

CSE 160

Lecture 23

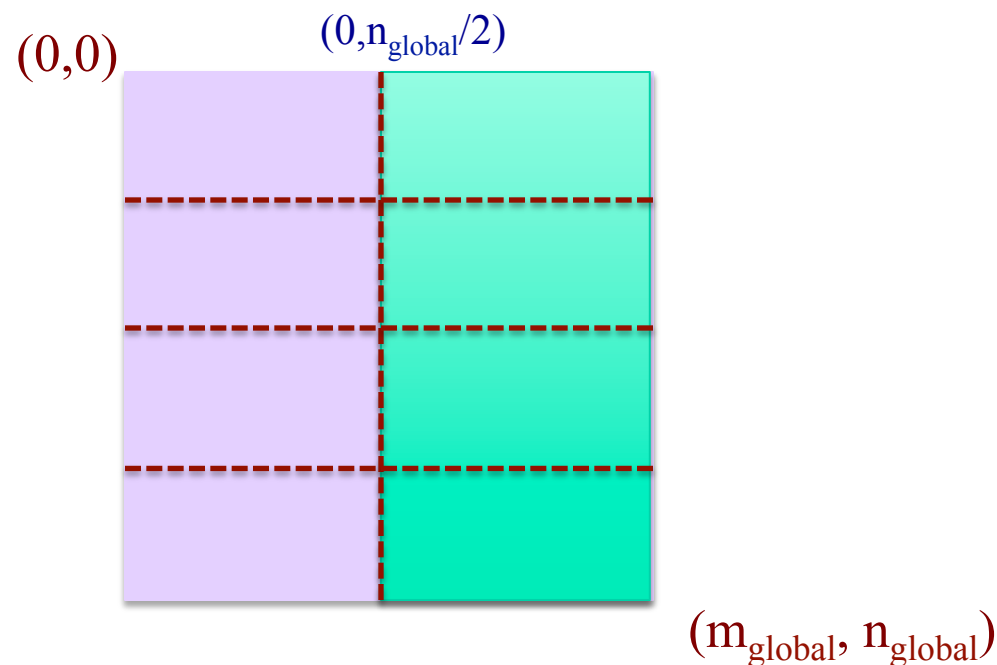
Matrix Multiplication – Continued
Managing communicators
Gather and Scatter (Collectives)

Today's lecture

- All to all communication
- Application to Parallel Sorting
- Blocking for cache

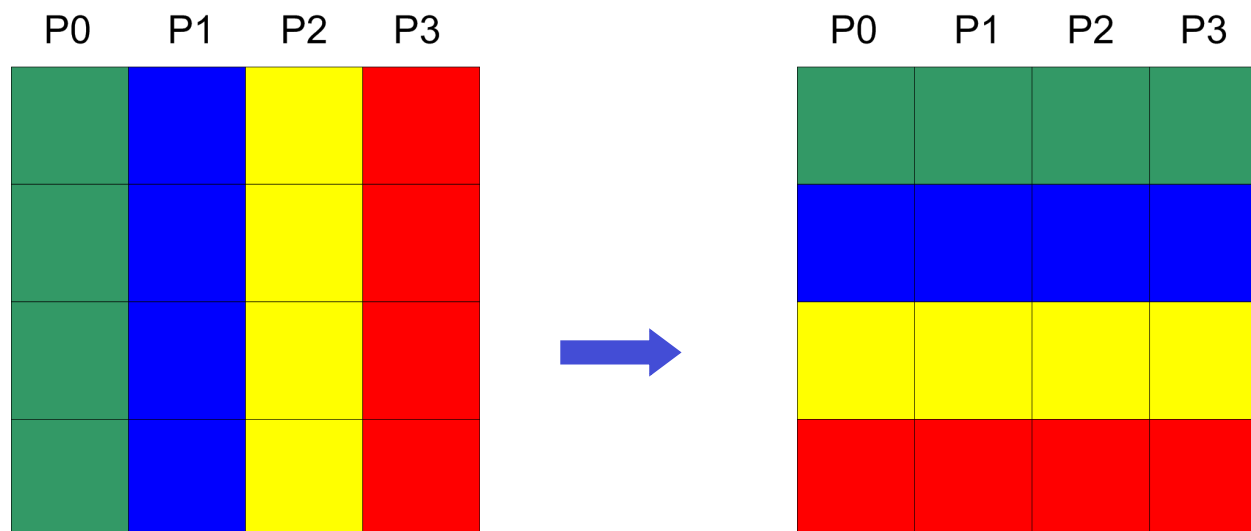
Global to local mapping

- In some applications, we need to compute a local to global mapping of array indices
- In the 4th assignment, we want to set all values to zero in certain regions of the problem



All to all

- Also called *total exchange* or *personalized communication*: a transpose
- Each process sends a different chunk of data to each of the other processes
- Used in sorting and the Fast Fourier Transform



Exchange algorithm

- n elements / processor (n total elements)
- $p - 1$ step algorithm
 - Each processor exchanges n/p elements with each of the others
 - In step i , process k exchanges with processes $k \pm i$

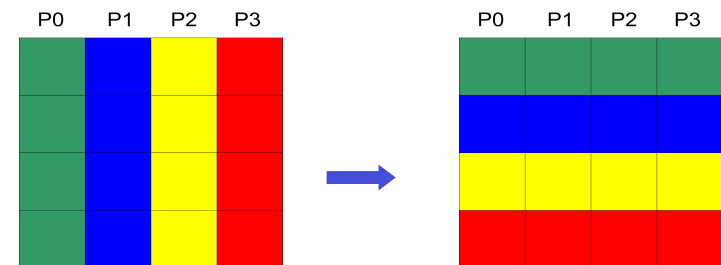
for $i = 1$ to $p-1$

src = $(\text{rank} - i + p) \bmod p$

dest = $(\text{rank} + i) \bmod p$

sendrecv(from src to dest)

end for



- Good algorithm for long messages
- Running time:

$$(p-1)\alpha + (p-1)\frac{n}{p}\beta \approx n\beta$$

Recursive doubling for short messages

- In each of $\lceil \log p \rceil$ phases all nodes exchange $\frac{1}{2}$ their accumulated data with the others
- Only $P/2$ messages are sent at any one time

$D = 1$

while ($D < p$)

 Exchange & accumulate data with rank $\otimes D$

 Left shift D by 1

end while

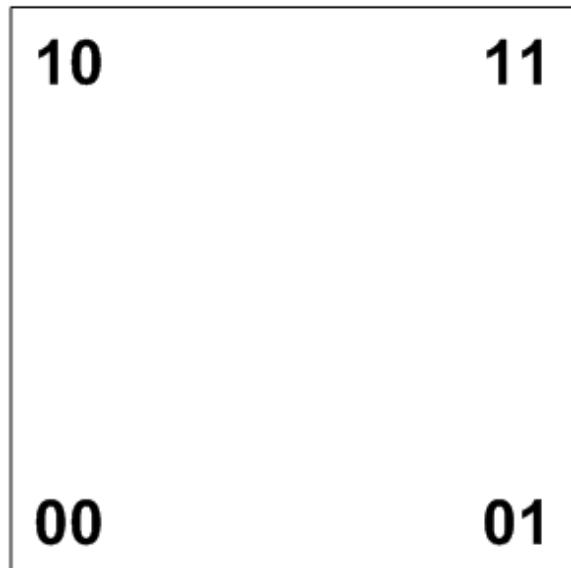
- Optimal running time for short messages

