Considerations in HPJava language design and implementation

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

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

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

Introduction

- 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

- Works best for algorithms that have essentially SIMD parallelism—identical updates across whole arrays. Many algorithms—even data parallel ones—easier to express with looser coupling between updates of the individual array elements.
- Many algorithms have completely irregular patterns of access to the elements of (nevertheless regular) arrays. May be expressible using (eg) FORALL, but hard for compiler to generate good code.
- Other applications simply don't have such regular arrays as major data structures—more general meshes needed.

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.

Features

- **Data parallel**: Generalized, HPF-like distributed arrays as language primitives.
- **MIMD**: processors have freedom to independently execute complex procedures on local data.
- **Distributed Memory**: All access to non-local array elements goes through library functions.
- **Distributed Control** constructs facilitate access to local elements of arrays, and simplify parallel array traversals.

Language Model

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

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.

Adlib

- Early development in **shpf** project at Southampton, UK. Reimplemented at Syracuse in PCRC.
- Designed specifically as runtime support for HPF translation.
- Kernel is C++ class library, using MPI. Fortran, C++ and Java interfaces available or under development.
- Communication primitives similar to Multiblock PARTI, plus all F90-style transformational intrinsic for array arithmetic (and gather/scatter for irregular).
- Array descriptor and communication library supports full HPF 1.0 distributed array model.

ad++

- C++ user-level interface to Adlib.
- Prototype of HPJava language model. It extends C++ only by class libraries and macros.
- Distributed arrays implemented as container classes.
- Introduces the three distributed control constructs, **on**, **at** and **overall**, implemented as macros.
- The current ad++ is inefficient (and concrete syntax is quite clumsy) compared with what could be achieved with a purpose-built compiler or preprocessor.

ad++ Fragment

```
Procs2 p(2, 2) ;

ON(p) {
  BlockRange x(100, p.dim(0)), y(10, p.dim(1)) ;

  Array2<float> a(x, y) ;

  Index i(x), j(y) ;
  OVERALL(i) {
    OVERALL(j) {
      a(i, j) = i + j ;
    } ALLOVER(j) ;
  } ALLOVER(i) ;

} NO(p) ;
```

HPJava

- Fix the efficiency problems of ad++ by using a preprocessor to optimize the parallel loops.
- Modelled semantically on ad++, but with a more manageable syntax.
- Implementation by a simple translator from HPJava to Java (in progress).

Distributed Arrays

- Distributed arrays flagged by double brackets. Orthogonal to ordinary Java arrays.
- Distribution expressed in terms of `Group` (subclass `Procs2`) and `Range` (subclass `BlockRange`).
- 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 ;
```

Motivating example: matrix multiplication

```
Procs2 p = new Procs2(P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;

  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, initialize `a`, `b` by

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

The on construct and the APG

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

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

- Language incorporates a formal idea of an **active process group** (APG). Inside `on(p)`, value of the APG changes to `p`.

Subscripting and Locations

- Distributed array access such as

```
a [17, 23] = 13 ;
```

  forbidden, because typical process does not hold the element.

- Introduce idea of a **location**—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.

- Furthermore, locations used in simple subscripting must be *named locations*, and named locations can only be scoped by **at** and **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.
- 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] ;
```

- Body of construct executes, conceptually in parallel, for every location in the range of its index.

- Due to rules on use of subscripts, **overall** can usually only combine elements of arrays with some simple alignment relation.

Translation scheme

- Forseeable HPJava compilers will be implemented as a source-to-source translators, converting HPJava program to a Java node program with calls to runtime functions and communication libraries.
- Initial implementation uses an ad++-like interface to Adlib, called **adJava**, as runtime support for distributed arrays.
- Translator generates Java wrapper classes representing each kind of distributed array actually 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

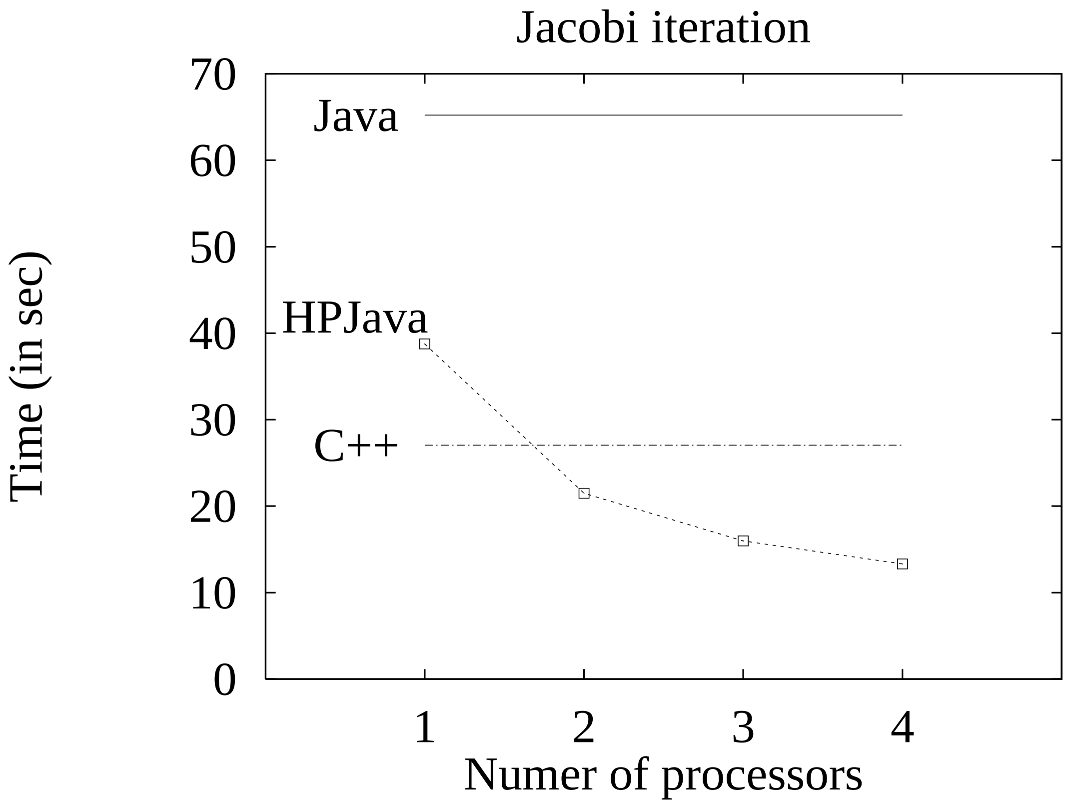
```
do parity = 0, 1

