

# From Flop to MegaFlops: Java for Technical Computing

José E. Moreira   Sam Midkiff   Manish Gupta

IBM T. J. Watson Research Center  
P.O. Box 218  
Yorktown Heights NY 10598



## Why consider Java for technical computing?

- Java is increasingly used as a first programming language in colleges: large supply of programmers.
- Java supports OO programming well, without too many complications.
- Java ports easily (write once, run everywhere).
- Java will be pervasive in the environment surrounding and driving numeric-intensive computing: high-level control, GUIs, communication, ...

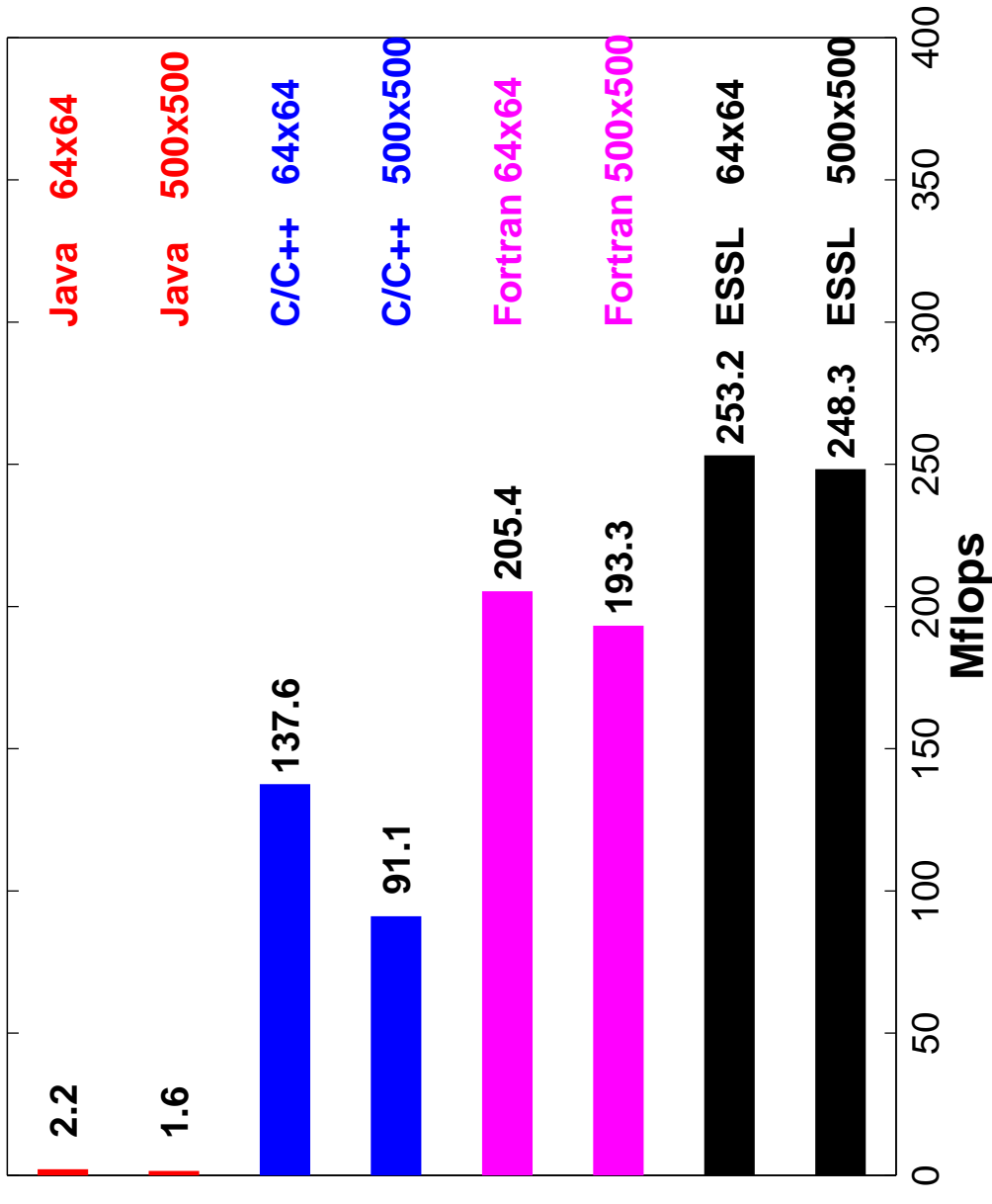
## What about performance?

