Parallel Matrix Algorithms (part 1)

Matrix-Vector Multiplication

- Multiplying a dense $n \times n$ matrix A with an $n \times 1$ vector x: $y = Ax$

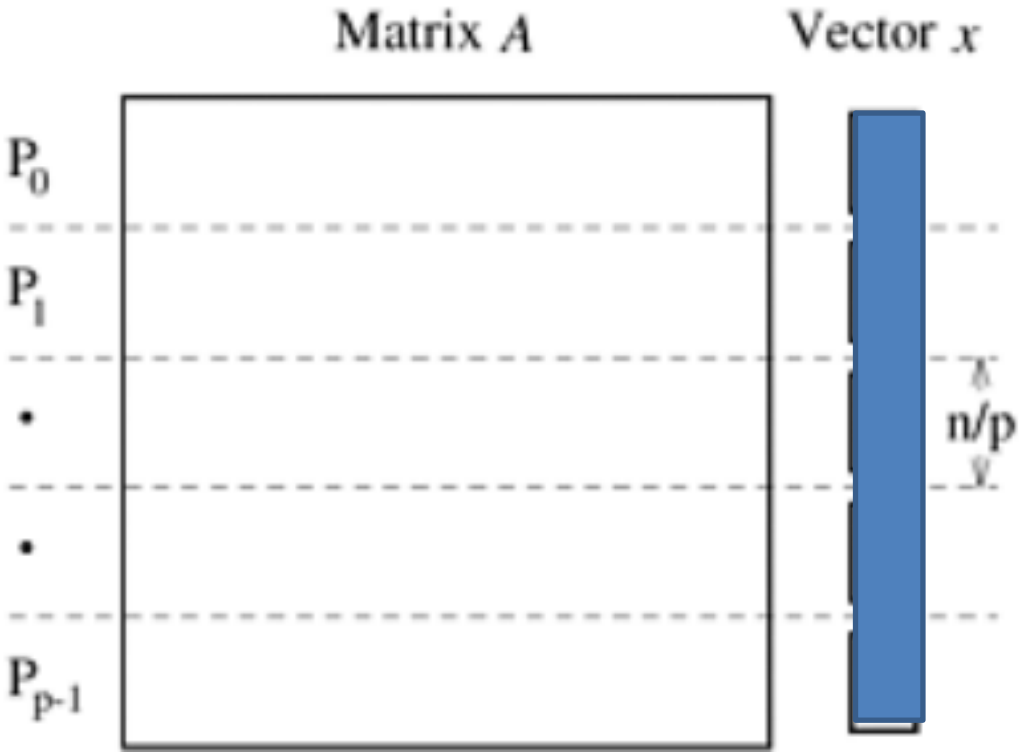
```
1.    procedure MAT_VECT ( A, x, y)
2.    begin
3.       for i := 0 to n - 1 do
4.       begin
5.          y[i]:=0;
6.          for j := 0 to n - 1 do
7.              y[i] := y[i] + A[i, j] × x[j];
8.       endfor;
9.    end MAT_VECT
```

- Sequential run time: $W = n^2$

Row-wise Block-Striped Decomposition

Step 1

- Row-wise 1D block partition is used to distribute matrix.
- An all-to-all broadcast is used to distribute the full vector among all the processes.



Step 2

- Each task performs dot product computation using rows mapped to it and replicated vector x.

Step 3

- Distribute the result vector y to all processes by collective communication.
- MPI_Allgatherv(**void** **sendbuf*, **int** *sendcount*, **MPI_Datatype** *sendtype*, **void** **recvbuf*, **int** **recvcounts*, **int** **displs*, **MPI_Datatype** *recvtype*, **MPI_Comm** *comm*)
  - Gather data from all tasks and deliver the combined data to all tasks
  - The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf. These blocks need not all be the same size.
  - *recvcounts*: element j of this array is the number of elements being gathered from process j.
  - *displs*: element j of this array is the displacement from the first element of *recvbuf* where the first element gathered from process j is to be stored.

Parallel Run Time Analysis

Message Passing Costs in Parallel Computers

- ***Startup time*** (t_s): The startup time is the time required to handle a message at the sending and receiving nodes. This include time to prepare the message(put in envelope), the time to execute routing algorithm, and the time to establish an interface between the local node and the router.
- ***Per-word transfer time*** (t_w): If the channel bandwidth is r words per second, then $t_w = \frac{1}{r}$.
- ***Per-hop time*** (t_h): After a message leaves a node, it takes a finite amount of time to reach the node in its next path.
- ***Cost model for communicating messages****:* Suppose that a message of size m is being transmitted through a network, Assume it traverses l links, the total communication cost is:

$$t_{comm} = t_s + lt_h + t_w m$$

- Assume that the # of processes p is less than n
- Assume that we run the program on a parallel machine adopting hypercube interconnection network (**Table 4.1** lists communication times of various communication schemes)

1. Each process is responsible for n/p rows of matrix. The complexity of the dot production portion of the parallel algorithm is $\Theta(n^2/p)$
2. Parallel communication time for all-to-all broadcast communication to replicate result vector y:
    - Each process needs to send a message of size n/p to all processes. This takes time $t_{comm} = t_s log p + t_w \left(\frac{n}{p}\right)(p-1)$. Assume p is large, then $t_{comm} = t_s log p + t_w n.$
3. The parallel run time for this program is:

$$T_p = \frac{n^2}{p} + t_s log p + t_w n$$

4. This program is cost-optimal for $p = O(n)$

Remark: This analysis neglect the one-to-all communication to broadcast vector b initially.

Scalability Analysis

- Let $K = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$, then $T(n,1) = KT_0(n,p)$
- $T_0 = t_s plogp + t_w np$
- Neglecting $t_w np$, $T(n,1) = Kt_s plogp$. This is the isoefficiency with respect to message startup time.
- Neglecting $t_s plogp$, $T(n,1) = Kt_w np$.

  Since $T(n,1) = n^2$, $T(n,1) = K^2 {t_w}^2 p^2$.

  In order to maintain a fixed efficiency, the problem size must increase with the rate of $\Theta(p^2)$.

Parallel Matrix Algorithms (part 2)

```
void create_mixed_xfer_arrays(
        int     id,
        int     p,
        int     n,
        int     **count,
        int     **disp)
{
        int     i;
        *count = my_malloc(id, p*sizeof(int));
        *disp = my_malloc(id, p*sizeof(int));
        (*count)[0] = BLOCK_SIZE(0,p,n);
        (*disp)[0] = 0;

        for(i = 1; i < p; i++)
        {
           (*disp)[i] = (*disp)[i-1] + (*count)[i-1];
           (*count)[i] = BLOCK_SIZE(i,p,n);
        }
}
```

This function creates the count and displacement arrays by scatter and gather functions, when the number of elements send/received to/from other processes varies

