

Parallel Programming in C with MPI and OpenMP

Michael J. Quinn



Chapter 8

Matrix-vector Multiplication

Chapter Objectives

- Review matrix-vector multiplication
- Propose replication of vectors
- Develop three parallel programs, each based on a different data decomposition

Outline

- Sequential algorithm and its complexity
- Design, analysis, and implementation of three parallel programs
 - ◆ Rowwise block striped
 - ◆ Columnwise block striped
 - ◆ Checkerboard block

Sequential Algorithm

$$\begin{array}{c} \begin{array}{|c|c|c|c|} \hline 2 & 1 & 0 & 4 \\ \hline 3 & 2 & 1 & 1 \\ \hline 4 & 3 & 1 & 2 \\ \hline 3 & 0 & 2 & 0 \\ \hline \end{array} & \times & \begin{array}{|c|} \hline 1 \\ \hline 3 \\ \hline 4 \\ \hline 1 \\ \hline \end{array} & = & \begin{array}{|c|} \hline 9 \\ \hline 14 \\ \hline 19 \\ \hline 11 \\ \hline \end{array} \end{array}$$

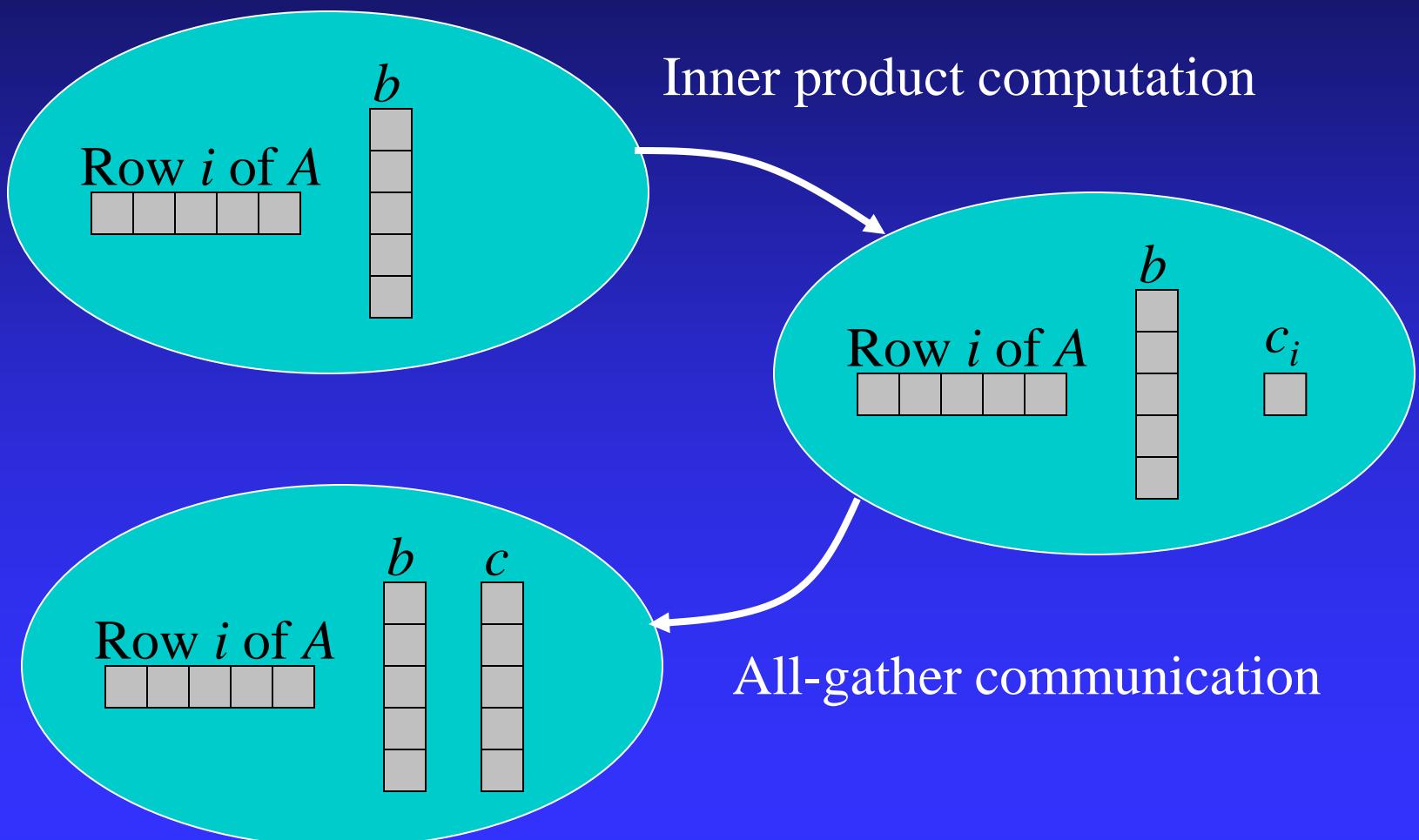
Storing Vectors

- Divide vector elements among processes
- Replicate vector elements
- Vector replication acceptable because vectors have only n elements, versus n^2 elements in matrices

Rowwise Block Striped Matrix

- Partitioning through domain decomposition
- Primitive task associated with
 - ◆ Row of matrix
 - ◆ Entire vector

Phases of Parallel Algorithm



Agglomeration and Mapping

- Static number of tasks
- Regular communication pattern (all-gather)
- Computation time per task is constant
- Strategy:
 - ◆ Agglomerate groups of rows
 - ◆ Create one task per MPI process

Complexity Analysis

- Sequential algorithm complexity: $\Theta(n^2)$
- Parallel algorithm computational complexity: $\Theta(n^2/p)$
- Communication complexity of all-gather: $\Theta(\log p + n)$
- Overall complexity: $\Theta(n^2/p + \log p)$