MATMUL benchmark description:

```
• static void matmul(double[][] A, double[][] B, double[][] C,  
  int m, int n, int p) {  
  int i, j, k;  
  for (i = 0; i < m; i++)  
    for (j = 0; j < p; j++)  
      for (k = 0; k < n; k++)  
        C[i][j] += A[i][k] * B[k][j]  
  }
```

- $m = n = p = 64$  (fits in cache of POWER2).
- $m = n = p = 500$  (too big for cache of POWER2).

## MATMUL results on the RS/6000 590

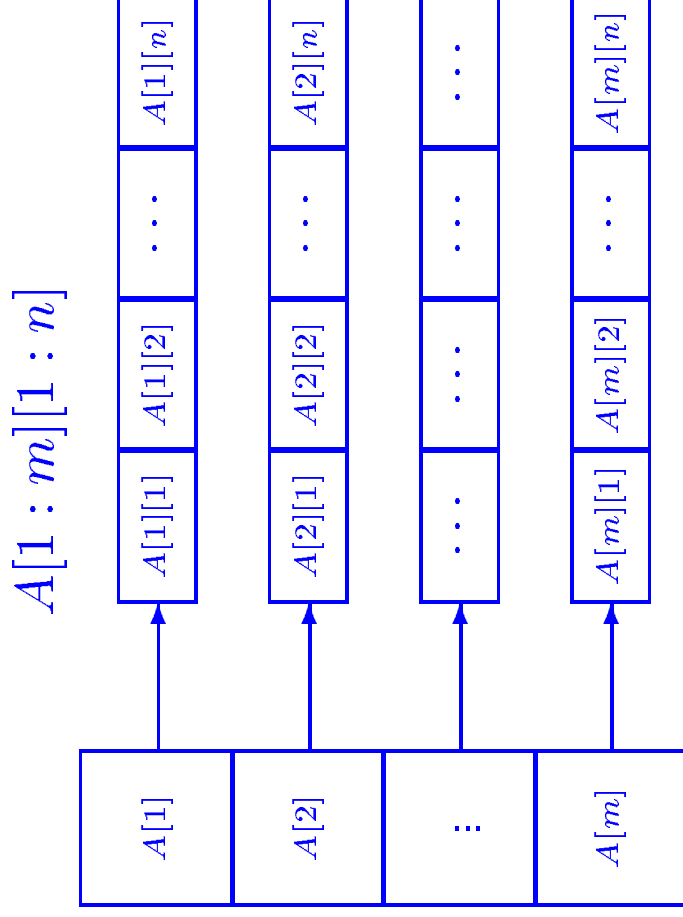


## Why is Fortran so much faster?

- At 205 Mflops, Fortran is 50% faster than C, and  $\sim 100$  times faster than Java!
- The IBM XLF 4.1 compiler uses several high-order transformations:
  - blocking, for better memory behavior,
  - multiple loop unrolling,
  - loop interchange,
  - scalar replacement,
  - loop fusion.
- This technology is moving to other languages as well, with the IBM Toronto Portable Optimizer (TPO).
- The same transformations can be applied to safe regions of Java! Normally, exceptions prevent reordering.

## Creating a safe region

- **First step:** localize array for thread safety. Much simpler with **Matrix** objects.
- Layout of 2-dimensional array:



## Privatizing a 2-dimensional array

- For array  $A$ :

```
matmul( $A', B', C', m, n, p$ ) {  
  double[][]  $A = \text{new double}[A'.length][]$   
   $A_{\text{rows}} = A.\text{length}$   
   $A_{\text{cols}} = \infty$   
  for ( $i = 0; i < A.\text{length}; i++$ ) {  
     $A[i] = A'[i]$   
     $A_{\text{cols}} = \min(A_{\text{cols}}, A[i].\text{length})$   
  }  
  ...  
}
```

- Do the same for  $B$  and  $C$ .

