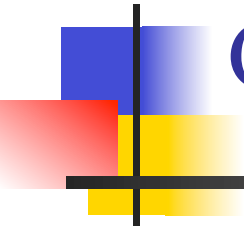
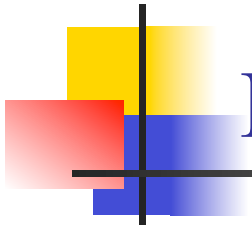


CS475 Parallel Programming

Dense Matrix Multiply

Wim Bohm, Colorado State University

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 license.



Mapping n x n matrix to p PEs

- Striped: allocate rows (or columns) on PEs
  - Row Block striped: consecutive rows to one PE, e.g.:

    PE# 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

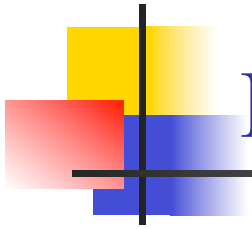
    Row 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  - Cyclic striped: interleaving rows onto Pes

    PE# 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

    Row 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  - Hybrid

    PE# 0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3

    Row 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  - Finest granularity
    - One row (or column) per PE, (p = n)



Mapping n x n matrix to p PEs (cont.)

- Blocked / Checkerboard
  - Map n/sqrt(p) x n/sqrt(p) blocks onto PEs
  - Maps well on a 2D mesh
  - Finest granularity: 1 element per PE, (p = n*n)
- Many matrix algorithms allow block formulation
  - Matrix add
  - Matrix multiply

n x n Matrix Multiply

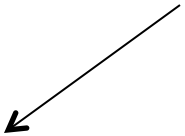
```
for i = 0 to n-1
    for j = 0 to n-1
        Cij = 0
        for k = 0 to n-1
            Cij += Aik * Bkj
```

We do not consider recursive < $O(n^3)$ algorithms (s.a. Strassen).

These could be the top level, driving the $O(n^3)$ algorithms described here.

n x n Matrix Multiply

```
for i = 0 to n-1
    for j = 0 to n-1
        Cij = 0
        for k = 0 to n-1
            Cij += Aik * Bkj
```

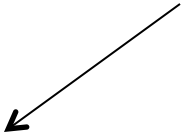


outer i,j indices of A and B determine the target C element or block

n x n Matrix Multiply

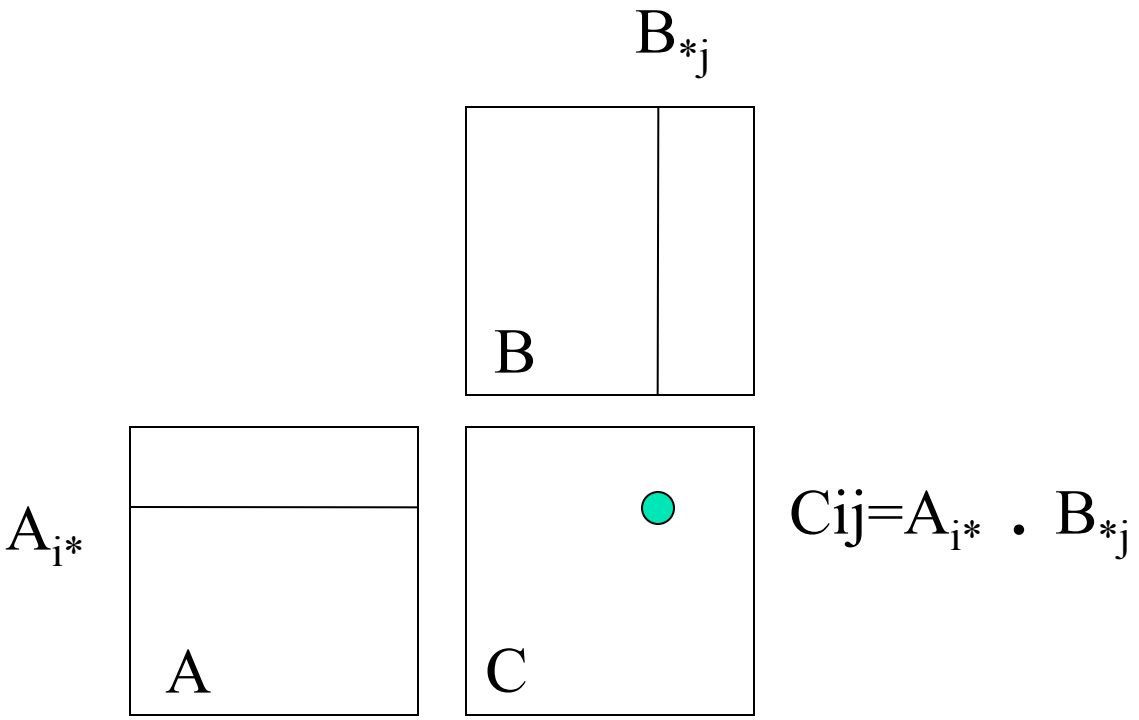
```
for i = 0 to n-1
    for j = 0 to n-1
        Cij = 0
        for k = 0 to n-1
            Cij += Aik * Bkj
```

A and B elements or block compute only if their inner index (k) is equal

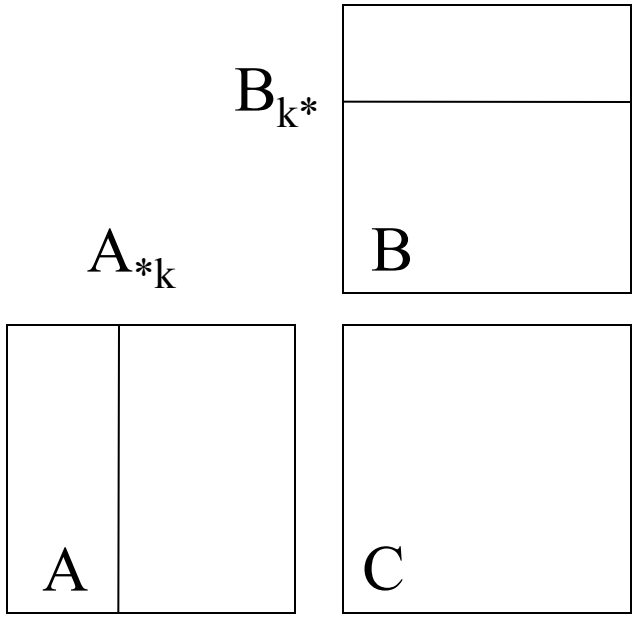


The order in which we compute and accumulate the blocks does not matter!

