Towards a Java Environment for SPMD Programming

Bryan Carpenter, Guansong Zhang, Geoffrey Fox,
Xiaoming Li, Xinying Li, Yuhong Wen

Northeast Parallel Architectures Centre,
Syracuse University,
Syracuse, New York

{**dbc**,zgs,gcf,lxm,xli,wen}@npac.syr.edu

Introduction

- HPJava is an environment for SPMD (Single Program, Multiple Data) parallel programming.
- Acronym "HPJava" coined in a draft white paper produced by members of the PCRC consortium early in 1996.
- Eventually incorporate bindings for general MIMD programming libraries, and preprocessor for an extended dialect of Java: recently released first component—**mpiJava**.
- This talk emphasizes **language** extensions.

Language issues

- Widely assumed that data-parallel programming has an important role in HPC.
- "Production" parallel computing still dominated by low-level SPMD programming. So far HPF has made limited inroads.
- HPJava takes a middle road between HPF and SPMD programming, combining some data parallel language features of HPF with free use of established SPMD libraries for distributed arrays.

Advantages of HPF

- Easy to understand semantics: equivalent to conventional Fortran program. Directives only change performance.
- Parallelism can be expressed explicitly through whole-array operations of Fortran 90/95.
- Performance can be tuned by changing a few directives to alter the distribution format of arrays.

Problems with HPF

- Promise of semantic equivalence between an HPF program and a sequential Fortran program leads to difficult implementation problems.
- Works best for algorithms that have essentially SIMD parallelism—identical updates across whole arrays. Many algorithms easier to express with looser coupling between updates of the local array elements. End up doing contortions.

HPJava Goals

Aim for a flexible hybrid of the data parallel and SPMD approaches, combining some established technologies for parallel processing.

Incorporates two complementary sectors:

- Syntax extensions for description of, and access to elements of, distributed arrays.
- Class library bindings for various communication paradigms.

Language Features

- Distributed array model semantically equivalent to to the HPF array model.
- Add **distributed control** constructs. HPF 2.0 has comparable mechanisms (`ON HOME`) as **optional** directives.
- Switch between data-parallel and low-level SPMD programming is a change in point of view—no **extrinsic interface** barrier.

HPJava semantics

- Semantics influenced by ad++, a C++ user-level interface to NPAC/Southampton runtime library.
- HPJava introduces a more convenient syntax, and an intelligent preprocessor to optimize distributed array subscripting.
- **Unlike HPF**, underlying model is simply SPMD programming. Other libraries for manipulating distributed arrays can be integrated into the same framework.

Communication Libraries

Ultimately would like, say:

- Regular collective (Adlib-like)
- Irregular collective (CHAOS-like)
- One-sided access to array sections (GA-like)
- Low-level SPMD (MPI)
- Bindings to higher-level application-oriented libraries (DAGH, ScaLAPACK, . . .)

Initially will have Adlib and MPI.

Distributed Arrays

- Distributed arrays flagged by double brackets. Unrelated to ordinary Java arrays.
- Distribution expressed in terms of `Group` (subclass `Procs2`) and `Range` (subclass `BlockRange`).
- Comparable with HPF processor arrangements and templates.

Distributed array construction

Example:

```
Procs2 p = new Procs2(3, 2) ;

Range x = new BlockRange(100, p.dim(0)) ;
Range y = new BlockRange(200, p.dim(1)) ;

float [[,]] a = new float [[x, y]] on p ;
```

Accessing elements: matrix example

```
on(p) {
  float [[,]]  c = new float [[x, y]] ;

  float [[,*]] a = new float [[x, N]] ;
  float [[*,]] b = new float [[N, y]] ;

  ... initialize 'a', 'b'

  overall(i = x)
    overall(j = y) {

      float sum = 0 ;
      for(int k = 0 ; k < N ; k++)
        sum += a [i, k] * b [k, j] ;

      c [i, j] = sum ;
    }
}
```

