p, x, y) ;

a = MATMUL(b, c) ;
```

Communication would be handled automatically inside array operations like MATMUL. Individual array elements would not be accessed in the Java program except, possibly, through *getElement*, *putElement* members.

This scheme can be implemented on top of an SPMD Java array library of the kind outlined in the previous section *or* by making the Java run as a coordination program controlling a parallel back end. It can be compared with [26, 16].

So far we have not attempted to implement (or specify in detail) such an approach for Java. The lack of user-defined operator-overloading may be particularly frustrating here.

5 Discussion

We have explored the practicality of doing parallel computing in Java, and of providing Java interfaces to High Performance Computing software. For various reasons, the success of this exercise was not a foregone conclusion. Java sits on a virtual machine model that is significantly different to the hardware-oriented model which C or Fortran exploit directly. Java discourages or prevents direct access to the some of the fundamental resources of the underlying hardware (most extremely, its memory).

Our earliest experiments in this direction (including the work described in section 4, which predates the MPI work) involved working entirely within Java, building new software on top of the communication facilities of the standard API. The more recent work in sections 3.2 and 3.3 involved creating a Java interface to an existing HPC package. Which is the better strategy? In the long term Java may become a major implementation language for large software packages like MPI. It certainly has advantages in respect of portability that could simplify implementations dramatically. In the immediate term recoding these packages does not appear so attractive. Java wrappers to existing software look more sensible. On a cautionary note, our experience with MPI suggests that interfacing Java to non-trivial communication packages may be less easy than it sounds. Nevertheless, we intend in the future to create a Java interface to an existing run-time library for data parallel computation.

So is Java, as it stands, a good language for High Performance Computing?

It still has to be demonstrated that Java can be compiled to code of efficiency comparable with C or Fortran. Many avenues are being followed simultaneously towards a *higher performance* Java. Besides the Java chip effort of Sun, it has been reported at this workshop that IBM is developing an optimizing Java compiler which produces binary code directly, that Rice University and Rochester University are working on optimization and restructuring of bytecode generated by *javac*, and that Indiana University is working on source restructuring to parallelize Java. Parallel interpretation of bytecode is also an emerging practice. For example, the IBM JVM, an implementation of JVM on shared memory architectures, was released in spring 1996, and UIUC has recently started work aimed at parallel interpretation of Java bytecode for distributed memory systems.

Another promising approach under investigation [18] is to integrate interpretation and compilation techniques for parallel execution of Java programs. In such a system, a partially ordered set of *interpretive frames* is generated by an *II/CVM compiler*. A frame is a description of some subtask, whose granularity may range from a single scalar assignment statement to a solver for a system of equations. Under supervision of the virtual machine (II/CVM), the actions specified in a frame may be performed in one of three ways:

- Executed by an *interpretive module* directly, which also incorporates JIT compilation capability.
- Some precompiled *computational library* function is invoked locally to accomplish the task; this function may be executed sequentially or in parallel.
- The frame is sent to some *registered remote system*, which will get the work done, once again either sequentially or in parallel.

With this approach, optimized binary codes for *well formed* computation subtasks exist in runtime libraries, supporting a high level interpretive environment. Task parallelism is observed among different frames executed by the three mechanisms simultaneously, while data parallelism is observed in the execution of some of the runtime functions.

Presuming these efforts satisfactorily address the performance issue, the second aspect in question concerns expressiveness of the Java language. Our final interface to MPI is quite elegant, and provides much of the functionality of the standard C and Fortran bindings. But creating this interface was a more difficult process than one might hope, both in terms of getting a good specification, and in terms of making the implementation work. In section 4 we noted that the lack of features like C++ templates (or any form of parametric polymorphism) and user-defined operator overloading (available in many modern languages, from functional programming languages to Fortran) made it difficult to produce a completely satisfying interface to a data parallel library. The Java language as currently defined imposes various limits to the creativity of the programmer.

In many respects Java is undoubtedly a better language than Fortran. It is object-oriented to the core and highly dynamic, and there is every reason to suppose that such features will be as valuable in scientific computing as in any other programming discipline. But to displace established scientific programming languages Java will surely have to acquire some of the facilities taken for granted in those languages.