n^2 inner products



n outer products

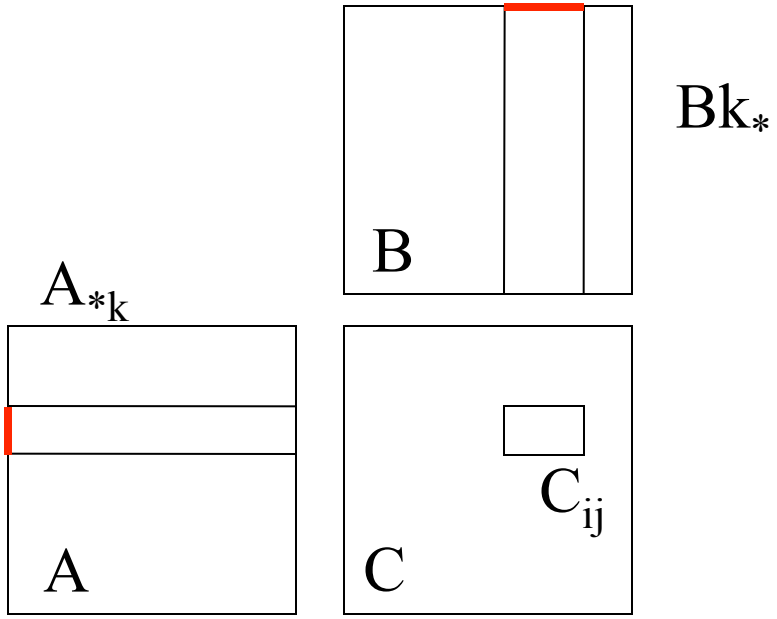



```
for k = 0 to n-1
  forall i in [0,n-1]
    forall j in [0,n-1]
      Cij += A ik * Bkj
```

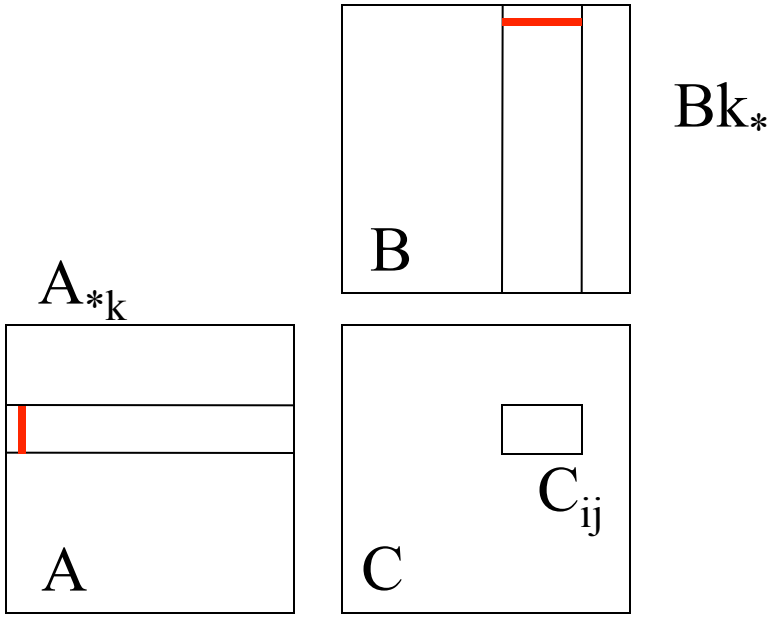
outer i,j indices of A and B are the target C indices

inner indices k of A and B are equal

Blocked outer product



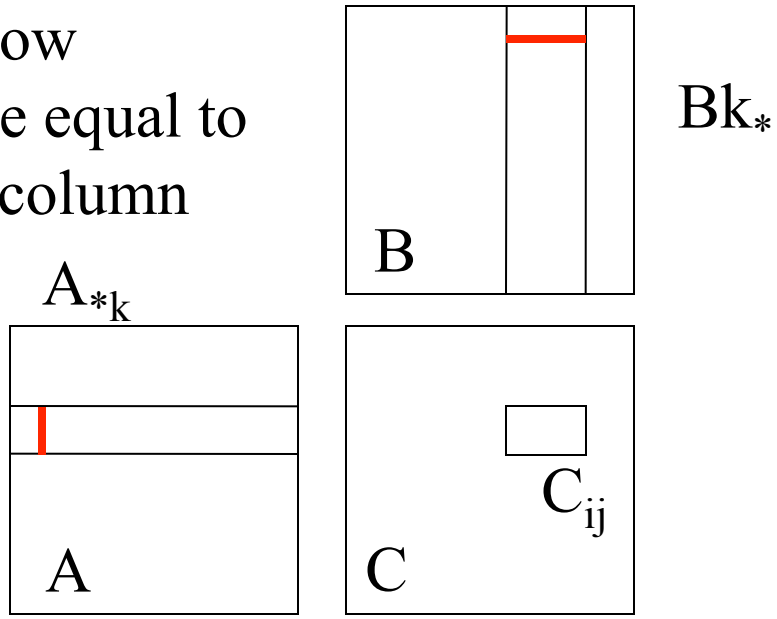
Blocked outer product

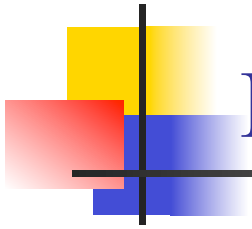


Blocked outer product

notice:

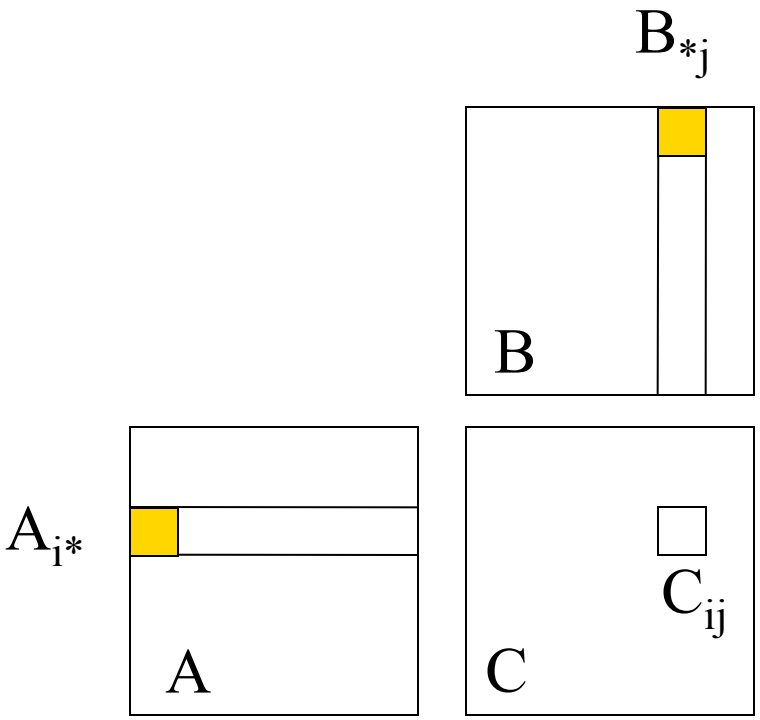
width of A block row
does not need to be equal to
width of B block column

