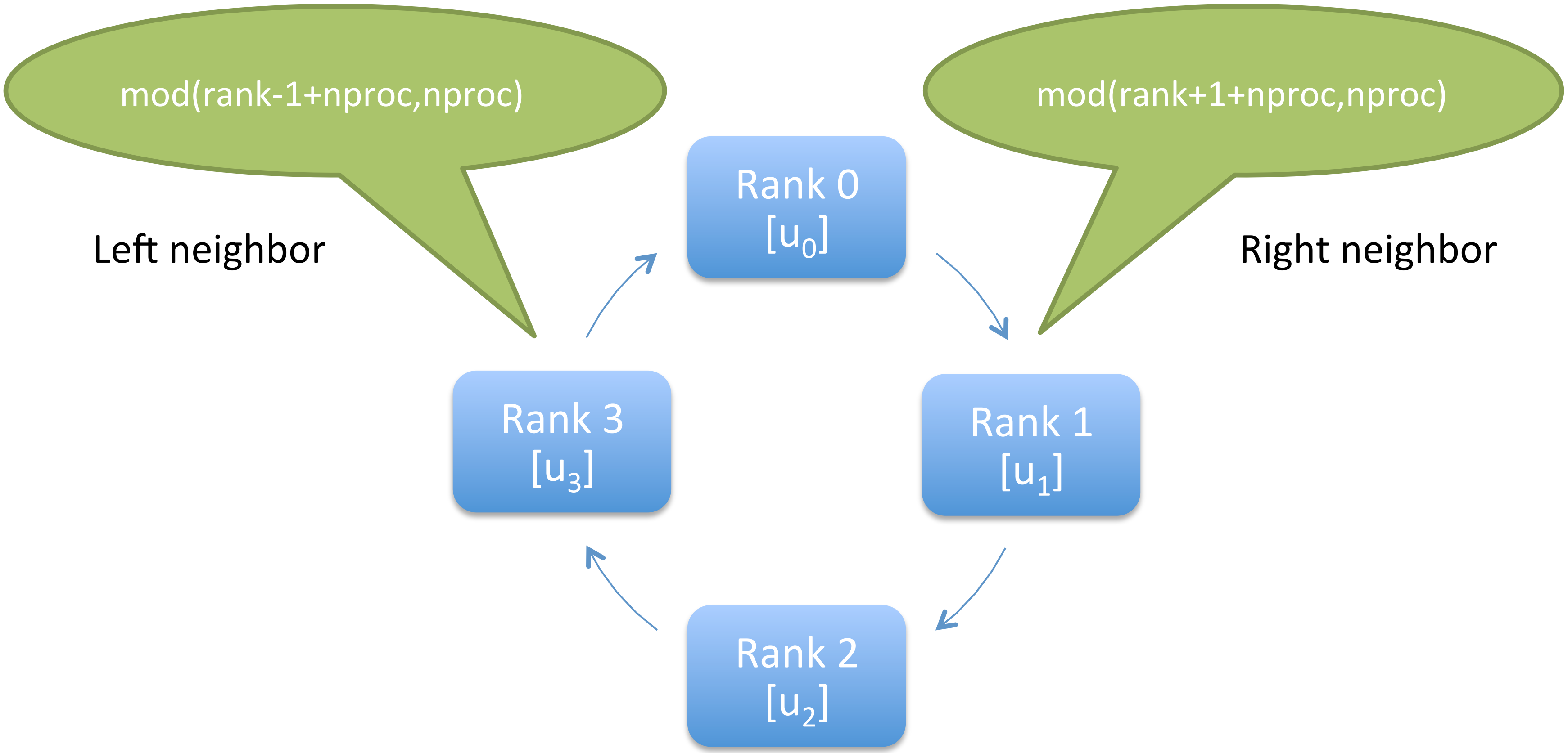
Hybrid Parallel Programming
Part 2

Bhupender Thakur
HPC @ LSU

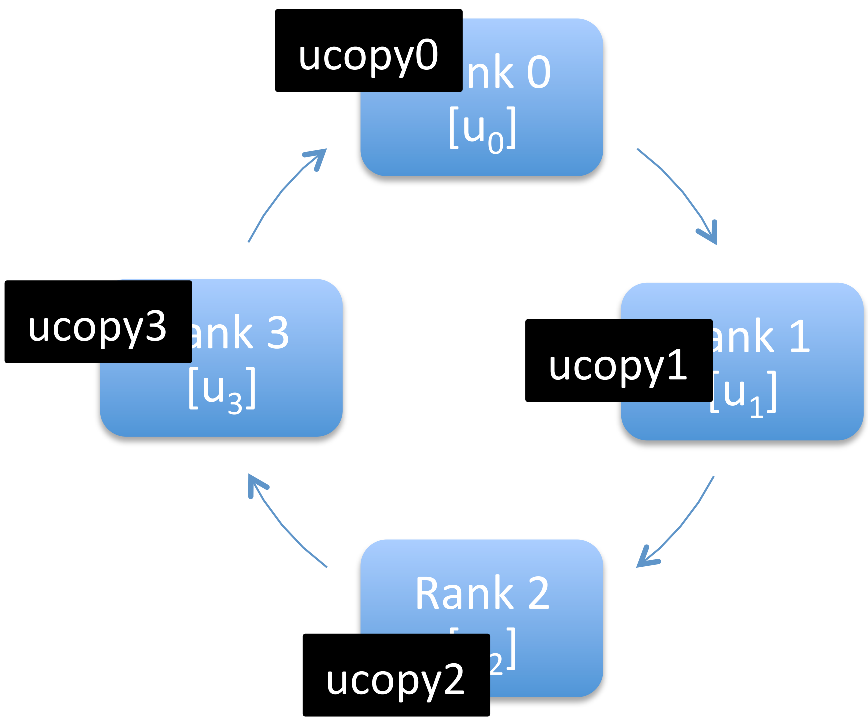
Overview

- Ring exchange
- Jacobi
- Advanced
- Overlapping

Exchange on a ring

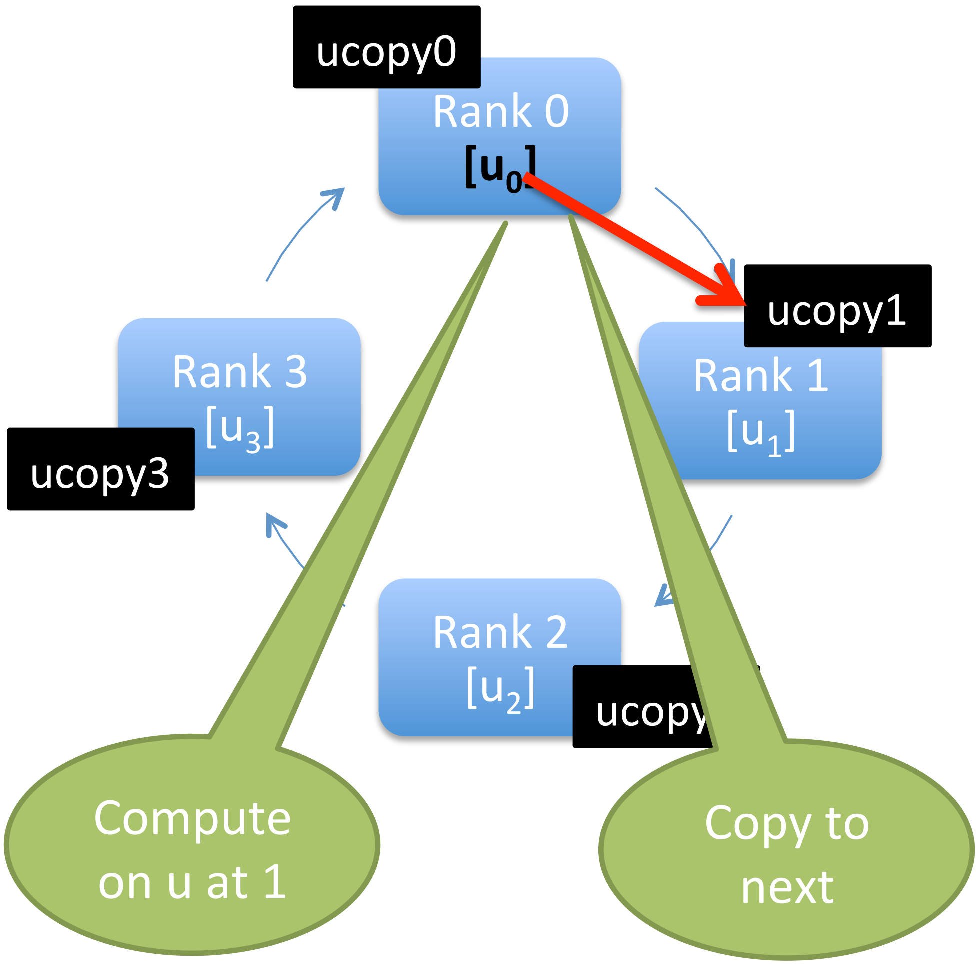


Exchange on a ring



1. Setup initial vector across mpi processes
2. Start copy of vector u to its neighbor in variable ucopy
3. In the meantime do useful work on u
4. At the end of computation and copy, replace u with newer value from ucopy
5. Repeat steps 1-3 until all vector components have been cycled

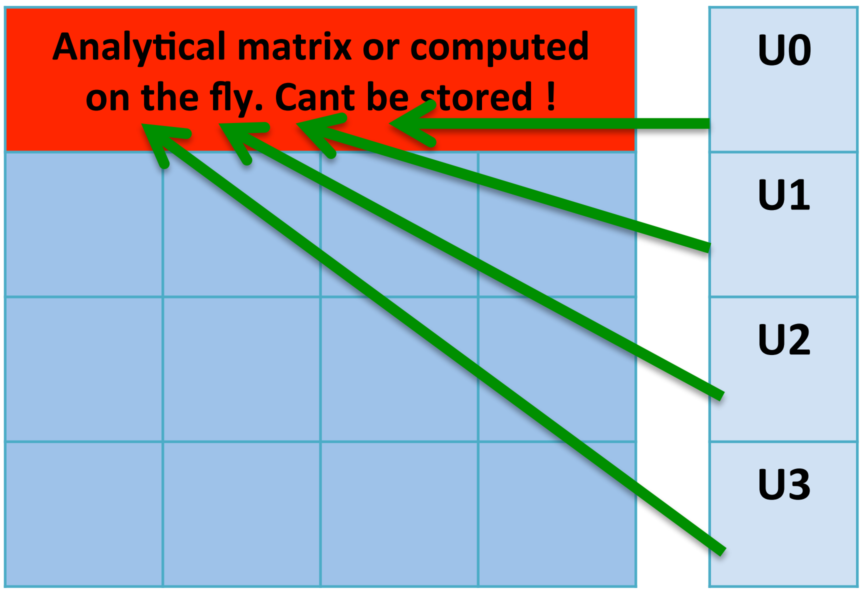
Exchange on a ring



Only two ways to improve performance:

- Reduce mpi processes so as to reduce communication and communication overhead
- Overlap computation and communication

Where is it useful



Vector space is **HUGE** and distributed vectors are the only way to go