$$\lceil \lg P \rceil \alpha + nP\beta \approx \lceil \lg P \rceil \alpha$$

Flow of information

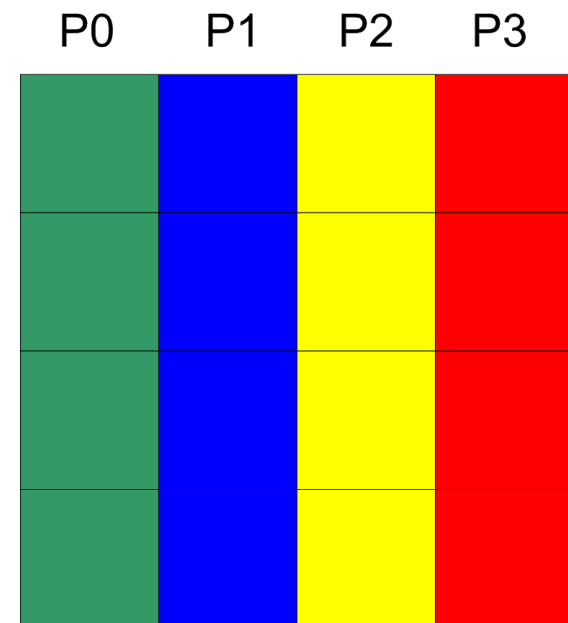
A B C D

A B C D



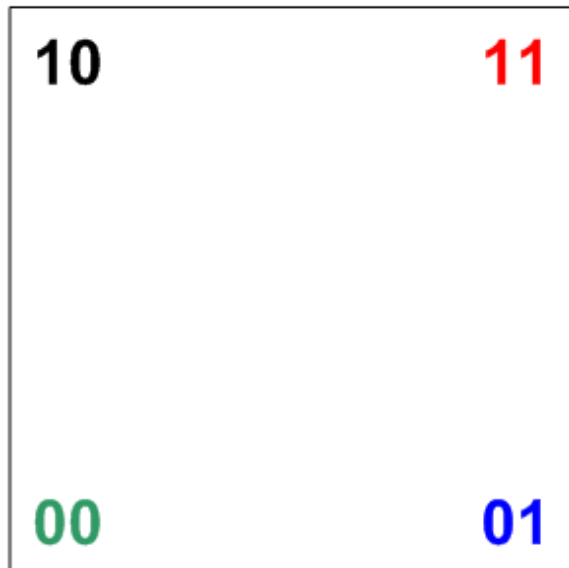
A B C D

A B C D



Flow of information

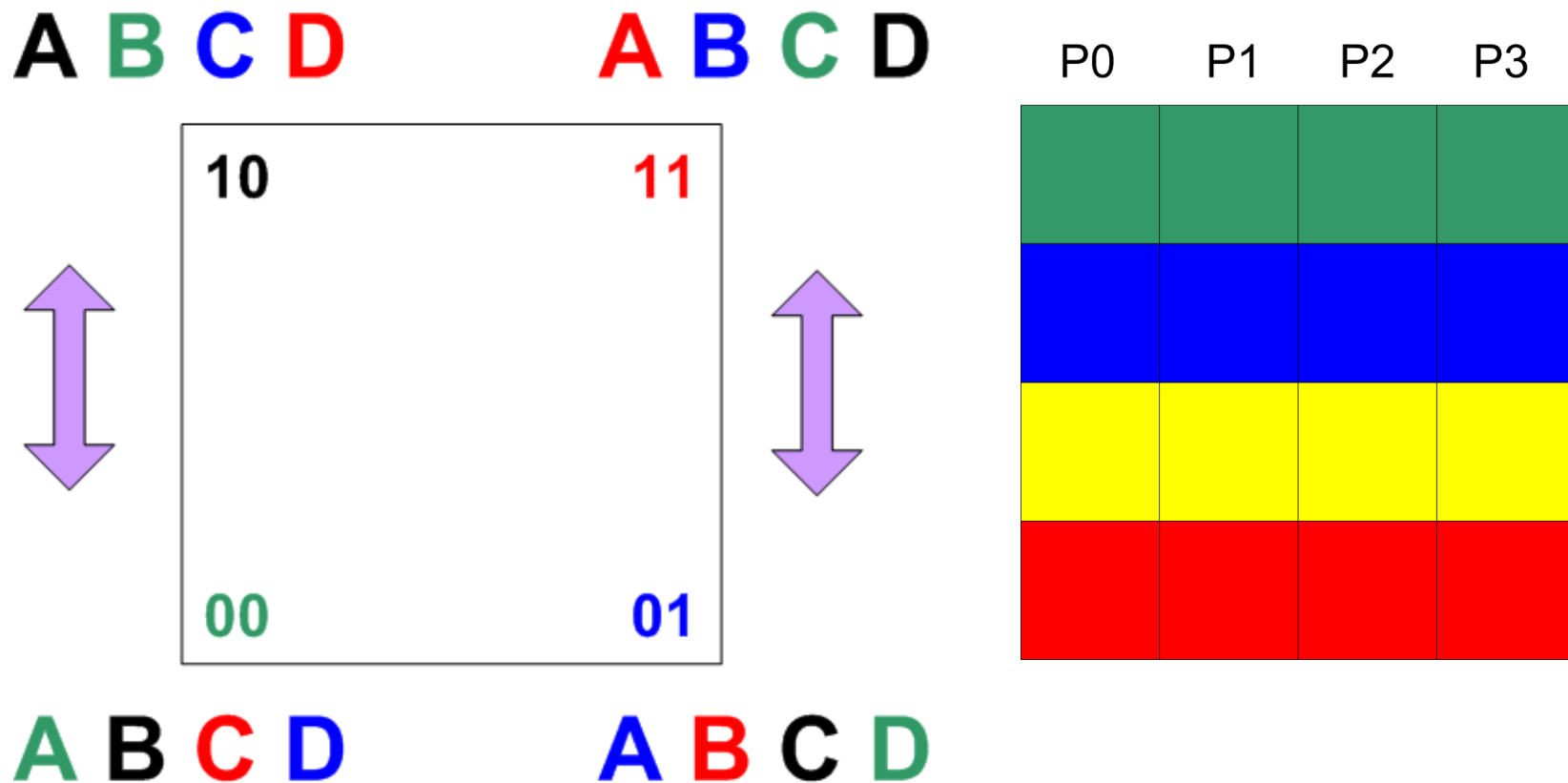
A B C D ↔ **A B C D**



A B C D ↔ **A B C D**

P0	P1	P2	P3
Green	Green	Yellow	Yellow
Blue	Blue	Red	Red
Green	Green	Yellow	Yellow
Blue	Blue	Red	Red

Flow of information



Summarizing all to all

- Short messages $\lceil \lg P \rceil \alpha$
- Long messages $\frac{P-1}{P} n \beta$

“Vector” All to All

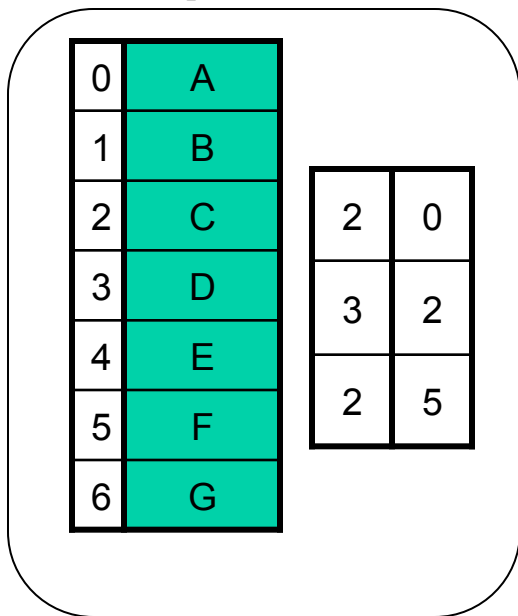
- Generalize all-to-all, gather, scatter, etc.
- Processes supply varying length data
- Gather/scatter vectors of different lengths
- Vector all-to-all [Used in sample sort (coming)]

`MPI_Alltoallv (`
 `void *sendbuf, int sendcounts[], int sDispl [],`
 `MPI_Datatype sendtype,`
 `void* recvbuf, int recvcnts[], int rDispl[],`
 `MPI_Datatype recvtype, MPI_Comm comm )`

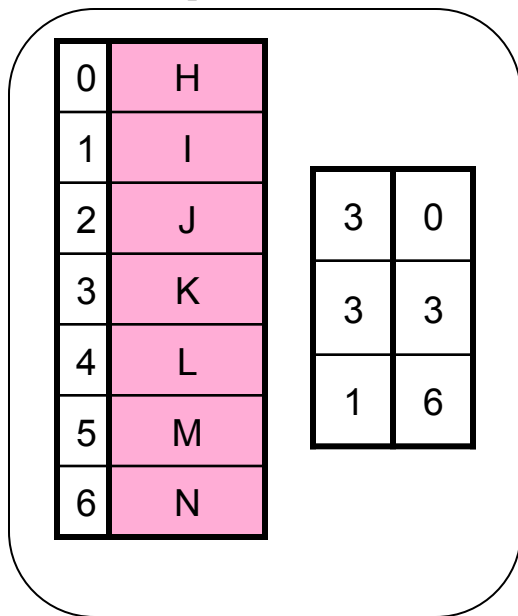
- Following diagrams courtesy of
Lori Pollock (U. Delaware)
www.eecis.udel.edu/~pollock/372/f06/

