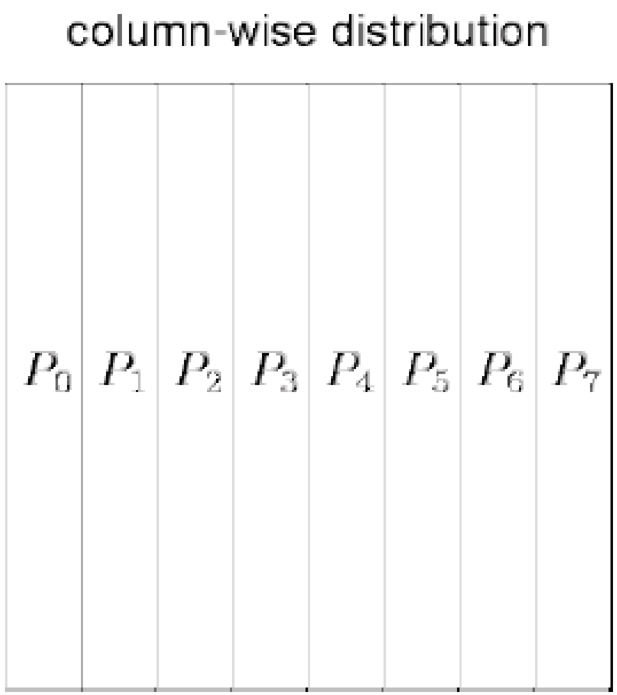
Lecture 5: Parallel Matrix Algorithms (part 2)

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 $A = [\boldsymbol{a}_1, \boldsymbol{a}_2, \dots, \boldsymbol{a}_n]$, $\boldsymbol{b} = [b_1, b_2, \dots, b_n]^T$, and $\mathbf{c} = [c_1, c_2, \dots, c_n]^T$
- 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.