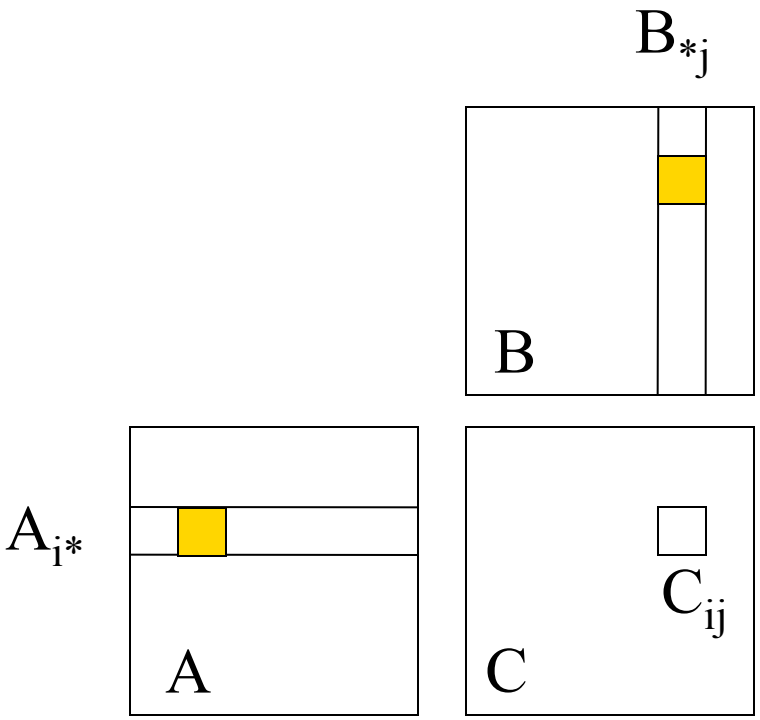


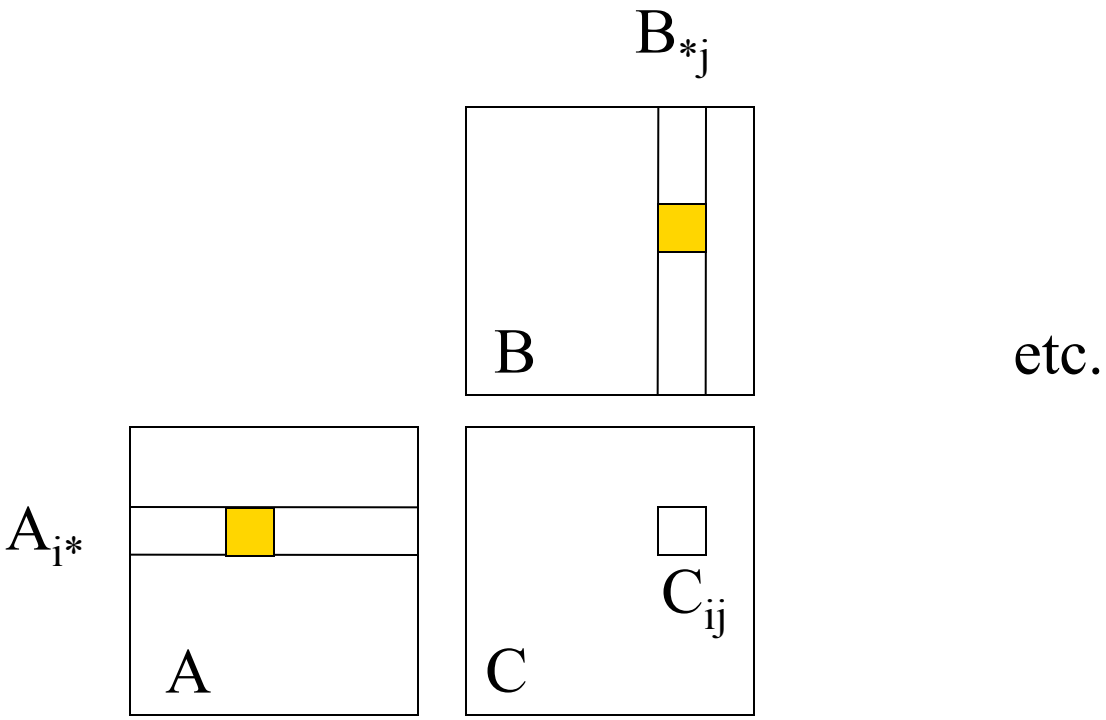


Blocked Matrix Multiply

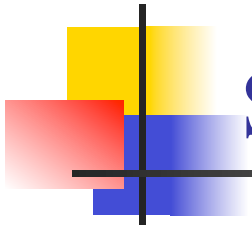
- Standard inner product algorithm can be blocked
- p processors: n/sqrt(p) * n/sqrt(p) sized blocks
- PEij has blocks Aij and Bij
  and computes block Cij
- Cij needs Aik and Bkj, k = 0 to n-1
  - Assuming blocked data distribution for all three matrices, some form of communication is needed


B_{*j}
B
A_{i*}
A
C
C_{ij}


B_{*j}
B
A_{i*}
A
C
C_{ij}

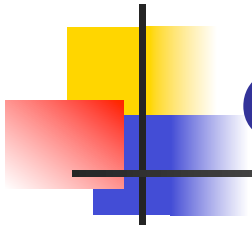


etc.



Simple Block Matrix Multiply

- All PEs need complete block rows of A
  - all-to-all block broadcast of A in row PEs
  - O(sqrt(p) * (n/sqrt(p)*n/sqrt(p)))
- All PEs need complete block columns of B
  - all-to-all block broadcast of B in column PEs
  - O(sqrt(p) * (n/sqrt(p)*n/sqrt(p)))
- Compute block Cij in PEij: n^3/p time
- Space use for A and B EXCESSIVE:
  - per PE: 2*sqrt(p)*(n/sqrt(p)*n/sqrt(p))
  - Total: 2*n*n*sqrt(p), **a replication factor of sqrt(p)**



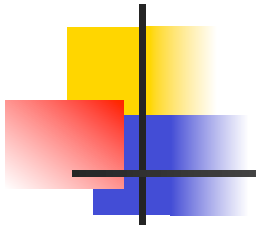
Cannon's Matrix Multiply

- Avoids space overhead
  - processsor has not more than 1 A block, 1 B block, and 1 C block
  - interleaves block moves and computation
- PEij computes block Cij
- Initial alignment of data
  - Circular left shift block Aij by i steps
  - Circular up shift block Bij by j steps
- Interleave computation and communication
  - Compute: block matrix multiplication
  - Communicate:
    - circular shift left A blocks
    - circular shift up B blocks

Initial state: p_{ij} owns $blocks_{ij}$

A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{01}	B_{02}	B_{03}
A_{10}	A_{11}	A_{12}	A_{13}	B_{10}	B_{11}	B_{12}	B_{13}
A_{20}	A_{21}	A_{22}	A_{23}	B_{20}	B_{21}	B_{22}	B_{23}
A_{30}	A_{31}	A_{32}	A_{33}	B_{30}	B_{31}	B_{32}	B_{33}

Cannon: align blocks so all can compute



A_{00} A_{01} A_{02} A_{03} B_{00} **B_{01}**

B_{10} **B_{11}**

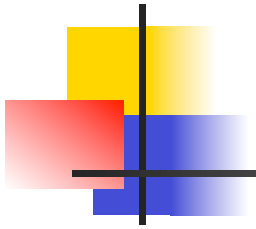
B_{20} **B_{21}**

B_{30} **B_{31}**

First row of A and column of B are in the right place, but which B block does A_{01} need to compute? B_{11}

So rotate B_{*1} 1 up

Cannon: align blocks so all can compute



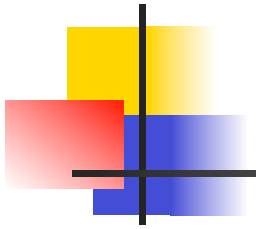
A_{00} A_{01} A_{02} A_{03} B_{00} **B_{11}**

B_{10} **B_{21}**

B_{20} **B_{31}**

B_{30} **B_{01}**

Cannon: align blocks so all can compute



A_{00} A_{01} A_{02} A_{03} B_{00} B_{11} **B_{02}**

B_{10} B_{21} **B_{12}**

B_{20} B_{31} **B_{22}**

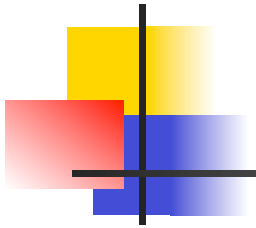
B_{30} B_{01} **B_{32}**

Which B block does A_{02} need to compute?