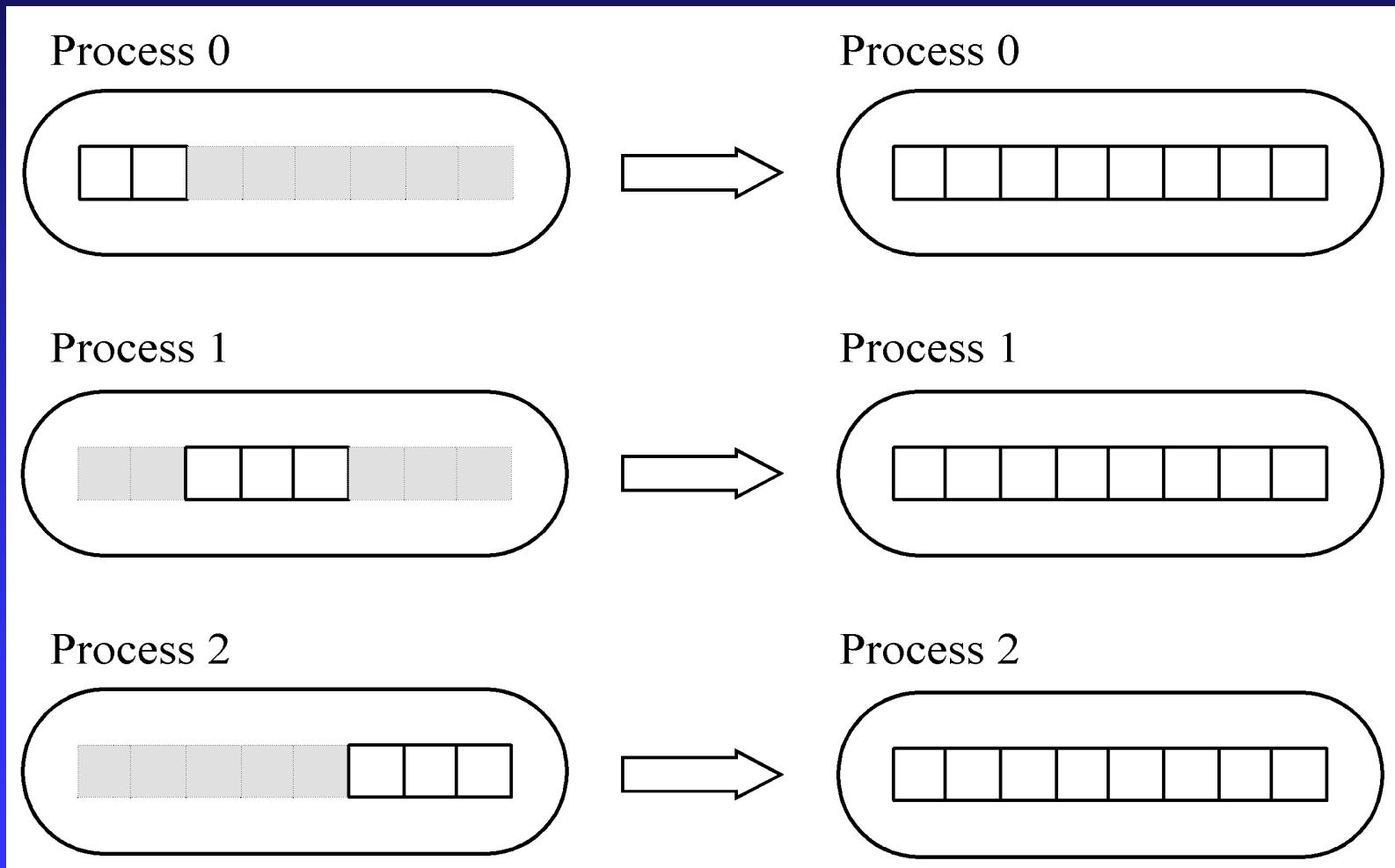
Isoefficiency Analysis

- Sequential time complexity: $\Theta(n^2)$
- Only parallel overhead is all-gather
 - ◆ When n is large, message transmission time dominates message latency
 - ◆ Parallel communication time: $\Theta(n)$
- $n^2 \geq Cpn \Rightarrow n \geq Cp$ and $M(n) = n^2$

