Matrix multiplication in HPJava

Bryan Carpenter, Guansong Zhang, Xinying Li (?)

September 22, 1997

The *Adlib* library contains a function called `matmul` for matrix multiplication. To illustrate various features of the language, this note gives three different algorithms for matrix multiplication in HPJava, without using `matmul`.

1 Algorithm 1

A well-known parallel algorithm for matrix multiplication is based on the observation that the N by N matrix product AB, or

$$\left[\begin{array}{ccccc} a_{11} & a_{12} & a_{13} & & a_{1,N} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2,N} \\ a_{31} & a_{32} & a_{33} & & a_{3,N} \\ & \vdots & & & \vdots \\ a_{N,1} & a_{N,2} & a_{N,3} & \ldots & a_{N,N} \end{array}\right] \left[\begin{array}{ccccc} b_{11} & b_{12} & b_{13} & & b_{1,N} \\ b_{21} & b_{22} & b_{23} & \ldots & b_{2,N} \\ b_{31} & b_{32} & b_{33} & & b_{3,N} \\ & \vdots & & & \vdots \\ b_{N,1} & b_{N,2} & b_{N,3} & \ldots & b_{N,N} \end{array}\right] \tag{1}$$

can be partitioned as follows

$$\left[\begin{array}{cccc} a_{11} & a_{12} & \ldots & a_{1,N} \\ \vdots & & & \vdots \\ a_{S,1} & a_{S,2} & \ldots & a_{S,N} \\ \hline a_{S+1,1} & a_{S+1,2} & \ldots & a_{S+1,,N} \\ \vdots & & & \vdots \\ a_{2S,1} & a_{2S,2} & \ldots & a_{2S,N} \\ \hline & \vdots & & \end{array}\right] \left[\begin{array}{ccc|ccc|c} b_{11} & \ldots & b_{1,S} & b_{1,S+1} & \ldots & b_{1,2S} & \ldots \\ b_{21} & & b_{2,S} & b_{2,S+1} & & b_{2,2S} & \\ \vdots & & \vdots & \vdots & & \vdots & \\ b_{N,1} & \ldots & b_{N,S} & b_{N,S+1} & \ldots & b_{N,2S} & \end{array}\right] \tag{2}$$

then expressed as

$$\left[\begin{array}{c} A_1 \\ A_2 \\ \vdots \end{array}\right] \left[\begin{array}{ccc} B_1 & B_2 & \ldots \end{array}\right] = \left[\begin{array}{ccc} A_1B_1 & A_1B_2 & \ldots \\ A_2B_1 & A_2B_2 & \\ \vdots & & \end{array}\right] \tag{3}$$

where A_i are the S by N bands of A, and B_i are the N by S blocks of B.

In the parallel algorithm the blocks of A and B are first allocated to individual processors with A_1, B_1 on the first processor, A_2, B_2 on the second processor, etc. At this stage the diagonal blocks of the result can be computed. The blocks of B are then cyclically shifted around the processors. A_1, B_2 are on the first processor, A_2, B_3 are on the second processor and so on up to the last (Pth) processor, which holds A_P and B_1. The corresponding off-diagonal blocks of the result are computed. The procedure is repeated P times (assuming $N = PS$) until all blocks of the result have been computed.

To express this in HPJava we represent the matrices as *3-dimensional* arrays. The unpartitioned dimensions are collapsed dimensions of size N. Each partitioned dimension is split into an on-processor, collapsed dimension of size S and a distributed process dimension. The complete program is given below

```
Procs1 p = new Procs1(P) ;
on(p) {
  Range d = p.dim(0) ;

  float a [[,*,*]] = new float [[d, S, N]] ;
  float b [[,*,*]] = new float [[d, N, S]] ;

  ... initialize 'a', 'b'

  float c [[,*,*]] = new float [[d, S, N]] ;

  float tmp [[,*,*]] = new float [[d, N, S]] ;

  for(int s = 0 ; s < P ; s++) {

    over(d) {
      const int base = S * ((d + s) % P) ;

      for(int ib = 0 ; ib < S ; ib++)
        for(int kb = 0 ; kb < S ; kb++) {
          float sum = 0 ;
          for(int j = 0 ; j < N ; j++)
            sum += a [[d, ib, j]] * b [[d, j, kb]] ;

          c [[d, ib, base + kb]] = sum ;
        }
    }

    Adlib.shift(tmp, b, 1, 0, CYCL) ;
    Adlib.copy(b, tmp) ;
  }
}
```

The main loop is over the P stages of the algorithm. Within this loop control is dispatched to the individual processors by the *over* construct. The inner loops implement a simple matrix product on the sections of the array at location `d`. The result is assigned to a section of `c` displaced `s` blocks from the diagonal by adding `base` to the last subscript. (Note the value `base` is computed in terms of the local process id given by `(int) d`.)

