Java as a Language for Scientific Parallel Programming

Bryan Carpenter, Yuh-Jye Chang, Geoffrey Fox, Xiaoming Li

Northeast Parallel Architectures Centre,

Syracuse University,

Syracuse, New York

Why Java?

From the Java whitepaper:

> *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.*

Goals of Java include

- Portability (cf, P4, MPI, HPF,. . .)
- Support for heterogenous networks (cf, PVM, MPI,. . .)
- Efficiency.

all major concerns in HPC.

Why Java?

- Java is a descendant of C++—used increasingly in scientific programming. Many variations on the theme of C++ for parallel computing (eg, CC++, pC++, A++/P++, . . .).
- Java omits features of C and C++ considered "difficult"—eg, pointers. Suggests Java (like Fortran) may be a suitable source language for highly optimizing compilers.
- Java comes with builtin multithreading. On a shared memory platform threads may be scheduled on different processors. In any case, very convenient in explicit message-passing styles (cf occam, Fortran M, . . .)

Approaches to Parallelism in Java

- Explicit parallelism through Java thread mechanism, scheduled on SMP.
- Automatic or semi-automatic parallelization (eg JAVAR).
- Language extensions (eg SPAR, pizza) or directives?
- Task parallelism. Applets as tasks (or task servers—eg Javelin)? Interacting "applications" as tasks?

Approaches to communication in Java

- Message-passing. Internet sockets? Higher-level class libraries—Channels? MPI?
- Communication through shared objects (Nexus, Linda, . . .).
- Distributed shared memory (Java/DSM).
- Data parallelism. High level "array syntax" with implicit communication (cf A++/P++)? Global SPMD style—explicit collective operations on distributed array classes (cf Adlib)?
- Libraries in Java? Libraries in native code?

This talk will focus on traditional message-passing and data-parallel approaches.

Part 1. Message passing in Java

Approaches investigated

- Application programming with `java.net` sockets.
- Fortran-M-like dynamic channels, implemented as a class library on top of `java.net`.
- "Native-methods" interface to standard MPI implementation.

Will only discuss the last approach.

MPI Interface

Have produced an experimental *native methods* interface to (subset of) MPI.

Interface modelled on proposed C++ bindings of MPI, eg

```
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) ;
  ...
}
```

Simple message passing

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

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

byte block[][] = new byte [blkSize + 2] [N] ;
...

  // Shift upper edge into next's lower ghost edge

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

  // Shift lower edge into prev's upper ghost edge

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

Derived data types

Description of user data buffers presents special problems in Java. Existing MPI bindings assume a flat, linear memory model.

MPI derived types expressible in Java binding have a uniquely defined *base type*—a Java primitive type.

- Interfaces to MPI_TYPE_HVECTOR and MPI_TYPE_HINDEXED provided, but strides and displacements in units of base type, not bytes.
- Interface to MPI_TYPE_STRUCT provided, but all component types must have same base type.

Send and receive buffers

Interface to standard mode send is

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

- Formal `buf` argument presented as a generic Java *Object*. Actual argument must be 1d array of elements of primitive type—the base type of `datatype`.
- `offset` argument is the offset in `buf` of first element of the message (cf `arrayCopy` in standard Java API).
- Other arguments as for conventional `MPI_Send` bindings.

Programming with derived data types

```
int sX = blkSizeX + 2, sY = blkSizeY + 2 ;
block = new byte [sX * sY] ;

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

Datatype edgeYType = MPI.BYTE.Vector(sX, 1, sY) ;
...
  p.Sendrecv(block, blkSizeX * sY, 1, edgeXType, pX.dst, 0,
             block, 0,             1, edgeXType, pX.src, 0) ;

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

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

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

Availability of “Java wrapper” for MPI

A Solaris version of the software, requiring MPICH/P4, is available from

`http://www.npac.syr.edu/users/yjchang/javaMPI`

This software has also be ported to the IBM/SP2.

Prototype software, made available in the hope of stimulated further development—not as a finished product!

Part 2. Data Parallelism in Java

Work to date based on a simplified version of the distributed data model of HPF.

Rather than language extensions, have taken a pure class library approach (cf A++/P++, HPC++ parallel STL, Adlib, . . .)

Take a relatively low-level approach—essentially SPMD, with classes for distributed arrays, plus collective communications.

Problems with container classes

Lack of parametric polymorphism (in C++, templates) makes container classes difficult. Rules out parallel STL approach.

Options:

- Container holds items of type `Object`. But then no array elements of primitive type; no compile-time type-checking (cf `void*` in C).
- Separate container class for each allowed type of element. Ad hoc and unnatractive.

Another worry: lack of operator overloading makes access to contents of container verbose.

Distributed Array Descriptor approach.

For now we have taken a lower level approach. Provide array descriptor (DAD) class, but allocate space for local elements in separate vector

```
Procs p = new Procs(this, 2, 2) ;

Range x = new Range(N, p, 0) ;    // distrib over 1st dim
Range y = new Range(N, p, 1) ;    // distrib over 2nd dim

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

byte [] w = new byte [r.seg()] ;  // main data array
```

`seg()` returns number of elements in local segment.

Programming with parallel arrays.

Fragment of a “life” demo program:

```
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;
  }
}
```

Neighbour arrays `cn_`, . . . allocated similarly to `w`.

Data-parallel example: comments

Characteristics of the data-parallel style of programming:

- Distribution format of arrays can be changed by altering a few parameters at start of program (eg, change `Range` to `CRange` for cyclic distribution). Main program insensitive to these details.
- Low level message-passing abstracted into high-level collective operations on distributed arrays.

Data-parallel example: lessons

- Need much more general collective operations than `shift` to make this idiom useful.
- Absence of templates and operator overloading make implementing arrays through a library less attractive than in C++.
- Using iterator functions `forall`, `test`, `next` to implement distributed loops is elegant but inefficient. Adds multiple procedure calls per iteration—in a compiled language perhaps one to two orders of magnitude performance hit for tight loops.

Future work

1. We are making a Java interface to the NPAC PCRC library. This provides array descriptors for the complete HPF 1.0 array model and very general collective communications and array intrinsics.
2. Plan a Java-to-Java translator adding minimal data parallel language extensions—parallel arrays and efficient implementation of the distributed loop.

Hope to provide a useful data-parallel programming environment for Java. Will also be a proving ground for general ideas on "global SPMD" language extensions.