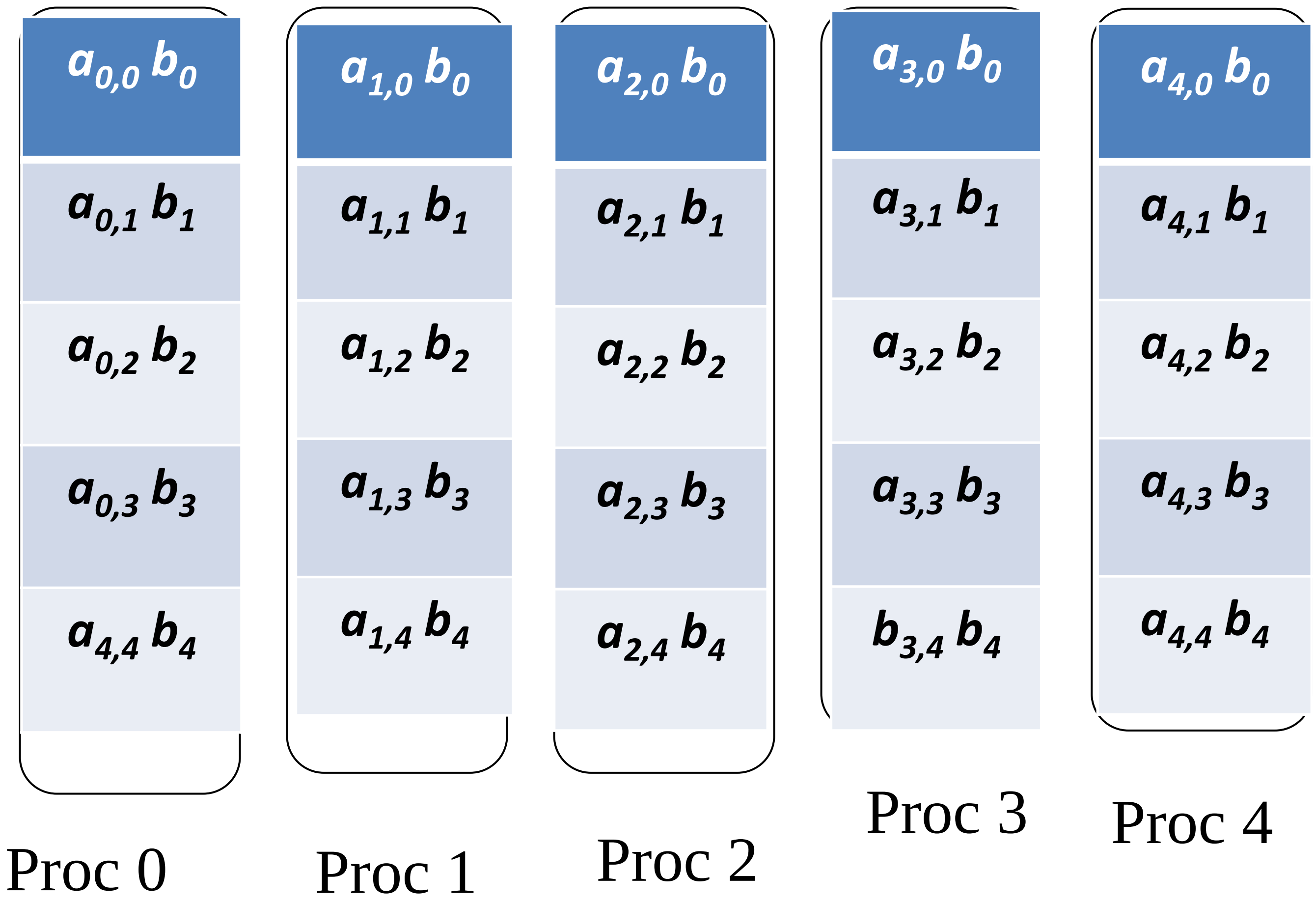
Proc 3’s init. comput

Proc 2’s init. comput

Processor 1’s initial computation

Processor 0’s initial computation

After All-to-All Communication



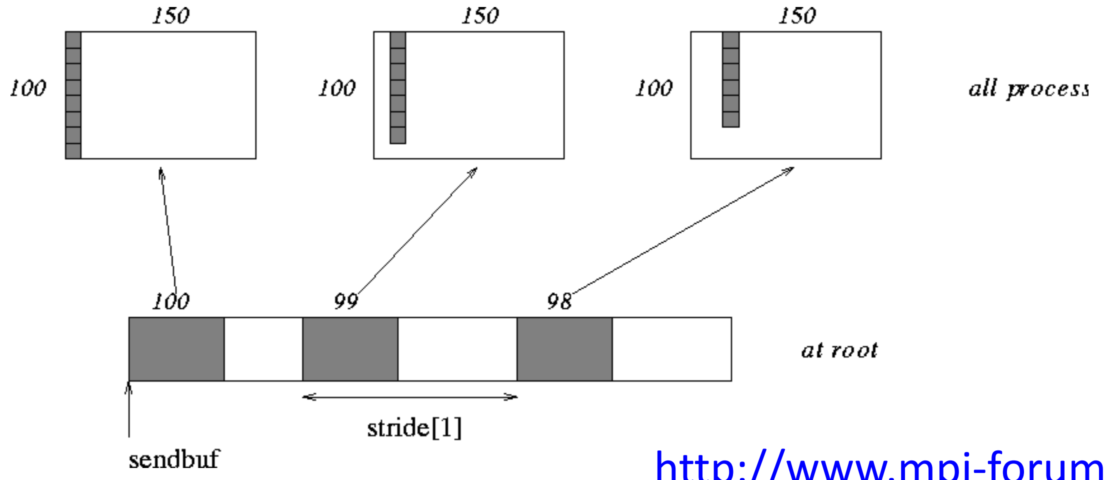
Reading a Column-wise Block-Striped Matrix

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

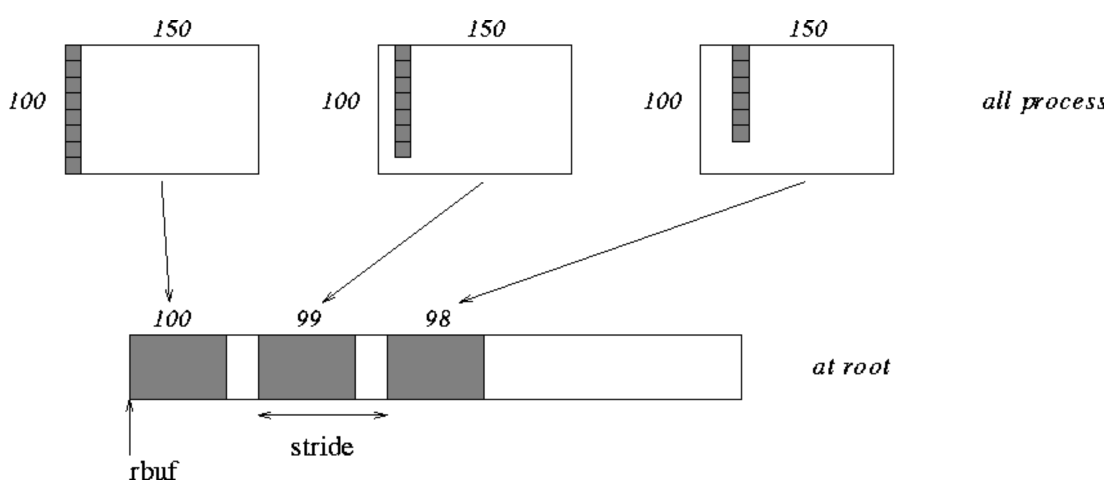
Printing a Colum-wise Block-Striped Matrix