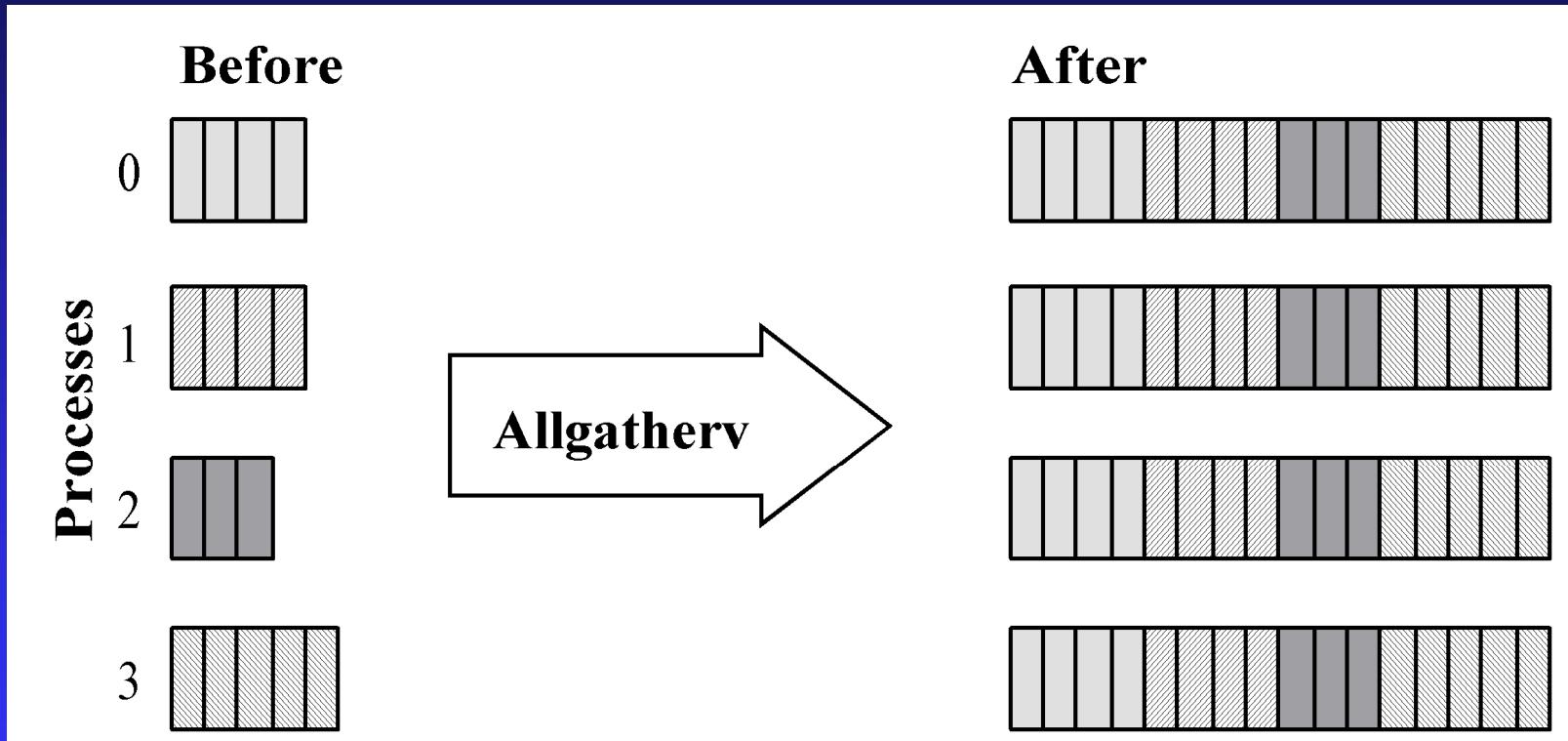
$$M(Cp)/p = C^2 p^2 / p = C^2 p$$

- System is not highly scalable

Block-to-replicated Transformation



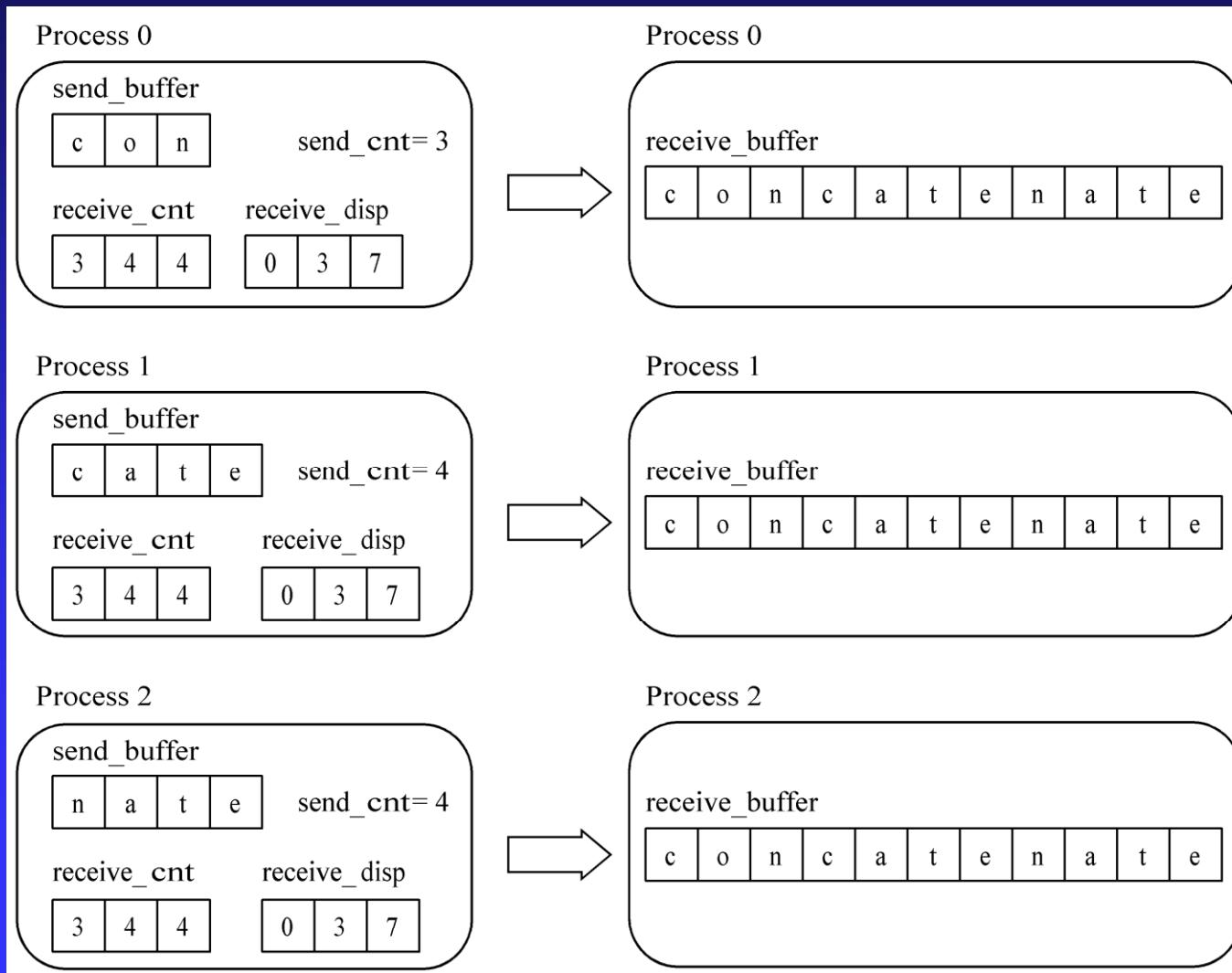
MPI_Allgatherv



MPI_Allgatherv

```
int MPI_Allgatherv (
    void *send_buffer,
    int send_cnt,
    MPI_Datatype send_type,
    void *receive_buffer,
    *receive_cnt,
    *receive_disp,
    MPI_Datatype receive_type,
    MPI_Comm communicator)
```

MPI_Allgatherv in Action



Function create_mixed_xfer_arrays

- First array
 - ◆ How many elements contributed by each process
 - ◆ Uses utility macro BLOCK_SIZE
- Second array
 - ◆ Starting position of each process' block
 - ◆ Assume blocks in process rank order

Function replicate_block_vector

- Create space for entire vector
- Create “mixed transfer” arrays
- Call **MPI_Allgatherv**

Function `read_replicated_vector`

- Process $p-1$
 - ◆ Opens file
 - ◆ Reads vector length
- Broadcast vector length (root process = $p-1$)
- Allocate space for vector
- Process $p-1$ reads vector, closes file
- Broadcast vector

Function `print_replicated_vector`

- Process 0 prints vector
- Exact call to `printf` depends on value of parameter `datatype`

Run-time Expression

- χ : inner product loop iteration time
- Computational time: $\chi n \lceil n/p \rceil$
- All-gather requires $\lceil \log p \rceil$ messages with latency λ
- Total vector elements transmitted:
$$(2^{\lceil \log p \rceil} - 1) / 2^{\lceil \log p \rceil}$$
- Total execution time: $\chi n \lceil n/p \rceil + \lambda \lceil \log p \rceil + (2^{\lceil \log p \rceil} - 1) / (2^{\lceil \log p \rceil} \beta)$

Benchmarking Results

		<i>Execution Time (msec)</i>			
<i>p</i>	<i>Predicted</i>	<i>Actual</i>	<i>Speedup</i>	<i>Mflops</i>	
1	63.4	63.4	1.00	31.6	
2	32.4	32.7	1.94	61.2	
3	22.3	22.7	2.79	88.1	
4	17.0	17.8	3.56	112.4	
5	14.1	15.2	4.16	131.6	
6	12.0	13.3	4.76	150.4	
7	10.5	12.2	5.19	163.9	
8	9.4	11.1	5.70	180.2	
16	5.7	7.2	8.79	277.8	

Columnwise Block Striped Matrix

- Partitioning through domain decomposition
- Task associated with
 - ◆ Column of matrix
 - ◆ Vector element