Matrix is analytical, so can be computed on the fly. It either does not need storage or cant be stored at all !

Code Walkthrough

Vector space is **HUGE** and distributed vectors are the only way to go

Matrix is analytical, so can be computed on the fly. It either does not need storage or cant be stored at all !

Exercise

- Replace Blocking communication in the code ring with non blocking
  calls

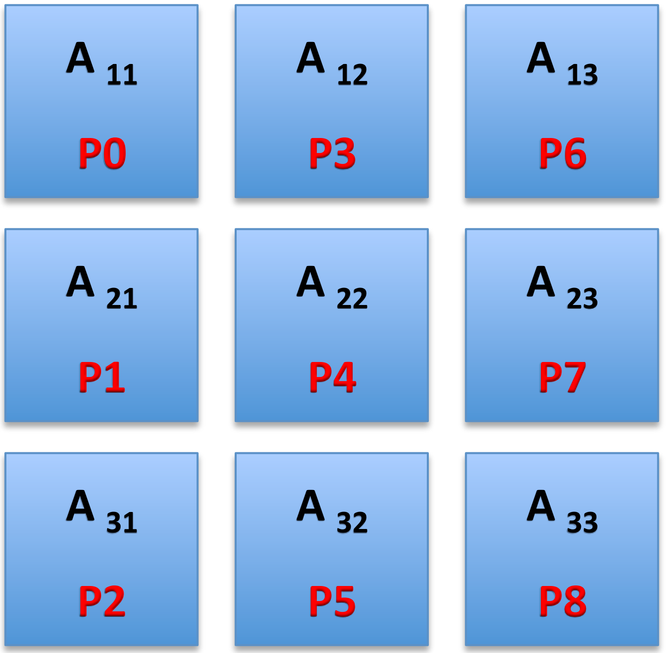
- Create a local vector c of same dimensions. Define a function that replicates a banded matrix

$$A(i,j) = \begin{cases} 1, & i=j \\ \text{random}(x), & 1 < \text{abs}(i-j) < 100 \end{cases}$$

Cannon’s Algorithm

Distributed matrix-multiplication

- **Matrix is distributed across a 2D grid of processors**
- **Matrix is too large to be present on one node**
- **Requires communication to get sub-matrices from other ranks**