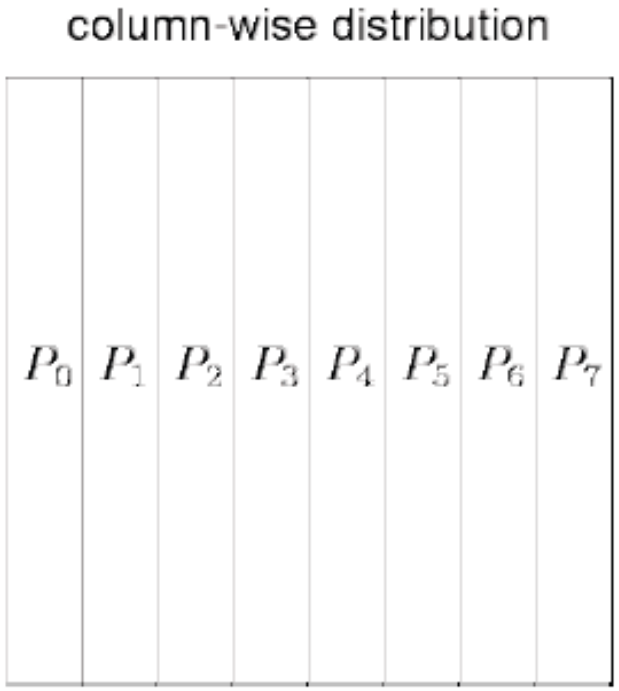
```
void replicate_block_vector(
      void      *ablock, /* block-distributed vector */
      int       n,
      void      *arep,   // replicated vector
      MPI_Datatype   dtype,
      MPI_Comm   comm)
{
      int       *cnt; // elements contributed by each process
      int       *disp;  // displacement in concatenated array
      int       id;
      int       p;

      MPI_Comm_size(comm, &p);
      MPI_Comm_rank(comm, &id);

      create_mixed_xfer_arrays(id, p, n, &cnt, &disp);
      MPI_Allgatherv(ablock, cnt[id], dtype, arep, cnt, disp, dtype, comm);
      free(cnt);
      free(disp);
}
```

replicate_block_vector() is used to transform a vector from a block distribution to a replicated distribution

Column-wise Block-Striped Decomposition

Summary of algorithm for computing $\mathbf{c} = A\boldsymbol{b}$

- Column-wise 1D block partition is used to distribute matrix.
- Let $A = [\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_n]$, $\boldsymbol{b} = [b_1, b_2, \ldots, b_n]^T$, and $\mathbf{c} = [c_1, c_2, \ldots, c_n]^T$
- Assume each task i has column $\boldsymbol{a}_i$, b_i and c_i (Assume a fine-grained decomposition for convenience)



1. Read in matrix stored in row-major manner and distribute by column-wise mapping
2. Each task i compute $b_i \boldsymbol{a}_i$ to result in a vector of partial result.
3. An all-to-all communication is used to transfer partial result: every partial result element j on task i must be transferred to task j.
4. At the end of computation, task i only has a single element of the result c_i by adding gathered partial results.

$$
\begin{aligned}
c_0 &= a_{0,0}\,b_0 + a_{0,1}\,b_1 + a_{0,2}\,b_2 + a_{0,3}\,b_3 + a_{4,4}\,b_4 \\
c_1 &= a_{1,0}\,b_0 + a_{1,1}\,b_1 + a_{1,2}\,b_2 + a_{1,3}\,b_3 + a_{1,4}\,b_4 \\
c_2 &= a_{2,0}\,b_0 + a_{2,1}\,b_1 + a_{2,2}\,b_2 + a_{2,3}\,b_3 + a_{2,4}\,b_4 \\
c_3 &= a_{3,0}\,b_0 + a_{3,1}\,b_1 + a_{3,2}\,b_2 + a_{3,3}\,b_3 + b_{3,4}\,b_4 \\
c_4 &= a_{4,0}\,b_0 + a_{4,1}\,b_1 + a_{4,2}\,b_2 + a_{4,3}\,b_3 + a_{4,4}\,b_4
\end{aligned}
$$

Proc 4's init.

