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. He also expect to obtain better efficiency.

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.

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 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 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

```
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.

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.

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.

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 run-time 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.

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.

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.

[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.

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 passing their ports to suitable members. A blocking `select` operations returns a handle to a port from the pool which presently has input data ready. Merge pools enable non-deterministic patterns of communication.

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 less actively than the data parallel approach.