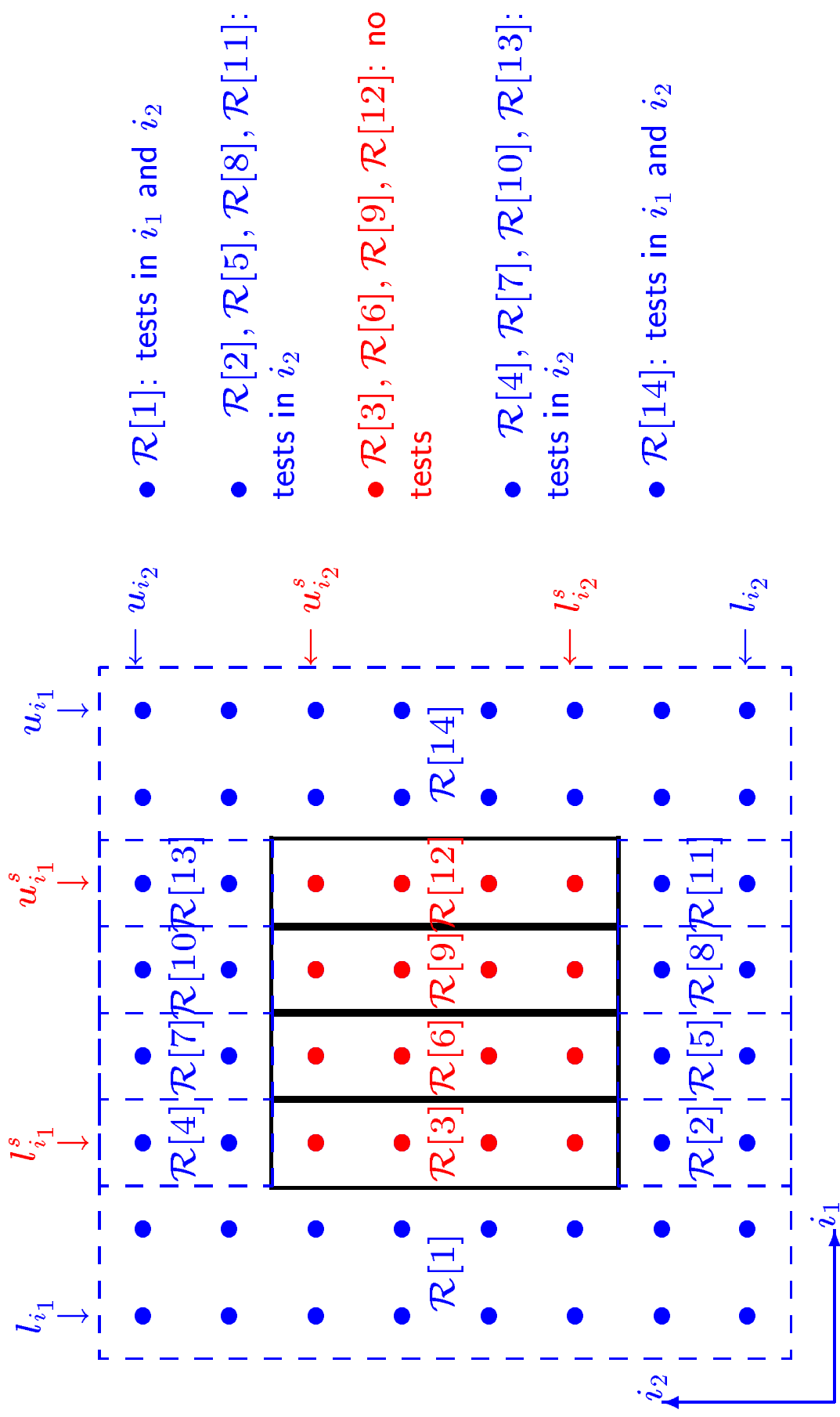
## Creating a safe region (continued)

- Second step: avoid index exceptions.

```
if ( (m ≤ Crows) ∧ (p ≤ Ccols) ∧ (m ≤ Arows) ∧ (n ≤ Acols) ∧  
    (n ≤ Brows) ∧ (p ≤ Bcols) ) {  
    for (i = 0; i < m; i++)  
        for (j = 0; j < p; j++)  
            for (k = 0; k < n; k++)  
                C[i][j] += A[i][k] * B[k][j]  
    } else {  
        for (i = 0; i < m; i++)  
            for (j = 0; j < p; j++)  
                for (k = 0; k < n; k++)  
                    C[i][j] += A[i][k] * B[k][j]  
    }
```

- The red version is safe! Bounds checking is not necessary and reordering is possible!