Matrix-Vector Multiplication

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

Processor 0's initial computation

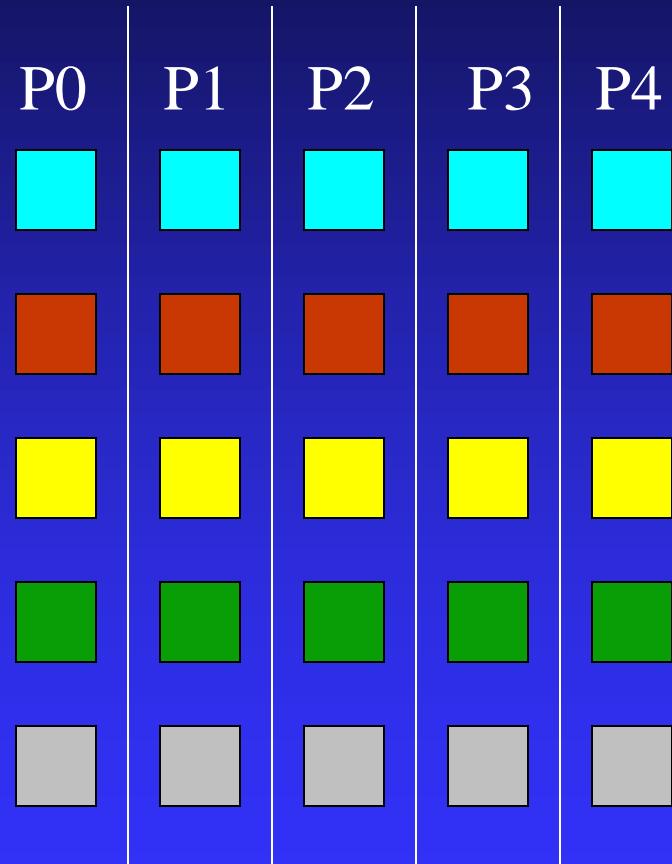
Processor 1's initial computation

Proc 2

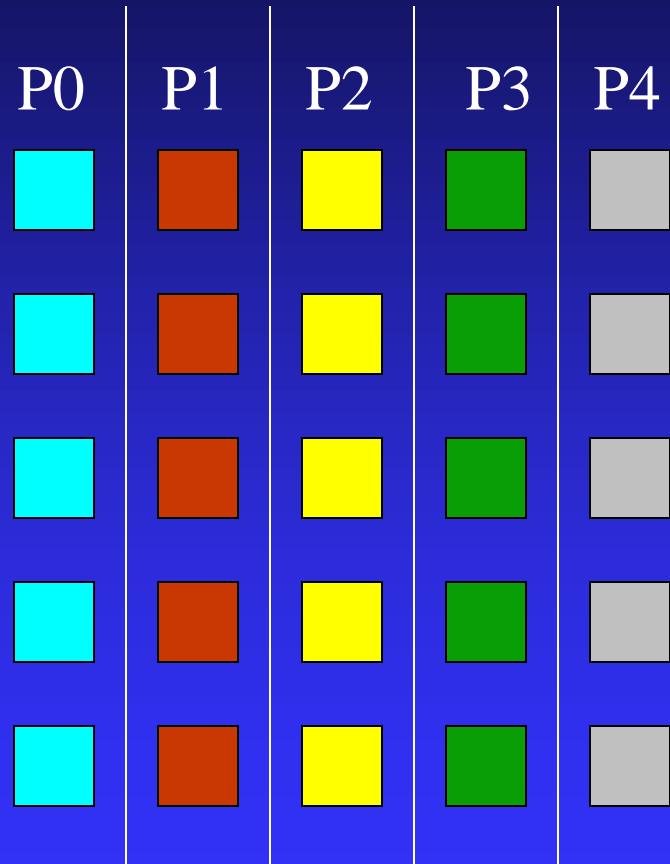
Proc 3

Proc 4

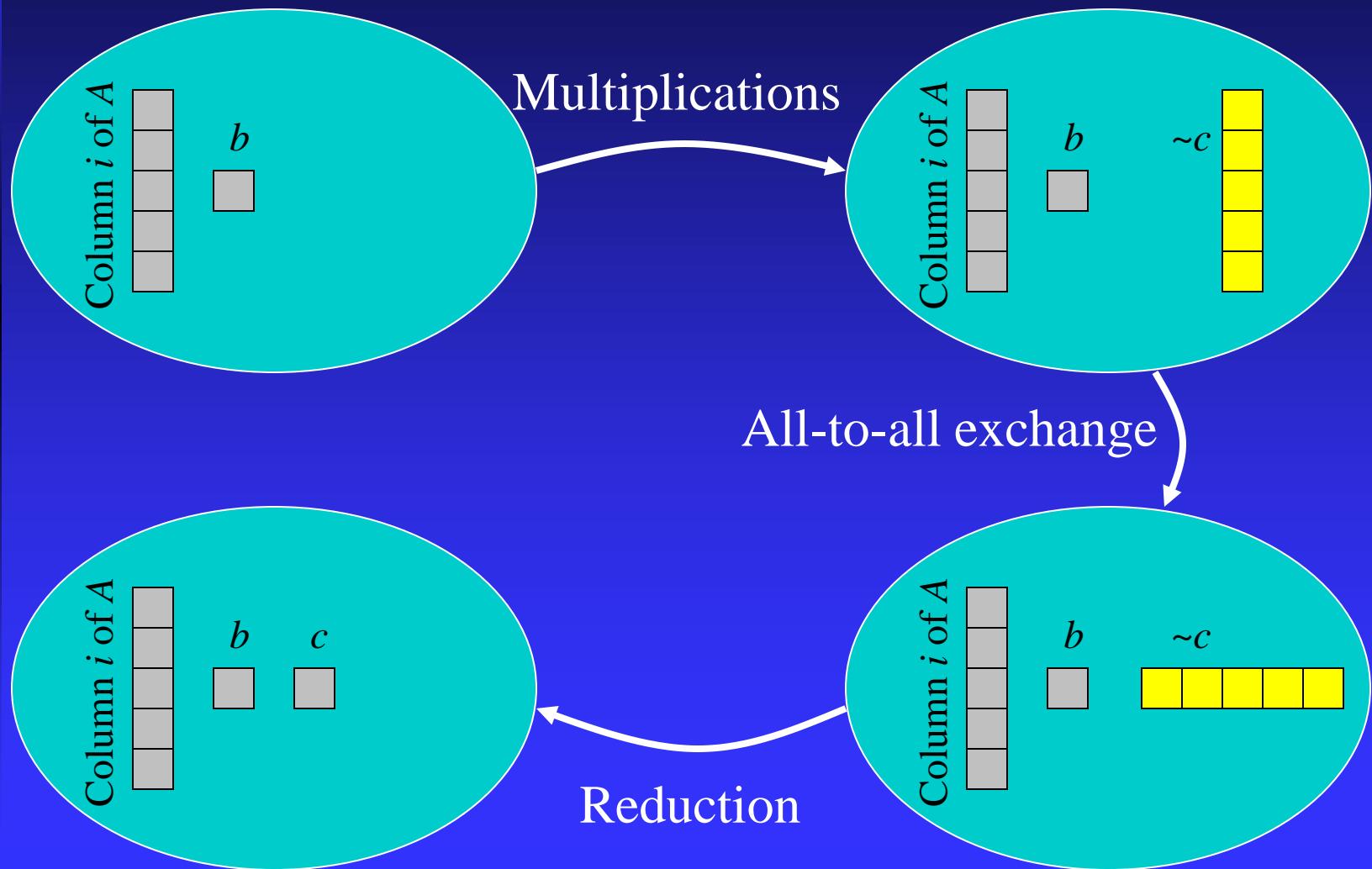
All-to-all Exchange (Before)



All-to-all Exchange (After)



Phases of Parallel Algorithm



Agglomeration and Mapping

- Static number of tasks
- Regular communication pattern (all-to-all)
- Computation time per task is constant
- Strategy:
 - ◆ Agglomerate groups of columns
 - ◆ Create one task per MPI process

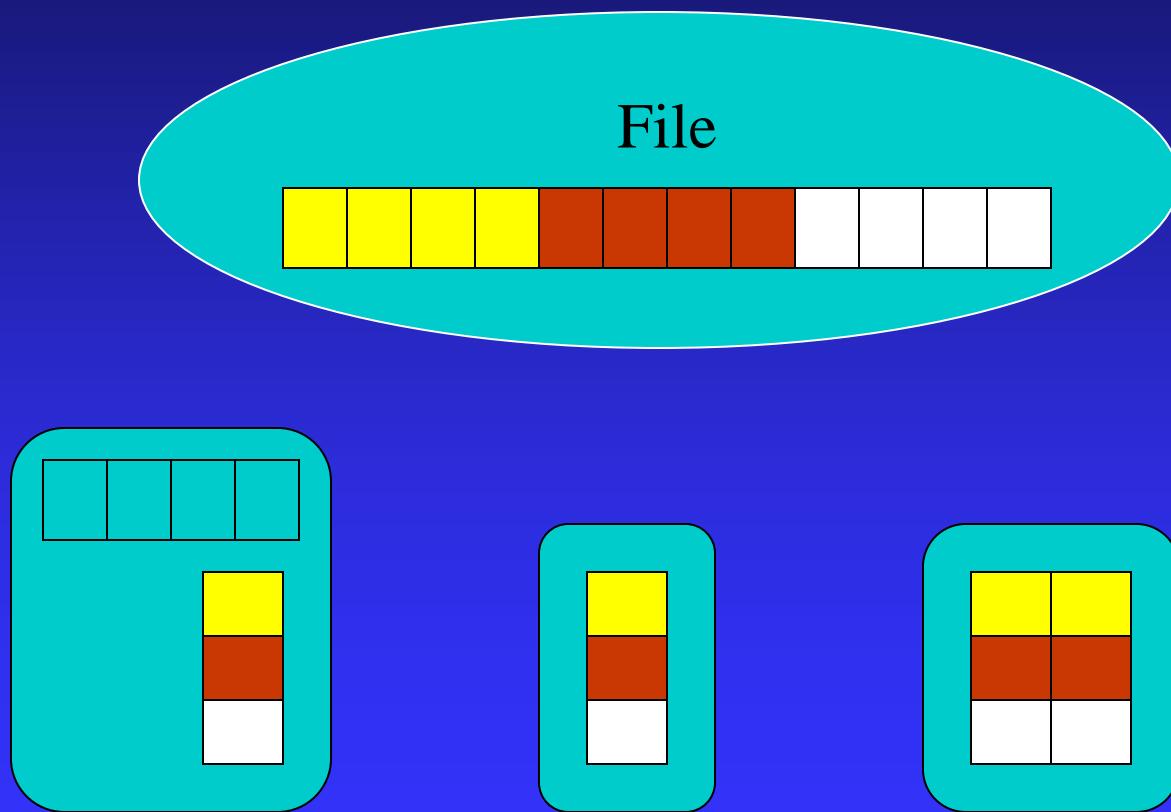
Complexity Analysis

- Sequential algorithm complexity: $\Theta(n^2)$
- Parallel algorithm computational complexity: $\Theta(n^2/p)$
- Communication complexity of all-to-all: $\Theta(p + n/p)$
- Overall complexity: $\Theta(n^2/p + \log p)$

