

Lecture 6: 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);
}

```

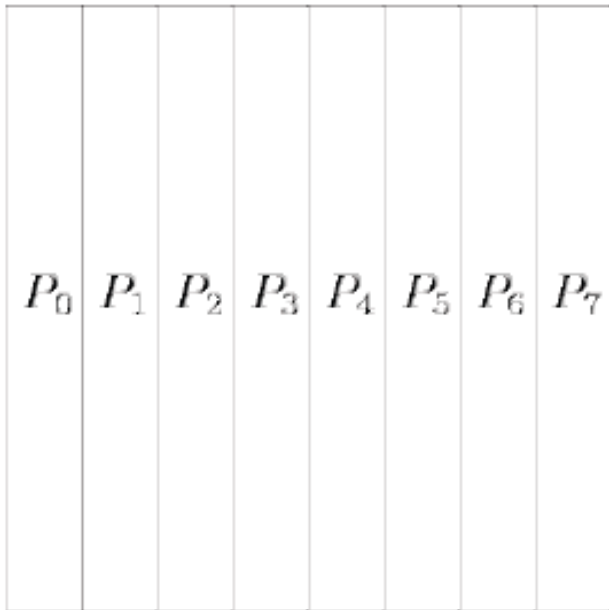
replicate_block_vector()
is used to transform a
vector from a block
distribution to a
replicated distribution

Column-wise Block-Striped Decomposition

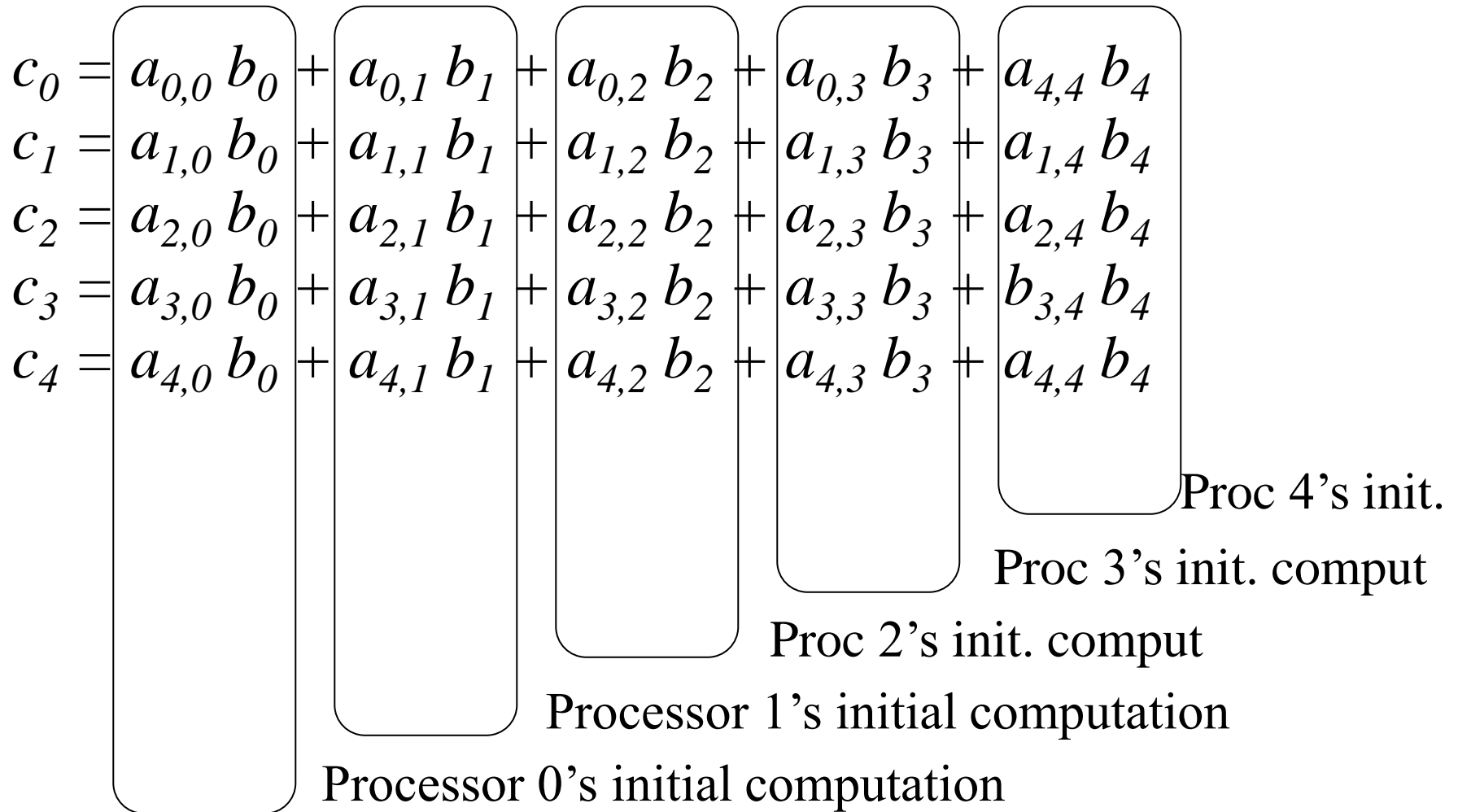
Summary of algorithm for computing $\mathbf{c} = A\mathbf{b}$