## Alternative: Tiling the iteration space with regions



## Traversing the iteration space

```
for ( $\delta = 1; \delta \leq u_\delta; \delta++$ ) {  
  if ( $(\mathcal{R}[\delta].\tau == \text{test})$ ) {  
    for ( $i_1 = l_{i_1}(\delta); i_1 \leq u_{i_1}(\delta); i_1++$ )  
      for ( $i_2 = l_{i_2}(\delta); i_2 \leq u_{i_2}(\delta); i_2++$ )  
        ...  
          for ( $i_d = l_{i_d}(\delta); i_d \leq u_{i_d}(\delta); i_d++$ )  
            Btest( $i_1, i_2, \dots, i_d$ )  
          } else {  
            for ( $i_1 = l_{i_1}(\delta); i_1 \leq u_{i_1}(\delta); i_1++$ )  
              for ( $i_2 = l_{i_2}(\delta); i_2 \leq u_{i_2}(\delta); i_2++$ )  
                ...  
                  for ( $i_d = l_{i_d}(\delta); i_d \leq u_{i_d}(\delta); i_d++$ )  
                    Bnotest( $i_1, i_2, \dots, i_d$ )  
                  }  
                }  
              }  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

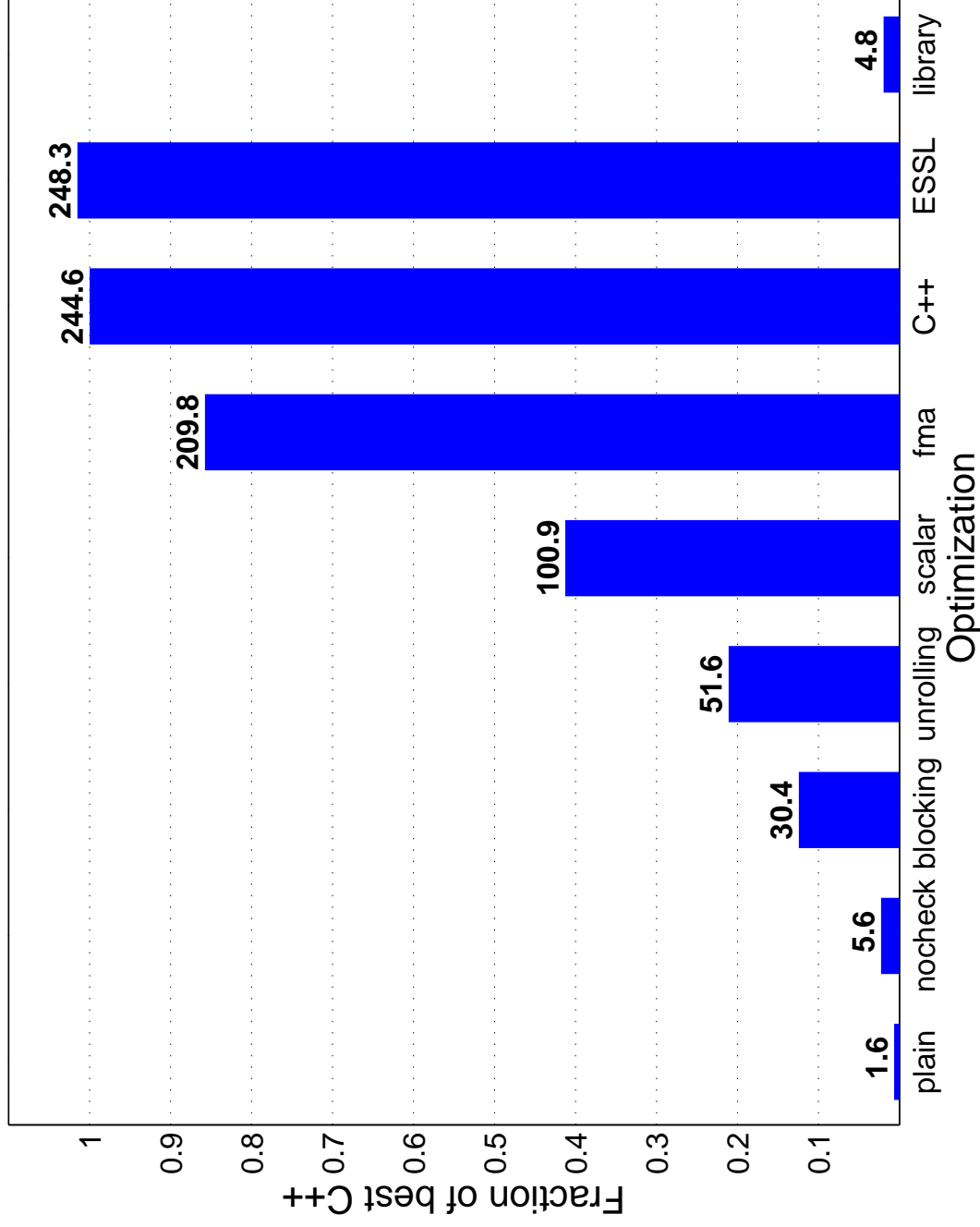
## Optimizing the safe region

- **blocking**: for better cache reuse.
- **unrolling**: of  $i$  and  $j$  loops for better register reuse and to exploit functional-unit parallelism. Note that  $k$  loop cannot be parallelized in Java!
- **scalar replacement**: requires pointer disambiguation. Can be done at run-time, much easier with **Matrix objects**.