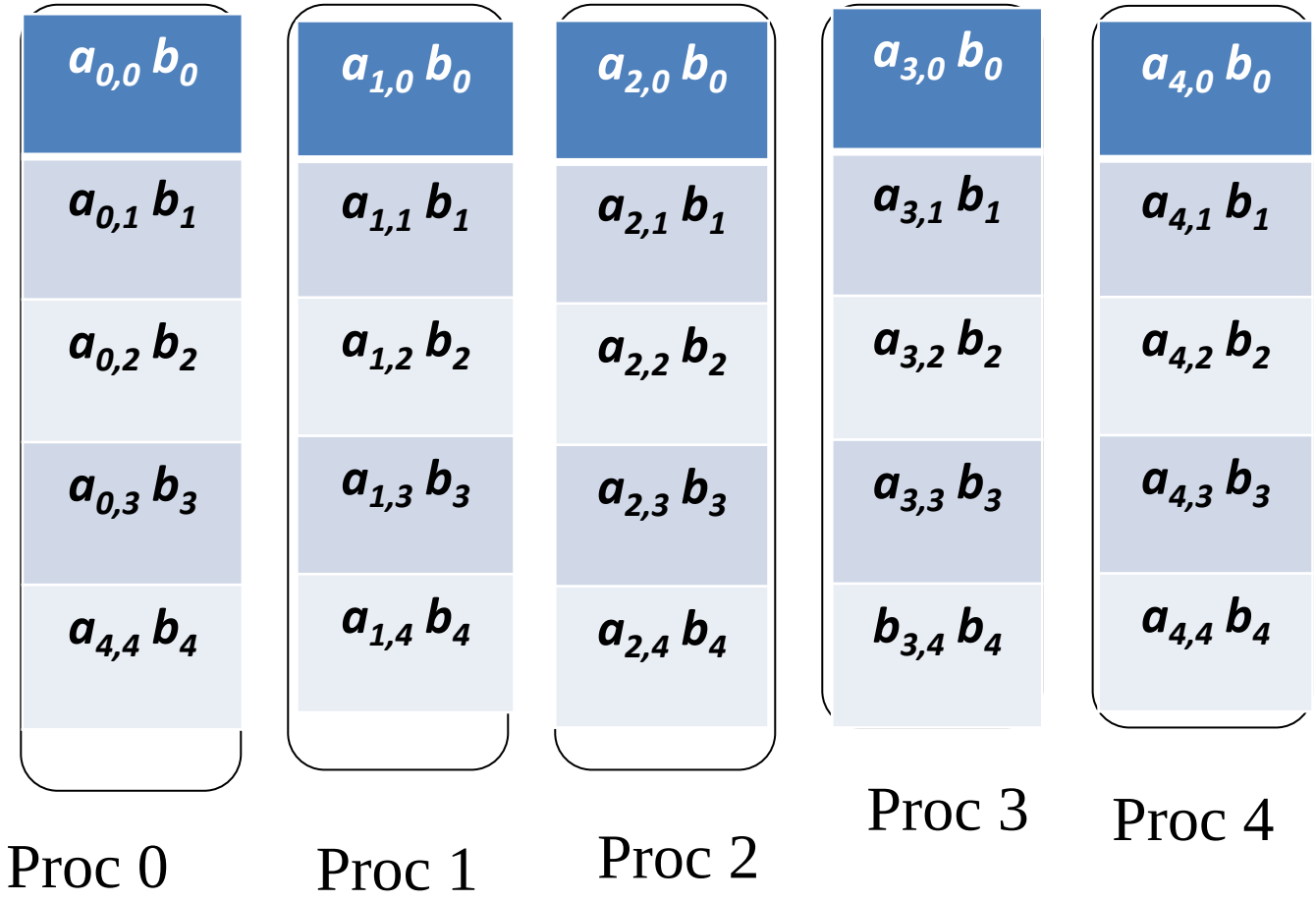
Proc 3's init. comput

Proc 2's init. comput

Processor 1's initial computation

Processor 0's initial computation

After All-to-All Communication



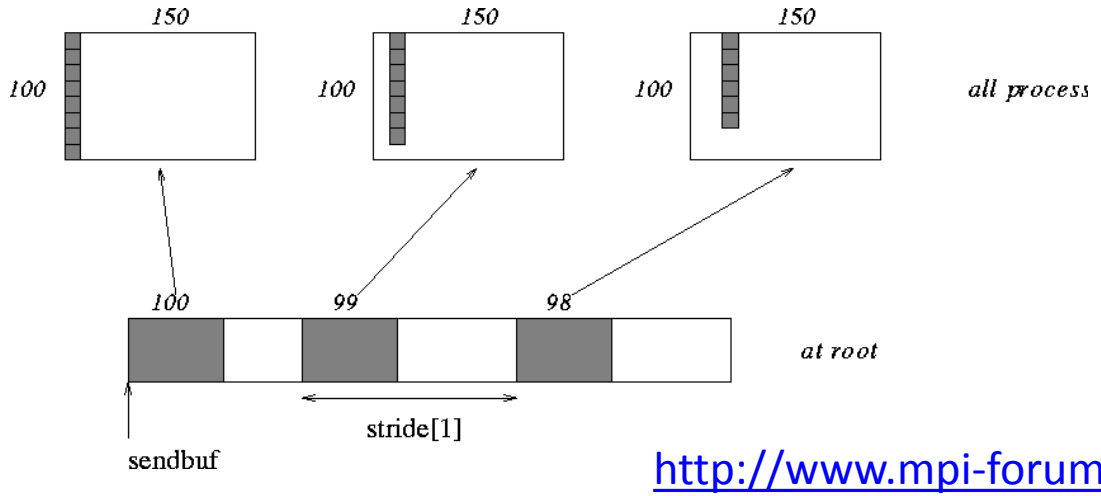
Reading a Column-wise Block-Striped Matrix

read_col_striped_matrix()

- Read from a file a matrix stored in row-major order and distribute it among processes in column-wise fashion.
- Each row of matrix must be scattered among all of processes.

```
read_col_striped_matrix()
{
    …
    // figure out how a row of the matrix should be distributed
    create_mixed_xfer_arrays(id,p, *n, &send_count, &send_disp);
    // go through each row of the matrix
     for(i = 0; i < *m; i++)
     {
         if(id == (p-1)) fread(buffer,datum_size, *n, infileptr);
         MPI_Scatterv(...);
     }
}
```

- **int MPI_Scatterv**(void **sendbuf,* int **sendcnts,* int **displs,* MPI_Datatype *sendtype,* void **recvbuf,* int *recvcnt,* MPI_Datatype *recvtype,* int *root,* MPI_Comm *comm*)
  - MPI_SCATTERV extends the functionality of MPI_SCATTER by allowing a varying count of data to be sent to each process.
  - *sendbuf*: address of send buffer
  - *sendcnts*: an integer array specifying the number of elements to send to each processor
  - *displs*: an integer array. Entry i specifies the displacement (relative to *sendbuf* from which to take the outgoing data to process i



http://www.mpi-forum.org/docs/mpi-11-html/node72.html

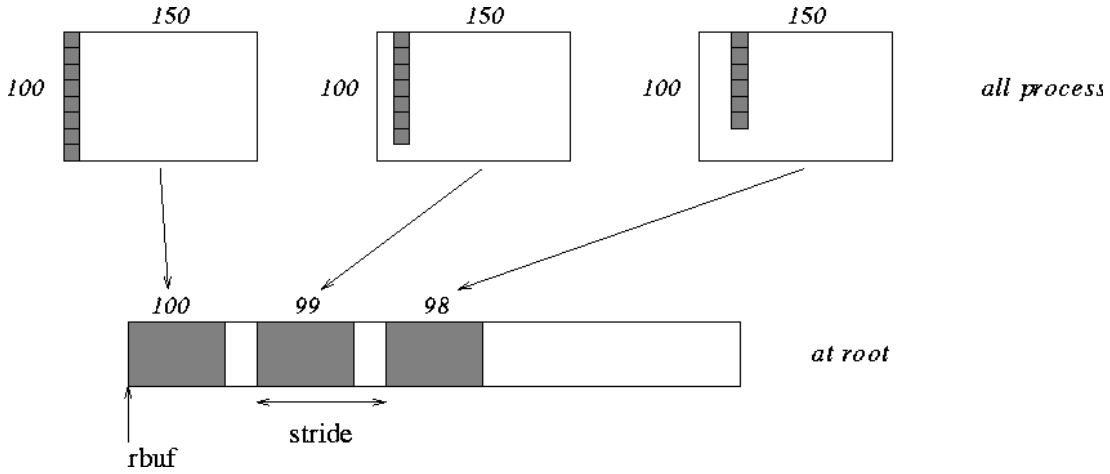
Printing a Colum-wise Block-Striped Matrix

print_col_striped_matrix()

- A single process print all values
- To print a single row, the process responsible for printing must gather together the elements of that row from entire set of processes

```
print_col_striped_matrix()
{
   …
   create_mixed_xfer_arrays(id, p, n, &rec_count, &rec_disp);
    // go through rows
    for(i =0; i < m; i++)
     {
         MPI_Gatherv(a[i], BLOCK_SIZE(id,p,n), dtype, buffer,
           rec_count, rec_disp, dtype, 0, comm);
         ....
     }
}
```

- int **MPI_Gatherv**(void **sendbuf*, int *sendcnt*, MPI_Datatype *sendtype*, void **recvbuf*, int **recvcnts*, int **displs*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)
  - Gathers into specified locations from all processes in a group.
  - *sendbuf*: address of send buffer
  - *sendcnt*: the number of elements in send buffer
  - *recvbuf*: address of receive buffer (choice, significant only at root)
  - *recvcounts*: integer array (of length group size) containing the number of elements that are received from each process (significant only atroot)
  - *displs*: integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i (significant only at root)