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

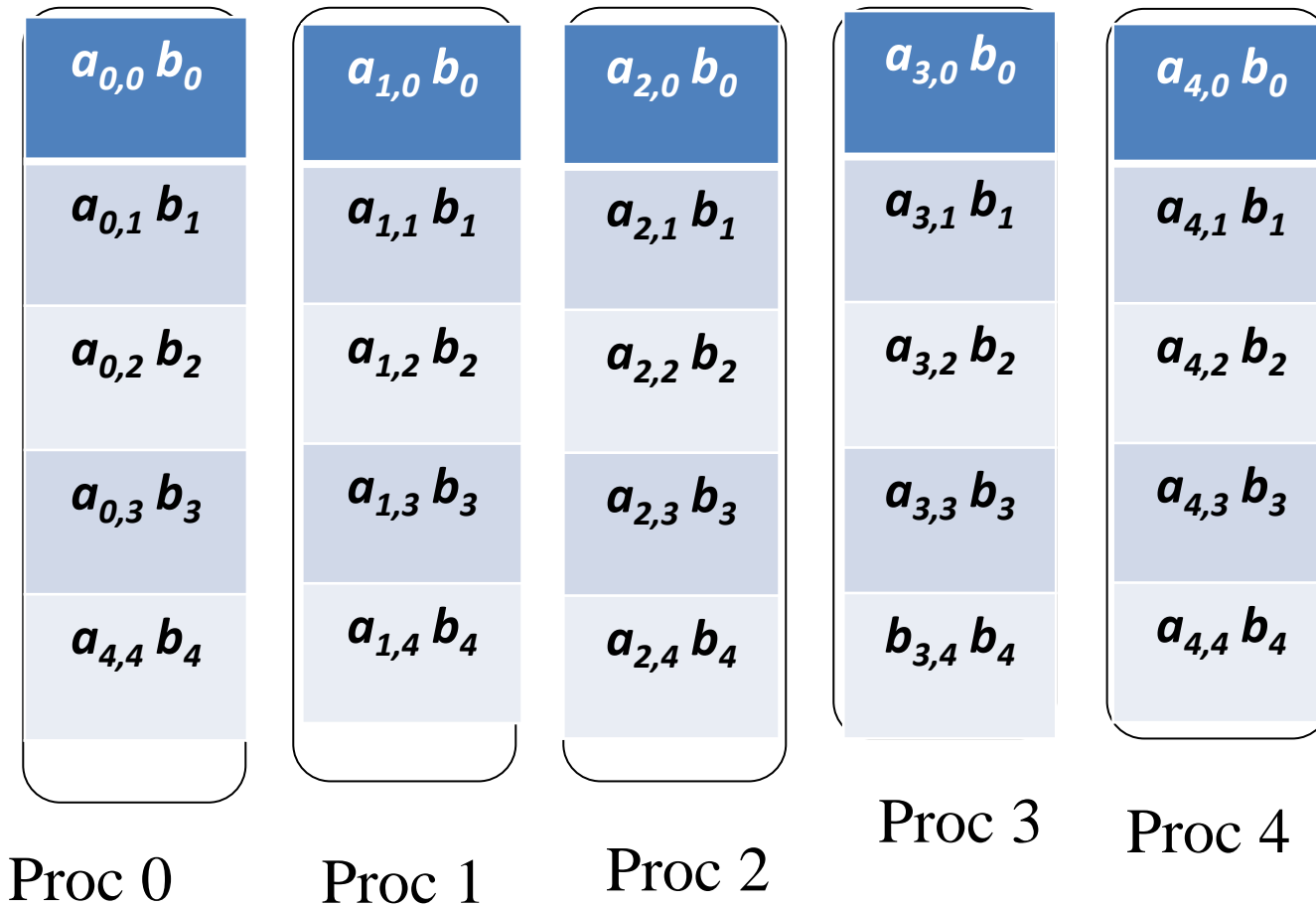
column-wise distribution



1. Read in matrix stored in row-major manner and distribute by column-wise mapping
2. Each task i compute $b_i \mathbf{a}_i$ 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.



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++)
```

```
{
```