Matrix multiplication—details

- Matrix `c` distributed in blocks.
- Processes contain complete rows of `a`, replicated over horizontal process dimension.
- Processes contain complete colums of `b`, replicated over vertical process dimension.
- For arbitrary distribution pattern of input data, could initialize `a`, `b` by

```
Adlib.remap(a, a_in) ;
Adlib.remap(b, b_in) ;
```

The **on** construct

- In SPMD programming, often necessary to restrict execution of a block of code to processors in a particular **active process group**, `p`.
- HPJava provides a special construct for this:

```
on(p) {
  ...
}
```

Subscripting and locality of reference

- Integer subscripts in distributed dimensions of arrays disallowed.

- A **location** is an abstract element, or "slot", of a distributed range.

- Distributed dimensions of arrays can **only** be subscripted using locations in the appropriate range of the array, scoped by `at` or `overall` constructs.

The at construct

- Analogous to **on**, except that its body is executed only on processes that hold the specified location.
- Construct scopes a named location, which can be used for subscripting:

```
at(i = x [17])
  at(j = y [23])
    a [i, j] = 13 ;
```

Distributed loops

- Last and most important distributed control construct is called `overall`. Implements a distributed parallel loop.
- Due to rules on use of subscripts, `overall` can usually only combine elements of arrays with some simple alignment relation.

Use of `overall`

Example:

```
float [[,]] a = new float [[x, y]],
            b = new float [[x, y]] ;

overall(i = x)
  overall(j = y)
    a [i, j] = 2 * b [i, j] ;
```

Locality of reference

- General rule: *a variable can only be accessed if the active process group is a subset in the home group of the variable.*
- In many cases, this rule is enforced by scope of scalar variables and syntactic and semantic constraints on subscripting.
- In other cases, run-time checks can be added.

Coherence

- Advisory rule: *a variable should only be updated if the active process group is identical to the home group of the variable.*
- Applied globally, this rule will ensure that values in variables are replicated consistently over all processes that hold a copy.

Translation

- HPJava translator is implemented as a source-to-source translator, converting HPJava program to a Java node program.
- Initial implementation uses an ad++-like interface to Adlib, called **adJava**, as runtime support for distributed arrays.
- Generate Java classes representing each kind of distributed array used in the program, on-the-fly.

Examples

Two examples follow:

- Cholesky decomposition
- Potts update

Choleski decomposition

In pseudocode, sequential algorithm is

$$
\begin{array}{l}
For\ k = 1\ to\ n - 1 \\
\quad l_{kk} = a_{kk}^{1/2} \\
\quad For\ s = k + 1\ to\ n \\
\quad\quad l_{sk} = a_{sk}/l_{kk} \\
\quad For\ j = k + 1\ to\ n \\
\quad\quad For\ i = j\ to\ n \\
\quad\quad\quad a_{ij} = a_{ij} - l_{ik}l_{jk} \\
l_{nn} = a_{nn}^{1/2}
\end{array}
$$

Note HPJava code should be enclosed by a suitable `on(p)`.

```
Range x = new CyclicRange(N, p.dim(0));

float [[*,]] a = new float [[N, x]] ;
float [[*]]  b = new float [[N]] ;  // buffer

for(int k = 0 ; k < N - 1 ; k++) {

  at(l = x [k]) {
    float d =  Math.sqrt(a [k, l]) ;
    a [k, l] = d ;
    for(int s = k + 1 ; s < N ; s++)
      a [s, l] /= d ;
  }

  Adlib.remap(b [[k + 1 : ]], a [[k + 1 : , k]]);

  overall(l = x [k + 1 : ])
    for(int i = x.idx(l) ; i < N ; i++)
      a [i, l] -= b [i] * b [x.idx(l)] ;
}

at(l = x [N - 1])
  a [N - 1, l] = Math.sqrt(a [N - 1, l]) ;
```