- **fma**: uses extended-precision fused-multiply-add instruction. Not legal yet.

In the next plot, **library** has all the optimizations of the **fma** version except for bounds checking. This is the best hand implementation with all checks explicit.

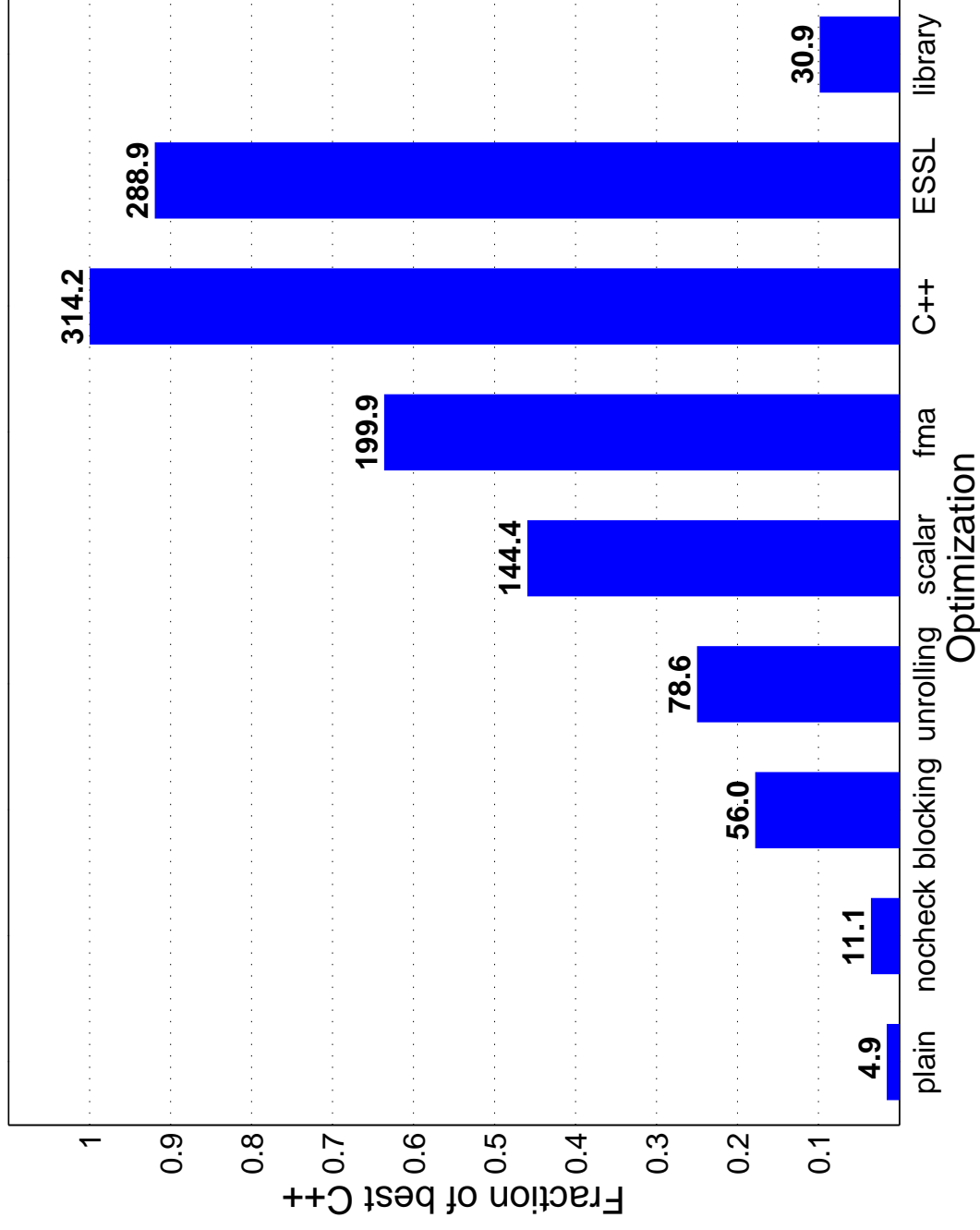
# 500 × 500 MATMUL

Performance of MATMUL on 67 MHz POWER2 (Mflops)



# 500 × 500 MATMUL

Performance of MATMUL on 332 MHz 604e (Mflops)



## Parallelizing MATMUL

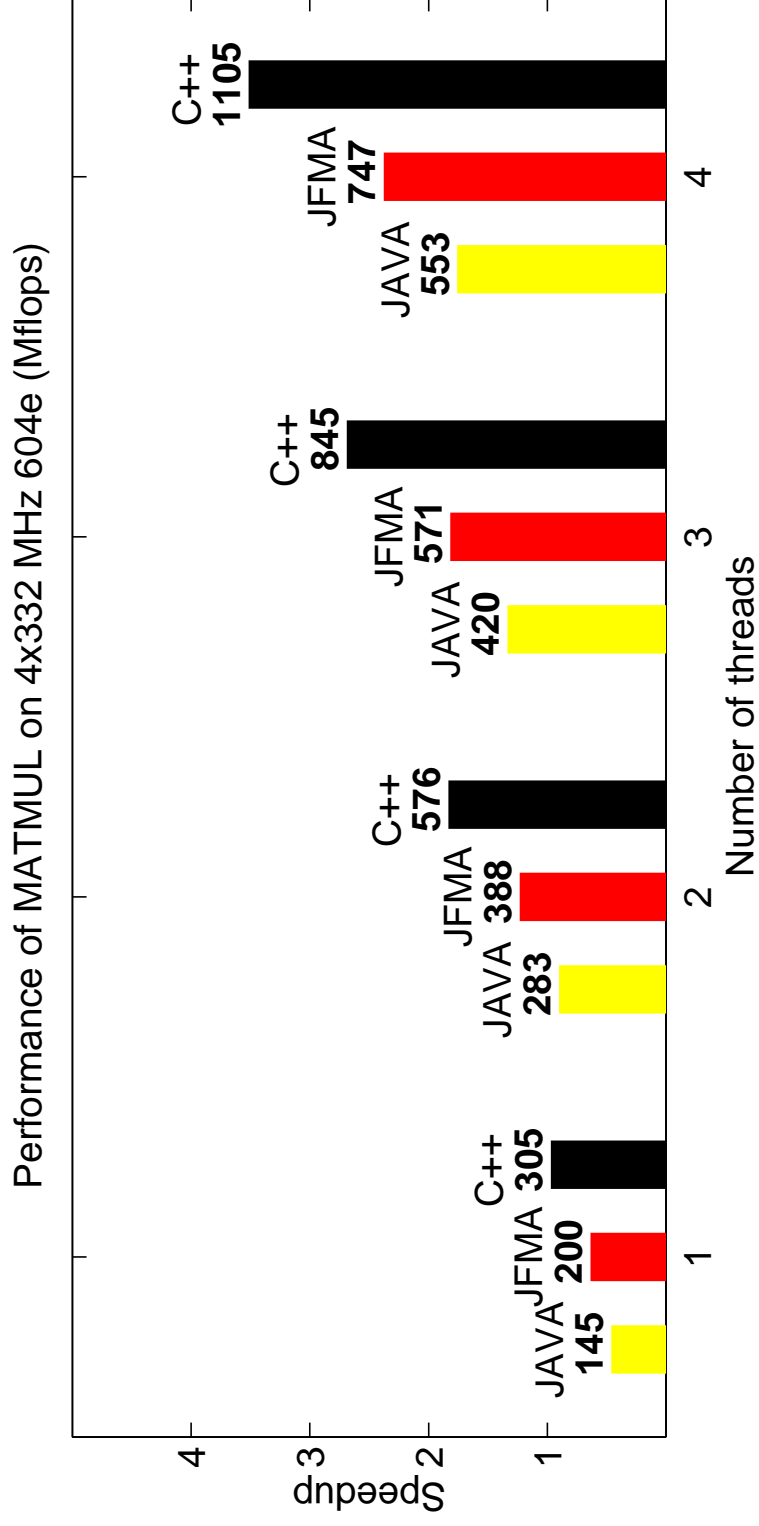
- The safe region can be multithreaded! Exploit parallelism on  $I$  and  $J$  block loops.
- $b_I, b_J$ , and  $b_K$  are the block sizes along  $I$ ,  $J$ , and  $K$ .  $t$  is the thread id,  $n_T$  is the number of threads.