References

[1] A. Agrawal, A. Sussman, and J. Saltz. An integrated runtime and compile-time approach for parallelizing structured and block structured applications. *IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

[2] Susan Atlas, Subhankar Banerjee, Julian C. Cummings, Paul J. Hinker, M. Srikant, John V. W. Reynders, and Marydell Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. In *Supercomputing '95*, 1995.

[3] Aart J.C. Bik and Dennis B. Gannon. Automatically exploiting implicit parallelism in Java. This workshop.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, 1987.

[5] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[6] D. B. Carpenter. Adlib: A distributed array library to support HPF translation, 1995. Presented at the 5th International Workshop on Compilers for Parallel Computers. URL: http://www.npac.syr.edu/users/dbc/Adlib.

[7] K.M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, page 24. MIT Press, 1993. ISBN: 0-262-01139-5.

[8] A. Choudhary, G. Fox, S. Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, and J. Saltz. Software support for irregular and loosely synchronous problems. *Computing Systems in Engineering*, 3:43–52, 1992.

[9] Parallel Compiler Runtime Consortium. Common runtime support for high-performance parallel languages. In *Supercomputing '93*. IEEE Computer Society Press, 1993.

[10] Parallel Compiler Runtime Consortium. HPCC and Java—a report by the Parallel Compiler Runtime Consortium, 1996. URL: http://www.npac.syr.edu/users/gcf/hpjava3.html.

[11] J.J. Dongarra, R. Pozo, and D.W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Object Oriented Numerics Conference*, 1993.

[12] Stephen J. Fink and Scott B. Baden. *The KeLP User's Guide*. University of California, San Diego, La Jolla, CA, March 1996. URL: http://www-cse.ucsd.edu/groups/hpcl/scg/kelp.html.

[13] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, special issue, 2, 1993.

[14] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. University of Tenessee, Knoxville, TN, June 1995. URL: http://www.mcs.anl.gov/mpi.

[15] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24, 1995.

[16] G.C. Fox and W. Furmanski. Towards interpreted run-time Fortran90D environment. Technical report, NPAC, 1992. URL: http://www.npac.syr.edu/projects/hpsin/hpfi.html.

[17] Geoffrey C. Fox and Wojtek Furmanski. Computing on the Web: new approaches to parallel processing—petaop and exaop performance in the year 2007, 1997. URL: http://www.npac.syr.edu/users/gcf/petastuff/petaweb/.

[18] Geoffrey C. Fox, Xiaoming Li, Yuhong Wen, and Guansong Zhang. Studies of integration and optimization of interpreted and compiled languages. Technical Report SCCS-780, NPAC, February 1997.

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Series. MIT Press, 1994. ISBN: 0-262-57108-0.

[20] James Gosling and Henry McGilton. *The Java Language Environment: A White Paper*. JavaSoft, Sun Microsystems, Inc, 1996. URL: http://www.javasoft.com/doc.

[21] A.S. Grimshaw. An introduction to parallel object-oriented programming with Mentat. Technical Report 91 07, University of Virginia, 1991.

[22] JavaSoft, Sun Microsystems, Inc. *RMI Documentation*, 1996. URL: http://java.sun.com/products/JDK/1.1/.

[23] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steel, Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994. ISBN: 0-262-61094-9.

[24] Inmos Ltd. *occam 2 Reference Manual*. Prentice-Hall International, 1988. ISBN: 0-13-629312-3.

[25] MPICH—a portable implementation of MPI. URL: http://www.mcs.anl.gov/mpi/mpich/.

[26] R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference calculations. In *Object Oriented Numerics Conference*, 1994.

[27] Jon Siegel. *CORBA Fundamentals and Programming*. Wiley, 1996. ISBN: 0471-12148-7.

[28] P. Sivilotti and P. Carlin. A tutorial for CC++. Technical Report CS-TR-94-02, Caltech, 1994.