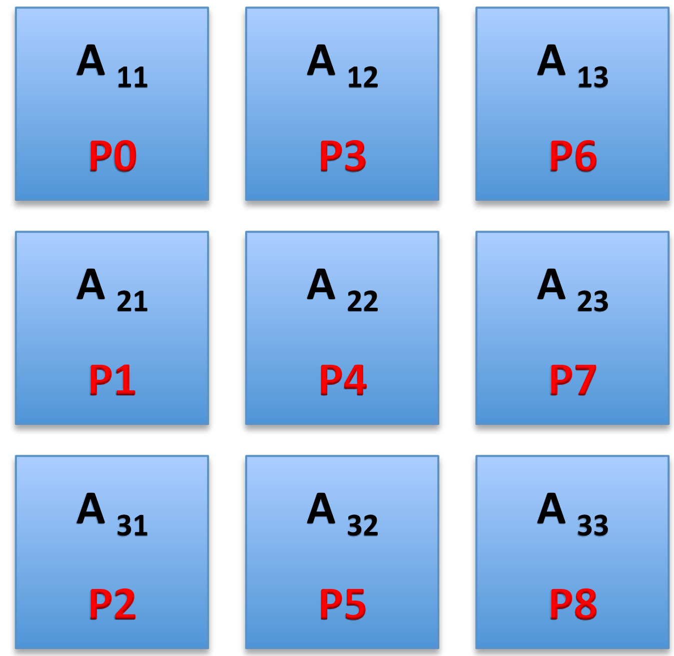
Cannon's Algorithm

Distributed matrix-multiplication

- **How Big?**

  Consider:
  Size =[1million X 1million]
  Memory=
  (10^6*10^6*(4))/(1024^3)

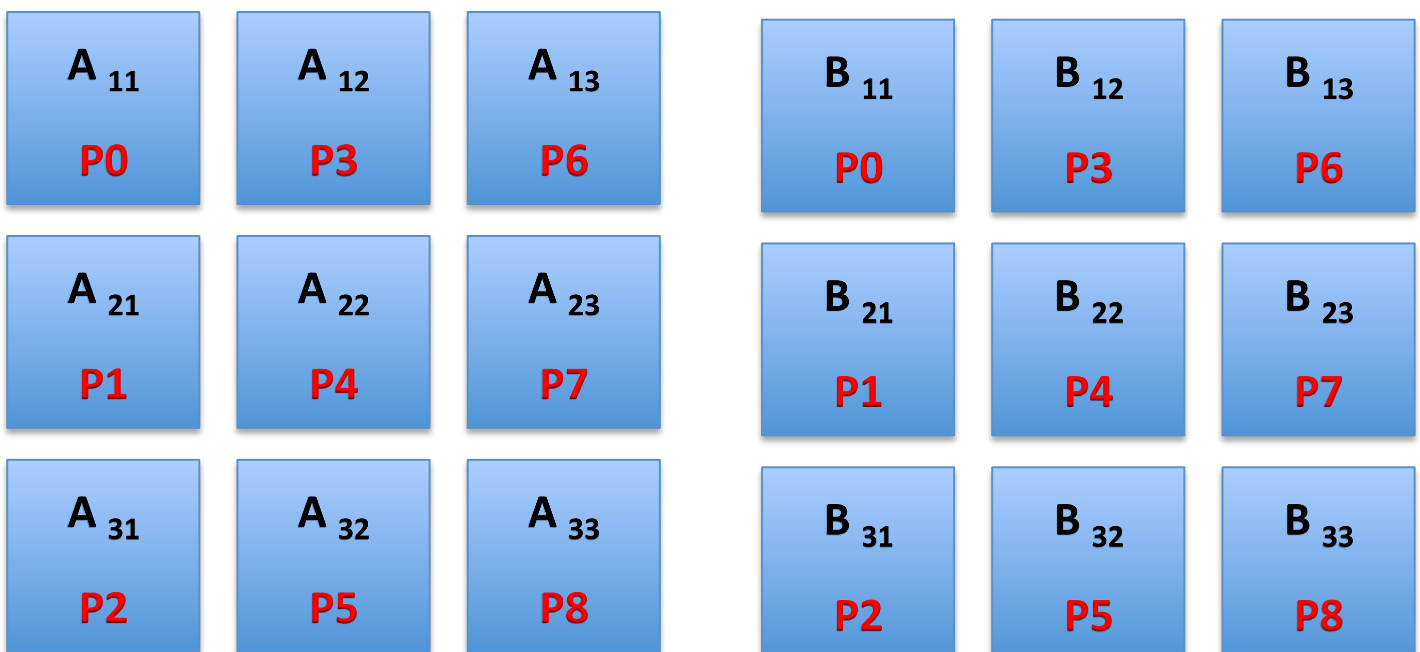
- **How many Supermike nodes?**
  3725/32=116 Nodes



Cannon’s Algorithm

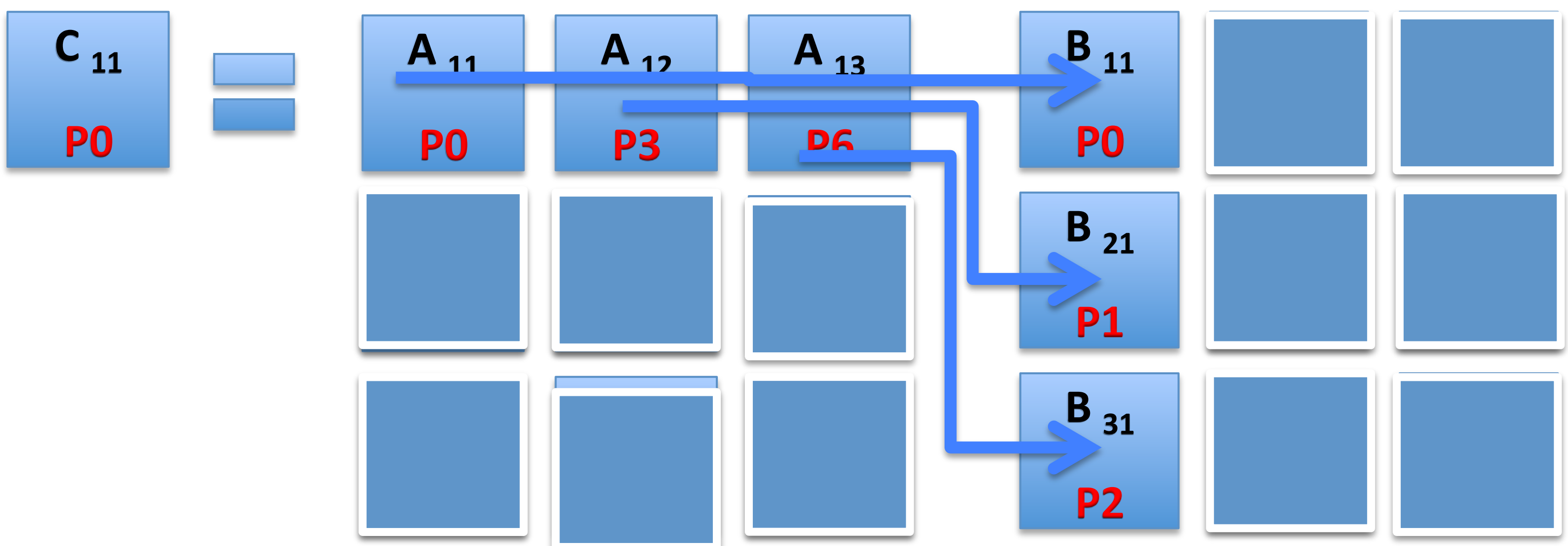
Distributed matrix-multiplication

- Input matrices are distributed, so should the output matrix be.