  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 ;
    }
```

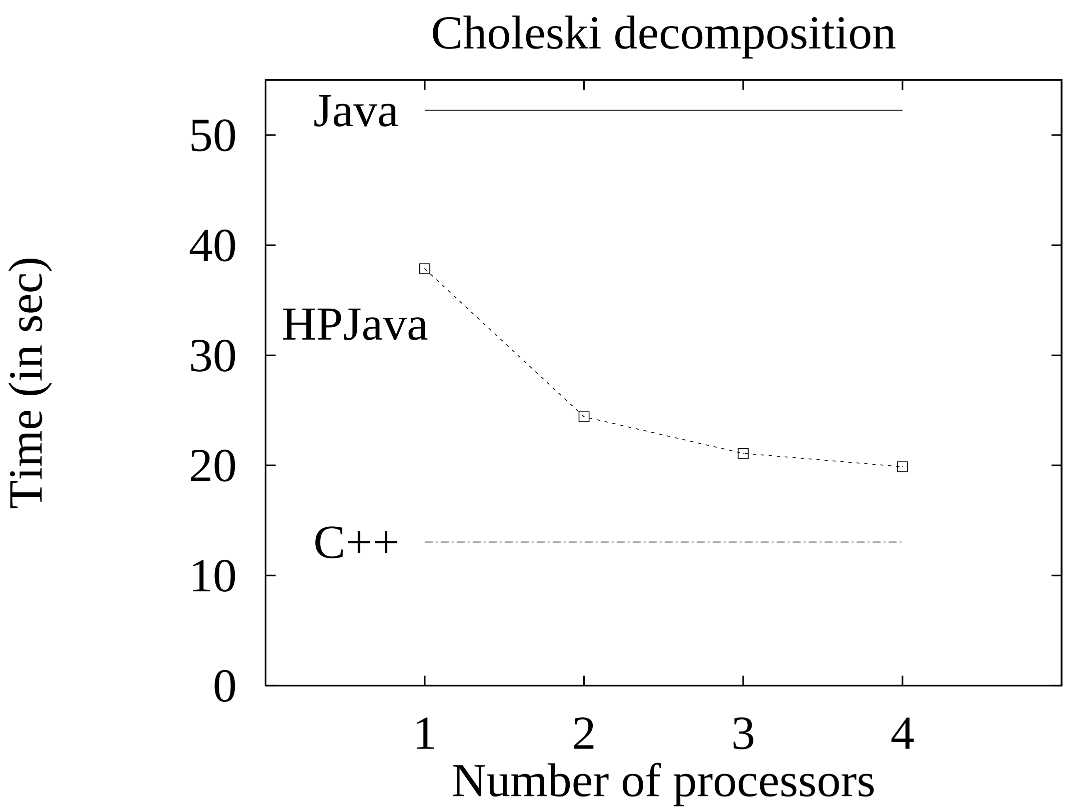
Preliminary Performance Data

- 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++
Time (in sec)
0
10
20
30
40
50
1
2
3
4
Number of processors

Conclusions

- HPJava language extensions provide much of the expressive power of HPF, but in a strictly SPMD enviroment.
- Allows programs to combine data parallel code and SPMD library calls directly.
- Similar Fortran extensions have been suggested (FAST proposal). May be implemented by Peking collaborators...