B_{22}

So rotate B_{*2} 2 up

Cannon: align blocks so all can compute



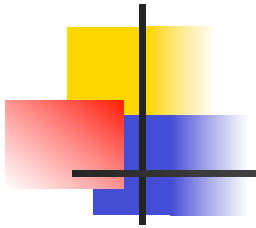
A_{00} A_{01} A_{02} A_{03} B_{00} B_{11} **B_{22}**

B_{10} B_{21} **B_{32}**

B_{20} B_{31} **B_{02}**

B_{30} B_{01} **B_{12}**

Cannon: align blocks so all can compute



A_{00} A_{01} A_{02} A_{03} B_{00} B_{11} B_{22} **B_{33}**

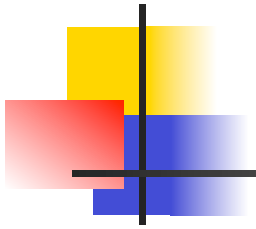
B_{10} B_{21} B_{32} **B_{03}**

B_{20} B_{31} B_{02} **B_{13}**

B_{30} B_{01} B_{12} **B_{23}**

and rotate B_{*3} 3 up

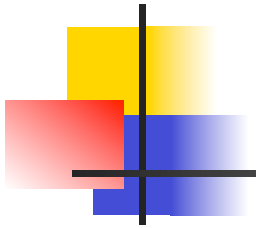
Cannon: align blocks so all can compute



A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{11}	B_{22}	B_{33}
				B_{10}	B_{21}	B_{32}	B_{03}
				B_{20}	B_{31}	B_{02}	B_{13}
				B_{30}	B_{01}	B_{12}	B_{23}

and now align A rows with B

Cannon: align blocks so all can compute



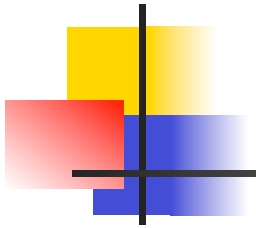
A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{11}	B_{22}	B_{33}
A_{11}	A_{12}	A_{13}	A_{10}	B_{10}	B_{21}	B_{32}	B_{03}
A_{22}	A_{23}	A_{20}	A_{21}	B_{20}	B_{31}	B_{02}	B_{13}
A_{33}	A_{30}	A_{31}	A_{32}	B_{30}	B_{01}	B_{12}	B_{23}

now all blocks are in the right place to multiply

for the next step:

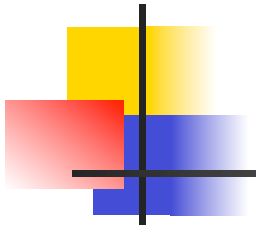
As cyclic shift left, and Bs cyclic shift up

Cannon: align blocks so all can compute



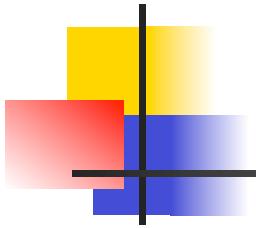
A_{01}	A_{02}	A_{03}	A_{00}	B_{10}	B_{21}	B_{32}	B_{03}
A_{12}	A_{13}	A_{10}	A_{11}	B_{20}	B_{31}	B_{02}	B_{13}
A_{23}	A_{20}	A_{21}	A_{22}	B_{30}	B_{01}	B_{12}	B_{23}
A_{30}	A_{31}	A_{32}	A_{33}	B_{00}	B_{11}	B_{22}	B_{33}

Cannon: align blocks so all can compute

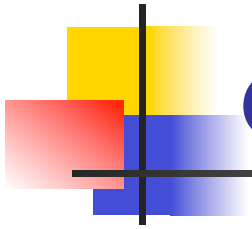


A_{02}	A_{03}	A_{00}	A_{01}	B_{20}	B_{31}	B_{22}	B_{13}
A_{13}	A_{10}	A_{11}	A_{12}	B_{30}	B_{01}	B_{32}	B_{23}
A_{20}	A_{21}	A_{22}	A_{23}	B_{00}	B_{11}	B_{02}	B_{33}
A_{31}	A_{32}	A_{33}	A_{30}	B_{10}	B_{21}	B_{12}	B_{03}

Cannon: align blocks so all can compute

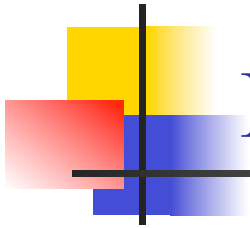


A_{03}	A_{00}	A_{01}	A_{02}	B_{30}	B_{01}	B_{22}	B_{13}
A_{10}	A_{11}	A_{12}	A_{13}	B_{00}	B_{11}	B_{32}	B_{23}
A_{21}	A_{22}	A_{23}	A_{20}	B_{10}	B_{21}	B_{02}	B_{33}
A_{32}	A_{33}	A_{30}	A_{31}	B_{20}	B_{31}	B_{12}	B_{03}



Cost of Cannon's Matrix Multiply

- Initial data alignment
  - Aligning A or B
    - Worst distance * size ≈ sqrt(p) * n^2/p
  - Total ≈ 2* sqrt(p) * n^2/p
- Interleave computation and communication
  - Compute: total = n^3 / p
  - Communicate:
    - A blocks circular shift left
    - B blocks circular shift up
    - Total cost = number of shifts * size ≈ 2* sqrt(p) * n^2/p
- Space: $3n^2/p$ per PE (A, B and C)



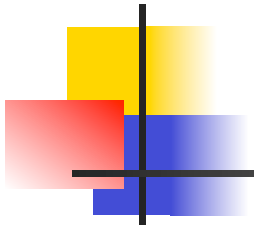
Fox's Matrix Multiply

- Also avoids space overhead
  - interleaves broadcasts for A blocks, block shifts for B, and computation
- Initial data distribution: standard block
- Initial computation
  - Broadcast Aii in row i
  - Compute: block matrix multiplication
- for j = 1 to sqrt(p)–1
  - Circular up shift B blocks
  - Broadcast Aik block in row i, where k = (j+i) mod sqrt(p)
  - Compute: block matrix multiplication and add to C block