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

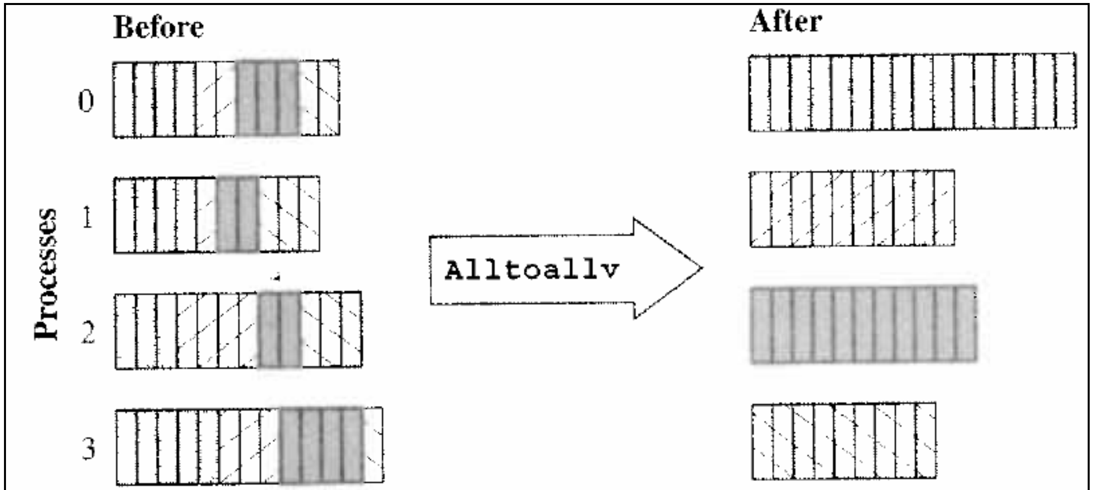


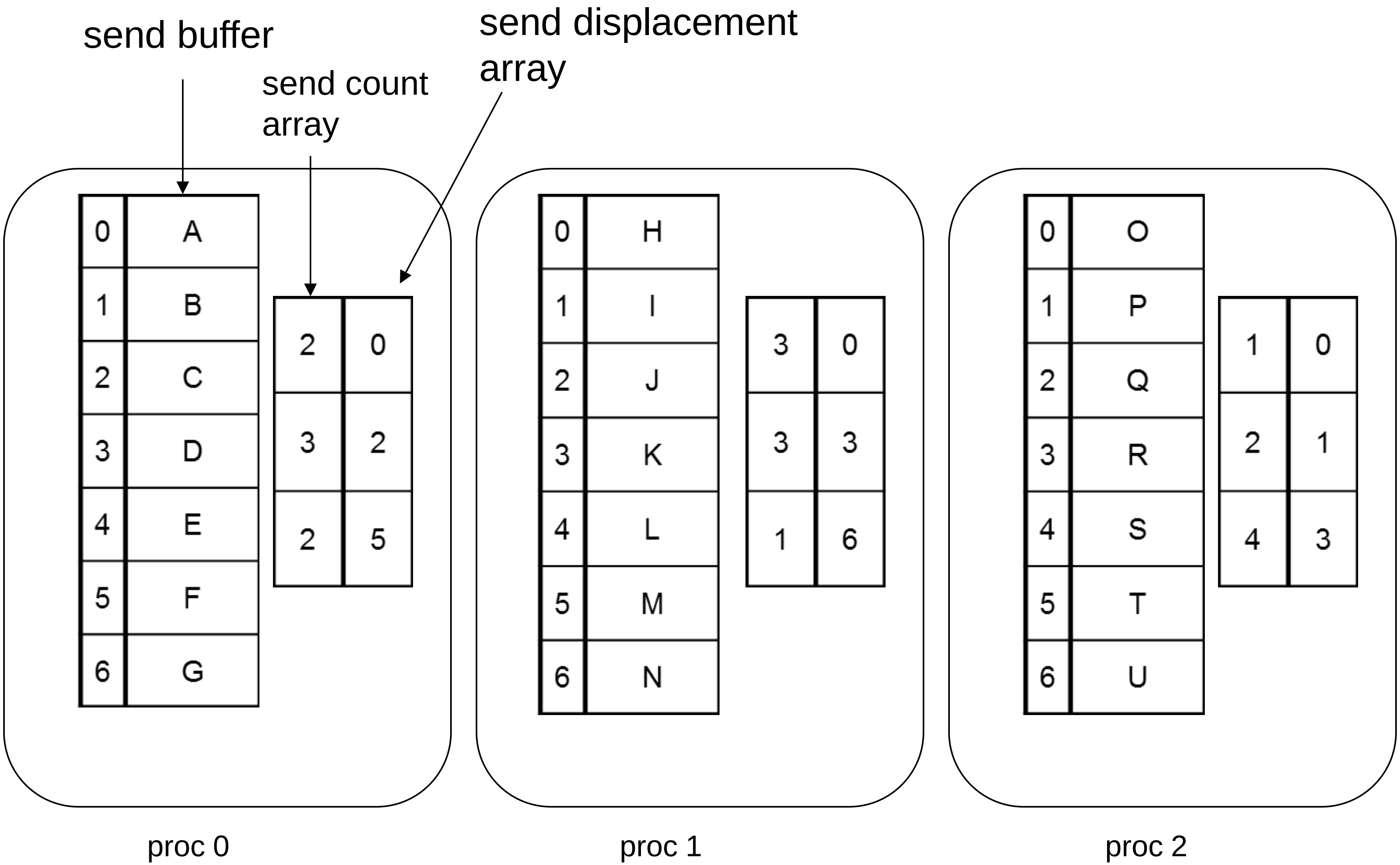
Figure 8.13 Function MPI_Alltoallv allows every MPI process to gather data items from all the processes in the communicator. The simpler function MPI_Alltoall should be used in the case where all of the groups of data items being transferred from one process to another have the same number of elements.