S
E
N
D

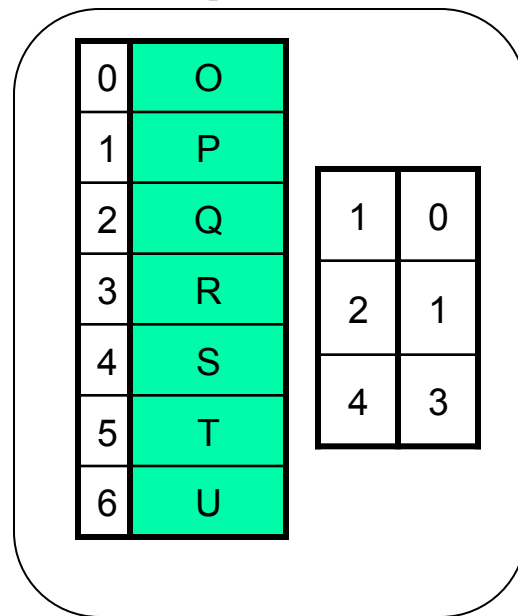
proc 0



proc 1

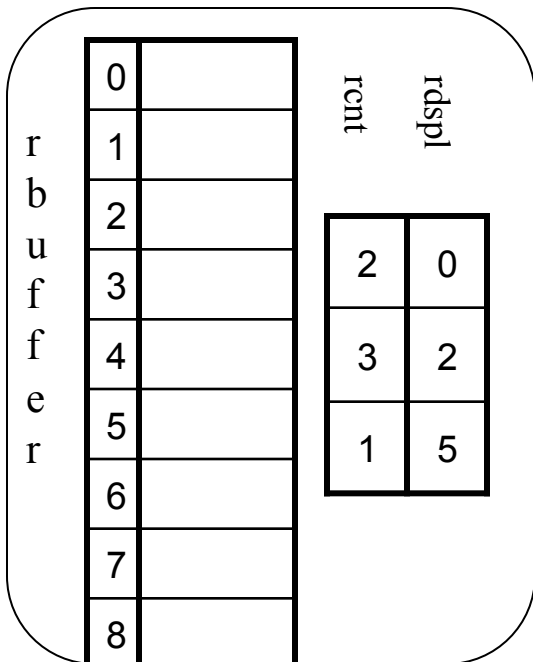


proc 2

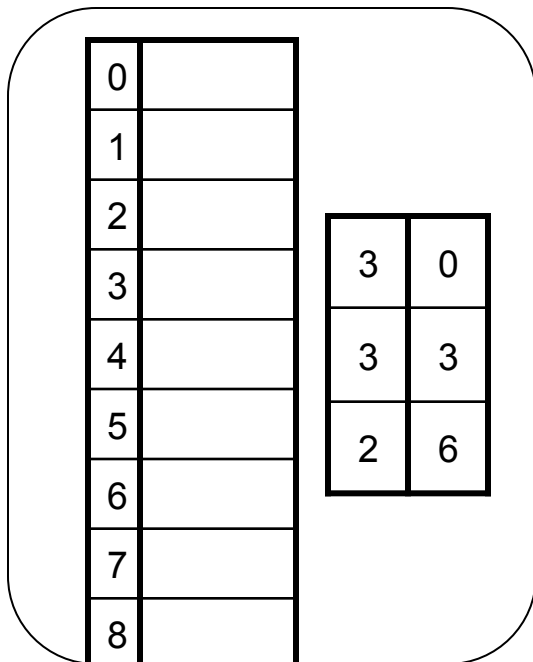


R
E
C
E
I
V
E

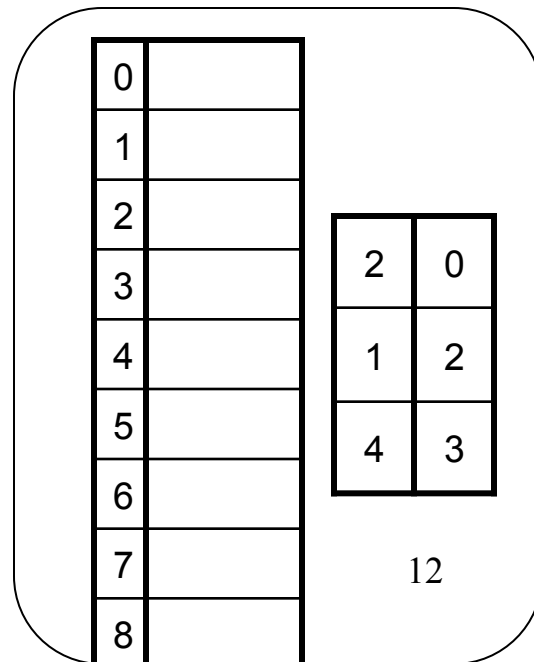
proc 0



proc 1

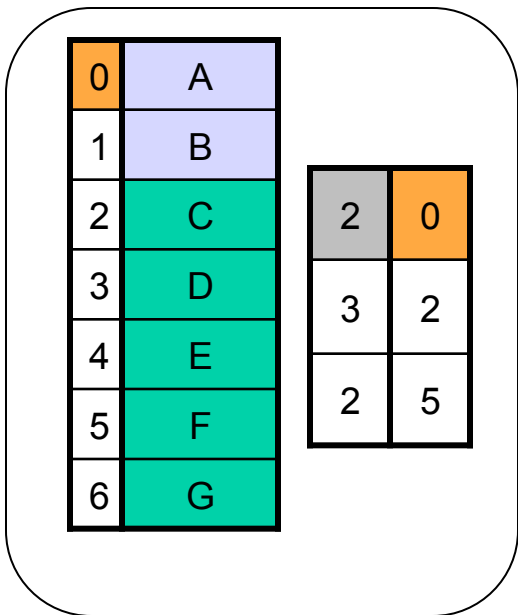


proc 2

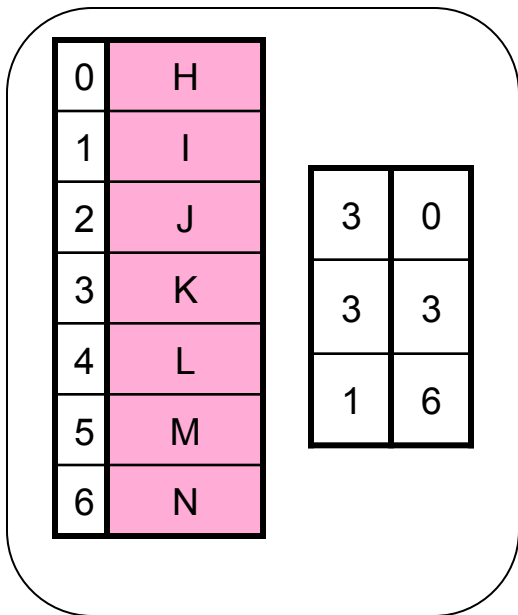


S
E
N
D

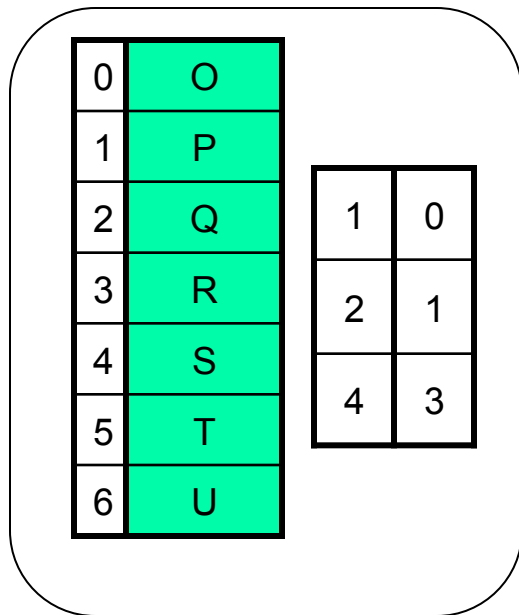
proc 0



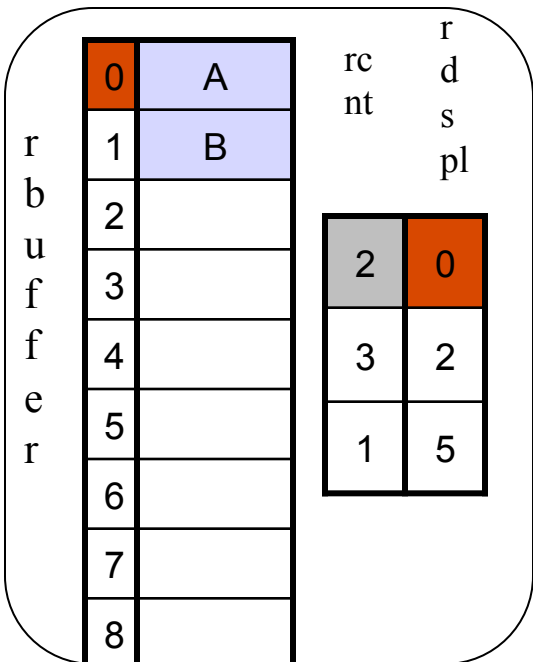
proc 1



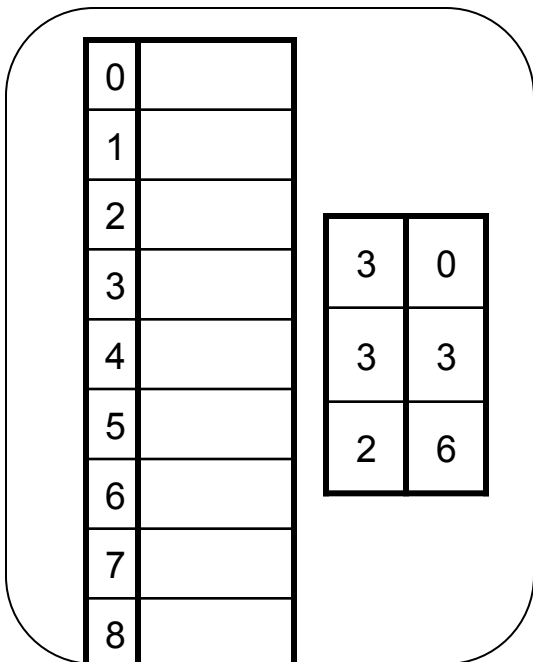
proc 2



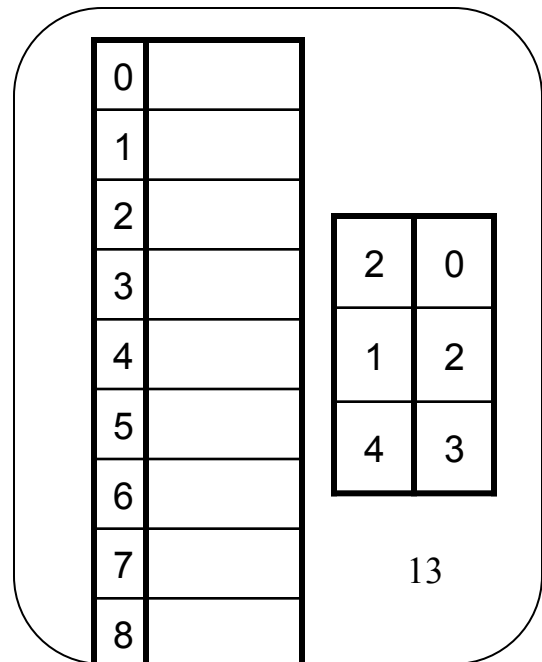
proc 0



proc 1



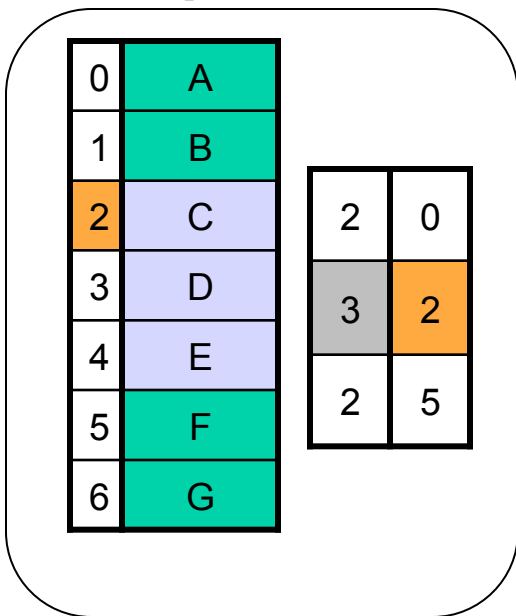
proc 2



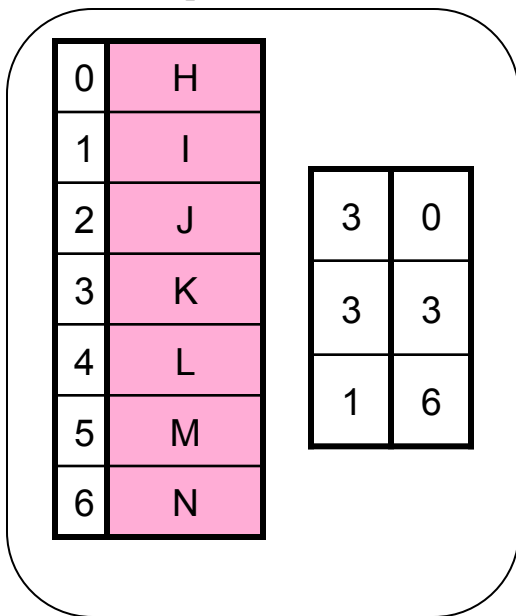
R
E
C
E
I
V
E

S
E
N
D

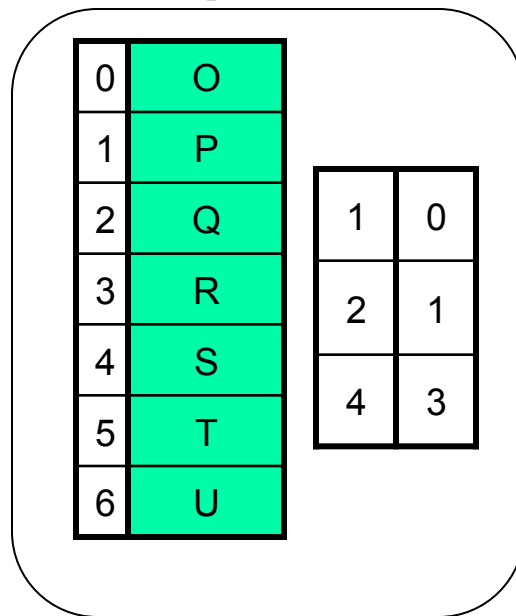
proc 0



proc 1

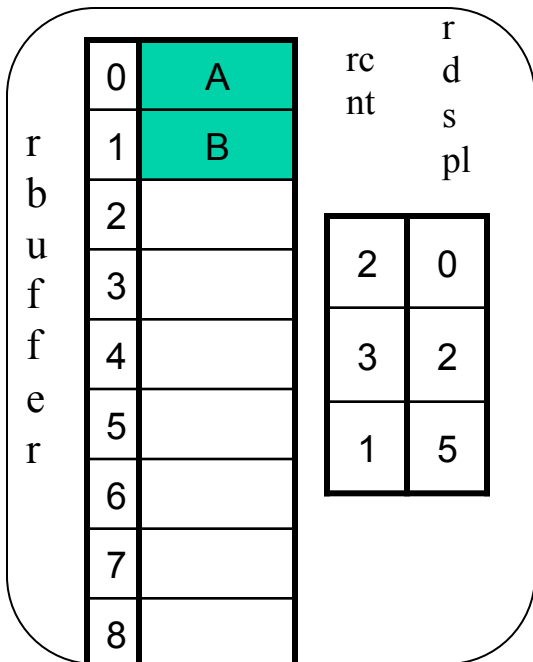


proc 2

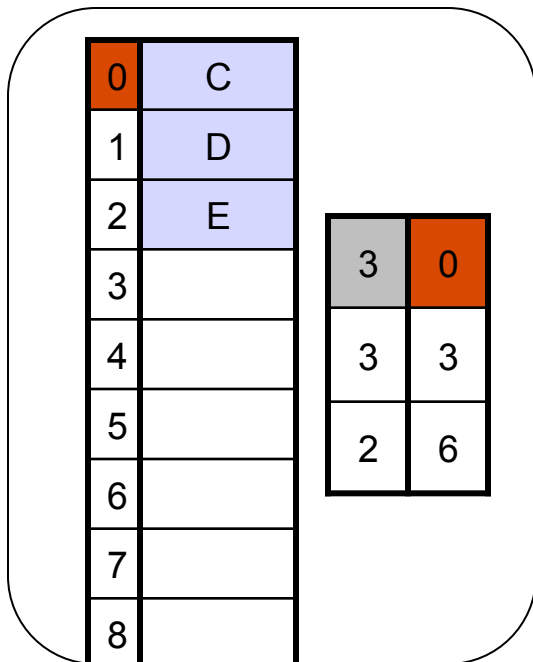


R
E
C
E
I
V
E

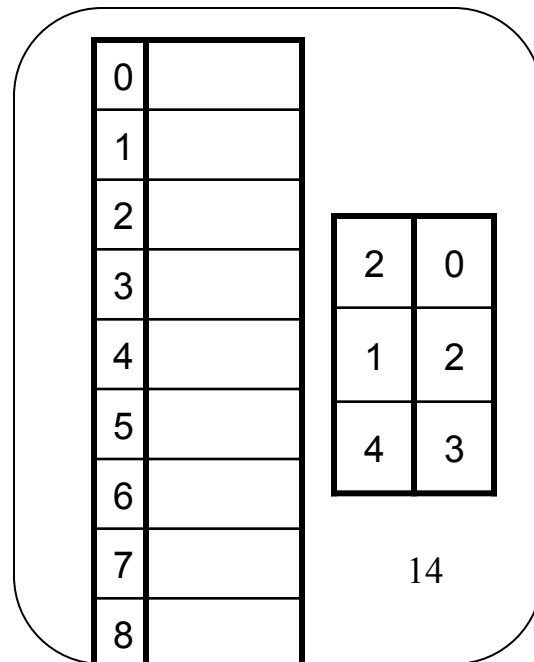
proc 0



proc 1

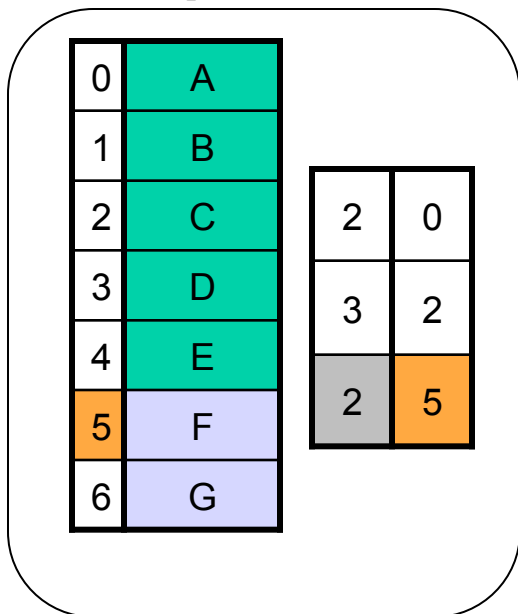


proc 2

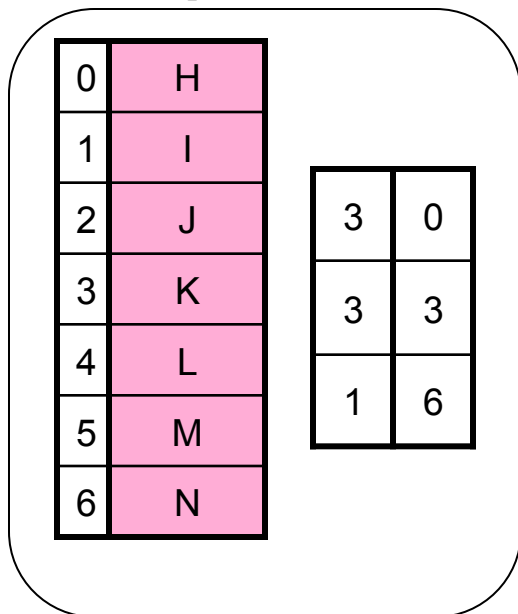


S
E
N
D