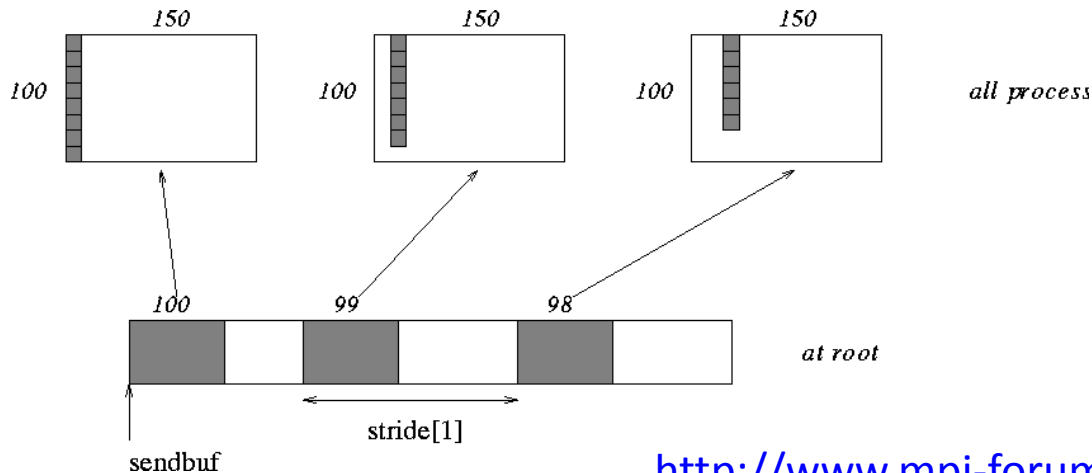
```
    if(id == (p-1)) fread(buffer,datum_size, *n, infileptr);
```

```
    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++)
```

```
{
```

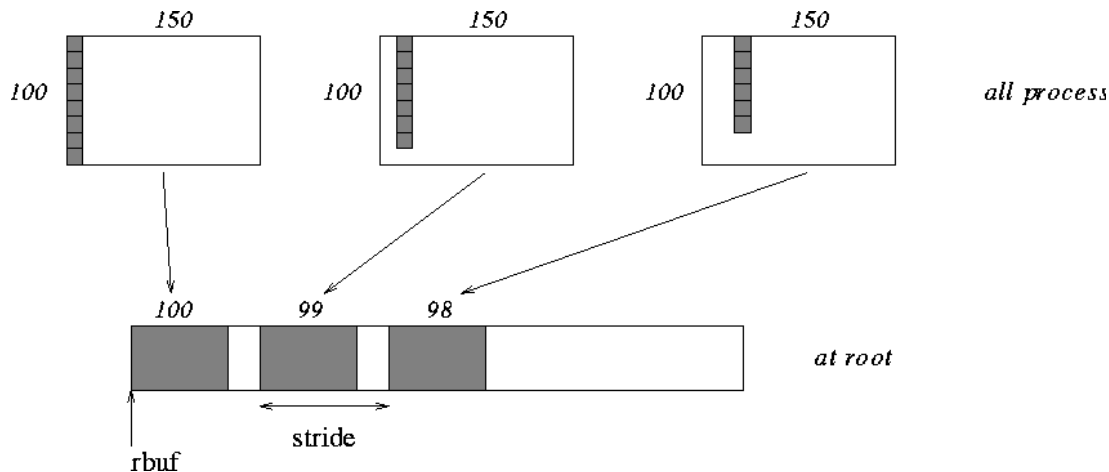
```
    MPI_Gatherv(a[i], BLOCK_SIZE(id,p,n), dtype, buffer,  
                rec_count, rec_disp, dtype, 0, comm);
```

```
....
```

```
}
```

```
}
```

- **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)
 - *recvcnts*: integer array (of length group size) containing the number of elements that are received from each process (significant only at root)
 - *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