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

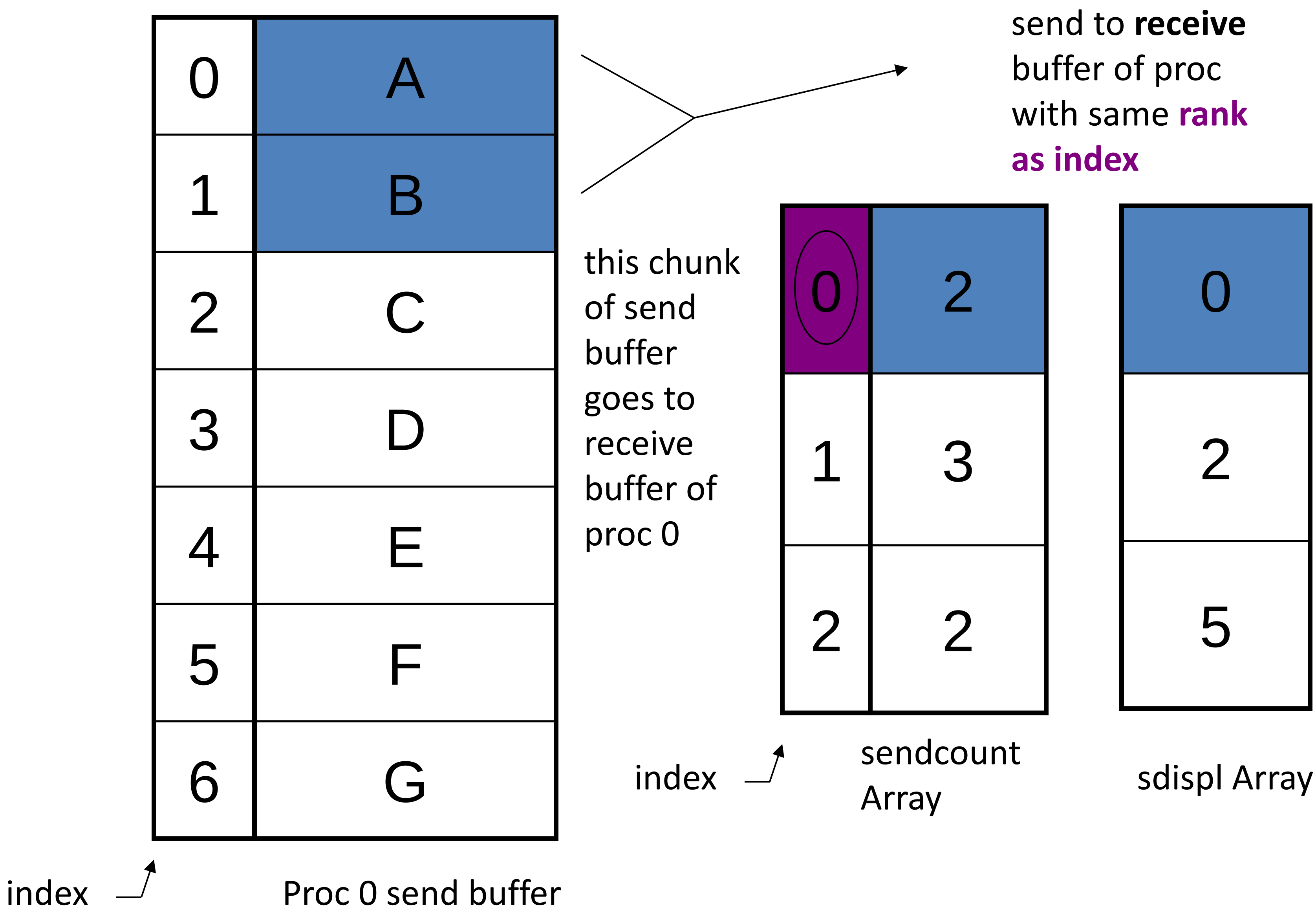
- *sendbuf*: starting address of send buffer (choice)
- *sendcounts*: integer array equal to the group size specifying the number of elements to send to each processor
- *sdispls*: integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing 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()