Cannon's Algorithm
Initial arrangement

- Idea is to bring required pieces of matrix to each processor
  Start with C_{11}

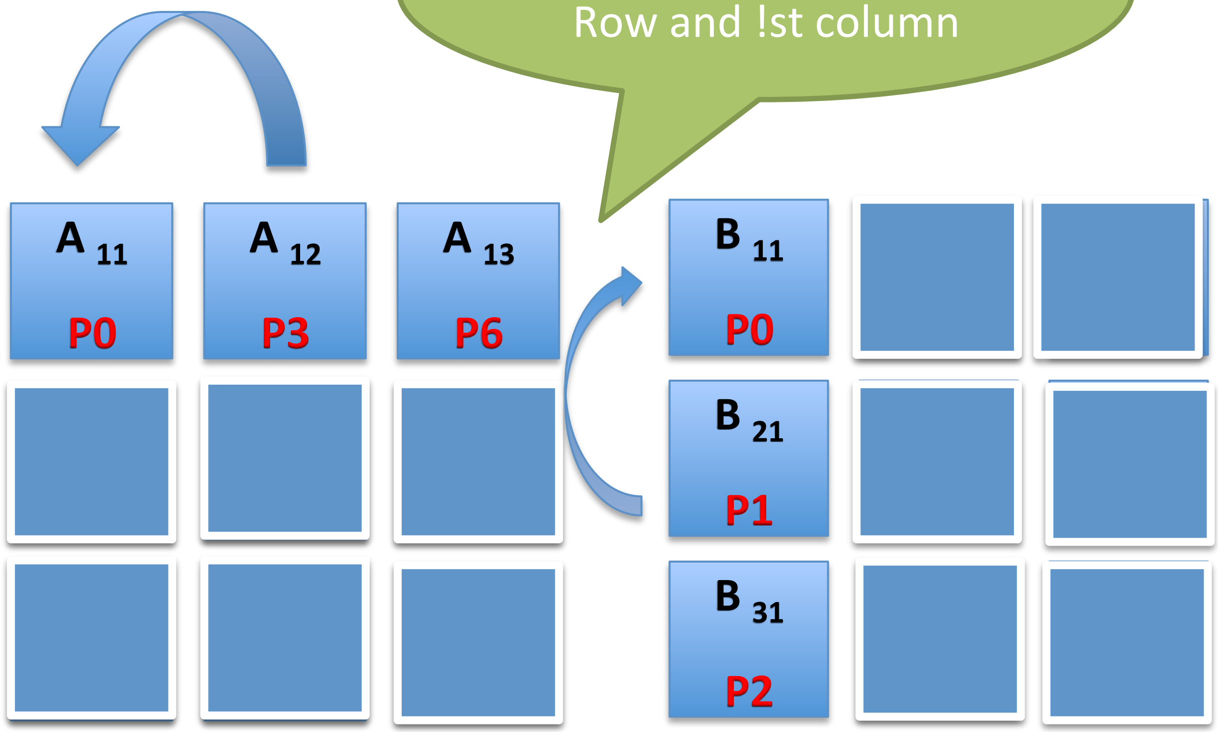


Cannon's Algorithm

Initial arrangement

No problem with Ist Row and !st column

First iteration partly computes C_{11}: $A_{11} * B_{11}$

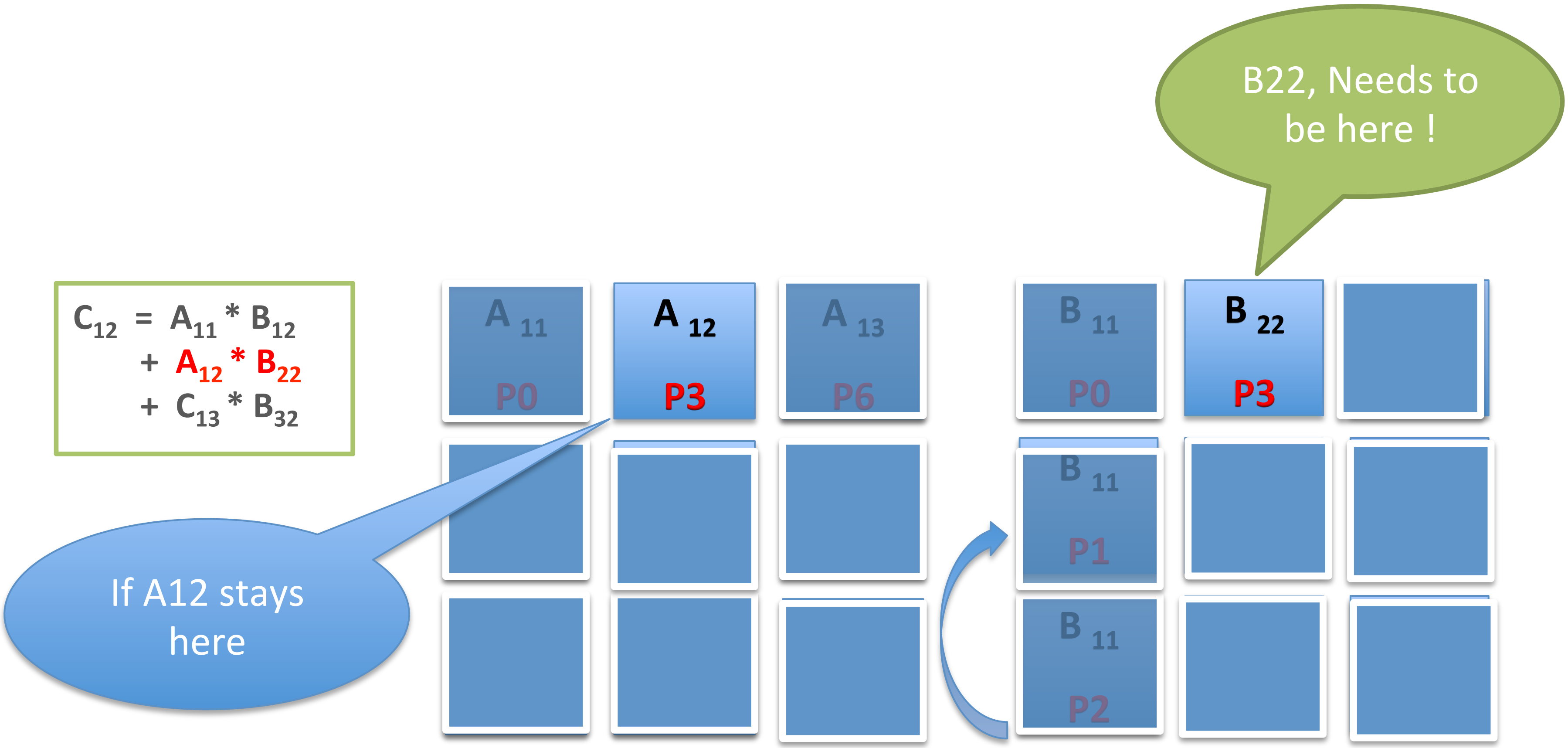


$$
\begin{aligned}
C_{11} = \; & A_{11} * B_{11} \\
& + A_{12} * B_{21} \\
& + C_{13} * B_{31}
\end{aligned}
$$