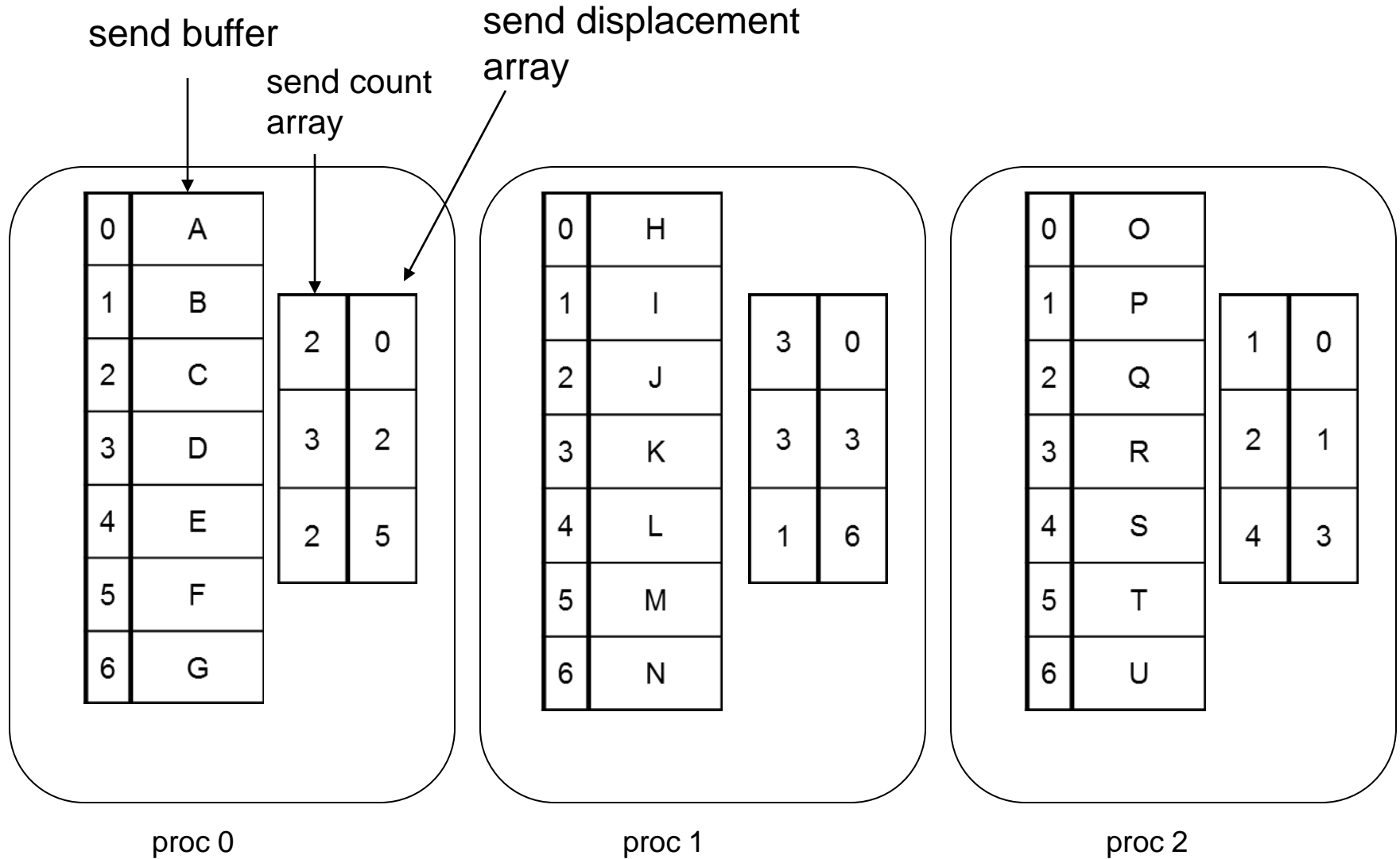
- $c_i = b_0 \mathbf{a}_{i,0} + b_1 \mathbf{a}_{i,1} + b_2 \mathbf{a}_{i,2} + \cdots + b_n \mathbf{a}_{i,n}$
- Each process need to distribute $n - 1$ terms to other processes and gather $n - 1$ terms from them (assume fine-grained decomposition).
 - MPI_Alltoallv() is used to do this **all-to-all**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