Initial state: p_{ij} owns $blocks_{ij}$

A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{01}	B_{02}	B_{03}
A_{10}	A_{11}	A_{12}	A_{13}	B_{10}	B_{11}	B_{12}	B_{13}
A_{20}	A_{21}	A_{22}	A_{23}	B_{20}	B_{21}	B_{22}	B_{23}
A_{30}	A_{31}	A_{32}	A_{33}	B_{30}	B_{31}	B_{32}	B_{33}

Fox: broadcast diagonal block in row

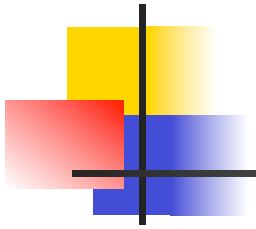


A_{00}	A_{01}	A_{02}	A_{03}	B_{00}	B_{01}	B_{02}	B_{03}
A_{10}	**A_{11}**	A_{12}	A_{13}	B_{10}	B_{11}	B_{12}	B_{13}
A_{20}	A_{21}	**A_{22}**	A_{23}	B_{20}	B_{21}	B_{22}	B_{23}
A_{30}	A_{31}	A_{32}	**A_{33}**	B_{30}	B_{31}	B_{32}	B_{33}

Fox: next diagonal, B rotates up

A_{00}	**A_{01}**	A_{02}	A_{03}	B_{10}	B_{11}	B_{12}	B_{13}
A_{10}	A_{11}	**A_{12}**	A_{13}	B_{20}	B_{21}	B_{22}	B_{23}
A_{20}	A_{21}	A_{22}	**A_{23}**	B_{30}	B_{31}	B_{32}	B_{33}
A_{30}	A_{31}	A_{32}	A_{33}	B_{00}	B_{01}	B_{02}	B_{03}

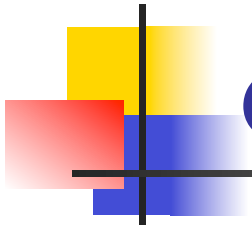
Fox: next diagonal, B rotates up



A_{00}	A_{01}	**A_{02}**	A_{03}	B_{20}	B_{21}	B_{22}	B_{23}
A_{10}	A_{11}	A_{12}	**A_{13}**	B_{30}	B_{31}	B_{32}	B_{33}
A_{20}	A_{21}	A_{22}	A_{23}	B_{00}	B_{01}	B_{02}	B_{03}
A_{30}	**A_{31}**	A_{32}	A_{33}	B_{10}	B_{11}	B_{12}	B_{13}

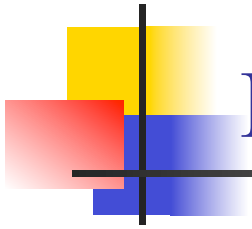
Fox: next diagonal, B rotates up

A_{00}	A_{01}	A_{02}	**A_{03}**	B_{30}	B_{31}	B_{32}	B_{33}
A_{10}	A_{11}	A_{12}	A_{13}	B_{00}	B_{01}	B_{02}	B_{03}
A_{20}	**A_{21}**	A_{22}	A_{23}	B_{10}	B_{11}	B_{12}	B_{13}
A_{30}	A_{31}	**A_{32}**	A_{33}	B_{20}	B_{21}	B_{22}	B_{23}



Cost of Fox's Matrix Multiply

- A: sqrt(p) times sqrt(p) broadcasts of blocks sized n^2/p (one-to-sqrt(p)): total volume n^2 (all of A)
- sqrt(p) circular shifts
  - Each circular shift (nearest neighbor): volume = n^2/p
- Computation time: $O(n^3/p)$



Dekel, Nassimi, Sahni Matrix Multiply

3D Mesh formulation:

- Z planes have equal k values

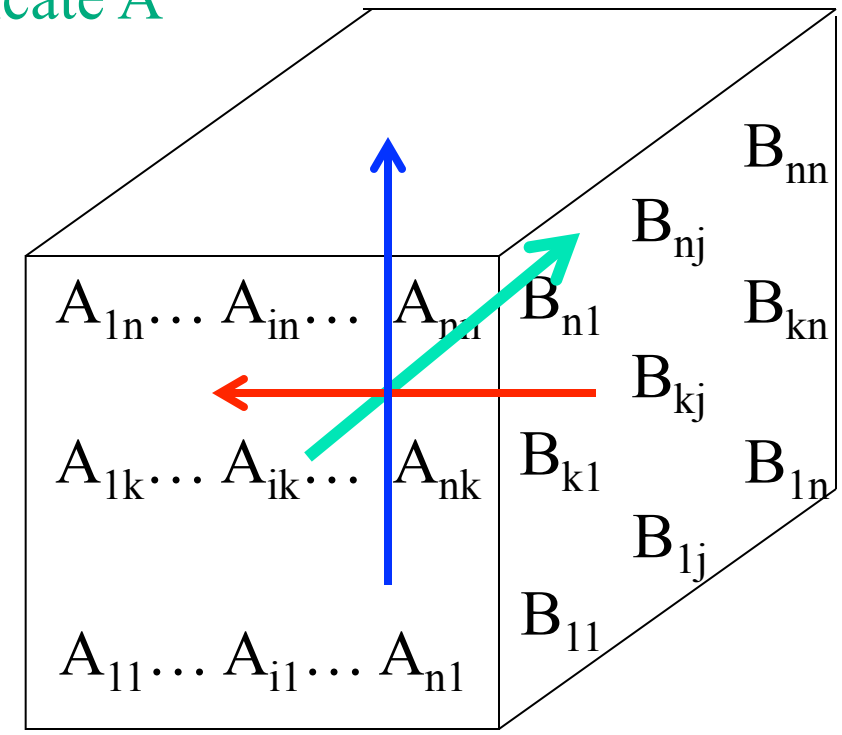
- A's columns distributed/replicated over Y planes
- B's rows distributed/replicated over X planes
  - lots of data replication

- Do all point to point multiplies in parallel
- Collapse sum reduction up / down the Z planes