Cannon's Algorithm

Initial arrangement

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22} + C_{13} * B_{32}$$

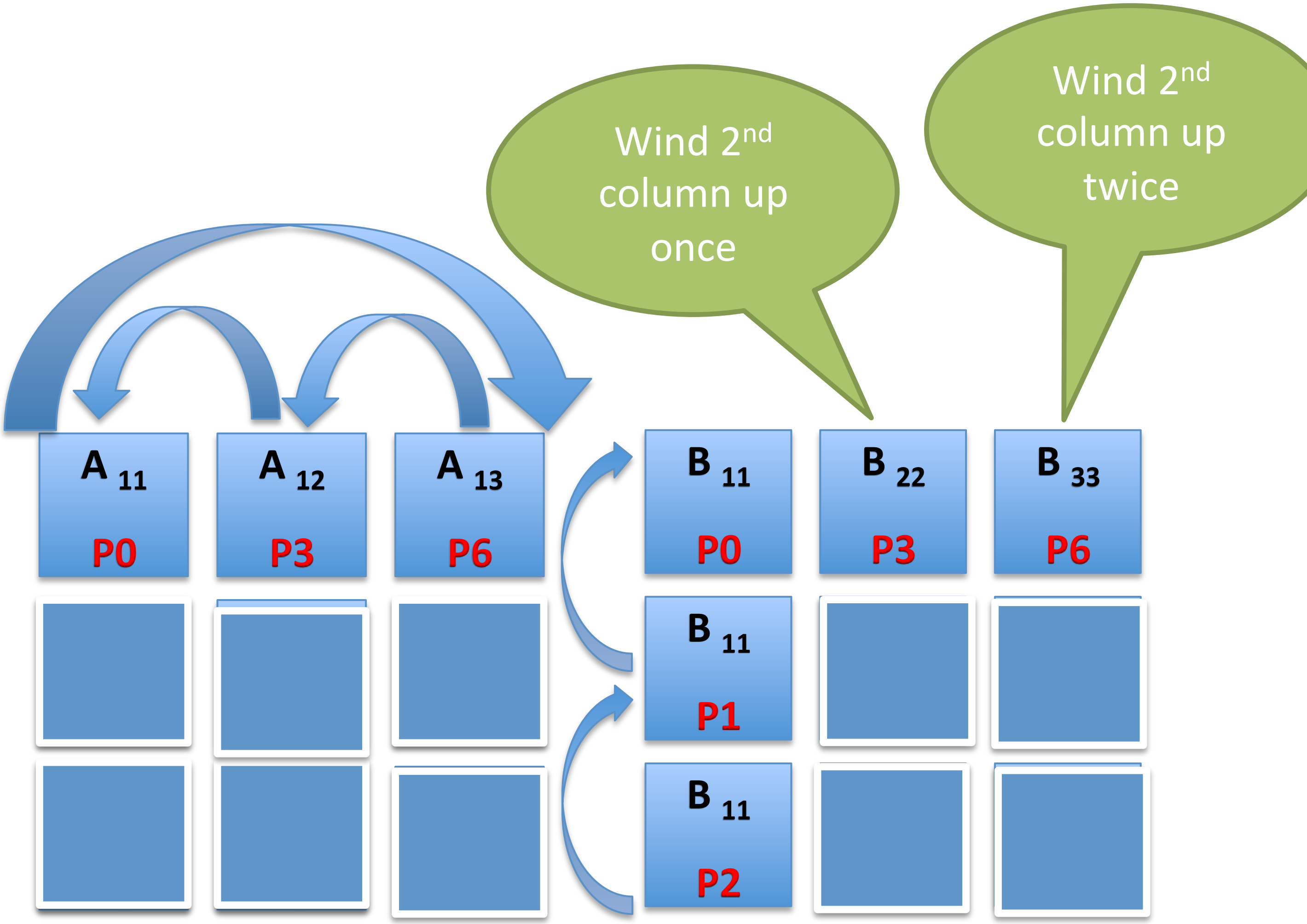


Cannon's Algorithm

Initial arrangement

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32}$$

$$C_{13} = A_{11} * B_{13} + A_{12} * B_{23} + A_{13} * B_{33}$$

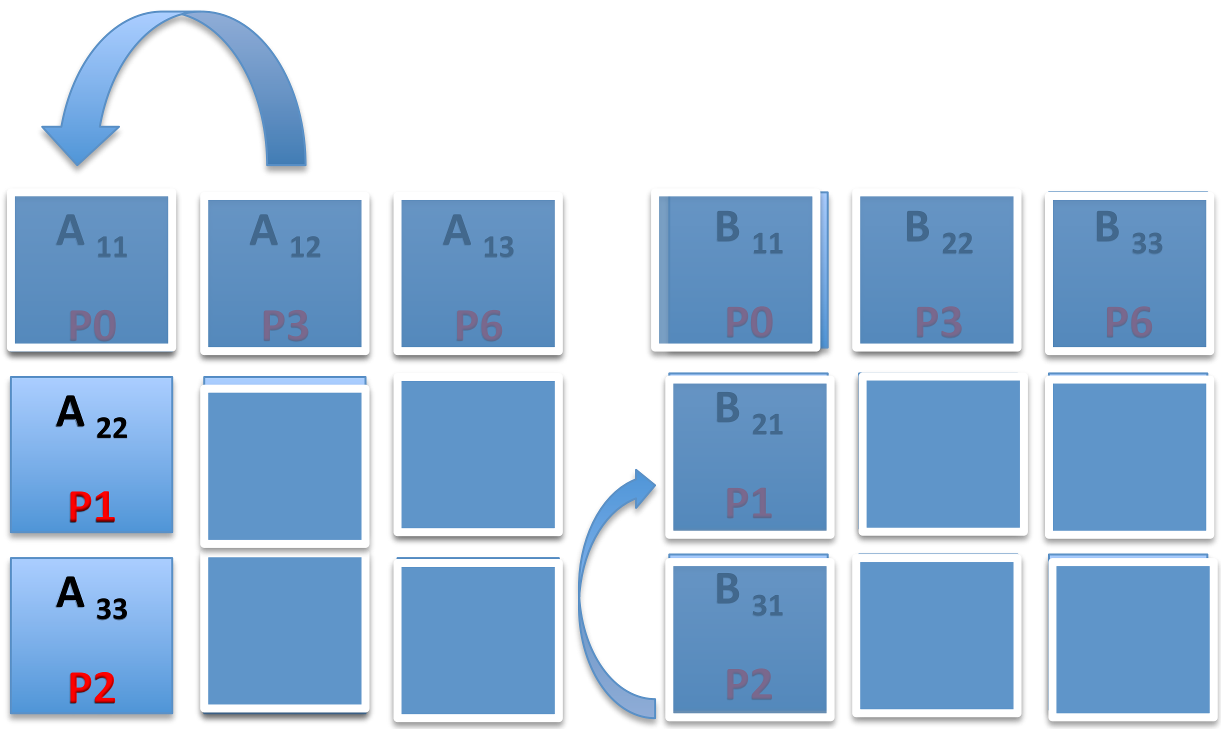


Cannon's Algorithm

Initial arrangement

$C_{21} = A_{21} * B_{11}$
