Matrix-vector Multiplication

- 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 methods of distributing the data

  - Rowwise block striped – each process calculates one or more elements of the final vector

  - Columnwise block striped – each process calculates a partial sum contributing to different elements of the vector

  - Checkerboard block – the matrix is divided into blocks and each process calculates a partial sum contributing to different elements of the final vector

Sequential Algorithm

Each row of the matrix multiplies the corresponding element in the vector

a[m,n] x b[n] = c[n]

$$\begin{bmatrix} 2 & 1 & 0 & 4 \\ 3 & 2 & 1 & 1 \\ 4 & 3 & 1 & 2 \\ 3 & 0 & 2 & 0 \end{bmatrix} \times \begin{bmatrix} 1 \\ 3 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ 14 \\ 19 \\ 11 \end{bmatrix}$$

c[i] = a[i,0]*b[0] + a[i,1]*b[1] + … + a[i,n-1]*b[n-1];

Parallel Algorithm Development

Identify the parallel tasks -

1. Multiplying a matrix element and a vector element
2. Adding up the products in step 1 to calculate an element of the result vector

This is data parallelism, but have to decide how to assign the tasks to processors to reduce communication.

Storing the vector elements

Must decide how the vector elements are distributed among the processes:

1. Divide vector elements among processes or
2. Replicate vector elements

Vector replication acceptable because vectors have only n elements, versus n^2 elements in matrices

Storing the elements of the matrix with the rowwise algorithm

As in Floyd's algorithm, several rows of the matrix can be assigned to each process.

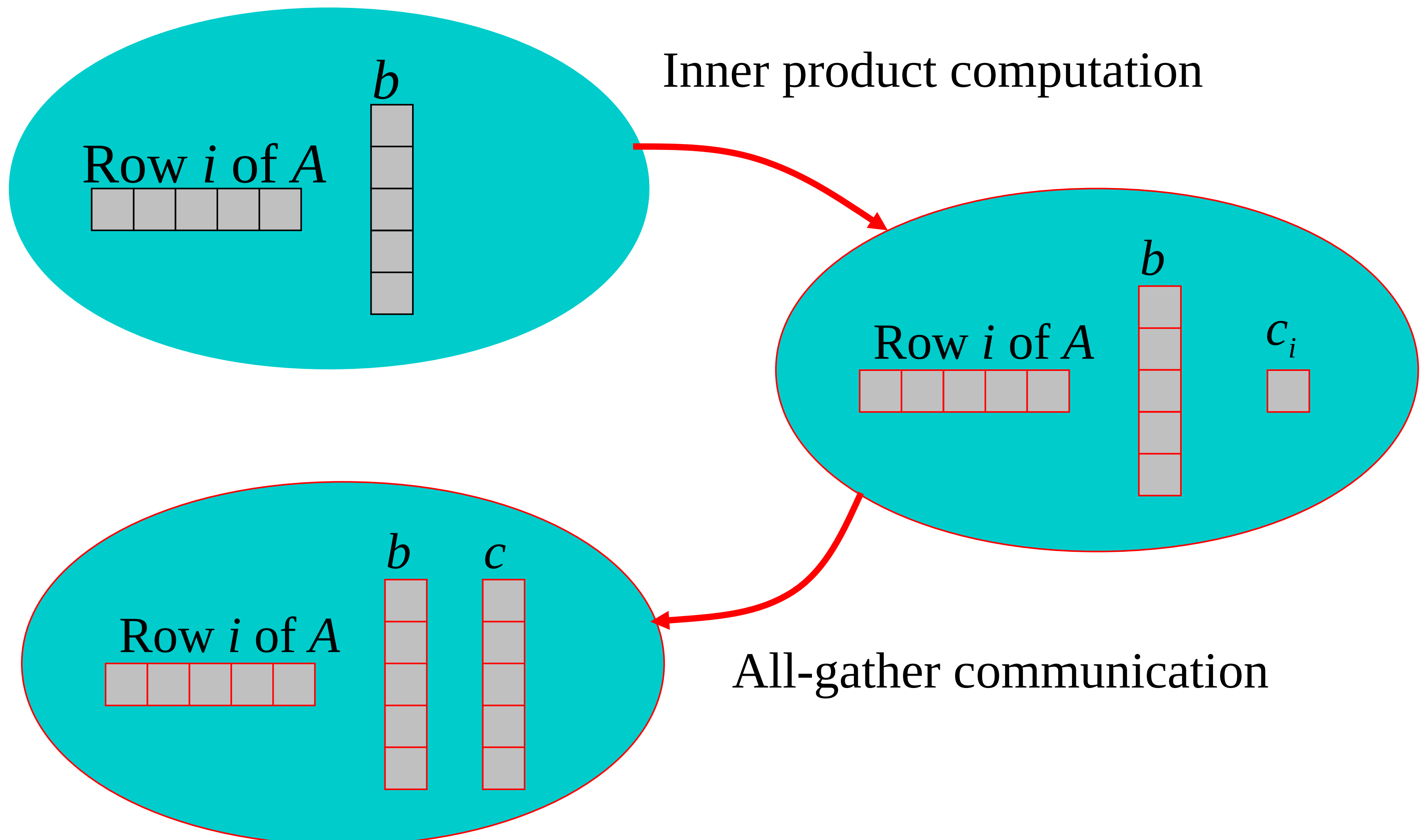
Rowwise block striped matrix:

- The tasks will involve the dot product of one row of the matrix with the vector.

$$c[i] = a[i,0]*b[0] + a[i,1]*b[1] + \ldots + a[i,n-1]*b[n-1];$$

- Each process will contribute one or more elements of the result vector, c[i].