Dekel, Nassimi, Sahni Matrix Multiply

Replicate A

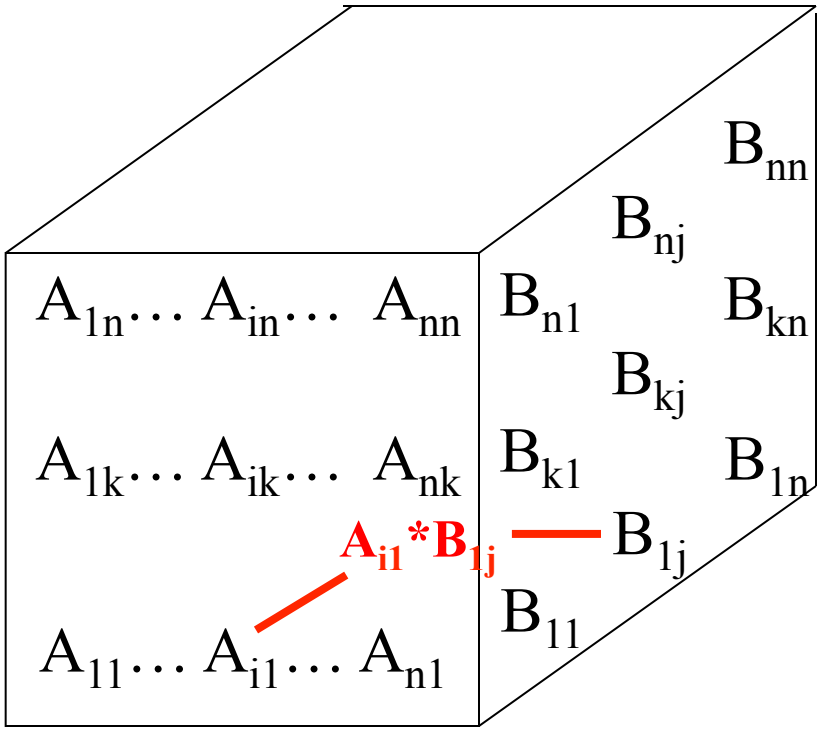
Replicate B



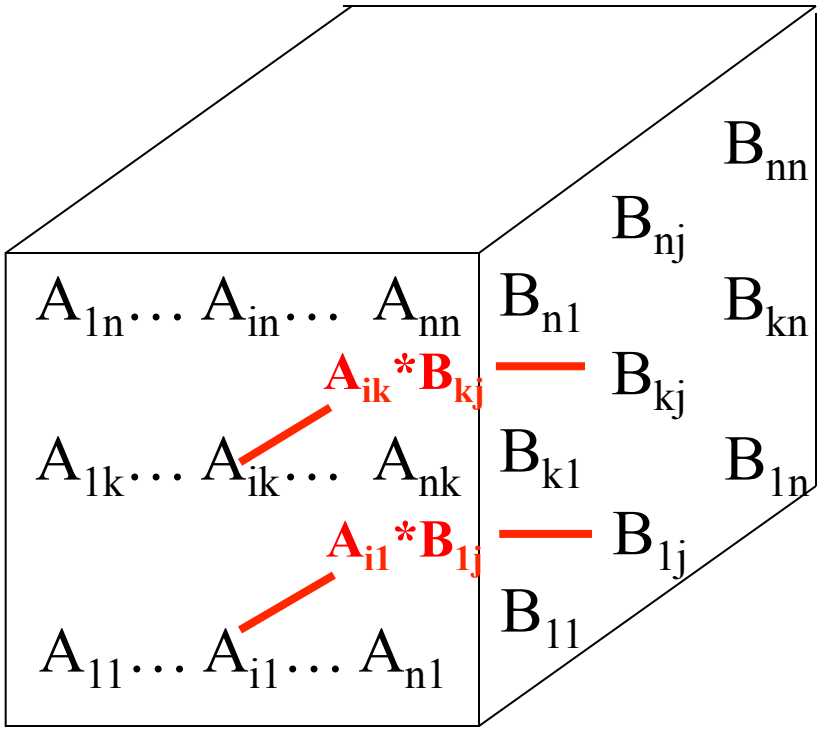
Sum reduce up

Block multiply

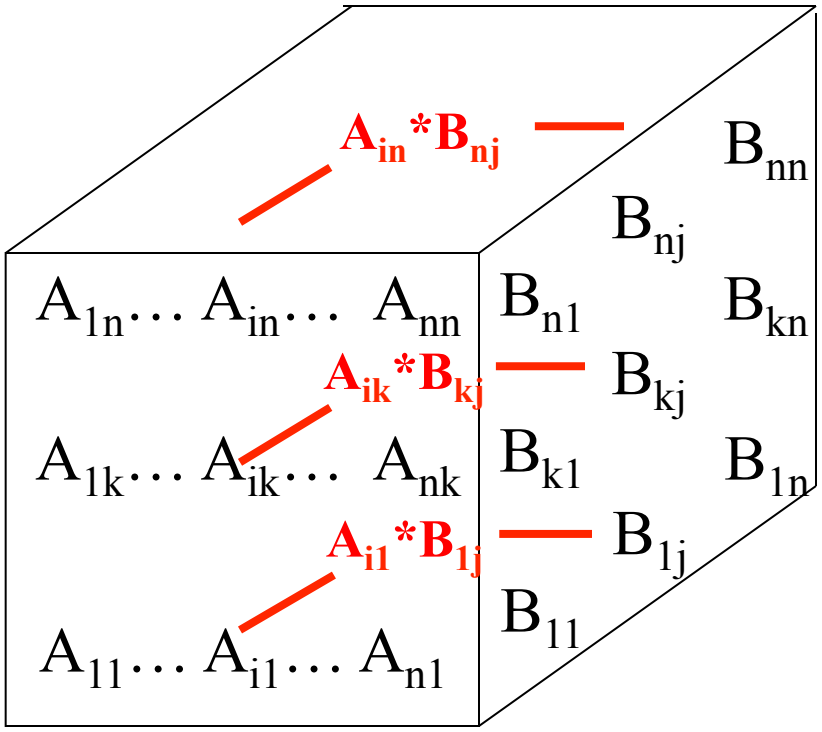
Dekel, Nassimi, Sahni Matrix Multiply



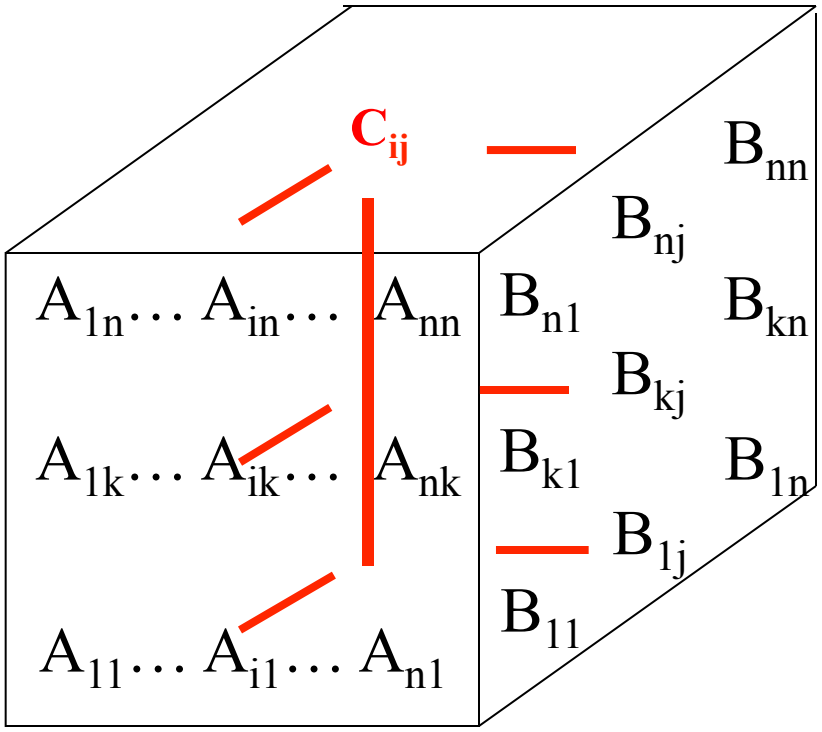
Dekel, Nassimi, Sahni Matrix Multiply



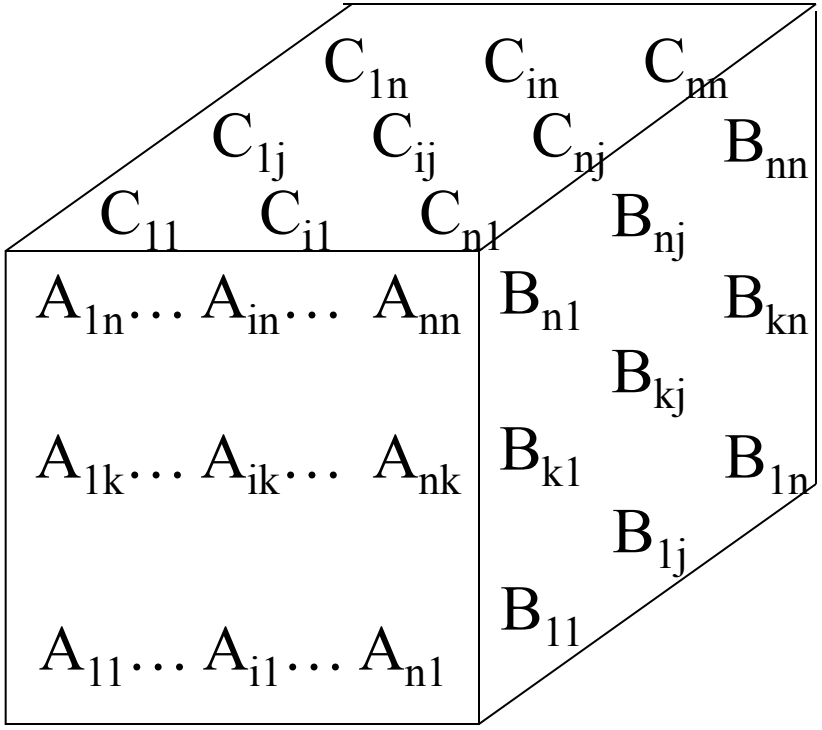
Dekel, Nassimi, Sahni Matrix Multiply

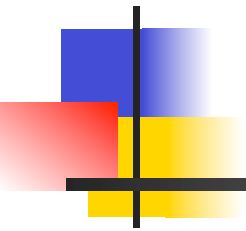


Dekel, Nassimi, Sahni Matrix Multiply



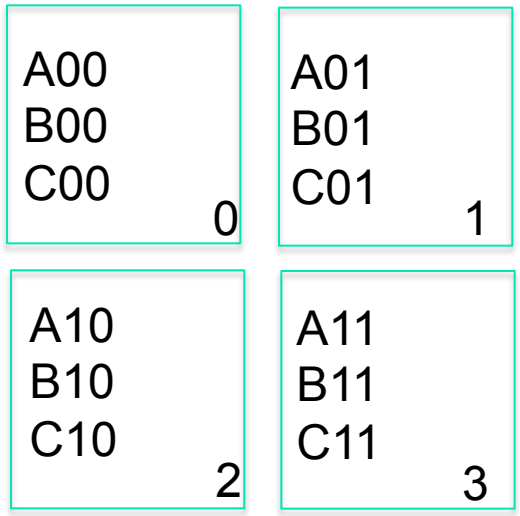
Dekel, Nassimi, Sahni Matrix Multiply



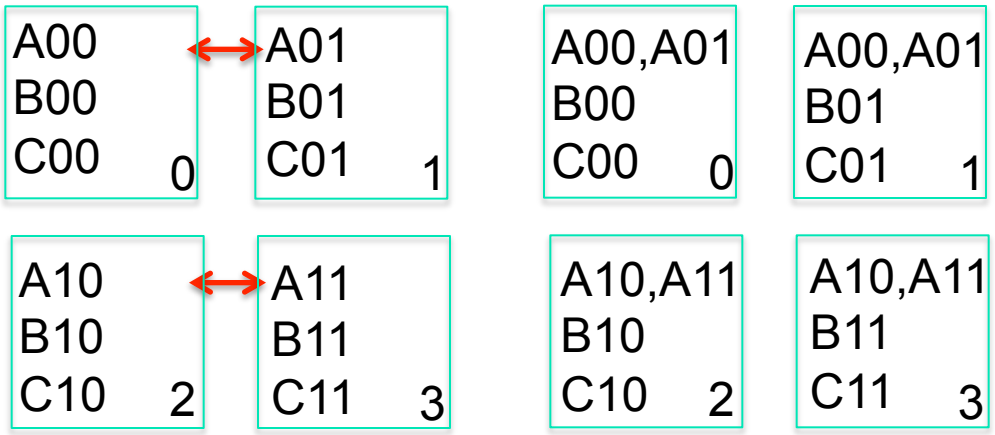


MPI 2x2 block matrix multiply

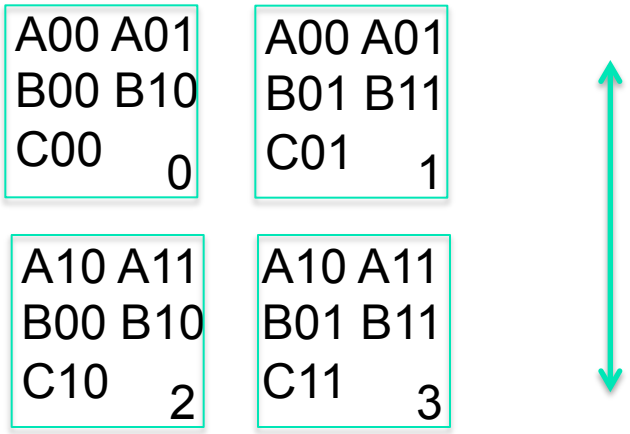
four processes, four blocks



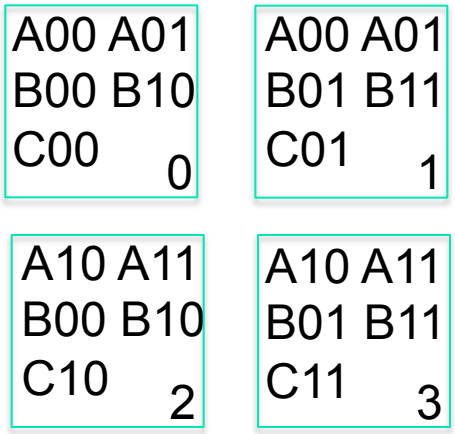
exchange rows



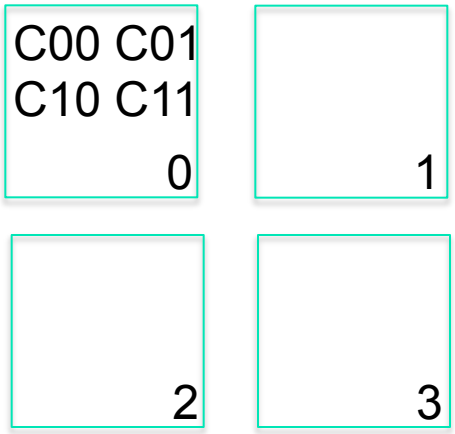
exchange columns

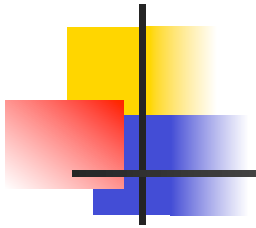


multiply


A00 A01
B00 B10
C00
0
A00 A01
B01 B11
C01
1
A10 A11
B00 B10
C10
2
A10 A11
B01 B11
C11
3

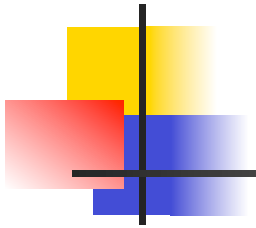
gather





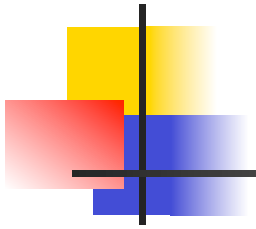
multiply a block

```
/* block size */
#define b 8

/* A, B, C are int* */
void multblock(int C[b][b], int A[b][b],  int B[b][b])  {
 int i,j,k;
 for(i=0;i<b;i++){
  for(j=0;j<b;j++){
   for(k=0;k<b;k++) C[i][j] += A[i][k]*B[k][j];
  }
 }
}
```



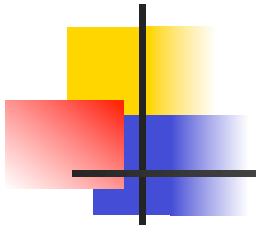
sequential main initialize

```
int main(int argc, char *argv[])  {
   int i,j,k, ioff, joff;

   int A00[b][b],  A01[b][b],  A10[b][b],  A11[b][b];