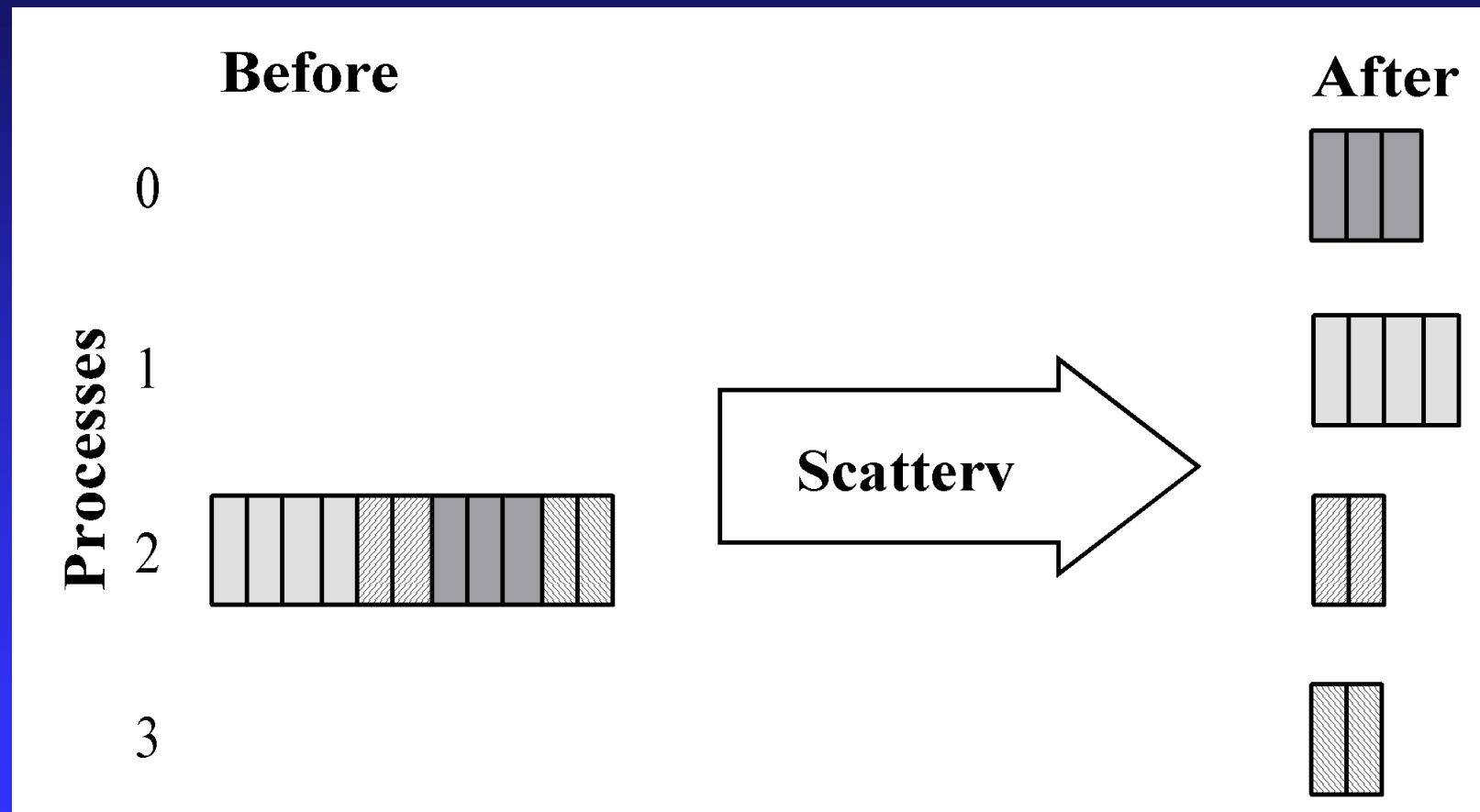
Isoefficiency Analysis

- Sequential time complexity: $\Theta(n^2)$
- Only parallel overhead is all-to-all
 - ◆ When n is large, message transmission time dominates message latency
 - ◆ Parallel communication time: $\Theta(n)$
- $n^2 \geq Cpn \Rightarrow n \geq Cp$
- Scalability function same as rowwise algorithm: C^2p

Reading a Block-Column Matrix



MPI_Scatterv



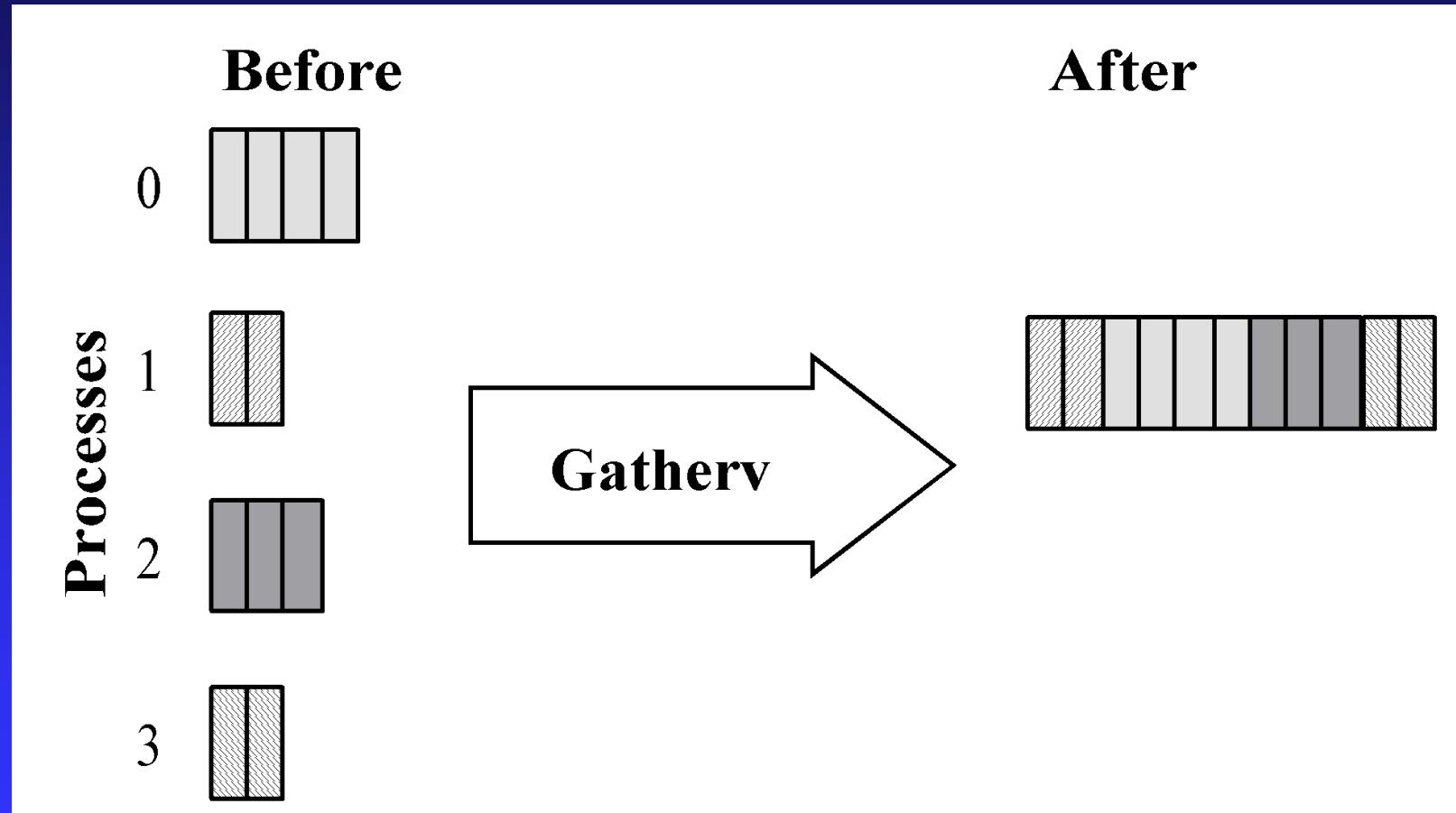
Header for MPI_Scatterv

```
int MPI_Scatterv (
    void           *send_buffer,
    int            *send_cnt,
    int            *send_disp,
    MPI_Datatype   send_type,
    void           *receive_buffer,
    int             receive_cnt,
    MPI_Datatype   receive_type,
    int             root,
    MPI_Comm       communicator)
```

Printing a Block-Column Matrix

- Data motion opposite to that we did when reading the matrix
- Replace “scatter” with “gather”
- Use “v” variant because different processes contribute different numbers of elements