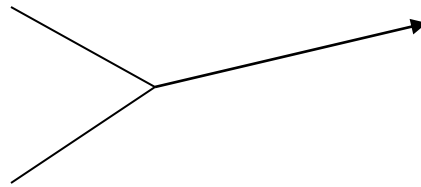


Process 0 Sends to Process 0

| | |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |

index ↗

Proc 0 send buffer



this chunk of send buffer goes to receive buffer of proc 0

send to **receive** buffer of proc with same **rank** as **index**

| | |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 2 |

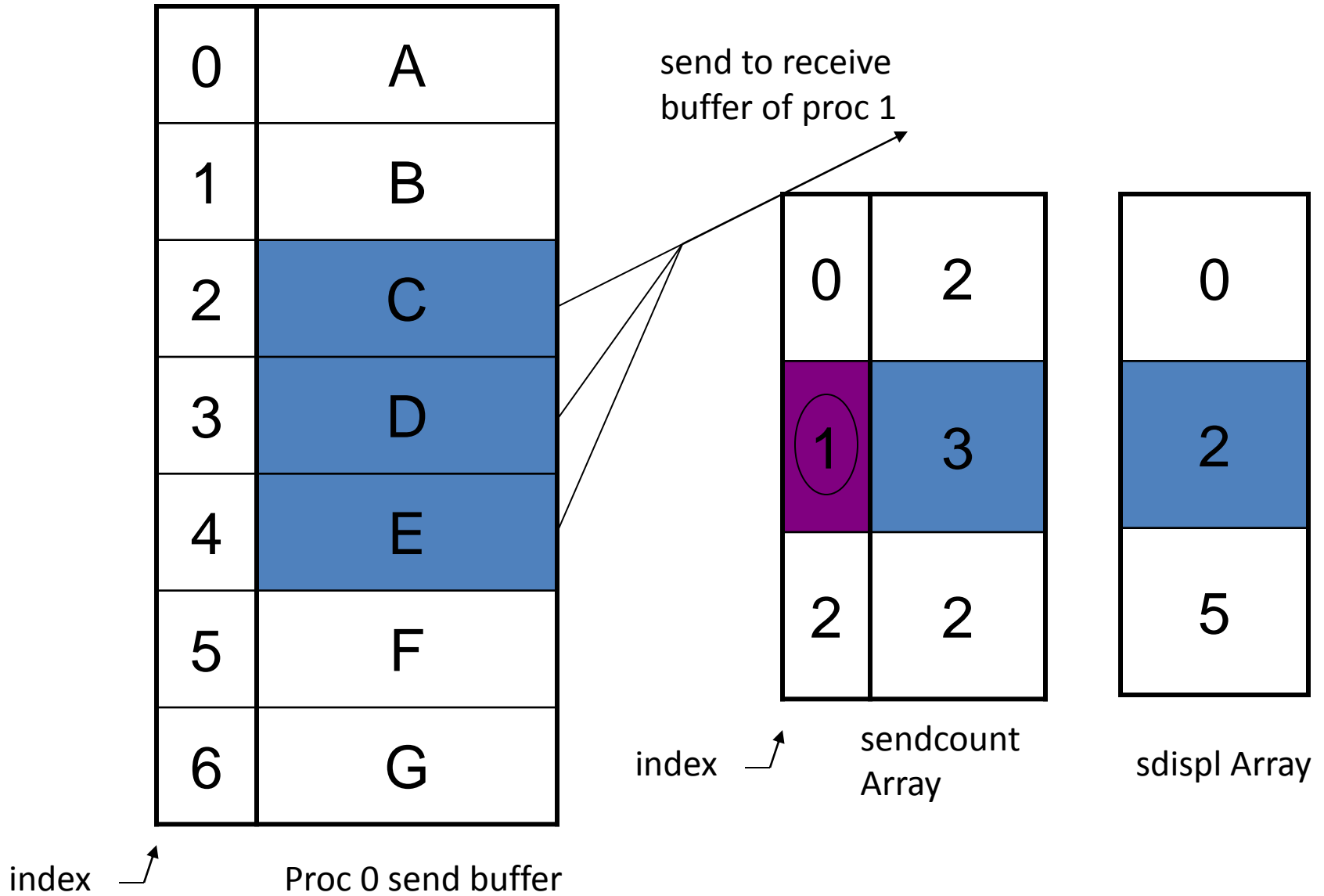
index ↗

sendcount Array

| |
|---|
| 0 |
| 2 |
| 5 |

sdispl Array