Distributing Partial Results

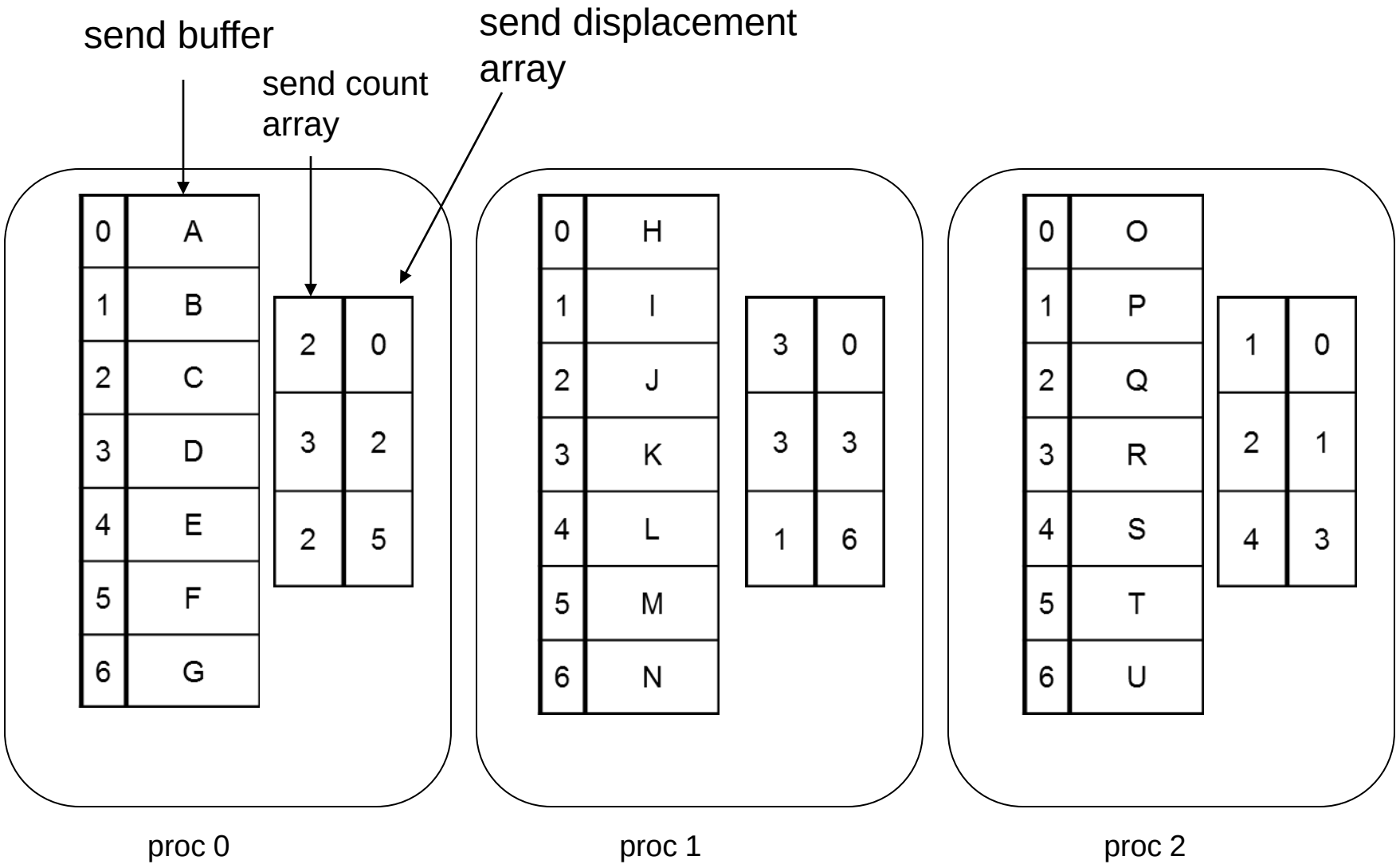
- $c_i = b_0\boldsymbol{a}_{i,0} + b_1\boldsymbol{a}_{i,1} + b_2\boldsymbol{a}_{i,2} + \cdots + b_n\boldsymbol{a}_{i,n}$
- Each process need to distribute $n-1$ terms to other processes and gather $n-1$ terms from them (assume fine-grained decomposition).
  - MPI_Alltoallv() is used to do this **all-to-all** exchange

int MPI_Alltoallv(void **sendbuf*, **int** **sendcnts*, **int** **sdispls*, **MPI_Datatype** *sendtype*, **void** **recvbuf*, **int** **recvcnts*, **int** **rdispls*, **MPI_Datatype** *recvtype*, **MPI_Comm** *comm* **);**

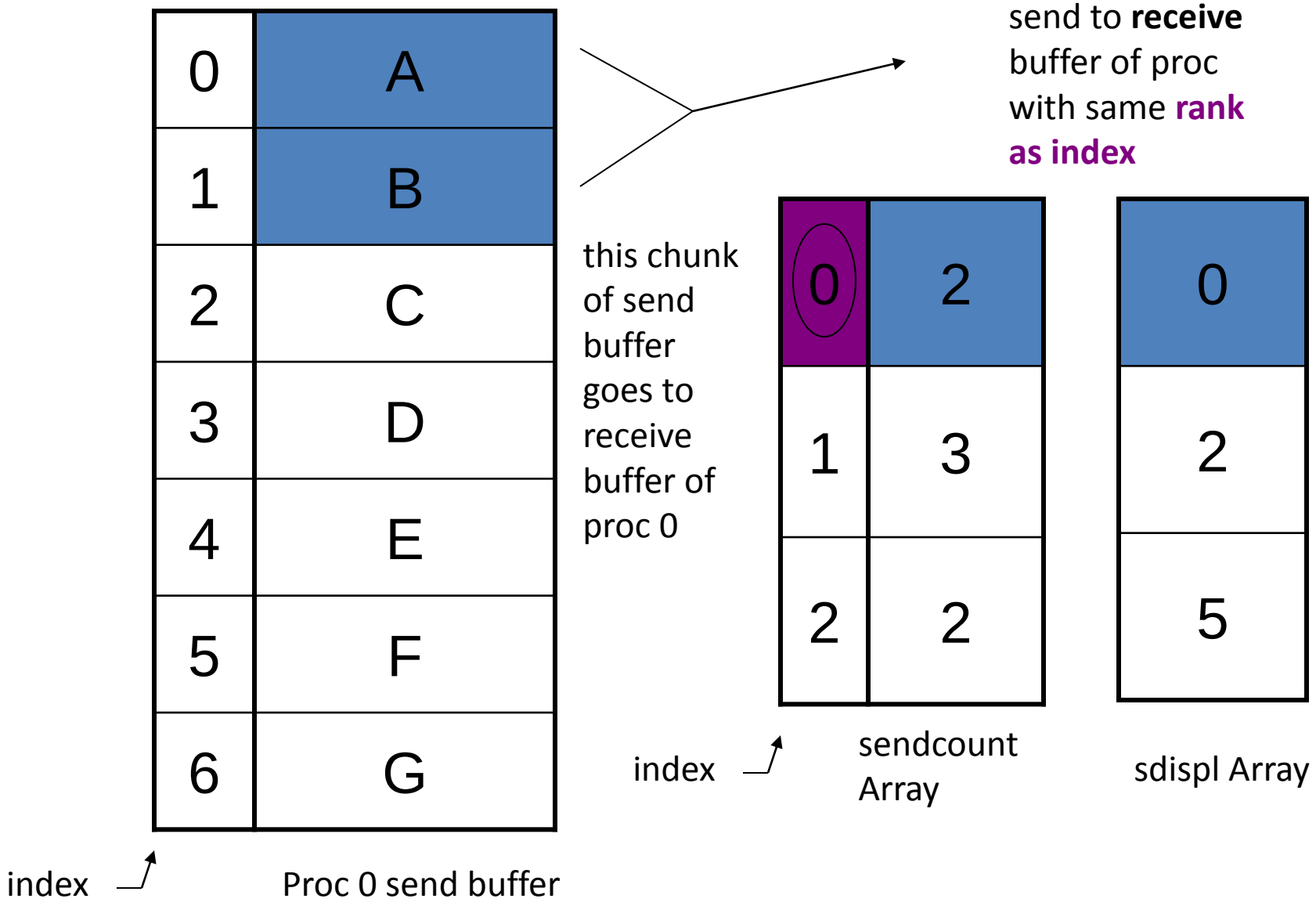
- *sendbuf*: starting address of send buffer (choice)
- *sendcounts*: integer array equal to the group size specifying the number of elements to send to each processor
- *sdispls*: integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for process j
- *recvbuf*: address of receive buffer (choice)
- *recvcounts*: integer array equal to the group size specifying the maximum number of elements that can be received from each processor
- *Rdispls*: integer array (of length group size). Entry i specifies the displacement (relative to recvbuf at which to place the incoming data from process i

Send of MPI_Alltoallv()