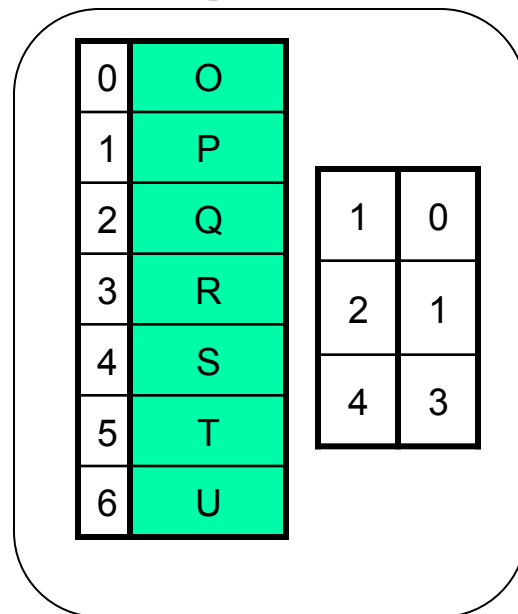
proc 0



proc 1

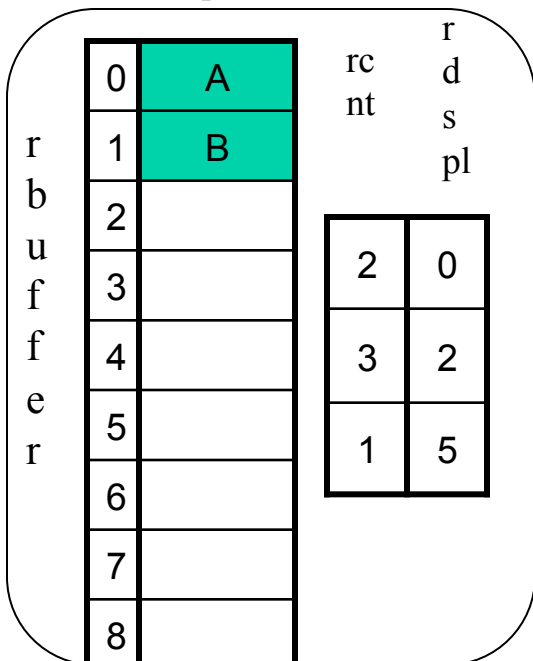


proc 2

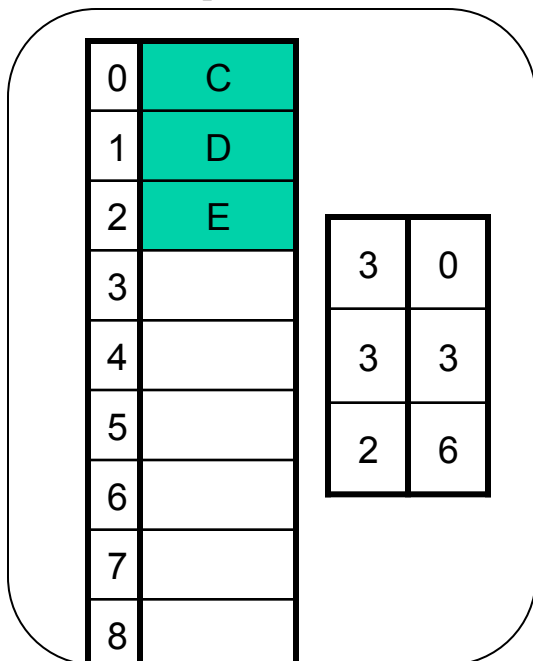


R
E
C
E
I
V
E

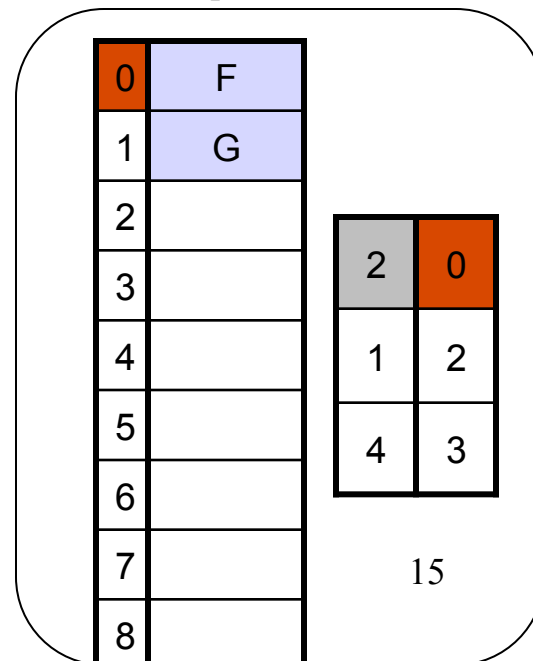
proc 0



proc 1

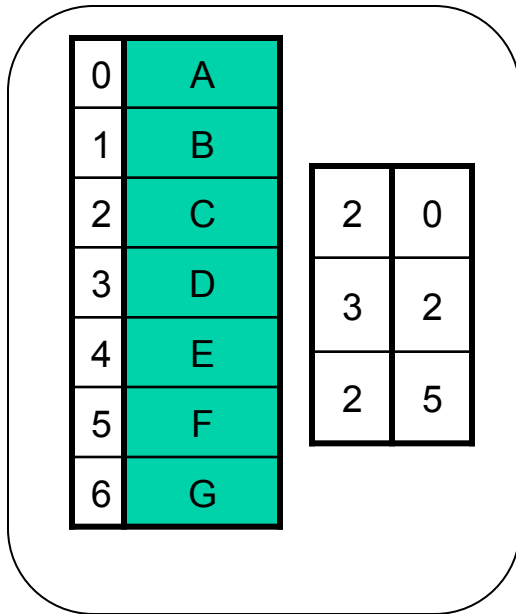


proc 2

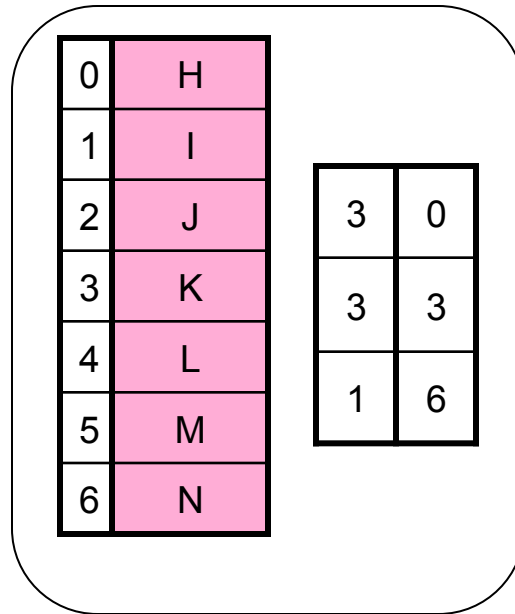


S
E
N
D

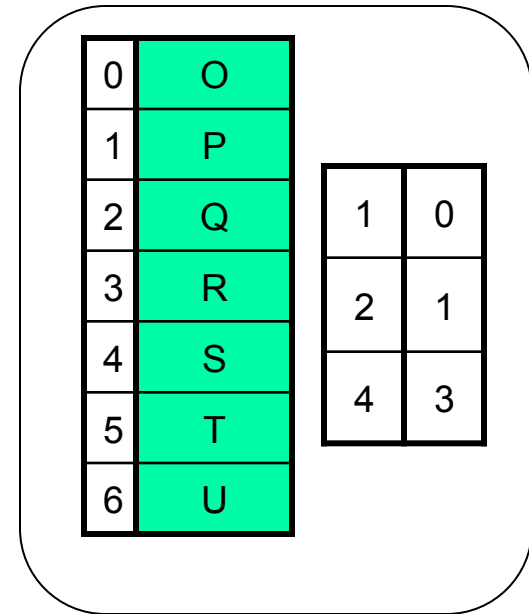
proc 0



proc 1

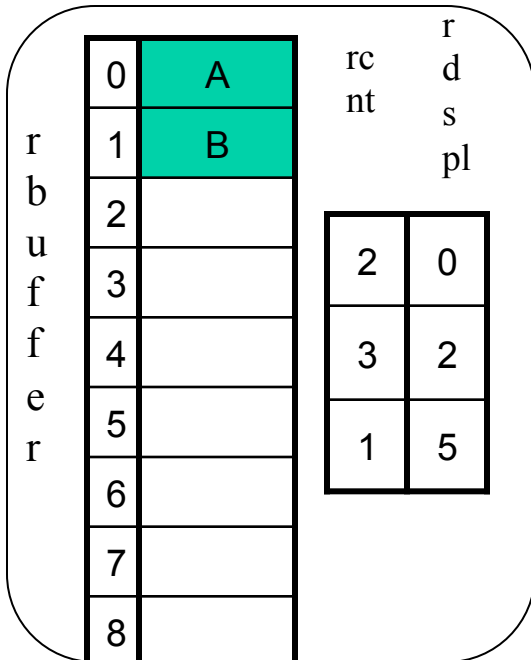


proc 2

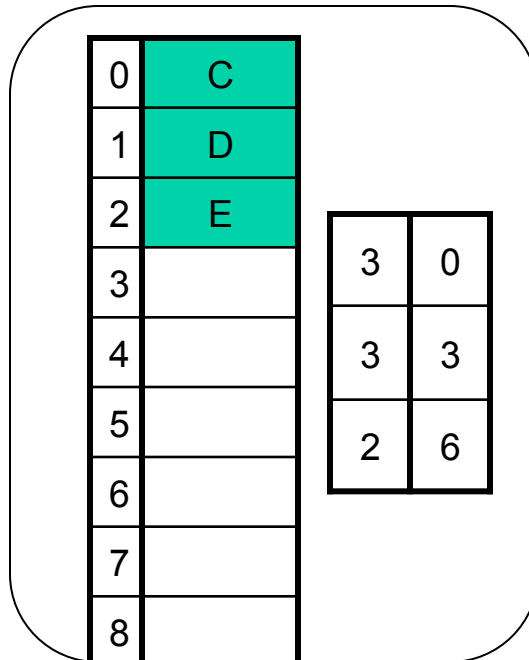


R
E
C
E
I
V
E

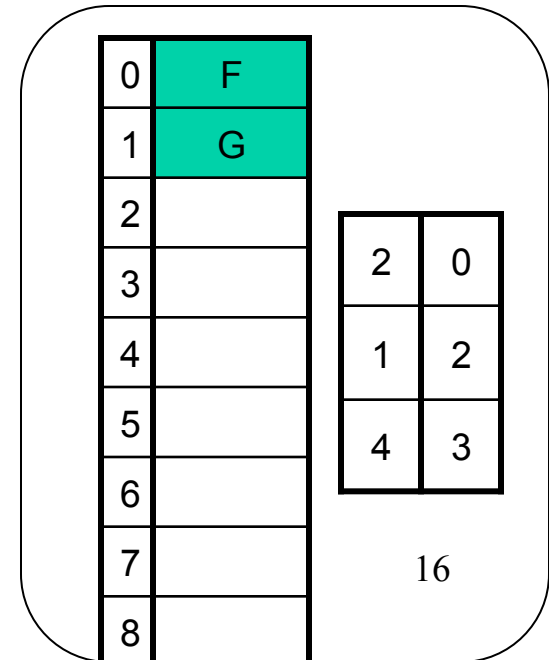
proc 0



proc 1



proc 2



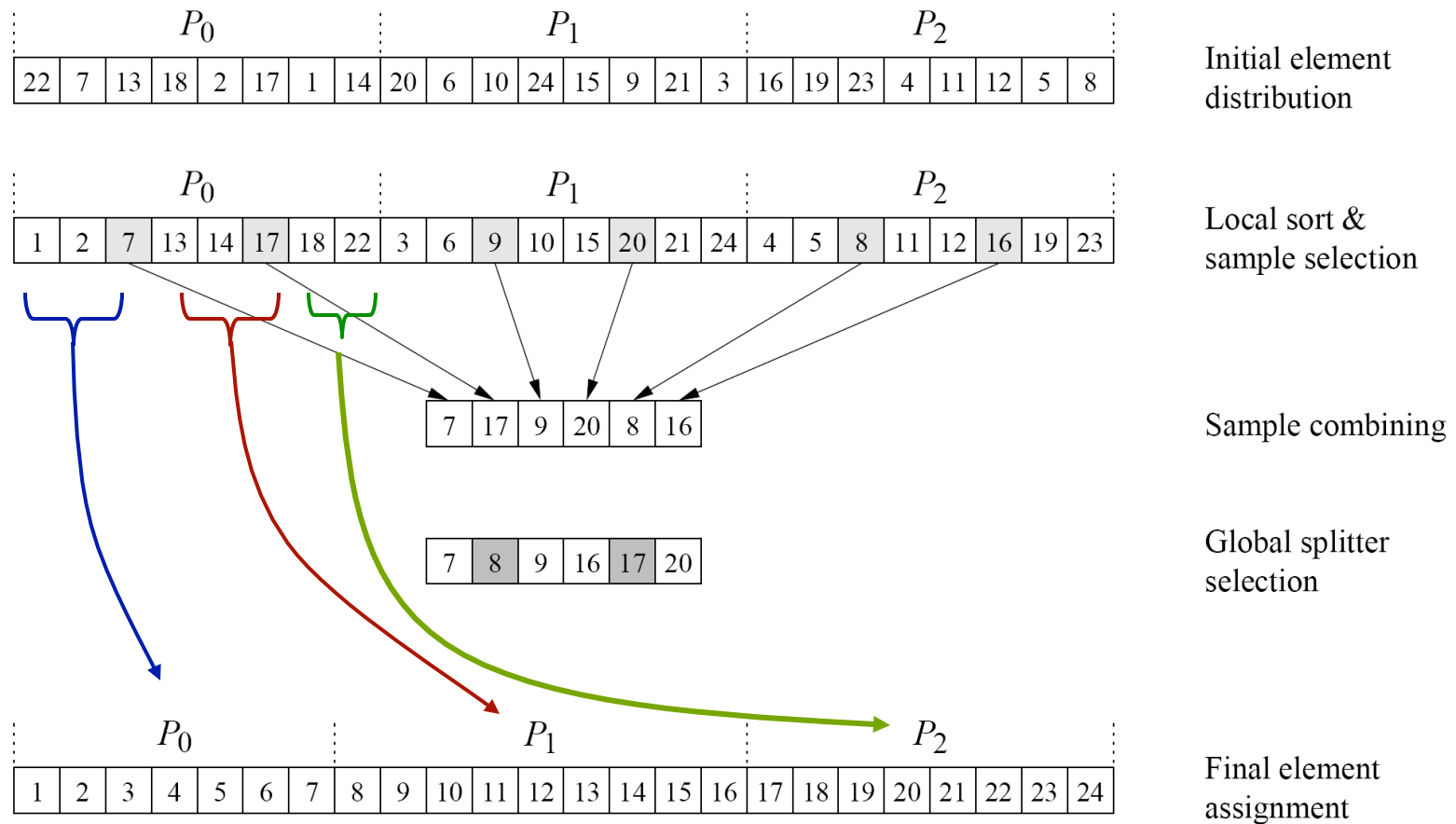
Today's lecture

- All to all communication
- **Application to Parallel Sorting**
- Blocking for cache

Recall sample sort

- Uses a heuristic to estimate the distribution of the global key range over the p threads
- Each processor gets about the same number of keys
- Sample the keys to determine a set of $p-1$ *splitters* that partition the key space into p disjoint regions (buckets)