   int B00[b][b],  B01[b][b],  B10[b][b],  B11[b][b];
   int C00[b][b],  C01[b][b],  C10[b][b],  C11[b][b];

 /* initialize A, B and C blocks */

 for(i=0,ioff=b;i<b;i++,ioff++){
  for(j=0,joff=b;j<b;j++,joff++){
   A00[i][j] = i+j; A01[i][j] = i+joff; A10[i][j] = ioff+j; A11[i][j] = ioff + joff;
   B00[i][j] = i-j; B01[i][j] = i-joff; B10[i][j] = ioff-j; B11[i][j] = ioff - joff;
   C00[i][j] =   0; C01[i][j] =      0; C10[i][j] =      0; C11[i][j] =           0;
 }}
```



sequential main compute

```
multblock(C00,A00,B00); multblock(C00,A01,B10);
printf("\n C00: "); printblock(C00);

multblock(C01,A00,B01); multblock(C01,A01,B11);
printf("\n C01: "); printblock(C01);

multblock(C10,A10,B00); multblock(C10,A11,B10);
printf("\n C10: "); printblock(C10);

multblock(C11,A10,B01); multblock(C11,A11,B11);
printf("\n C11: "); printblock(C11);
```

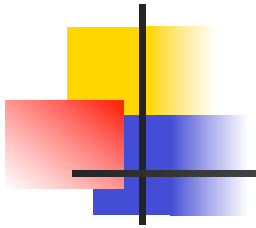


MPI code

- all pe-s declare all blocks (easiest)
- each pe initializes it's A ,B and C blocks
- exchange rows
- exchange cols
- compute
- gather

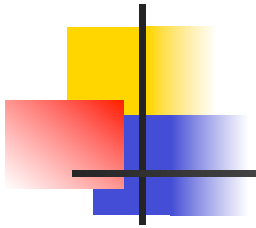
I only used blocking sends and recvs

making sure sends and recvs correctly ordered



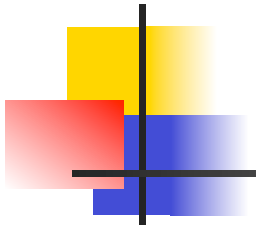
pe 0 initialize

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &my_id );
MPI_Comm_size( MPI_COMM_WORLD, &p );

MPI_Barrier(MPI_COMM_WORLD);
switch(my_id)  {

case 0:
  printf("PE0: Init\n");
  /* Initialize A00, BOO  and C00 */
   for(i=0;i<b;i++){
    for(j=0,joff=b;j<b;j++,joff++){
     A00[i][j] = i+j;  B00[i][j] = i-j;   C00[i][j] =0;
     }