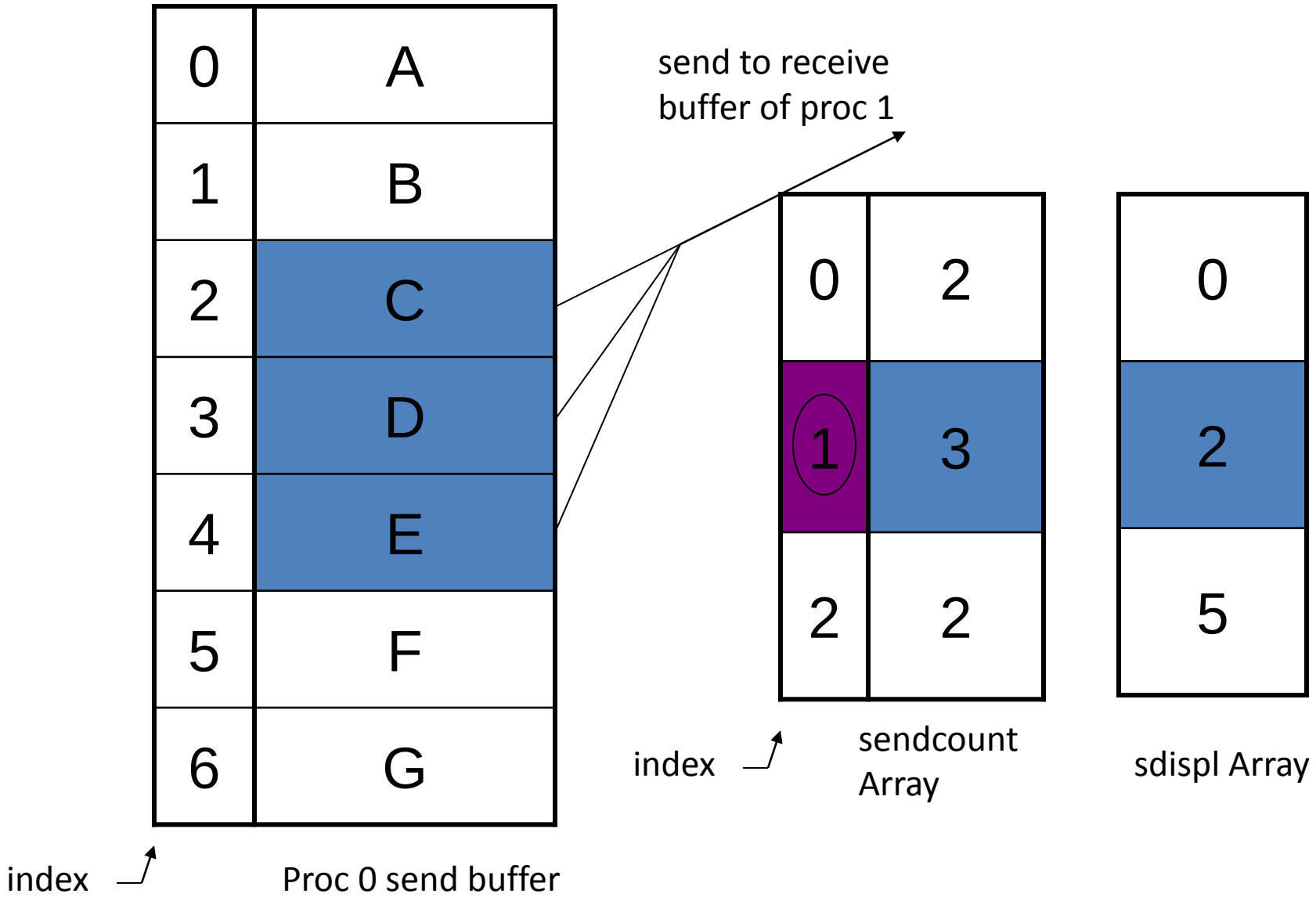
Each node in parallel community has



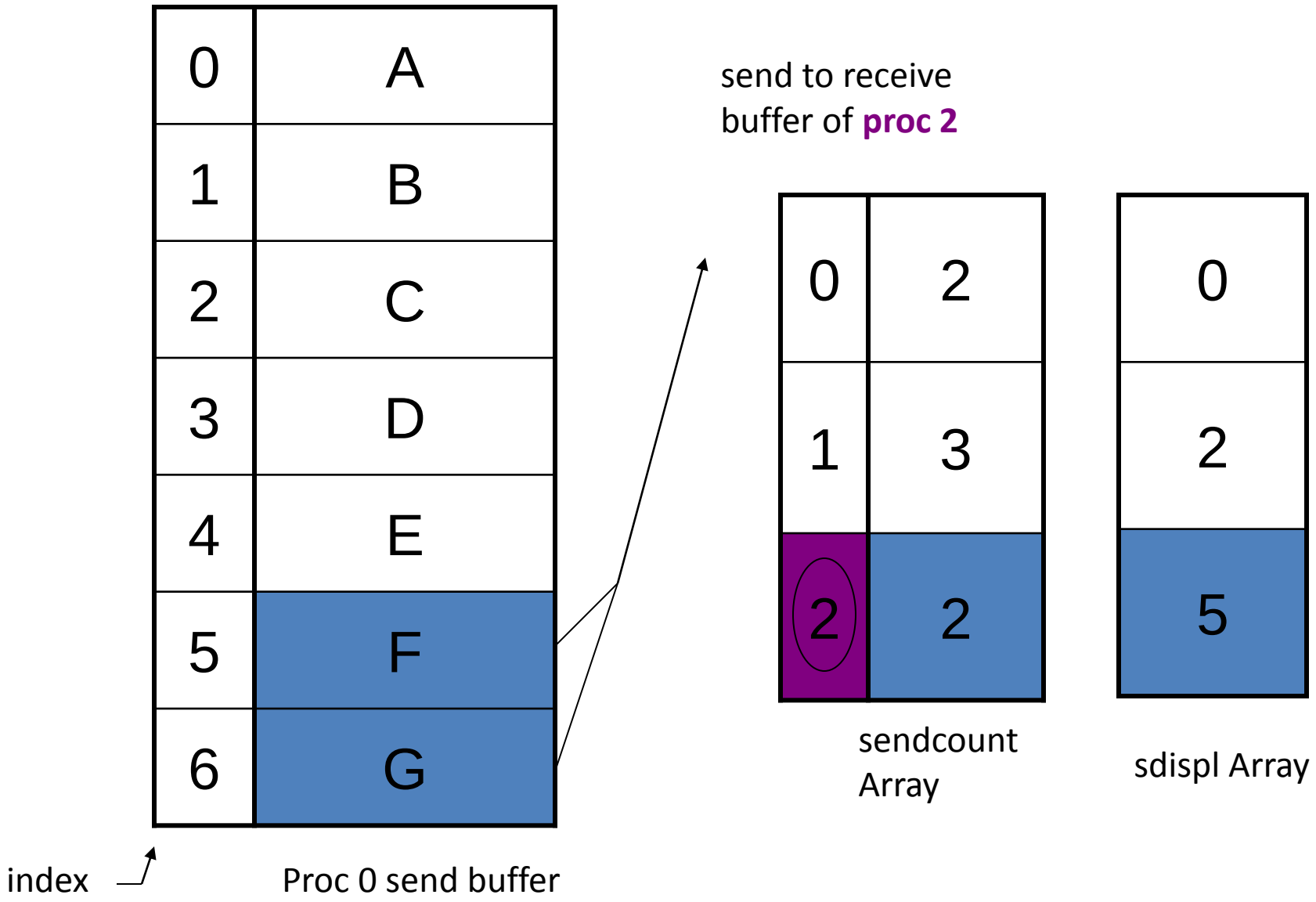
Process 0 Sends to Process 0



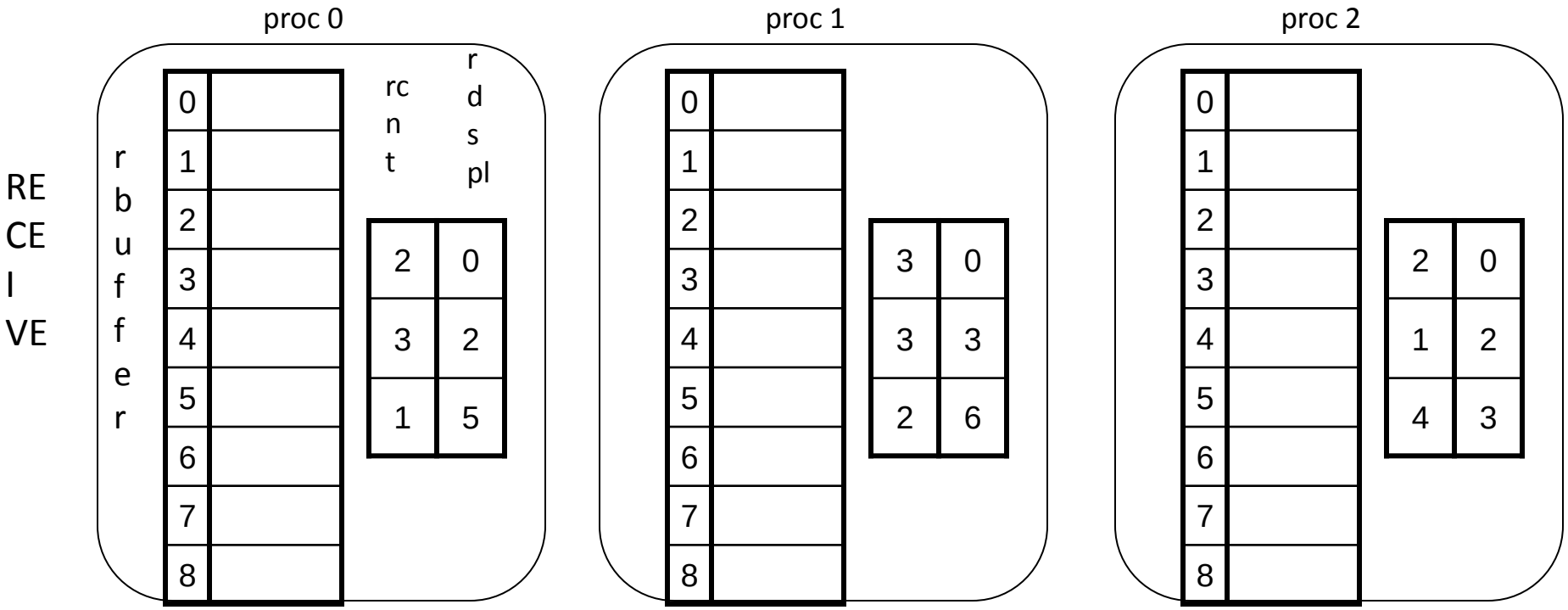
Process 0 Sends to Process 1

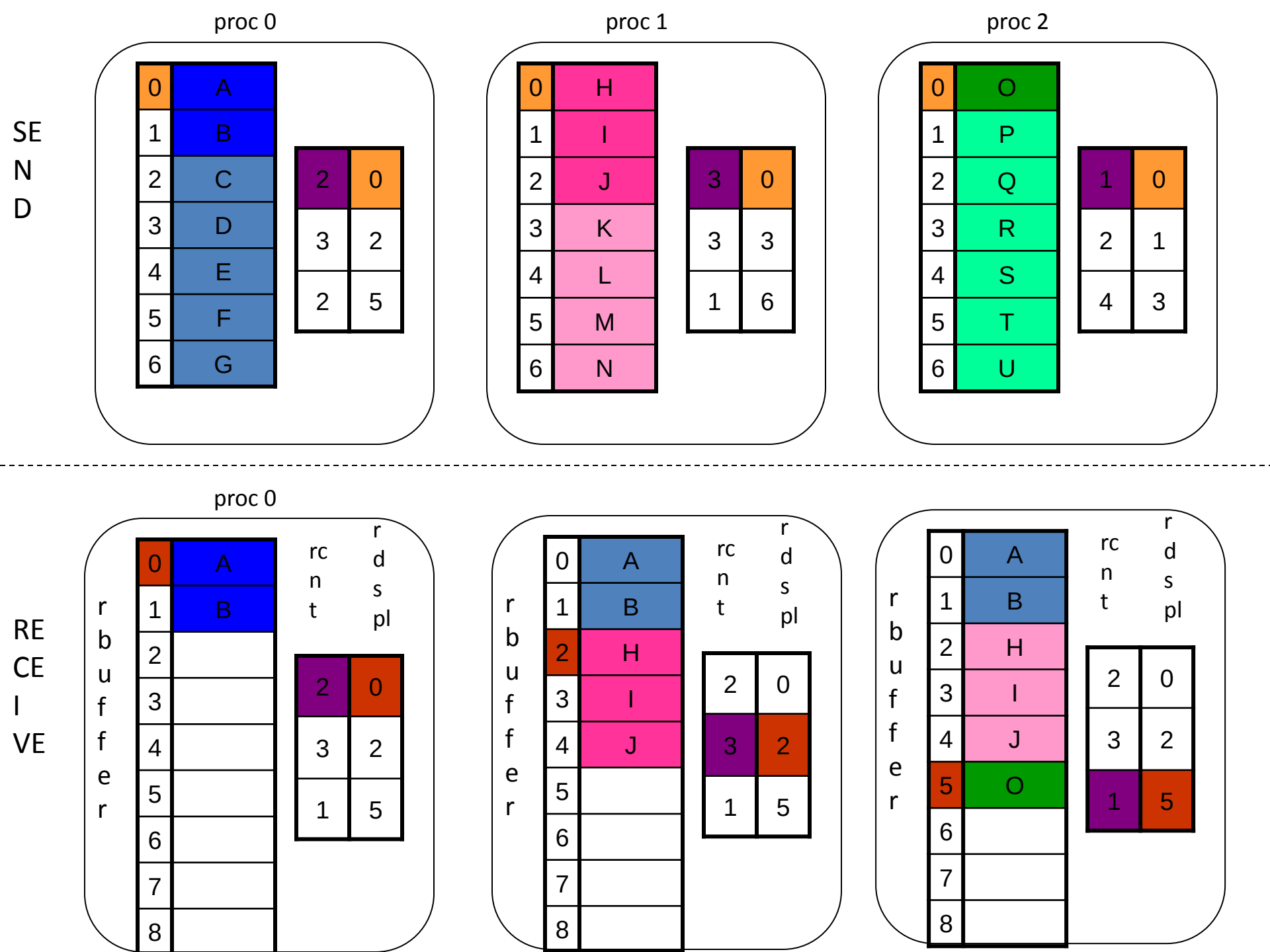


Process 0 Sends to Process 2



Receive of MPI_Alltoallv()




proc 0
proc 1
proc 2
SEND
0 A
1 B
2 C
3 D
4 E
5 F
6 G
2 0
3 2
2 5
0 H
1 I
2 J
3 K
4 L
5 M
6 N
3 0
3 3
1 6
0 O
1 P
2 Q
3 R
4 S
5 T
6 U
1 0
2 1
4 3
RECEIVE
proc 0
rbuffer
rcnt
rdspl
0 A
1 B
2
3
4
5
6
7
8
2 0
3 2
1 5
0 A
1 B
2 H
3 I
4 J
5
6
7
8
2 0
3 2
1 5
0 A
1 B
2 H
3 I
4 J
5 O
6
7
8
2 0
3 2
1 5

Parallel Run Time Analysis (Column-wise)

- Assume that the # of processes p is less than n
- Assume that we run the program on a parallel machine adopting hypercube interconnection network (**Table 4.1** lists communication times of various communication schemes)

1. Each process is responsible for n/p columns of matrix. The complexity of the dot production portion of the parallel algorithm is $\Theta(n^2/p)$
2. After *all-to-all personalized* communication, each processor sums the partial vectors. There are p partial vectors, each of size n/p . The complexity of the summation is $\Theta(n)$.
3. Parallel communication time for all-to-all *personalized* broadcast communication:
   - Each process needs to send p messages of size n/p each to all processes.