Alltoallv used in sample sort



Introduction to Parallel Computing, 2nd Ed., A.Grama, A.I Gupta, G. Karypis, and V. Kumar, Addison-Wesley, 2003.

The collective calls

- Processes transmit varying amounts of information to the other processes
- This is an `MPI_Alltoallv`
(`SKeys`, `send_counts`, `send_displace`, `MPI_INT`,
`RKeys`, `recv_counts`, `recv_displace`, `MPI_INT`,
`MPI_COMM_WORLD`)
- Prior to making this call, all processes must cooperate to determine how much information they will exchange
 - ▶ The *send list* describes the number of keys to send to each process *k*, and the offset in the local array
 - ▶ The *receive list* describes the number of incoming keys for each process *k* and the offset into the local array

Determining send & receive lists

- After sorting, each process scans its local keys from left to right, marking where the splitters divide the keys, in terms of **send counts**
- Perform an all to all to transpose these send counts into **receive counts**

```
MPI_Alltoall(send_counts, 1, MPI_INT,  
             recv_counts, 1, MPI_INT, MPI_COMM_WORLD)
```

- A simple loop determines the displacements

```
for (p=1; p < nodes; p++){  
    s_displ[p] = s_displ[p-1] + send_counts[p-1];  
    r_displ[p] = r_displ[p-1] + recv_counts[p-1];  
}
```

Today's lecture

- All to all communication
- Application to Parallel Sorting
- **Blocking for cache – matrix multiplication**

Matrix Multiplication

- Given two *conforming* matrices A and B , form the matrix product $A \times B$
 A is $m \times n$
 B is $n \times p$
- Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
- Discussion follows from Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html

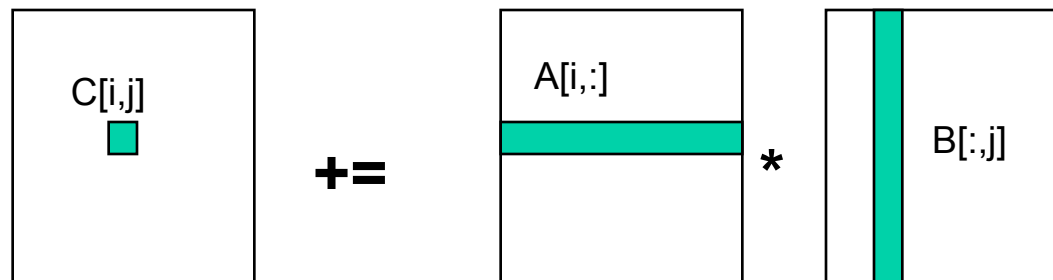
Unblocked Matrix Multiplication

for $i := 0$ **to** $n-1$

for $j := 0$ **to** $n-1$

for $k := 0$ **to** $n-1$

$$C[i,j] += A[i,k] * B[k,j]$$



Analysis of performance

for $i = 0$ to $n-1$

// for each iteration i , load all of B into cache

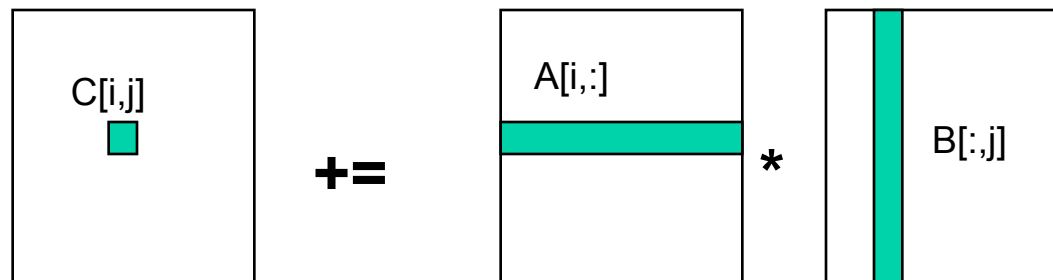