```

for (b = t; b < n_I n_J; b += n_T) {
    I = (b/n_I) × b_I
    J = (b mod n_I) × b_J
    for (K = 0; K < n; K += b_K) {
        for (i = I; i < min(m, I + b_I); i++)
            for (j = J; j < min(p, J + b_J); j++)
                for (k = K; k < min(n, K + b_K); k++)
                    C[i][j] += A[i][k] * B[k][j]
    }
}

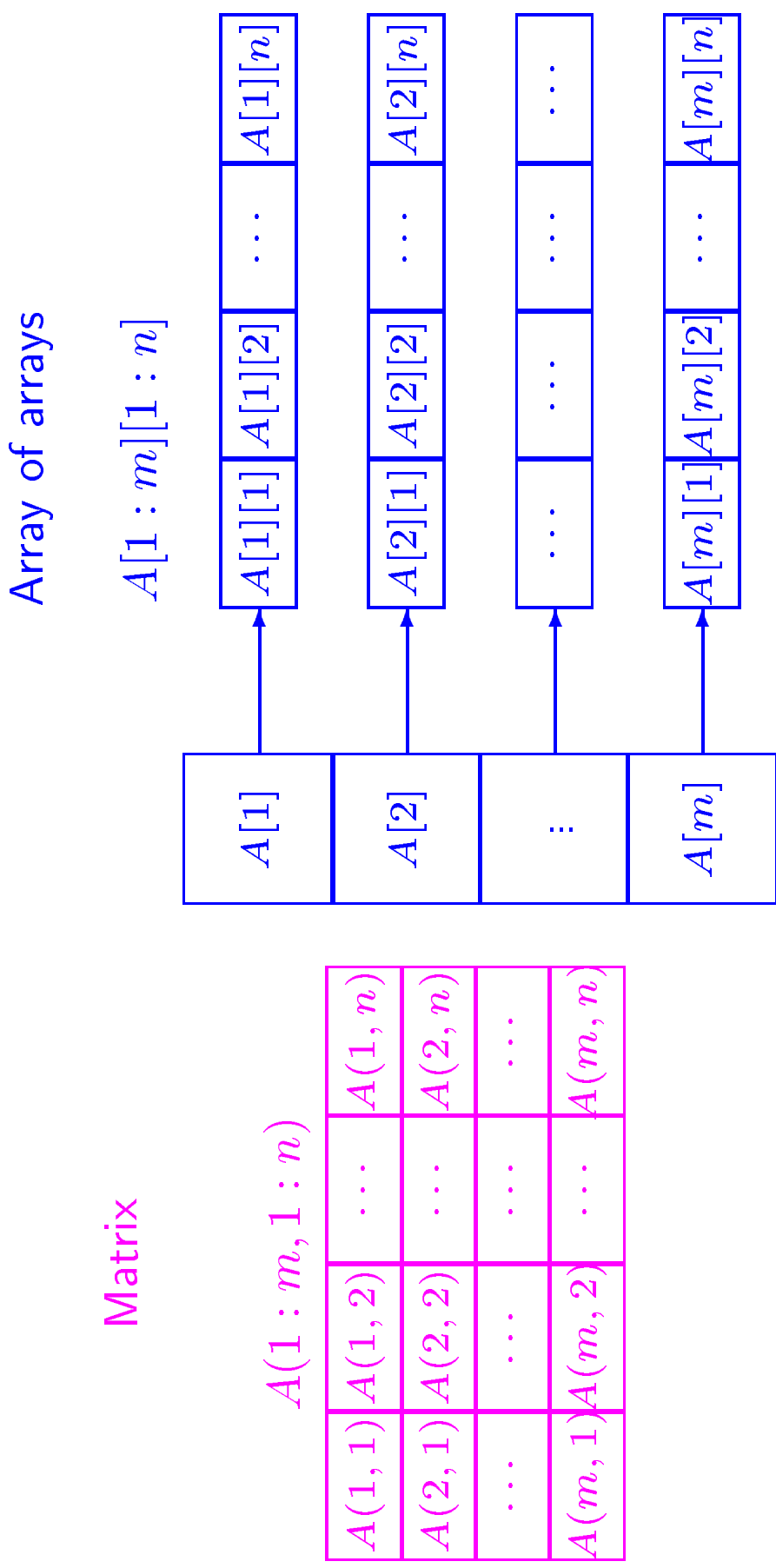
```

## Parallelizing MATMUL – results



Parallel ESSL: 1183 Mflops.

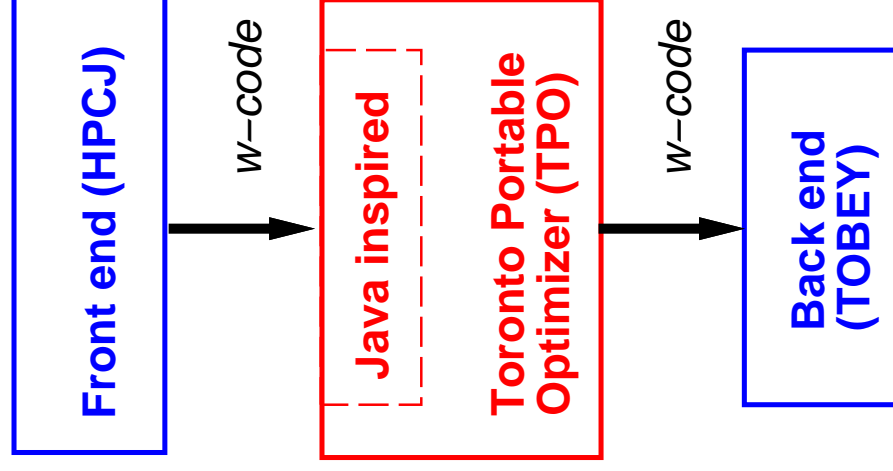
## Matrix objects



## Matrices vs. arrays of arrays

1. The location of an element  $A(i, j)$  can be computed using simple arithmetic. Finding  $A[i][j]$  involves, in general, pointer chasing.
2. There is no guarantee that  $A[1 : m][1 : n]$  is rectangular. Even if it was rectangular when created it may have been modified.
3. Aliasing disambiguation. Given two matrix objects  $A$  and  $B$  it is sufficient to show that  $A \neq B$  to prove there is no aliasing between them. In the case of two arrays of arrays  $A[][]$  and  $B[][]$  one has to show that  $A[i] \neq B[j] \forall i, j$ .
4. Regular array sections for matrices can be represented compactly.
5. Disambiguation between two array sections reduces to proving that the source arrays are different or that the sections do not intersect.
6. Privatizing matrix  $A(1 : m, 1 : n)$ , for thread safety, requires copying only one reference.