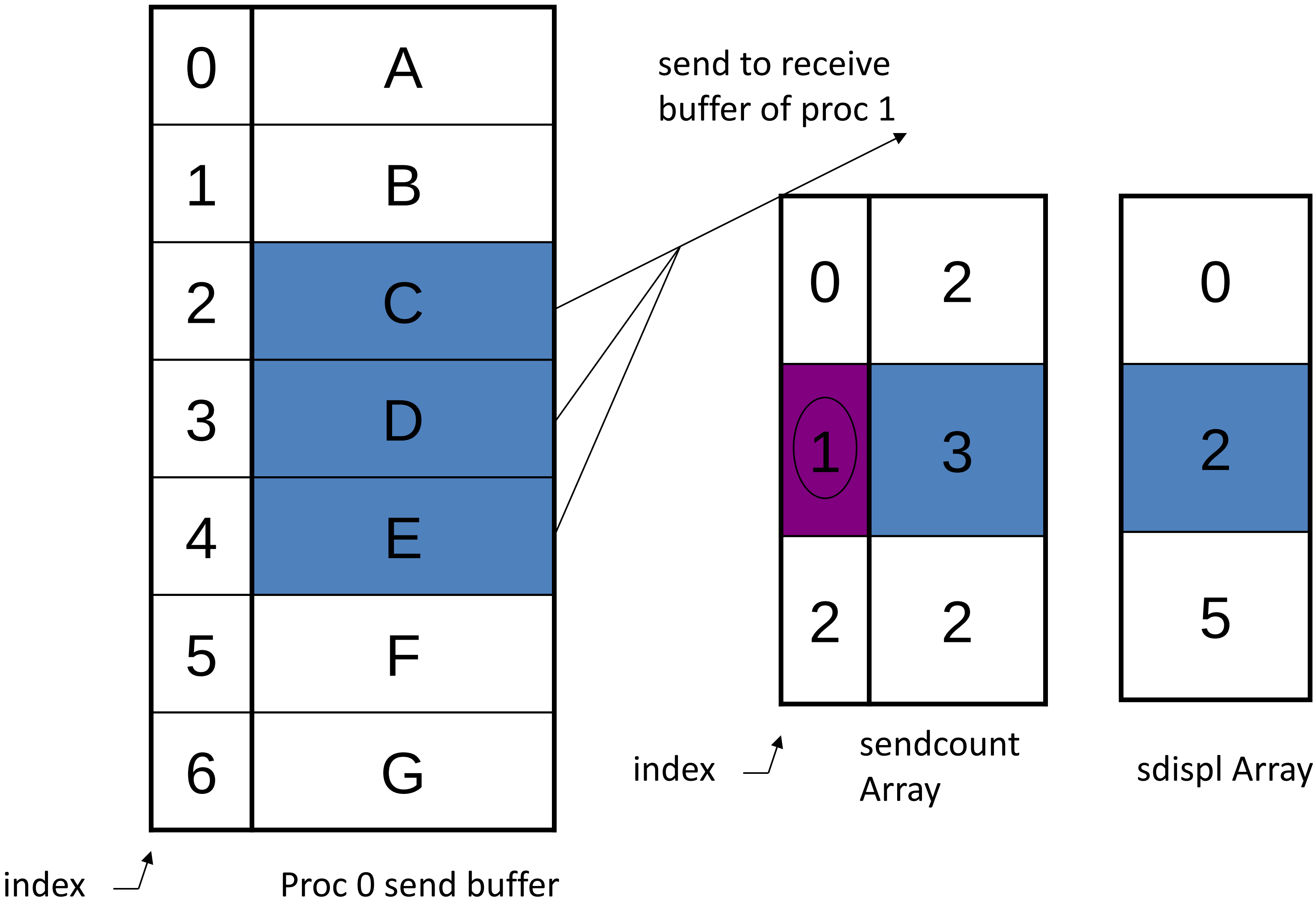
Each node in parallel community has



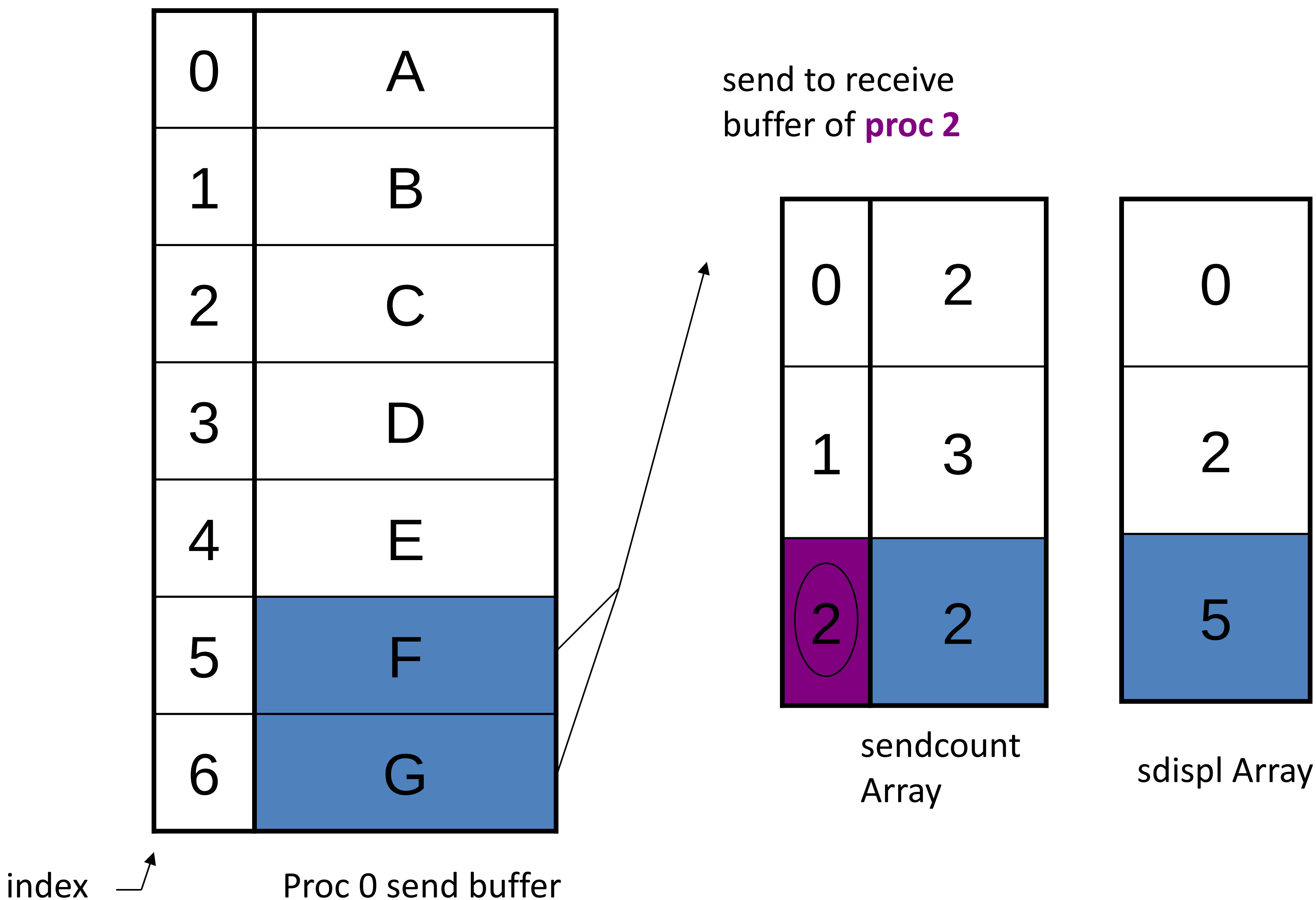
Process 0 Sends to Process 0



Process 0 Sends to Process 1

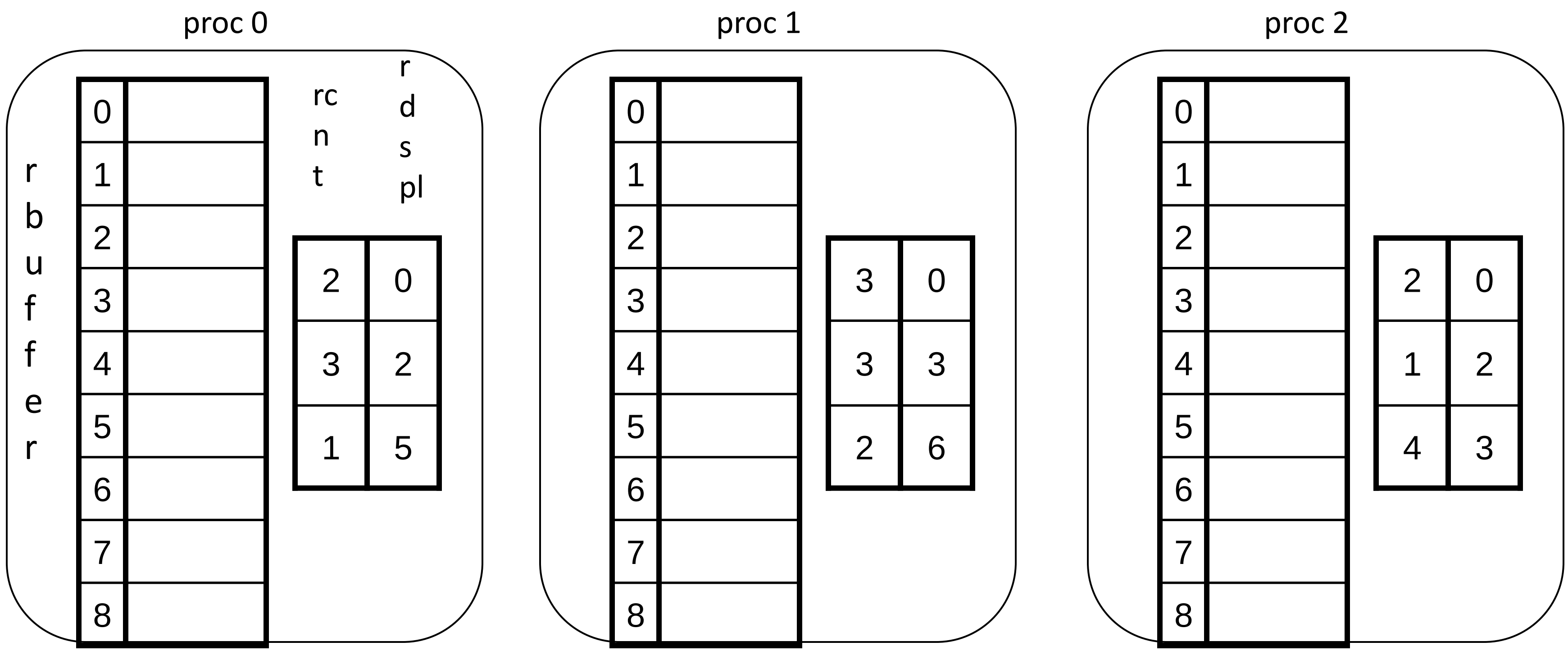


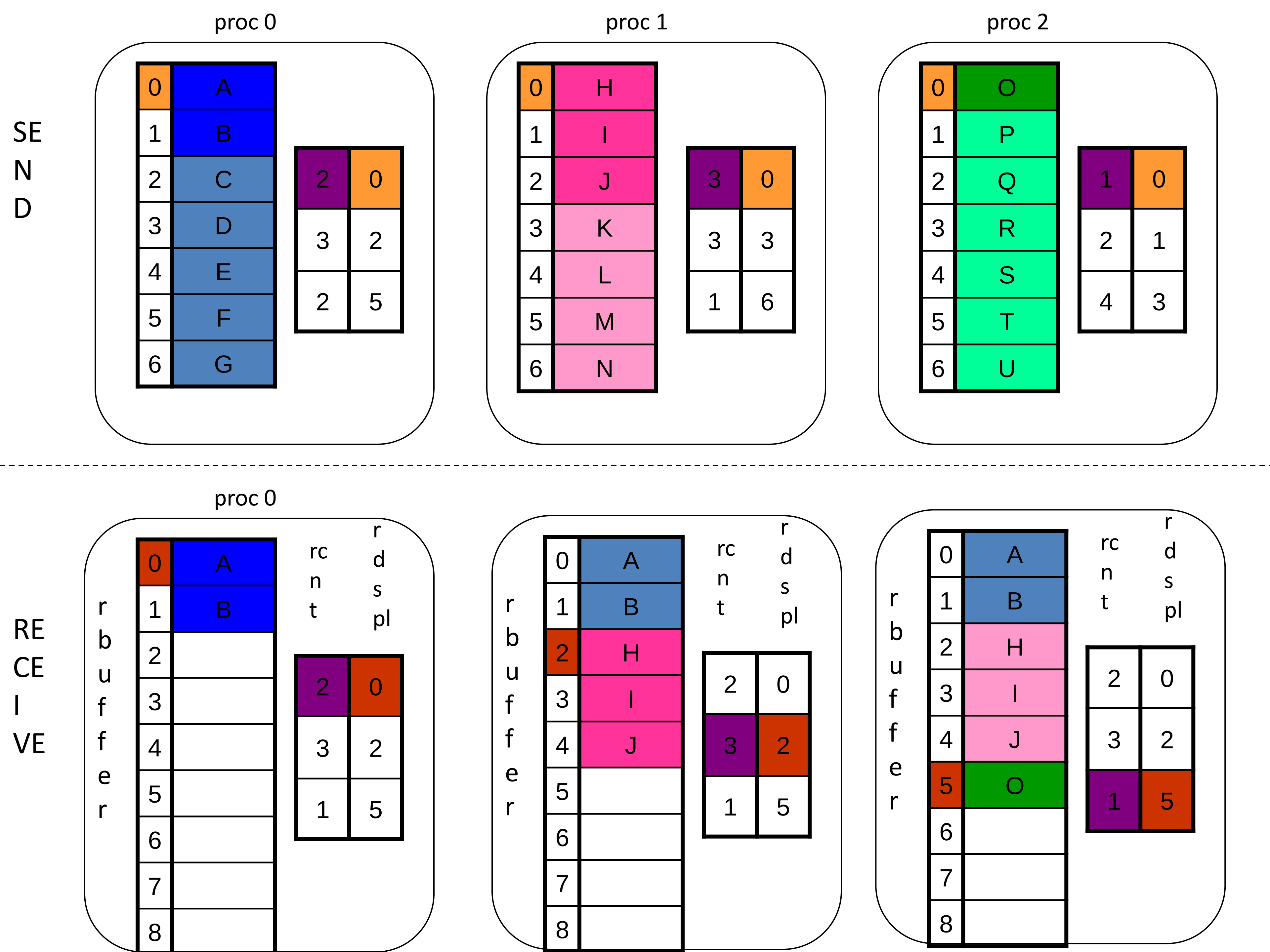