Potts update pseudocode

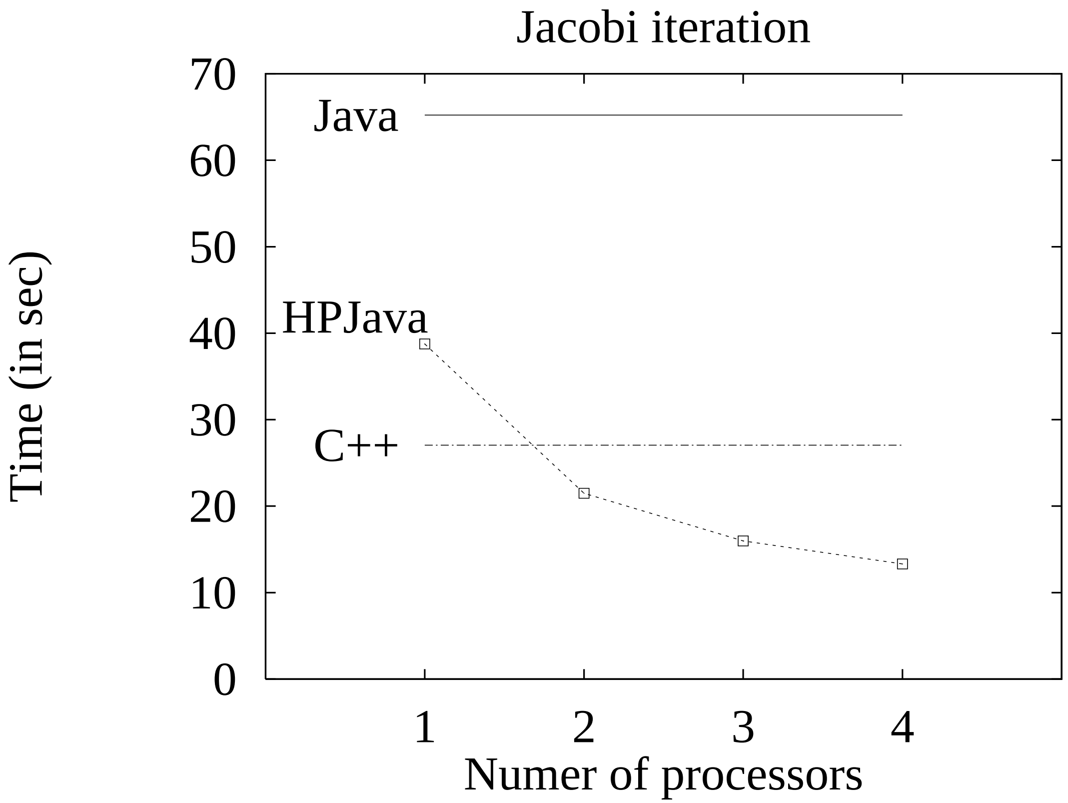
```
do parity = 0, 1

  forall i, j such that i + j mod 2 = parity
    old  = latt [i, j]

    cand = (old + (int) (rand() * q - 1) + 1) % q

    e_old = (old == latt [i + 1, j]) + (old == latt [i - 1, j]) +
            (old == latt [i, j + 1]) + (old == latt [i, j - 1])

    e_new = (cand == latt [i + 1, j]) + (cand == latt [i - 1, j]) +
            (cand == latt [i, j + 1]) + (cand == latt [i, j - 1])

    if(rand() < exp((e_old - e_new) * cup))
      latt [i, j] = cand
  end forall

end do
```

```
for (int parity = 0; parity < 2; parity ++) {

  Adlib.writeHalo(latt, w, w, mode) ;

  overall(i = x)
    overall(j = y [(parity + x.idx(i)) % 2 : : 2]) {
      int old  = latt [i, j] ;

      int cand = (old + (int) (Math.erand48(xsubi) * q - 1) + 1) % q ;

      float e_old =
          (old == latt [i + 1, j]) + (old == latt [i - 1, j]) +
          (old == latt [i, j + 1]) + (old == latt [i, j - 1]) ;

      float e_new =
          (cand == latt [i + 1, j]) + (cand == latt [i - 1, j]) +
          (cand == latt [i, j + 1]) + (cand == latt [i, j - 1]) ;

      if (Math.erand48(xsubi) < Math.exp((e_old - e_new) * cup))
        latt [i, j] = cand ;
    }
```