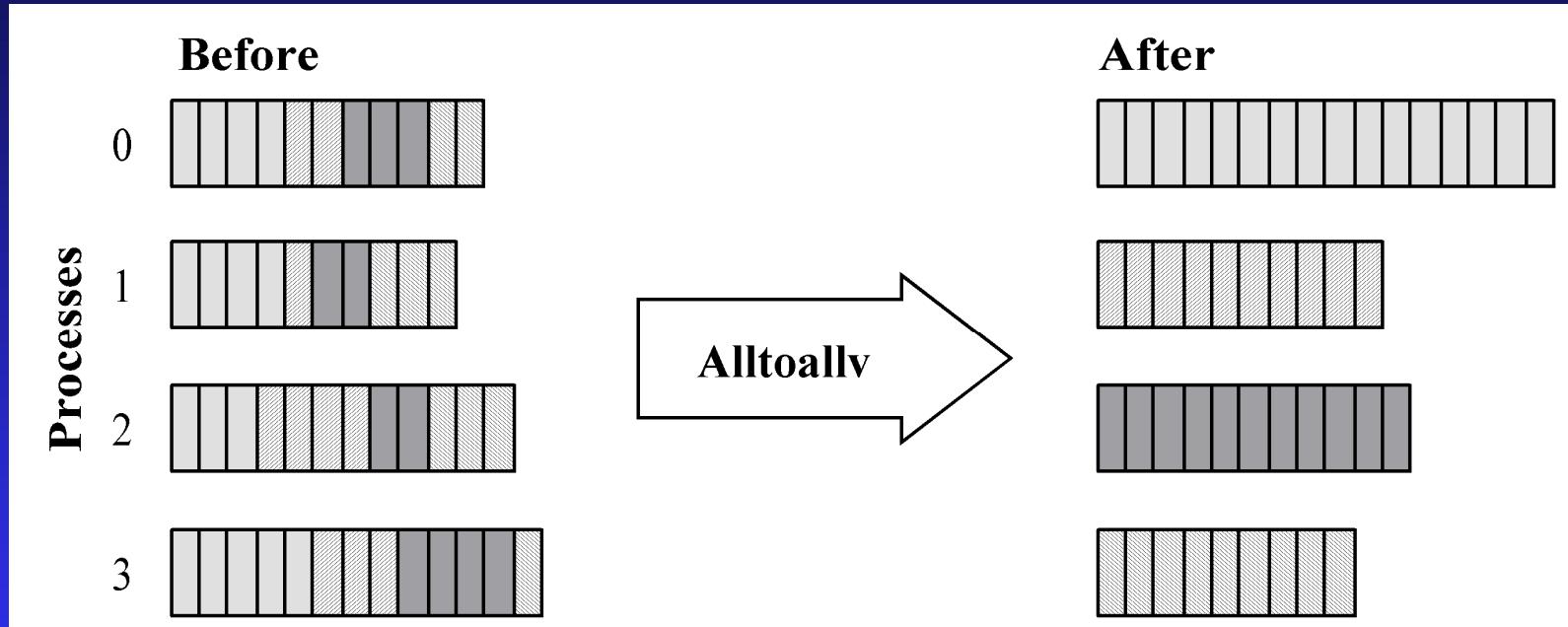
Function MPI_Gatherv



Header for MPI_Gatherv

```
int MPI_Gatherv (  
    void *send_buffer,  
    int send_cnt,  
    MPI_Datatype send_type,  
    void *receive_buffer,  
    int receive_cnt,  
    MPI_Datatype receive_type,  
    int root,  
    MPI_Comm communicator)
```

Function MPI_Alltoallv



Header for MPI_Alltoallv

```
int MPI_Gatherv (  
    void *send_buffer,  
    int send_cnt,  
    int send_disp,  
    MPI_Datatype send_type,  
    void *receive_buffer,  
    int receive_cnt,  
    int receive_disp,  
    MPI_Datatype receive_type,  
    MPI_Comm communicator)
```

Count/Displacement Arrays

- MPI_Alltoallv requires `create_mixed_xfer_arrays` to build these count/displacement arrays
- First pair for values being sent
 - ◆ `send_cnt`: number of elements
 - ◆ `send_disp`: index of first element
- Second pair for values being received
 - ◆ `recv_cnt`: number of elements
 - ◆ `recv_disp`: index of first element

Function create_uniform_xfer_arrays

- First array
 - ◆ How many elements received from each process (always same value)
 - ◆ Uses ID and utility macro `block_size`
- Second array
 - ◆ Starting position of each process' block
 - ◆ Assume blocks in process rank order

Run-time Expression

- χ : inner product loop iteration time
- Computational time: $\chi n \lceil n/p \rceil$
- All-gather requires $p-1$ messages, each of length about n/p
- 8 bytes per element
- Total execution time:
$$\chi n \lceil n/p \rceil + (p-1)(\lambda + (8n/p)/\beta)$$

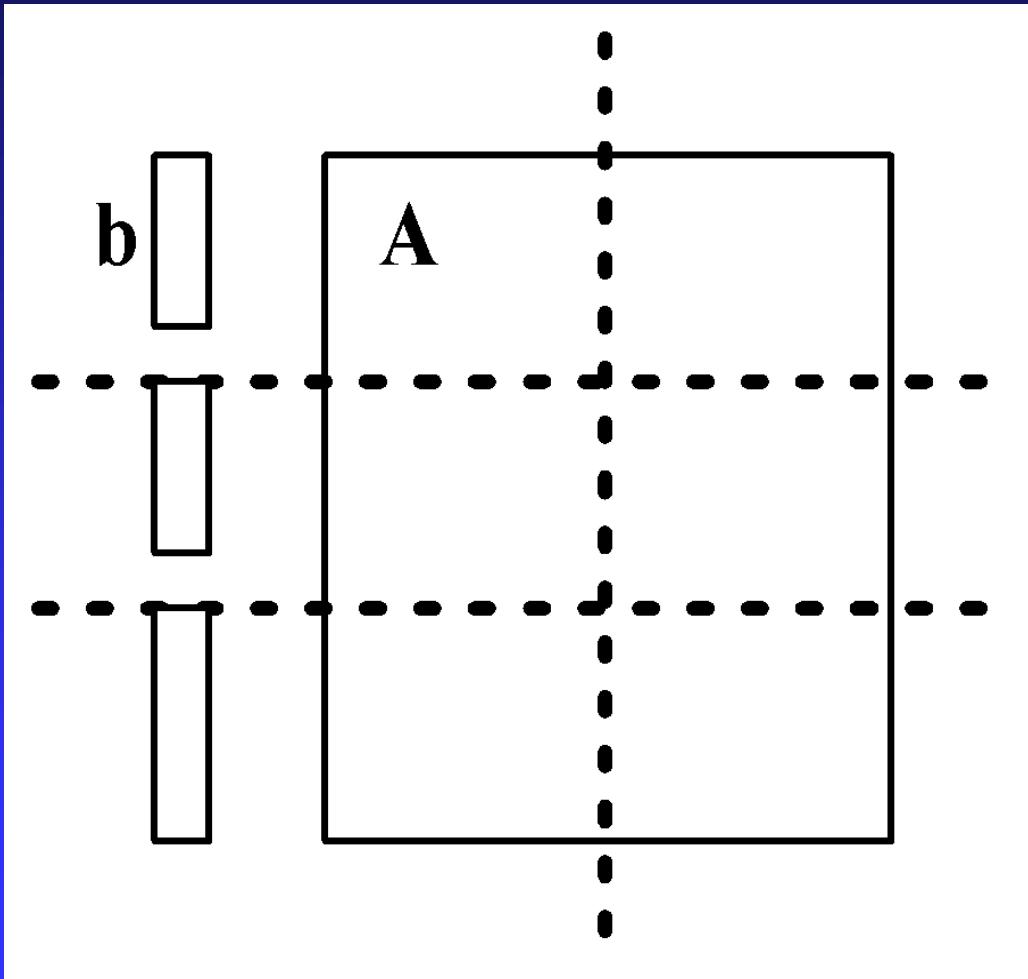
Benchmarking Results

p	<i>Execution Time (msec)</i>		<i>Speedup</i>	<i>Mflops</i>
	<i>Predicted</i>	<i>Actual</i>		
1	63.4	63.8	1.00	31.4
2	32.4	32.9	1.92	60.8
3	22.2	22.6	2.80	88.5
4	17.2	17.5	3.62	114.3
5	14.3	14.5	4.37	137.9
6	12.5	12.6	5.02	158.7
7	11.3	11.2	5.65	178.6
8	10.4	10.0	6.33	200.0
16	8.5	7.6	8.33	263.2