The cyclic shift operation is analogous to the `CSHIFT` function in Fortran 90. Its array arguments `tmp` and `b` must be aligned arrays. The shift amount is 1 (third argument) in dimension 0 (second argument)[1]. The last argument is the mode of the shift, cyclic in this case. In-place shifts aren't allowed (the array arguments of `shift` must be distinct) so values are shifted into `tmp` then copied back to `b`.

2 Algorithm 2

The algorithm in this section that presumes that the matrices are replicated appropriately over a three dimensional grid of processors, then uses dot product reduction operations to calculate the elements elements of the result. In theory this is a very fast parallel algorithm, because it only involves a single parallel step using a reduction operation that can be evaluated in parallel in "logarithmic time". For various reasons it is not a good practical algorithm, but it illustrates some interesting features of the HPJava language. A version of the code is given below

```
Procs3 p = new Procs3(P, P, P) ;
on(p) {
  Range x = new BlockRange(N, p.dim(0)) ;
  Range y = new BlockRange(N, p.dim(1)) ;
  Range z = new BlockRange(N, p.dim(2)) ;

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

  ... initialize 'a', 'b'

  float [[,]] c = new float [[x, y]] ;

  over(x)
    over(y)
      c [[x, y]] = Adlib.dotProduct(a [x, z], b [z, y]) ;
}
```

In this case the matrices are represented directly as two-dimensional arrays. For definiteness the ranges of size `N` are block-distributed over the process array, although this algorithm is insensitive to the distribution format.

[1]Dimensions are labelled $0, 1, \ldots$ so dimension 0 refers to the first, distributed dimension of `b` and `tmp`.

The matrix a is replicated over the second dimension of p and the matrix b is replicated over the first dimension. The result c is replicated over the last dimension of p.

The dotProduct operation is applied in parallel to combine the rows of a and columns of b. These are selected using section subscripting, with the parametric ranges of the *over* constructs appearing as subscripts (standing for *locations* in this context—a parametric range used as an array subscript is always converted to a Location).

The example illustrates how HPJava supports a powerful form of *nested parallelism*. Inside the parallel *over* constructs, further parallelism is available because the process dimension of z has not been restricted. The dotProduct operation will work in parallel across this dimension. Automatically, its result will be broadcast to all members of the process group active at this point.

3 Algorithm 3

This algorithm is a modification of the last one which dispenses with the parallelism in the third dimension. The z dimension is assumed to be collapsed, and the dot product operation is evaluated "in line". A direct transcription is

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

  ... initialize 'a', 'b'

  float [[,]] c = new float [[x, y]] ;

  over(x)
    over(y) {
      float sum = 0 ;
      for(int j = 0 ; j < N ; j++)
        sum += a [[x, j]] * b [[j, y]] ;

      c [[x, y]] = sum ;
    }
}
```

Because we are moving closer to a practical algorithm here, we will give a version that eliminates the assumptions about the layout of the input data. The ranges and process group of c are determined by using the inquiry functions rng and grp, which are applicable to any distributed array. The a and b matrices are copied to temporary arrays with the proper layout relative to `c`, using the `remap` function.

```
Group p = c.grp() ;

Range x = c.rng(0) ;
Range y = c.rng(1) ;

on(p) {
  float [[,*]] tmp1 = new float [[x, N]] ;
  float [[*,]] tmp2 = new float [[N, y]] ;

  Adlib.remap(tmp1, a) ;
  Adlib.remap(tmp2, b) ;

  over(x)
    over(y) {
      float sum = 0 ;
      for(int j = 0 ; j < N ; j++)
        sum += tmp1 [[x, j]] * tmp2 [[j, y]] ;

      c [[x, y]] = sum ;
    }
}
```

The only shortcoming of this algorithm is that needs a lot of space. Because the temporaries are each collapsed in one of their dimensions, the local memory requirements for the temporaries may be larger than the local memory requirements for the original arrays. The library function `matmul` uses a variant of this algorithm which bounds the temporary space by splitting the calculation into several phases dealing with successive bands of the input matrices (reminiscent of the blocking in algorithm 1, but this time dividing up the shared inner product range rather than the outer product ranges).