Process 0 Sends to Process 1



Process 0 Sends to Process 2

| | |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |

index ↗

Proc 0 send buffer

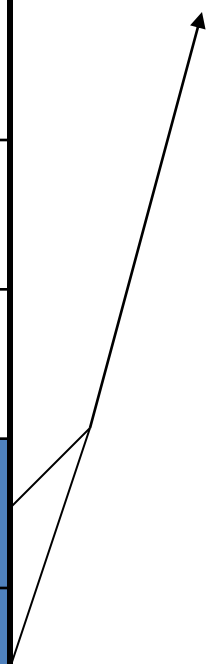
send to receive
buffer of **proc 2**

| | |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 2 |

sendcount
Array

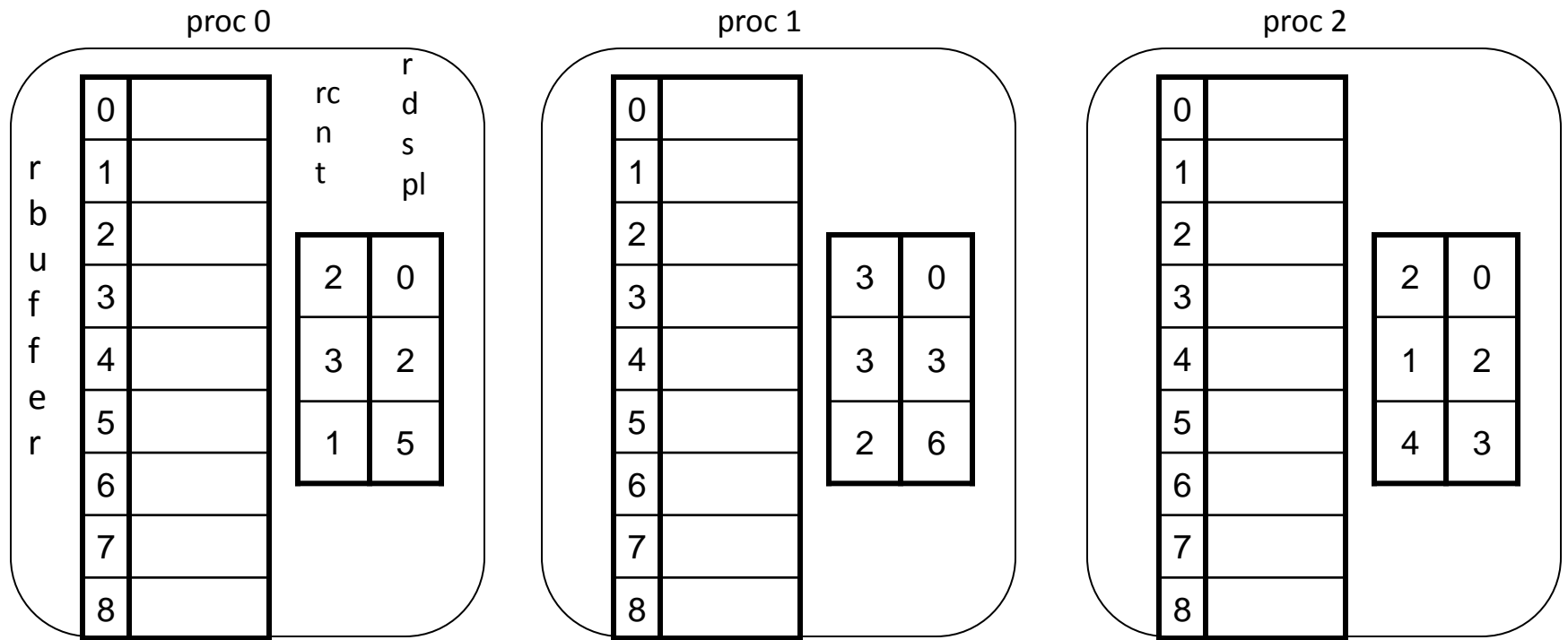
| |
|---|
| 0 |
| 2 |
| 5 |

sdispl Array

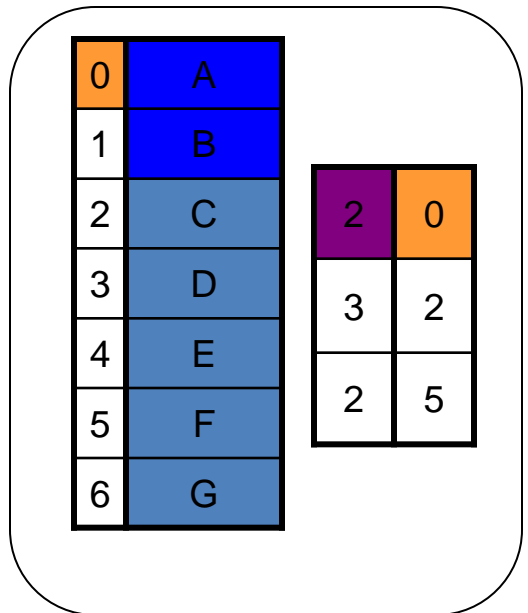


Receive of MPI_Alltoallv()