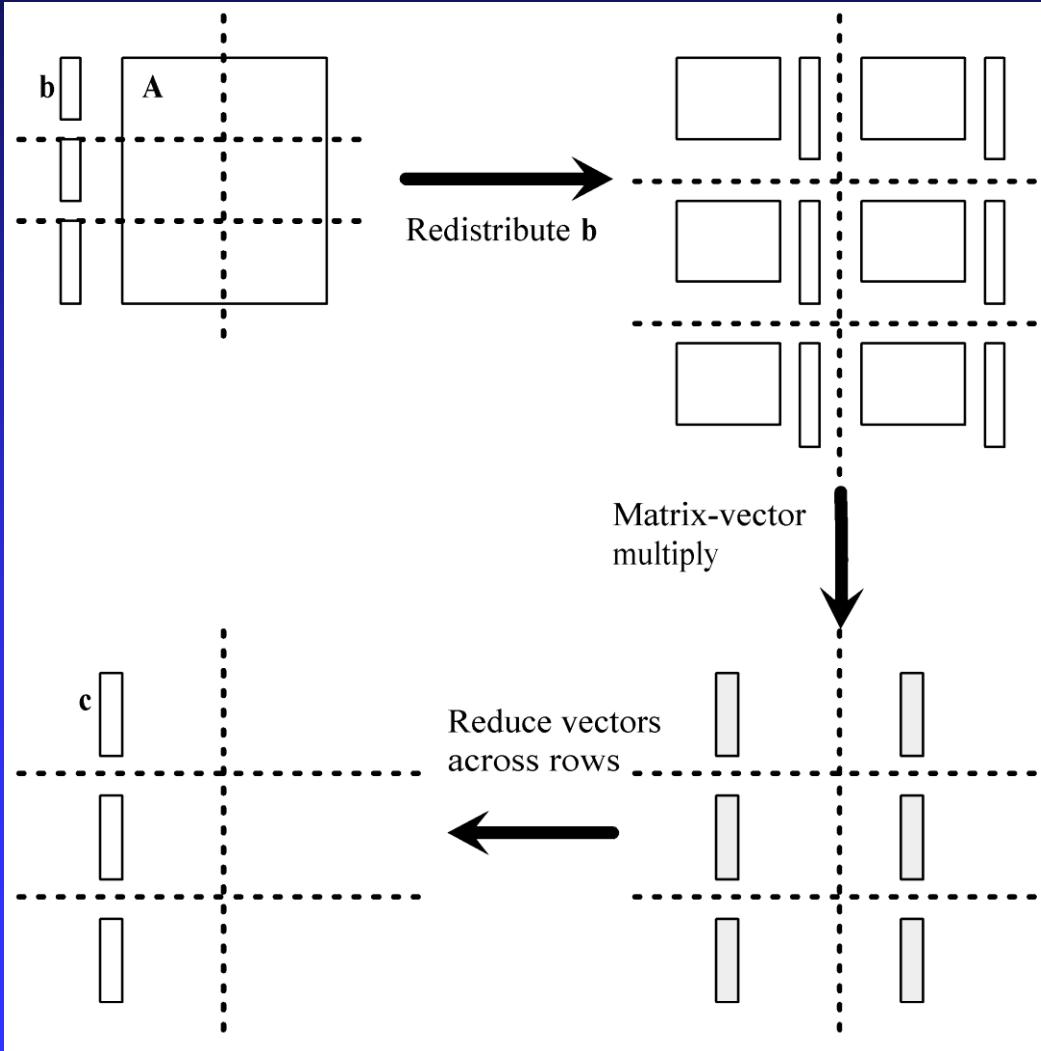
Checkerboard Block Decomposition

- Associate primitive task with each element of the matrix \mathbf{A}
- Each primitive task performs one multiply
- Agglomerate primitive tasks into rectangular blocks
- Processes form a 2-D grid
- Vector \mathbf{b} distributed by blocks among processes in first column of grid

Tasks after Agglomeration



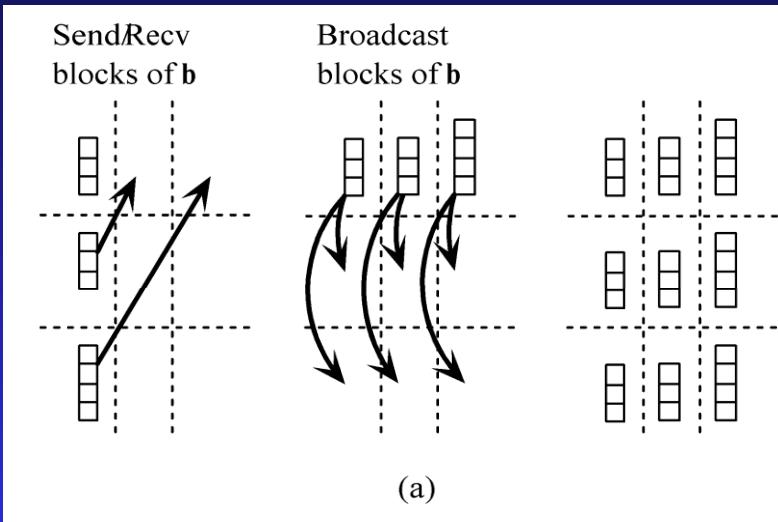
Algorithm's Phases



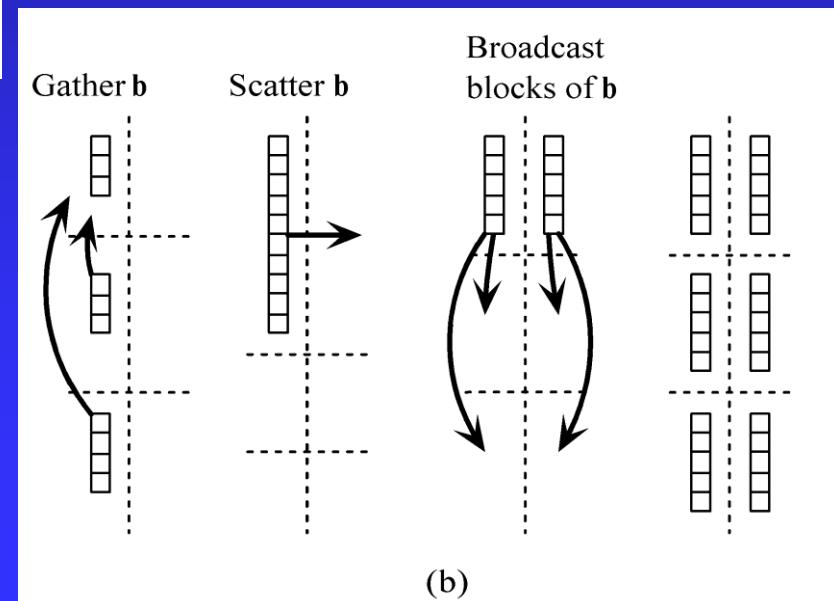
Redisistributing Vector **b**

- Step 1: Move **b** from processes in first row to processes in first column
 - ◆ 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 procs
- Step 2: First row processes scatter **b** within columns

Redisistributing Vector \mathbf{b}



When p is a square number



When p is not a square number

Complexity Analysis

- Assume p is a square number
 - ◆ If grid is $1 \times p$, devolves into columnwise block striped
 - ◆ If grid is $p \times 1$, devolves into rowwise block striped

Complexity Analysis (continued)

- Each process does its share of computation:
 $\Theta(n^2/p)$
- Redistribute **b**: $\Theta(n / \sqrt{p} + \log p(n / \sqrt{p})) =$
 $\Theta(n \log p / \sqrt{p})$
- Reduction of partial results vectors:
 $\Theta(n \log p / \sqrt{p})$
- Overall parallel complexity:
 $\Theta(n^3/p + n \log p / \sqrt{p})$

Isoefficiency Analysis

- Sequential complexity: $\Theta(n^2)$
- Parallel communication complexity:
 $\Theta(n \log p / \sqrt{p})$
- Isoefficiency function:
 $n^2 \geq Cn \sqrt{p} \log p \Rightarrow n \geq C \sqrt{p} \log p$

$$M(C\sqrt{p} \log p) / p = C^2 p \log^2 p / p = C^2 \log^2 p$$

- This system is much more scalable than the previous two implementations

Creating Communicators

- Want processes in a virtual 2-D grid
- Create a custom communicator to do this
- Collective communications involve all processes in a communicator
- We need to do broadcasts, reductions among subsets of processes
- We will create communicators for processes in same row or same column

What's in a Communicator?