Process 0 Sends to Process 2



Receive of MPI_Alltoallv()

RECEIVE





proc 0
proc 1
proc 2
SE
N
D
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
RE
CE
I
VE
proc 0
rbuffer
rcnt
rdspl
0 A
1 B
2
3
4
5
6
7
8
2 0
3 2
1 5
rbuffer
rcnt
rdspl
0 A
1 B
2 H
3 I
4 J
5
6
7
8
2 0
3 2
1 5
rbuffer
rcnt
rdspl
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: $T_p = \dfrac{n^2}{p} + n + t_s(p-1) + t_w n$

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 $\boldsymbol{b}$.

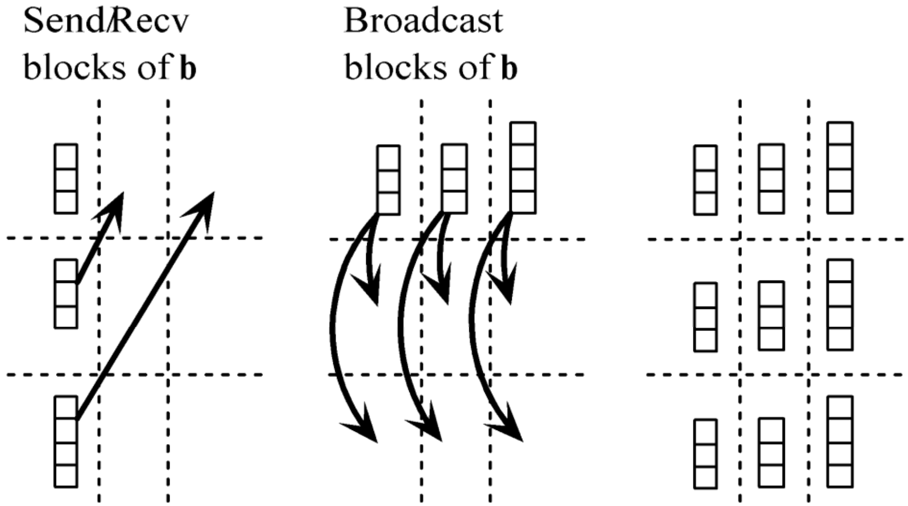