for $j = 0$ to $n-1$

// for each iteration (i,j) , load $A[i,:]$ into cache

// for each iteration (i,j) , load and store $C[i,j]$

for $k = 0$ to $n-1$

$C[i,j] += A[i,k] * B[k,j]$



Analysis of performance

for $i = 0$ to $n-1$

// $n \times n^2 / L$ loads = n^3/L , L =cache line size $B[:,:]$

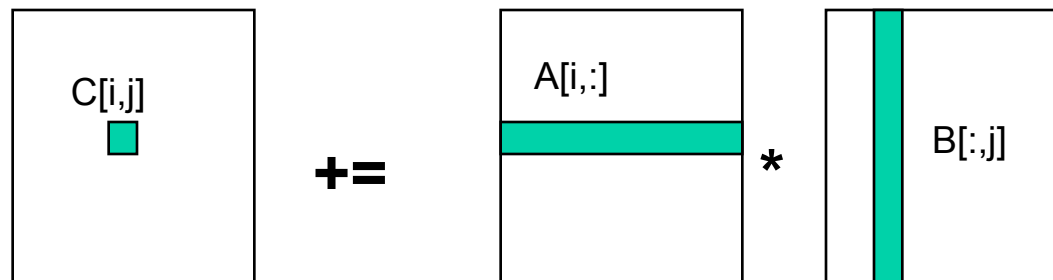
for $j = 0$ to $n-1$

// n^2 / L loads = n^2/L $A[i,:]$

// n^2 / L loads + n^2 / L stores = $2n^2 / L$ $C[i,j]$

for $k = 0$ to $n-1$

$C[i,j] += A[i,k] * B[k,j]$ **Total: $(n^3 + 3n^2) / L$**



Flops to memory ratio

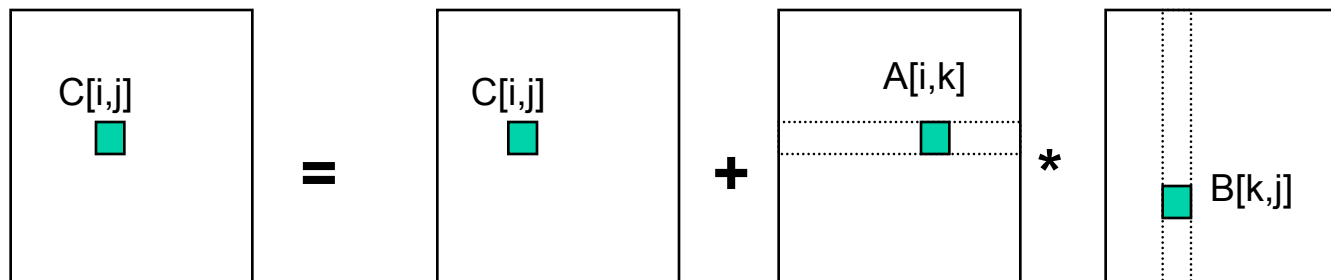
Let $q = \# \text{ flops} / \text{main memory reference}$

$$q = \frac{2n^3}{n^3 + 3n^2}$$

$$\approx 2 \text{ as } n \rightarrow \infty$$

Blocked Matrix Multiply

- Divide A, B, C into $N \times N$ sub blocks
- Assume we have a good quality library to perform matrix multiplication on subblocks
- Each sub block is $b \times b$
 - $b=n/N$ is called the **block size**
 - How do we establish b ?