     $t_{comm} = (t_s + t_w\left(\frac{n}{p}\right))(p-1)$. Assume p is large, then

     $t_{comm} = t_s(p-1) + t_w n.$

- The parallel run time: $T_p = \frac{n^2}{p} + n + t_s(p-1) + t_w n$

2D Block Decomposition

Summary of algorithm for computing $\boldsymbol{y} = A\mathbf{b}$

- 2D block partition is used to distribute matrix.
- Let $A = [a_{ij}]$, $\mathbf{b} = [b_1, b_2, \ldots, b_n]^T$, and $\boldsymbol{y} = [y_1, y_2, \ldots, y_n]^T$
- Assume each task is responsible for computing $d_{ij} = a_{ij} b_j$ (assume a fine-grained decomposition for convenience of analysis).
- Then $y_i = \sum_{j=0}^{n-1} d_{ij}$: for each row i, we add all the d_{ij} to produce the ith element of $\boldsymbol{y}$.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

1. Read in matrix stored in row-major manner and distribute by 2D block mapping. Also distribute $\boldsymbol{b}$ so that each task has the correct portion of $\boldsymbol{b}$.

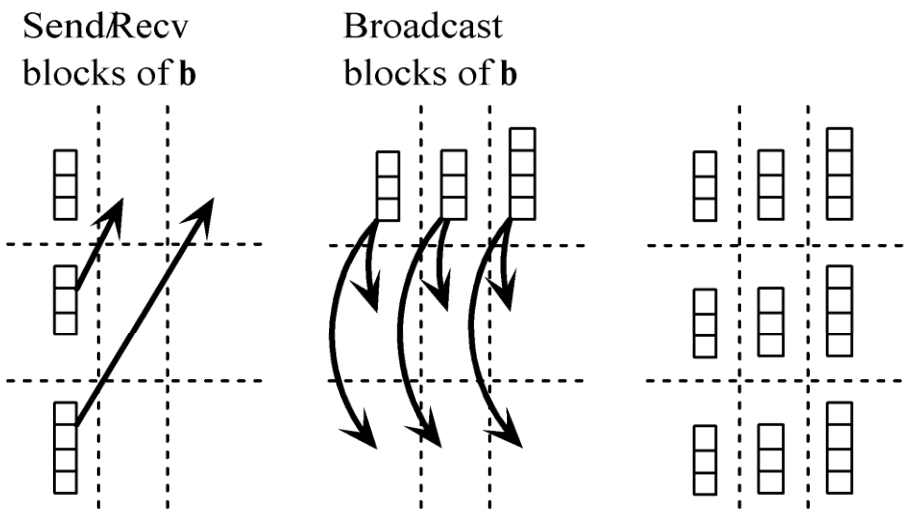
2. Each task computes a matrix-vector multiplication using its portion of A and $\boldsymbol{b}$.

3. Tasks in each row of the task grid perform a sum-reduction on their portion of $\boldsymbol{y}$.

4. After the sum-reduction, $\boldsymbol{y}$ is distributed by blocks among the tasks in the first column of the task grid.

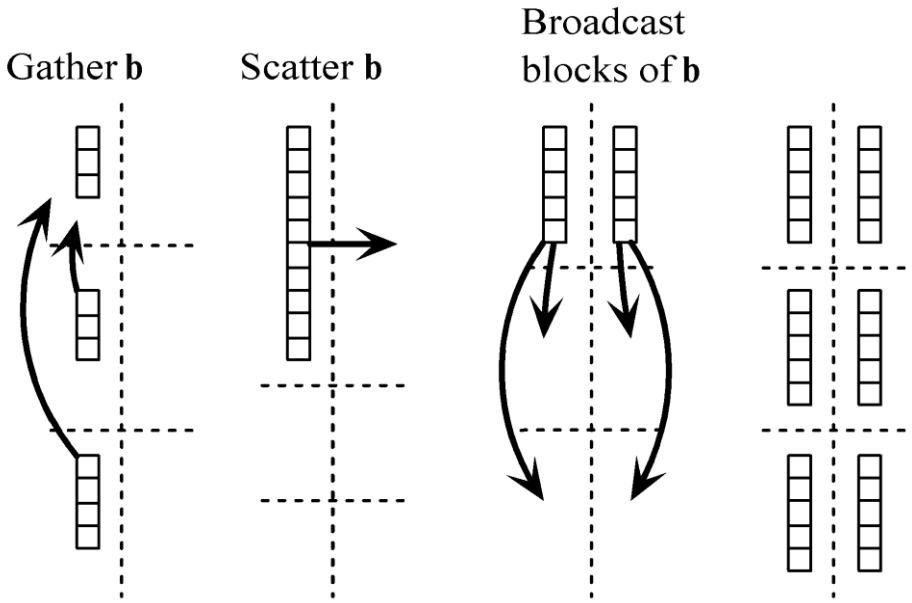
Distributing $\boldsymbol{b}$

- Initially, **b** is divided among tasks in the first column of the task grid.
- Step 1:
  - If *p* square
    - First column/first row processes send/receive portions of **b**
  - If *p* not square
    - Gather **b** on process 0, 0
    - Process 0, 0 broadcasts to first row processes
- Step 2: First row processes scatter **b** within columns



(a)

When p is a square number



(b)

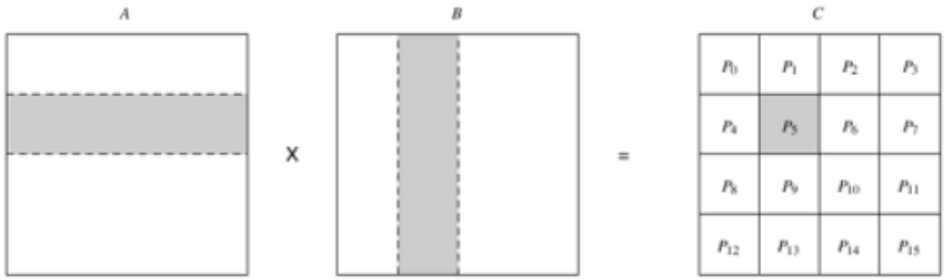
When p is not a square number

Parallel Matrix Algorithms (part 3)

A Simple Parallel Matrix-Matrix Multiplication

Let $A = [a_{ij}]_{n \times n}$ and $B = [b_{ij}]_{n \times n}$ be n × n matrices. Compute $C = AB$

- Computational complexity of sequential algorithm: $O(n^3)$
- Partition A and B into p square blocks $A_{i,j}$ and $B_{i,j}$ $(0 \leq i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Use Cartesian topology to set up process grid. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Remark: Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < \sqrt{p}$.