$+ A_{22} * B_{21}$
$+ A_{23} * B_{31}$

$C_{31} = A_{31} * B_{11}$
$+ A_{32} * B_{21}$
$+ A_{33} * B_{31}$

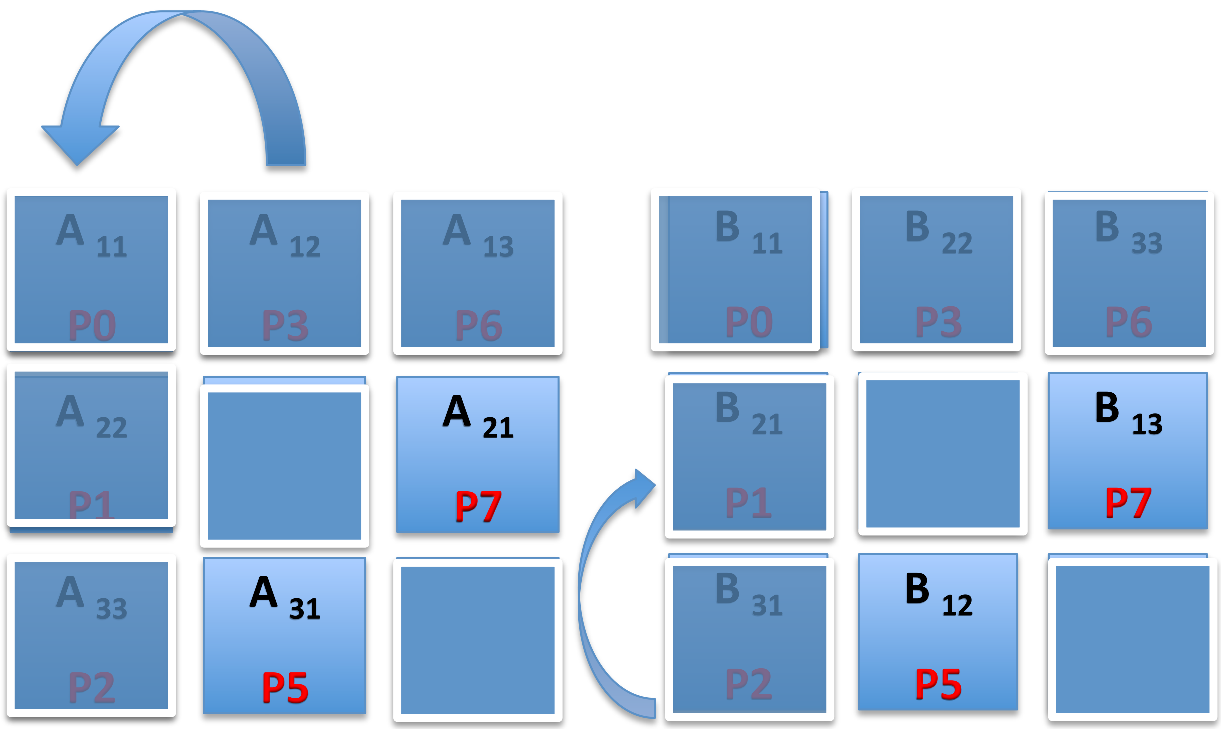


Cannon's Algorithm

Initial arrangement

$C_{32} = A_{31} * B_{12}$
$\quad + \sout{A_{32} * B_{22}}$
$\quad + \sout{A_{33} * B_{32}}$

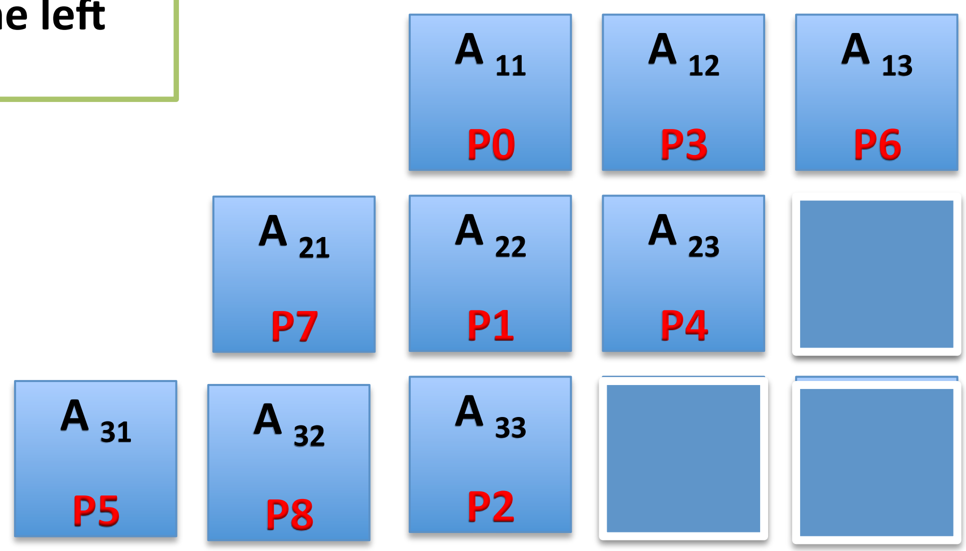
$C_{23} = A_{21} * B_{13}$
$\quad + \sout{A_{22} * B_{23}}$
$\quad + \sout{A_{23} * B_{33}}$