## Compiler architecture



- Current IBM compiler infrastructure (HPCJ, TPO, TOBEY) provide a solid foundation, and integrate optimizations developed for Java, C, and Fortran.
- Automatic parallelization technology developed for Fortran and C also good for Java!
- Use of *w-code* as an intermediate language allows standard IBM compiler components to be used.
- It is portable to multiple architectures: e.g., RS/6000, S/390, Intel.

## Related Work

- Java for technical computing by Atwood et al., Boisvert et al., Casanova et al, and Schwab et al. Note that Java is more competitive in slower machines.
- Work to determine that a run-time index check is unnecessary by Cousot et al., Harrison, and Schwarz et al.
- Work on hoisting tests out of loops by Markstein et al.
- Work on test coverage, to identify redundant tests, by Asuru, Gupta, and Kolte et al.
- Java precise exception model and **try** blocks present new challenges.
- Work on Java parallelization by Bik et al.

## Conclusions

- There are good reasons to use Java for technical computing.
- Performance problems can be handled:
  - run-time checks with compiler optimizations;
  - matrices by standard classes, language extensions, compiler technology;
  - `fma` and reordering by relaxing semantics;
  - safe regions enable high-order transformations and parallelization;
- Java performance in MATMUL can be 65-85% of C++, Fortran, and library versions.