- Algorithm:
  - Perform all-to-all broadcast of blocks of A in each row of processes
  - Perform all-to-all broadcast of blocks of B in each column of processes
  - Each process $P_{i,j}$ perform $C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,k} B_{k,j}$

Performance Analysis

- $\sqrt{p}$ rows of all-to-all broadcasts, each is among a group of $\sqrt{p}$ processes. A message size is $\frac{n^2}{p}$, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$
- $\sqrt{p}$ columns of all-to-all broadcasts, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$
- Computation time: $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$
- Parallel time: $T_p = \frac{n^3}{p} + 2\left(t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)\right)$

Memory Efficiency of the Simple Parallel Algorithm

- Not memory efficient
  - Each process $P_{i,j}$ has $2\sqrt{p}$ blocks of $A_{i,k}$ and $B_{k,j}$
  - Each process needs $\Theta(n^2/\sqrt{p})$ memory
  - Total memory over all the processes is $\Theta(n^2 \times \sqrt{p})$, i.e., $\sqrt{p}$ times the memory of the sequential algorithm.

Cannon's Algorithm of Matrix-Matrix Multiplication

Goal: to improve the memory efficiency.

Let $A = [a_{ij}]_{n\times n}$ and $B = [b_{ij}]_{n\times n}$ be n × n matrices. Compute $C = AB$

- Partition A and B into p square blocks $A_{i,j}$ and $B_{i,j}$ $(0 \le i, j < \sqrt{p})$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each.
- Use Cartesian topology to set up process grid. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.
- Remark: Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \le k < \sqrt{p}$.
- **The contention-free formula**:

$$C_{i,j} = \sum_{k=0}^{\sqrt{p}-1} A_{i,(i+j+k)\%\sqrt{p}} B_{(i+j+k)\%\sqrt{p},j}$$

Cannon's Algorithm

// make initial alignment

for i, j :=0 **to** $\sqrt{p} - 1$ **do**

Send block $A_{i,j}$ to process $\left(i, \left(j - i + \sqrt{p}\right) mod \sqrt{p}\right)$ and block $B_{i,j}$ to process $\left(\left(i - j + \sqrt{p}\right) mod \sqrt{p}, j\right)$;

endfor;

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

// compute-and-shift. A sequence of one-step shifts pairs up $A_{i,k}$ and $B_{k,j}$

// on process $P_{i,j}$. $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

for step :=1 **to** $\sqrt{p} - 1$ **do**

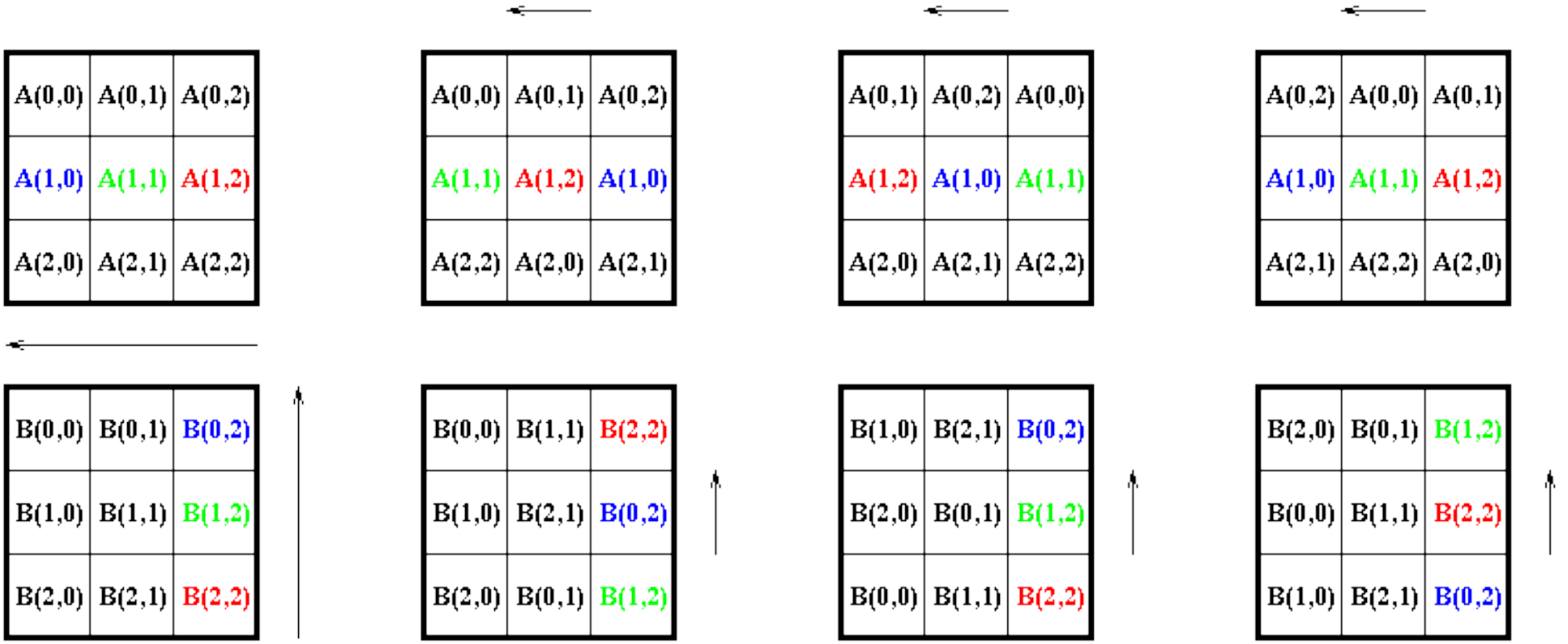
Shift $A_{i,j}$ one step left (with wraparound) and $B_{i,j}$ one step up (with wraparound);

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

Endfor;

Remark: In the initial alignment, the send operation is to: shift $A_{i,j}$ to the left (with wraparound) by i steps, and shift $B_{i,j}$ to the up (with wraparound) by j steps.

Cannon's Algorithm for 3 × 3 Matrices



Initial A, B

A, B initial alignment

A, B after shift step 1

A, B after shift step 2

Performance Analysis

- In the initial alignment step, the maximum distance over which block shifts is $\sqrt{p} - 1$
  - The circular shift operations in row and column directions take time: $t_{comm} = 2(t_s + \frac{t_w n^2}{p})$
- Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase takes time: $t_s + \frac{t_w n^2}{p}$.
- Multiplying $\sqrt{p}$ submatrices of size $(\frac{\mathrm{n}}{\sqrt{p}}) \times (\frac{\mathrm{n}}{\sqrt{p}})$ takes time: n^3/p.
- Parallel time: $T_p = \frac{n^3}{p} + 2\sqrt{p}\left(t_s + \frac{t_w n^2}{p}\right) + 2(t_s + \frac{t_w n^2}{p})$

int MPI_Sendrecv_replace(void **buf*, **int** *count*, **MPI_Datatype** *datatype*, **int** *dest*, **int** *sendtag*, **int** *source*, **int** *recvtag*, **MPI_Comm** *comm*, **MPI_Status** **status* **);**

- Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received.
- *buf*[in/out]: initial address of send and receive buffer

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
   int    myid, numprocs, left, right;
   int    buffer[10];
   MPI_Request request;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);

   right = (myid + 1) % numprocs;
   left = myid - 1;
   if (left < 0)
      left = numprocs - 1;

   MPI_Sendrecv_replace(buffer, 10, MPI_INT, left, 123, right, 123, MPI_COMM_WORLD,
&status);

   MPI_Finalize();
   return 0;
}
```