Blocked Matrix Multiplication

for i = 0 to N-1

for j = 0 to N-1

// load each block C[i,j] into cache, once : n^2

// b = n/N = block size

for k = 0 to N-1

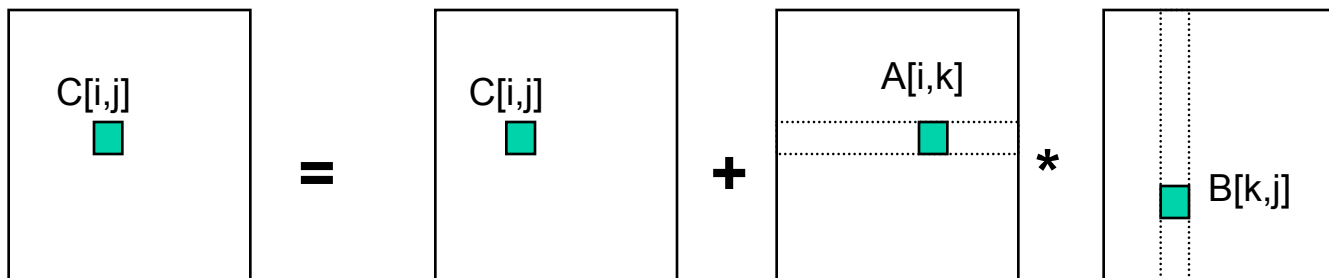
// load each block A[i,k] and B[k,j] N^3 times

// = $2N^3 \times (n/N)^2 = 2Nn^2$

C[i,j] += A[i,k] * B[k,j] // do the matrix multiply

// write each block C[i,j] once : n^2

Total: $(2*N+2)*n^2$



The results

N,B	Unblocked Time	Blocked Time
256, 64	0.6	0.002
512,128	15	0.24

Amortize memory accesses by
increasing memory reuse

Flops to memory ratio

- Since data motion has become increasingly expensive, the ratio of floating point work to data motion is a factor in determining performance
- Let $q = \# \text{ flops} / \text{main memory reference}$

$$q = \frac{2n^3}{(2N+2)n^2} = \frac{n}{N+1}$$

$$\approx n/N = \mathbf{b}$$

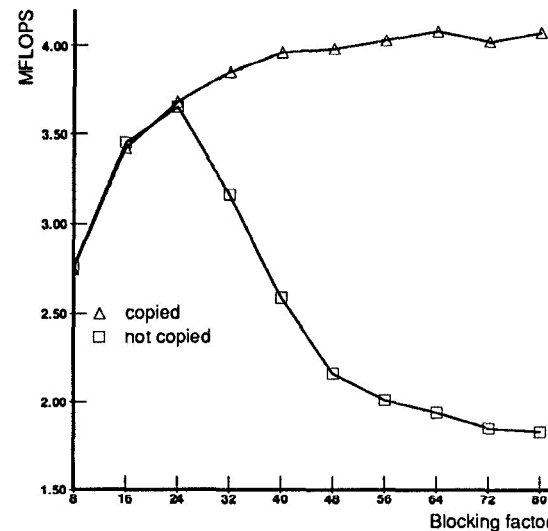
as $n \rightarrow \infty$

More on blocked algorithms

- Data in the sub-blocks are contiguous within rows only
- We may incur conflict cache misses
- Idea: since re-use is so high... let's copy the subblocks into contiguous memory before passing to our matrix multiply routine

“The Cache Performance and Optimizations of Blocked Algorithms,”
M. Lam et al., *ASPLOS IV*, 1991

<http://www-suif.stanford.edu/papers/lam91.ps>



Revisiting broadcast
(last lecture but not posted in initial
version of the slides)

Revisiting Broadcast

- P may not be a power of 2
- MPI-CH uses a binomial tree algorithm for short messages (We can use the hypercube algorithm to illustrate the special case of $P=2^k$)
- We use a different algorithm for long messages

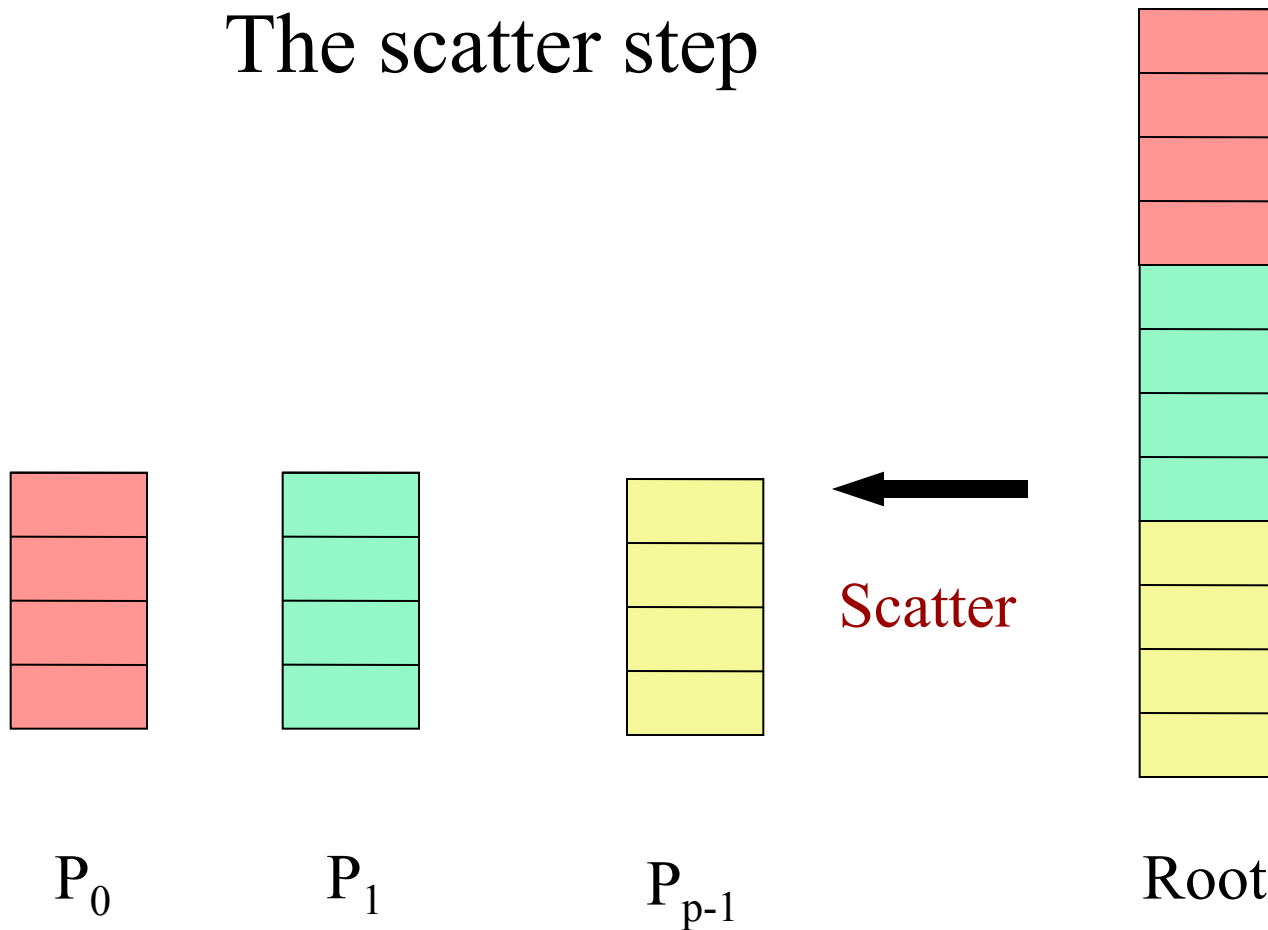
Strategy for long messages

- Based van de Geijn's strategy
- Scatter the data
 - Divide the data to be broadcast into pieces, and fill the machine with the pieces
- Do an Allgather
 - Now that everyone has a part of the entire result, collect on all processors
- Faster than MST algorithm for long messages

$$2 \frac{p-1}{p} n\beta \ll \lceil \lg p \rceil n\beta$$

Algorithm for long messages

The scatter step



Algorithm for long messages