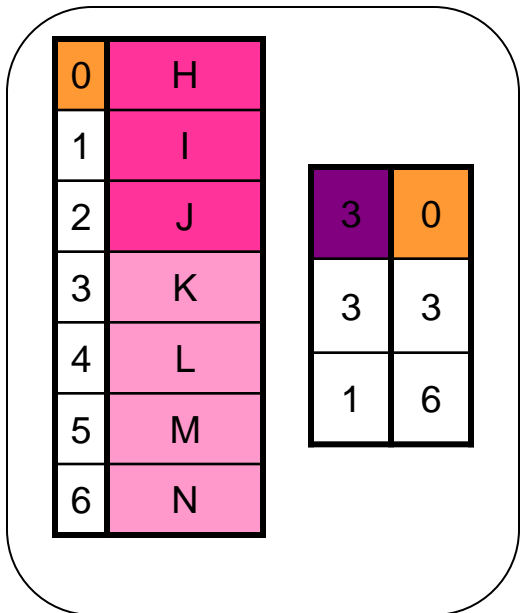
RE
CE
I
VE



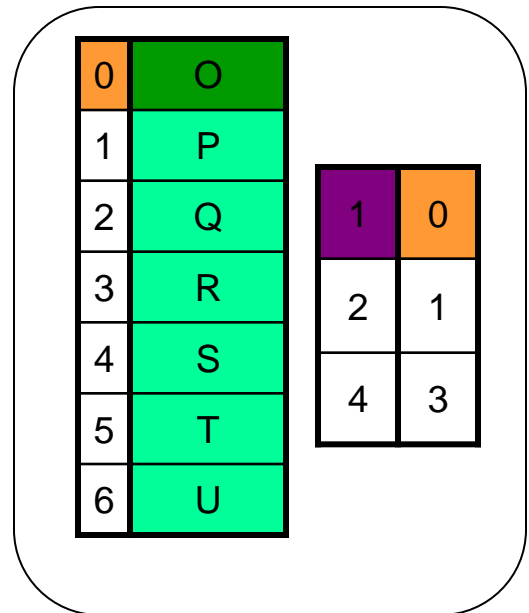
proc 0



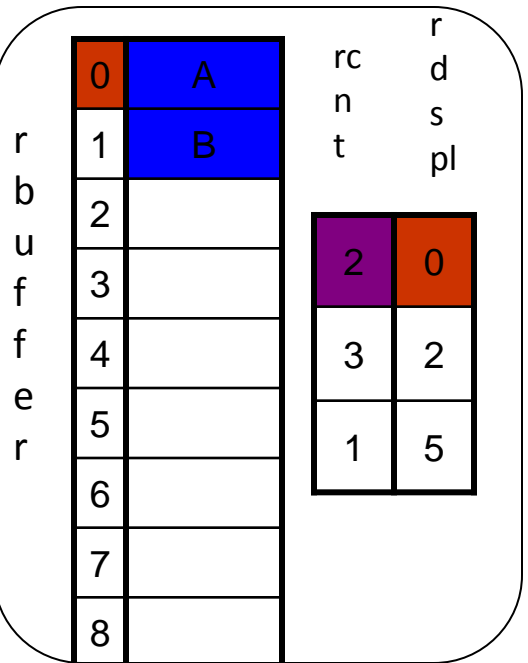
proc 1



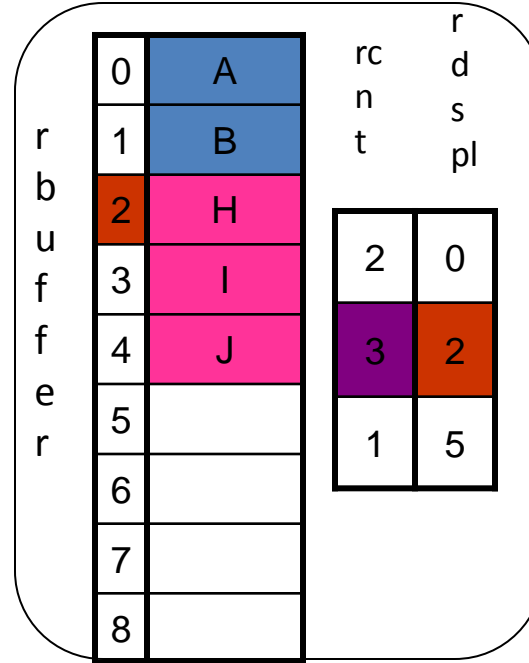
proc 2



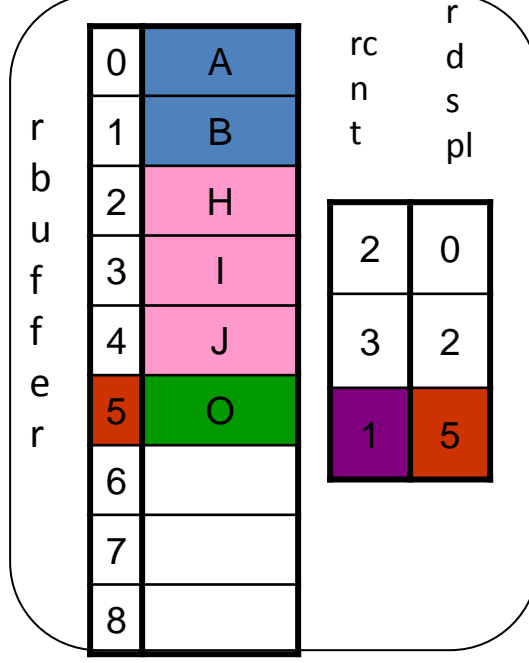
proc 0



proc 1



proc 2



SE
ND

RE
CE
I
VE

r
b
u
f
f
e
r

r
b
u
f
f
e
r

r
b
u
f
f
e
r

rc
nt
rd
spl

rc
nt
rd
spl

rc
nt
rd
spl

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.