Cannon's Algorithm

Initial arrangement

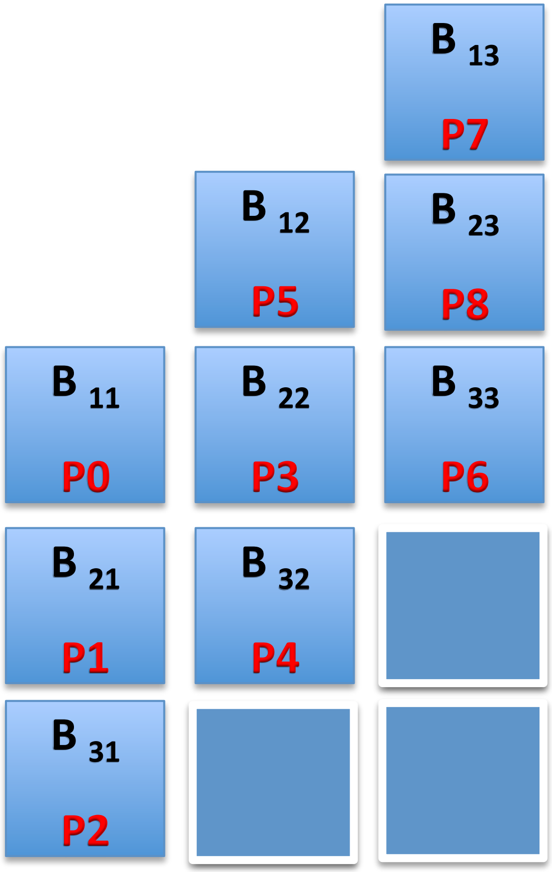
Rows have been shifted to the left



Cannon's Algorithm

Initial arrangement

Columns have been shifted upwards



Cannon's Algorithm

Initial arrangement

Matrices after initial skewing

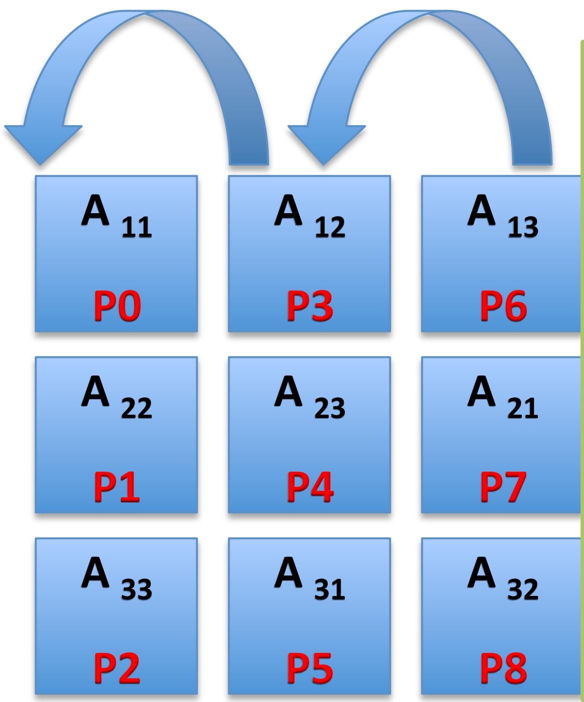
A_{11} P0	A_{12} P3	A_{13} P6
A_{22} P1	A_{23} P4	A_{21} P7
A_{33} P2	A_{31} P5	A_{32} P8

B_{11} P0	B_{22} P3	B_{33} P6
B_{21} P1	B_{32} P4	B_{13} P7
B_{31} P2	B_{12} P5	B_{23} P5

Cannon's Algorithm

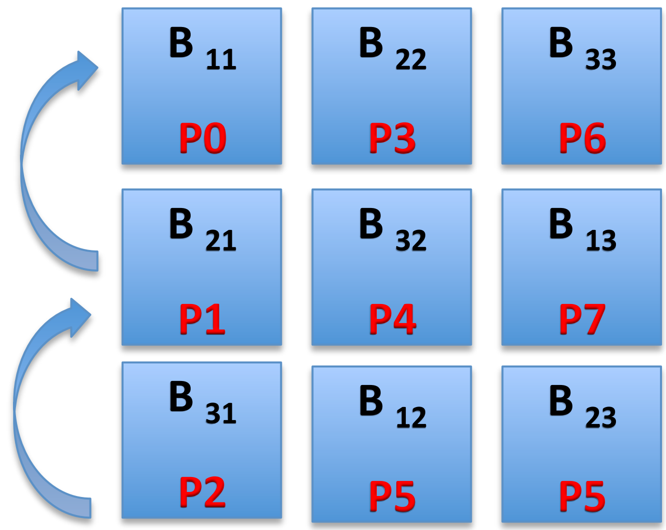
Algorithm: Iterative compute and transfer

The iteration in every step multiplies local submatrix of A with local submatrix of B
Partial results are added to sub-matrix of C. Full matrices are never needed.



Sub-matrices among rows are copied from the right neighbor to the left

Sub-matrices among columns are copied from the bottom neighbor to the top

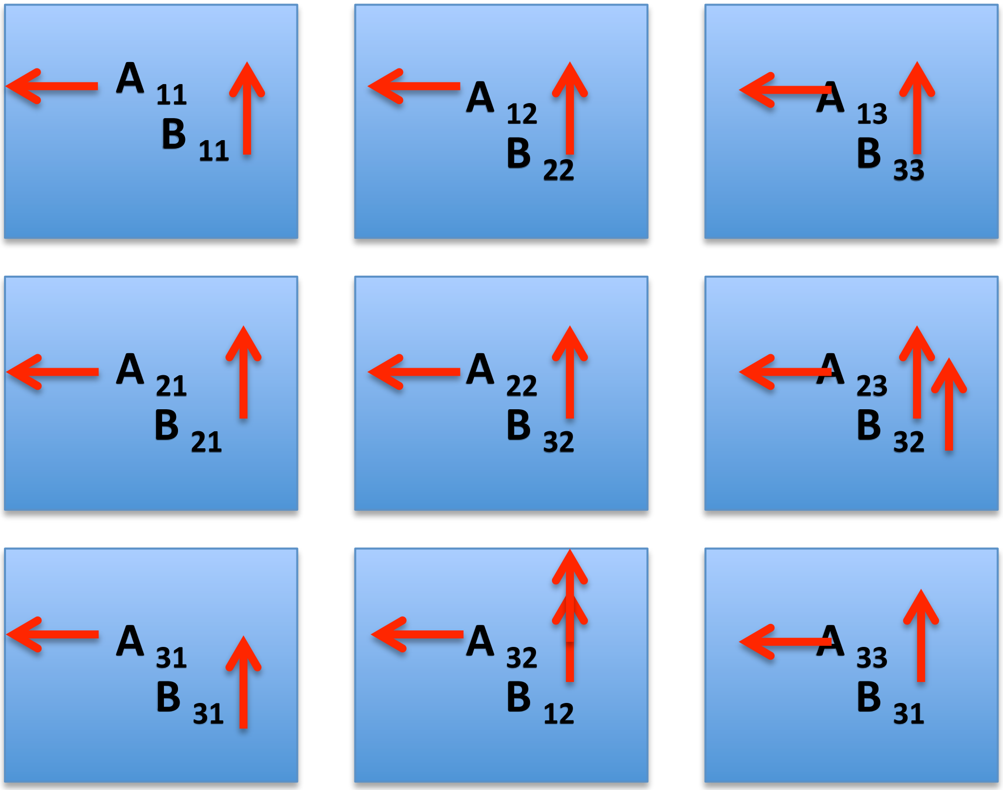


Cannon's Algorithm

Algorithm: Iterative compute and transfer

Sub-matrices of A are copied from the right neighbor to the left, and sub-matrices for B are copied from bottom to the top neighbor