2. Each task computes a matrix-vector multiplication using its portion of A and $\boldsymbol{b}$.

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.

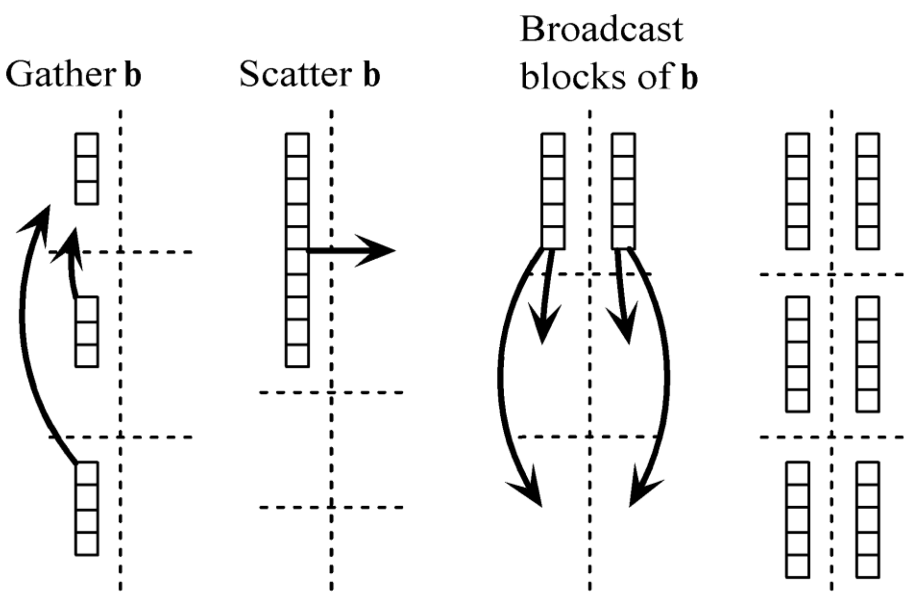
Distributing $\boldsymbol{b}$

- 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