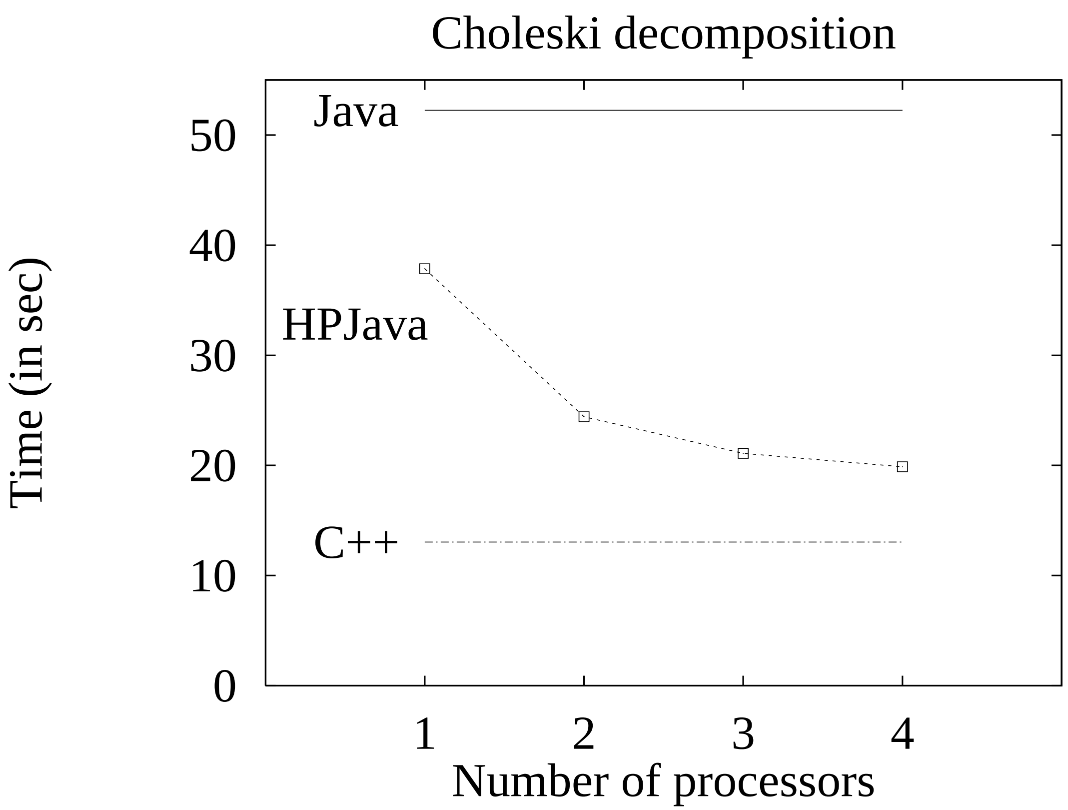
Preliminary Performance Data

- Examples are Jacobi update for 2d Laplace and Cholesky decomposition.
- HPJava code was hand-translated to Java, applying only optimizations that will be incorporated in the initial translator.
- Parallel Java on 4 Ultrasparc processors, using JDK 1.2 JIT compiler Generated code uses a JNI interface to Adlib, with MPICH for underlying communications.
- Comparisions are with simple Java and C++ versions.
- Overhead in parallel case attributable to MPI—SunHPC may reduce this by an order of magnitude.

Jacobi Update



Choleski


Choleski decomposition
Java
HPJava
C++
50
40
30
20
10
0
1
2
3
4
Time (in sec)
Number of processors

Conclusions

- HPJava language provides much of the expressive power of HPF, in an object-oriented, SPMD environment.
- Allows programs to combine data parallel code and SPMD library calls directly.
- Similar Fortran extensions have been suggested (FAST proposal).