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

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

- 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 $\mathbf{y} = A\mathbf{b}$

- 2D block partition is used to distribute matrix.
- Let $A = [a_{ij}]$, $\mathbf{b} = [b_1, b_2, \dots, b_n]^T$, and $\mathbf{y} = [y_1, y_2, \dots, y_n]^T$
- 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 i th element of \mathbf{y} .

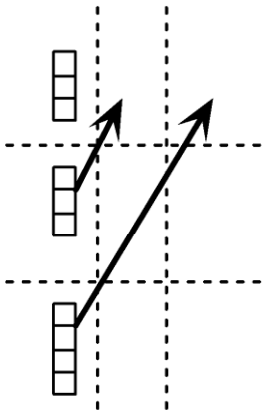
| | | | |
|----------|----------|----------|----------|
| P_0 | P_1 | P_2 | P_3 |
| P_4 | P_5 | P_6 | P_7 |
| P_8 | P_9 | P_{10} | P_{11} |
| P_{12} | P_{13} | P_{14} | P_{15} |

1. Read in matrix stored in row-major manner and distribute by 2D block mapping. Also distribute \mathbf{b} so that each task has the correct portion of \mathbf{b} .
2. Each task computes a matrix-vector multiplication using its portion of A and \mathbf{b} .
3. Tasks in each row of the task grid perform a sum-reduction on their portion of \mathbf{y} .
4. After the sum-reduction, \mathbf{y} is distributed by blocks among the tasks in the first column of the task grid.

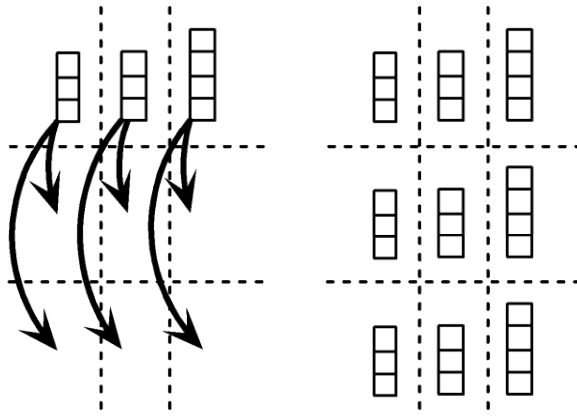
Distributing \mathbf{b}

- Initially, \mathbf{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 \mathbf{b}
 - If p not square
 - Gather \mathbf{b} on process 0, 0
 - Process 0, 0 broadcasts to first row processes
- Step 2: First row processes scatter \mathbf{b} within columns

Send/Recv
blocks of \mathbf{b}



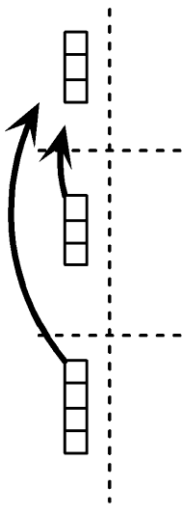
Broadcast
blocks of \mathbf{b}



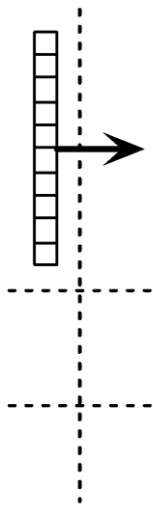
When p is a square number

(a)

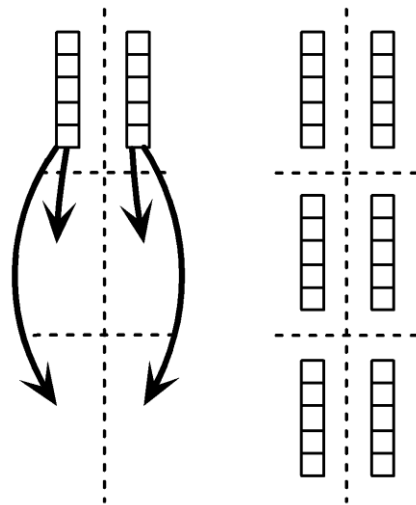
Gather \mathbf{b}



Scatter \mathbf{b}



Broadcast
blocks of \mathbf{b}



When p is not a square number

(b)