   }
```



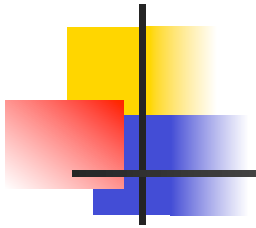
some exchanges

```
/* Row Exchange 0 <--> 1 */
printf("PE0: <--> PE1: Row Exchange\n");
 MPI_Recv( (int *)A01, b*b, MPI_INT, 1, 1, MPI_COMM_WORLD, &status);
 MPI_Send( (int *)A00, b*b, MPI_INT, 1, 2, MPI_COMM_WORLD);

/* Col Exchange 0 <--> 2 */
printf("PE0: <--> PE2: Col Exchange\n");
 MPI_Recv( (int *)B10, b*b, MPI_INT, 2, 3, MPI_COMM_WORLD, &status);
 MPI_Send( (int *)B00, b*b, MPI_INT, 2, 4, MPI_COMM_WORLD);

Row  EXCHANGE  in PE1:

 /* Row Exchange 0 <--> 1 */
printf("PE1: <--> PE0: Row Exchange\n");
 MPI_Send( (int *)A01, b*b, MPI_INT, 0, 1, MPI_COMM_WORLD);
 MPI_Recv( (int *)A00, b*b, MPI_INT, 0, 2, MPI_COMM_WORLD, &status);
```



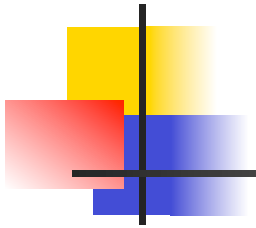
pe 0 computes

```
/* Block Multiply C00 = A00*B00 + A01*B10 */
 multblock(C00,A00,B00);
 multblock(C00,A01,B10);
```



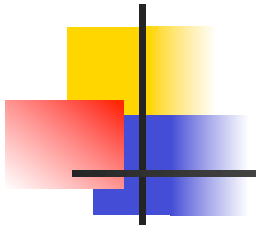
pe 0 gathers

```
/* Gather */
printf("PE0: Gather C01 <-- PE1\n");
 MPI_Recv( (int *)C01, b*b, MPI_INT, 1, 5, MPI_COMM_WORLD, &status);
printf("PE0: Gather C10 <-- PE2\n");
 MPI_Recv( (int *)C10, b*b, MPI_INT, 2, 6, MPI_COMM_WORLD, &status);
printf("PE0: Gather C11 <-- PE3\n");
 MPI_Recv( (int *)C11, b*b, MPI_INT, 3, 7, MPI_COMM_WORLD, &status);
```



pe 0 prints

```
/* Print */
printf("PE0: Print blocks\n");
printf("\n C00: ");  printblock(C00);
printf("\n C01: ");  printblock(C01);
printf("\n C10: ");  printblock(C10);
printf("\n C11: ");  printblock(C11);
printf("\n");
break;
```



all pe-s happy ☺

```
EXIT:
    MPI_Finalize();
```