- Process group
- Context
- Attributes
 - ◆ Topology (lets us address processes another way)
 - ◆ Others we won't consider

Creating 2-D Virtual Grid of Processes

- **MPI_Dims_create**
 - ◆ Input parameters
 - ◆ Total number of processes in desired grid
 - ◆ Number of grid dimensions
 - ◆ Returns number of processes in each dim
- **MPI_Cart_create**
 - ◆ Creates communicator with cartesian topology

MPI_Dims_create

```
int MPI_Dims_create (
    int nodes,
        /* Input - Procs in grid */

    int dims,
        /* Input - Number of dims */

    int *size)
        /* Input/Output - Size of
           each grid dimension */
```

MPI_Cart_create

```
int MPI_Cart_create (
    MPI_Comm old_comm, /* Input - old communicator */

    int dims, /* Input - grid dimensions */

    int *size, /* Input - # procs in each dim */

    int *periodic,
        /* Input - periodic[j] is 1 if dimension j
           wraps around; 0 otherwise */

    int reorder,
        /* 1 if process ranks can be reordered */

    MPI_Comm *cart_comm)
    /* Output - new communicator */
```

Using MPI_Dims_create and MPI_Cart_create

```
MPI_Comm cart_comm;
int p;
int periodic[2];
int size[2];
...
size[0] = size[1] = 0;
MPI_Dims_create (p, 2, size);
periodic[0] = periodic[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, size,
                    1, &cart_comm);
```

Useful Grid-related Functions

■ MPI_Cart_rank

- ◆ Given coordinates of process in Cartesian communicator, returns process rank

■ MPI_Cart_coords

- ◆ Given rank of process in Cartesian communicator, returns process' coordinates

Header for MPI_Cart_rank

```
int MPI_Cart_rank (
    MPI_Comm comm,
    /* In - Communicator */
    int *coords,
    /* In - Array containing process'
       grid location */
    int *rank)
    /* Out - Rank of process at
       specified coords */
```

Header for MPI_Cart_coords

```
int MPI_Cart_coords (
    MPI_Comm comm,
    /* In - Communicator */
    int rank,
    /* In - Rank of process */
    int dims,
    /* In - Dimensions in virtual grid */
    int *coords)
    /* Out - Coordinates of specified
       process in virtual grid */
```

MPI_Comm_split

- Partitions the processes of a communicator into one or more subgroups
- Constructs a communicator for each subgroup
- Allows processes in each subgroup to perform their own collective communications
- Needed for columnwise scatter and rowwise reduce

Header for MPI_Comm_split

```
int MPI_Comm_split (
    MPI_Comm old_comm,
    /* In - Existing communicator */

    int partition, /* In - Partition number */

    int new_rank,
    /* In - Ranking order of processes
       in new communicator */

    MPI_Comm *new_comm)
    /* Out - New communicator shared by
       processes in same partition */
```

Example: Create Communicators for Process Rows

```
MPI_Comm grid_comm; /* 2-D process grid */

MPI_Comm grid_coords[2];
/* Location of process in grid */

MPI_Comm row_comm;
/* Processes in same row */

MPI_Comm_split (grid_comm, grid_coords[0],
grid_coords[1], &row_comm);
```

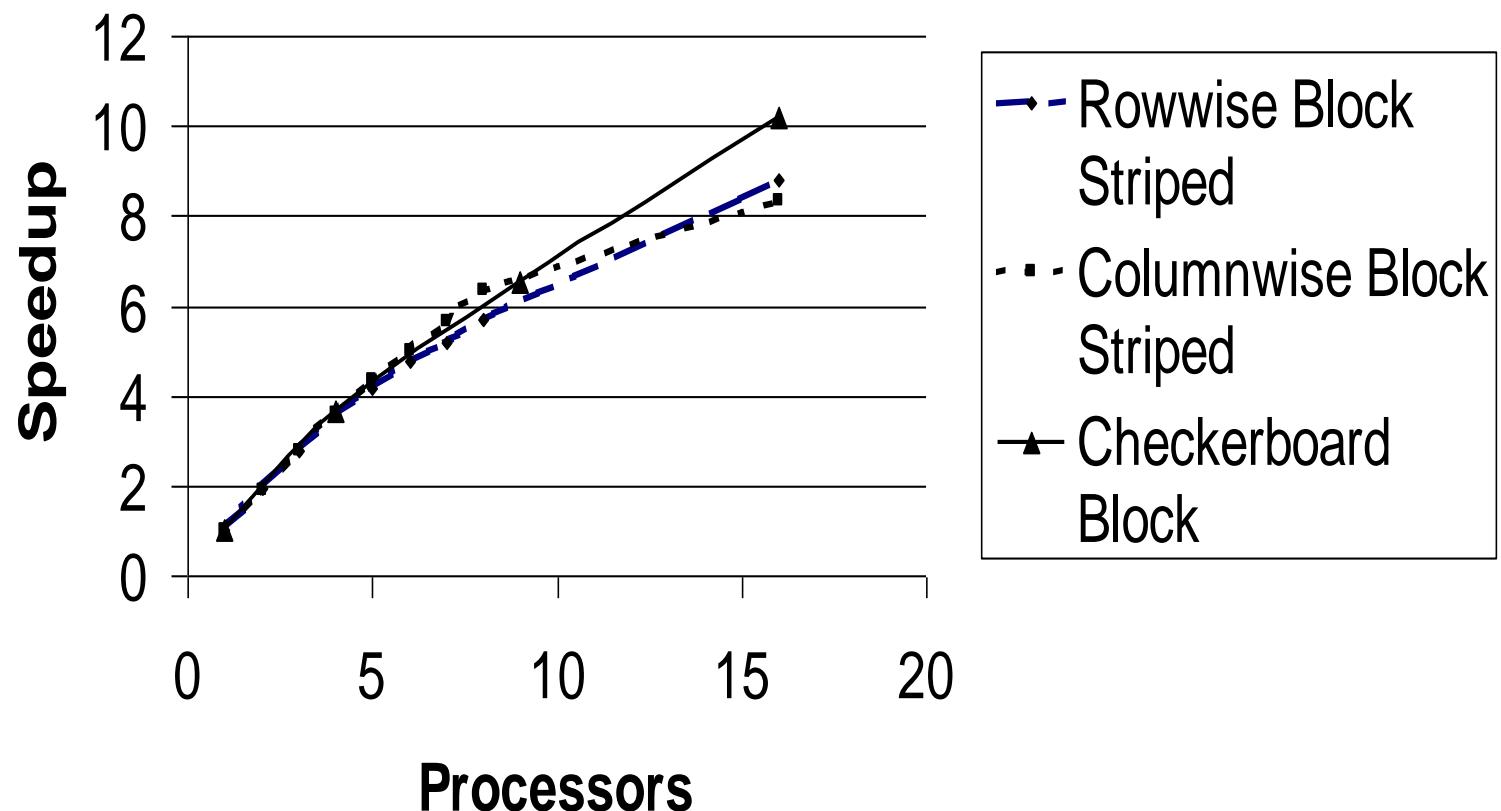
Run-time Expression

- Computational time: $\chi \lceil n/\sqrt{p} \rceil \lceil n/\sqrt{p} \rceil$
- Suppose p a square number
- Redistribute \mathbf{b}
 - ◆ Send/recv: $\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta$
 - ◆ Broadcast: $\log \sqrt{p} (\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta)$
- Reduce partial results:
 $\log \sqrt{p} (\lambda + 8 \lceil n/\sqrt{p} \rceil / \beta)$

Benchmarking

Procs	Predicted (msec)	Actual (msec)	Speedup	Megaflops
1	63.4	63.4	1.00	31.6
4	17.8	17.4	3.64	114.9
9	9.7	9.7	6.53	206.2
16	6.2	6.2	10.21	322.6

Comparison of Three Algorithms



Summary (1/3)

- Matrix decomposition \Rightarrow communications needed
 - ◆ Rowwise block striped: all-gather
 - ◆ Columnwise block striped: all-to-all exchange
 - ◆ Checkerboard block: gather, scatter, broadcast, reduce
- All three algorithms: roughly same number of messages
- Elements transmitted per process varies
 - ◆ First two algorithms: $\Theta(n)$ elements per process
 - ◆ Checkerboard algorithm: $\Theta(n/\sqrt{p})$ elements
- Checkerboard block algorithm has better scalability

Summary (2/3)

- Communicators with Cartesian topology
 - ◆ Creation
 - ◆ Identifying processes by rank or coords
- Subdividing communicators
 - ◆ Allows collective operations among subsets of processes

Summary (3/3)

- Parallel programs and supporting functions much longer than C counterparts
- Extra code devoted to reading, distributing, printing matrices and vectors
- Developing and debugging these functions is tedious and difficult
- Makes sense to generalize functions and put them in libraries for reuse

MPI Application Development