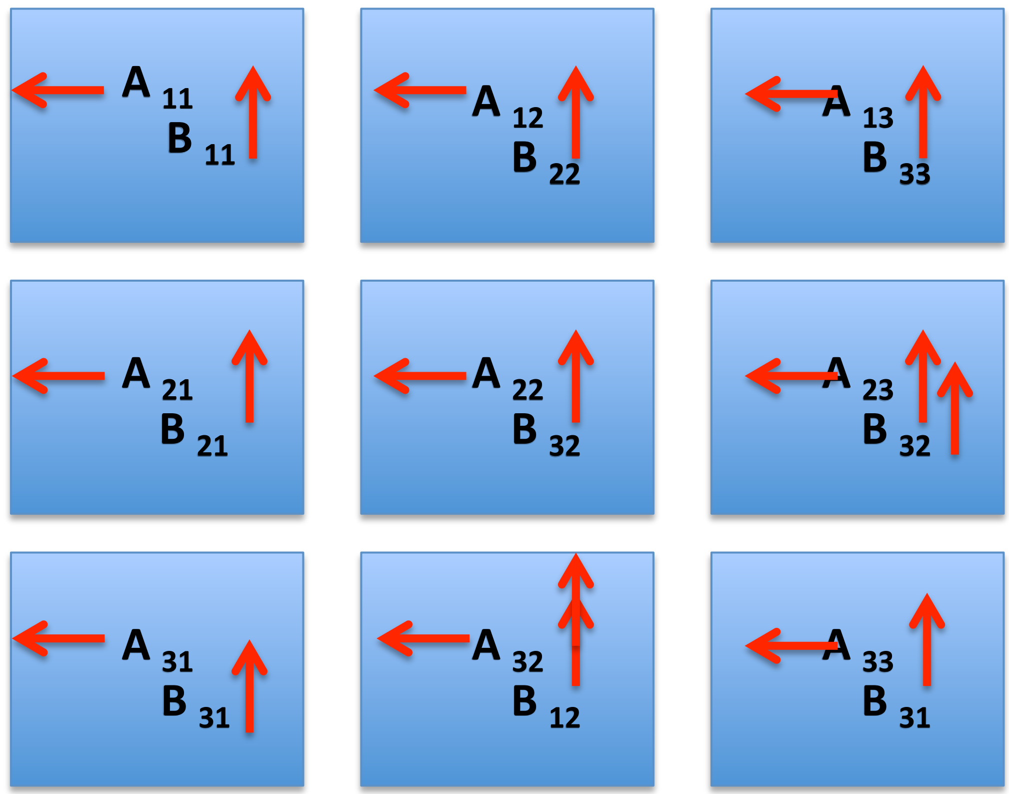
Partial results from each iteration are added to local matrix C



Cannon's Algorithm

Algorithm: Iterative compute and transfer

Algorithm begs for overlap of computation and communication !



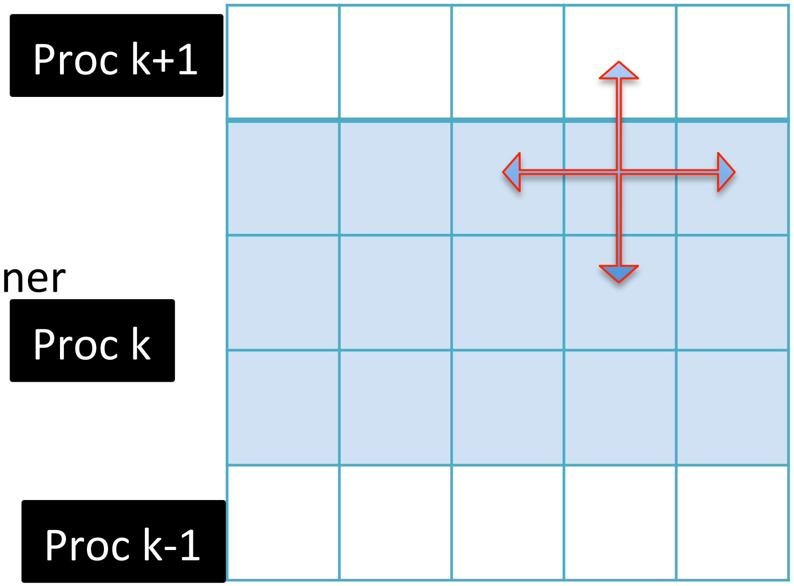
Jacobi solver

1. Master calculates boundaries
   Isends and Irecvs them on other processes
   Send down, Send Up, Recv down Recv up

2. All threads chime in and start working on inner points. Check convergence

3. Wait for communication to finish
   Copy new to old

4. Proceed with another iteration if needed



Jacobi solver

```
/* Use master thread to calculate and communicate boundies */
  #pragma omp master
  {
   /* Loop over top and bottom boundry */
   for (k = 1; k <= NC; k++){
     /*Calculate average of neighbors as new value (Point Jacobi method) */
     t[*new][1][k] = 0.25 *
                 (t[old][2][k] + t[old][0][k] +
                  t[old][1][k+1] + t[old][1][k-1]);
     t[*new][nrl][k] = 0.25 *
                   (t[old][nrl+1][k] + t[old][nrl-1][k] +
                    t[old][nrl][k+1] + t[old][nrl][k-1]);
     /* Calculate local maximum change from last step */
     /* Puts thread's max in d */
     d = MAX(fabs(t[*new][1][k] - t[old][1][k]), d);
     d = MAX(fabs(t[*new][nrl][k] - t[old][nrl][k]), d);
   }
```

Jacobi solver

```
if (nPEs!=1){
      /* Exchange boundries with neighbor tasks */
      if (myPE < nPEs-1 )
        /* Sending Down; Only npes-1 do this */
        MPI_Isend(&(t[*new][nrl][1]), NC, MPI_FLOAT,
                  myPE+1, DOWN, MPI_COMM_WORLD, &request[0]);
      if (myPE != 0)
       /* Sending Up; Only npes-1 do this */
       MPI_Isend(&t[*new][1][1], NC, MPI_FLOAT,
                 myPE-1, UP, MPI_COMM_WORLD, &request[1]);
      if (myPE != 0)
        /* Receive from UP */
        MPI_Irecv(&t[*new][0][1], NC, MPI_FLOAT,
                  MPI_ANY_SOURCE, DOWN, MPI_COMM_WORLD, &request[2]);
      if (myPE != nPEs-1)
        /* Receive from DOWN */
        MPI_Irecv(&t[*new][nrl+1][1], NC, MPI_FLOAT,
                  MPI_ANY_SOURCE, UP, MPI_COMM_WORLD, &request[3]);
    }
   }

         Computation goes here

MPI_Wait
```

Jacobi solver

Code walkthrough

```
/* Everyone calculates values and finds local max change */
  #pragma omp for schedule(runtime) nowait
    for (i = 2; i <= nrl-1; i++)
      for (j = 1; j <= NC; j++){
        t[*new][i][j] = 0.25 *
                        (t[old][i+1][j] + t[old][i-1][j] +
                         t[old][i][j+1] + t[old][i][j-1]);
        d = MAX(fabs(t[*new][i][j] - t[old][i][j]), d);
      }

  /*Local max change become taks-global max change */
  #pragma omp critical
    dt = MAX(d, dt); /* Finds max of the d's */
}
```

Advanced Hybrid Programming