Steps in the 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 – assume a square matrix

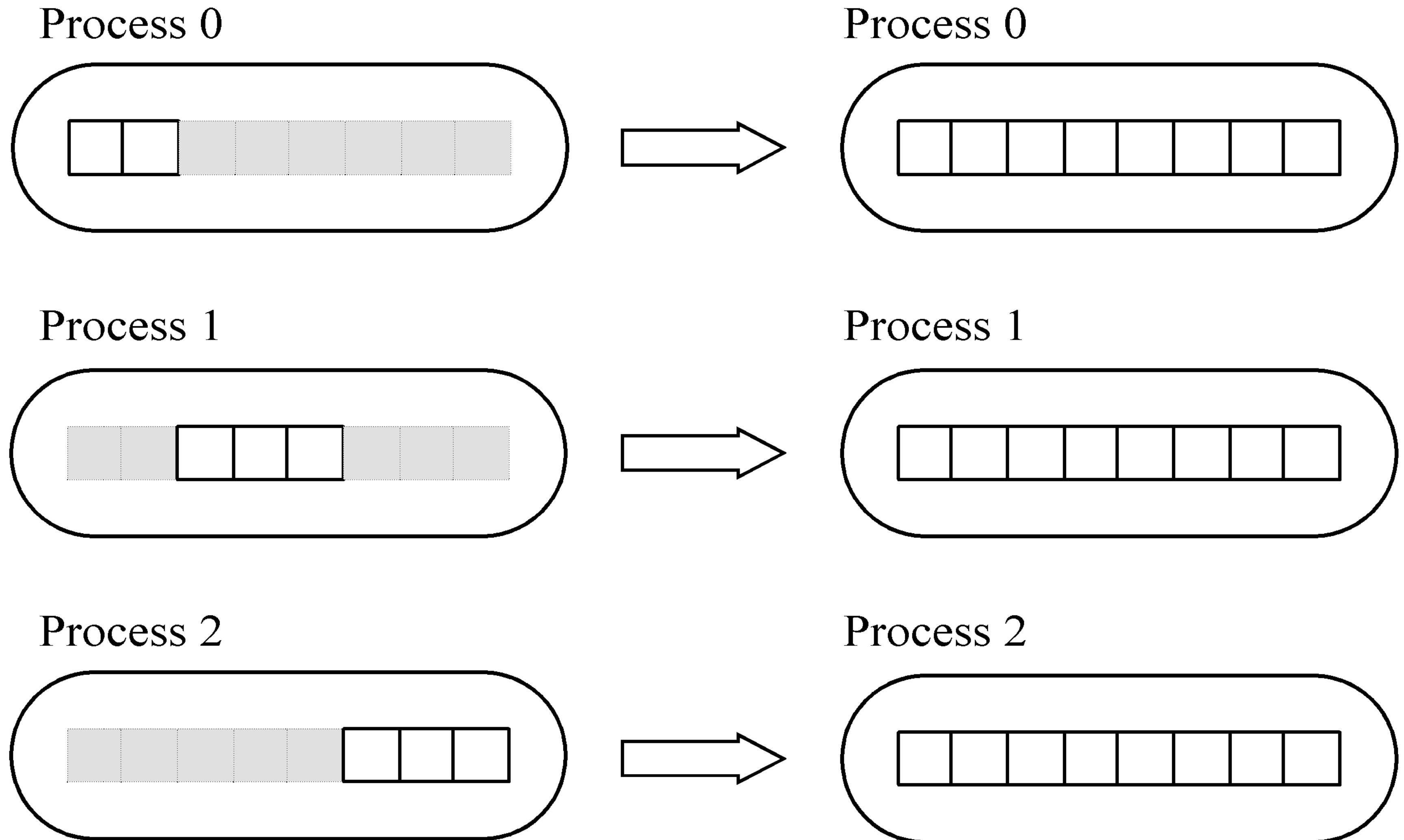
- Sequential algorithm complexity: $\Theta(n^2)$ – multiplying n elements of each row of the matrix times n elements of the vector
- Parallel algorithm computational complexity: $\Theta(n^2/p)$
- Communication complexity of all-gather: $\Theta(\log p + n)$

  Why? All processes sending log p results to one process. Assuming that p is a square number.
- Overall complexity: $\Theta(n^2/p + \log p)$

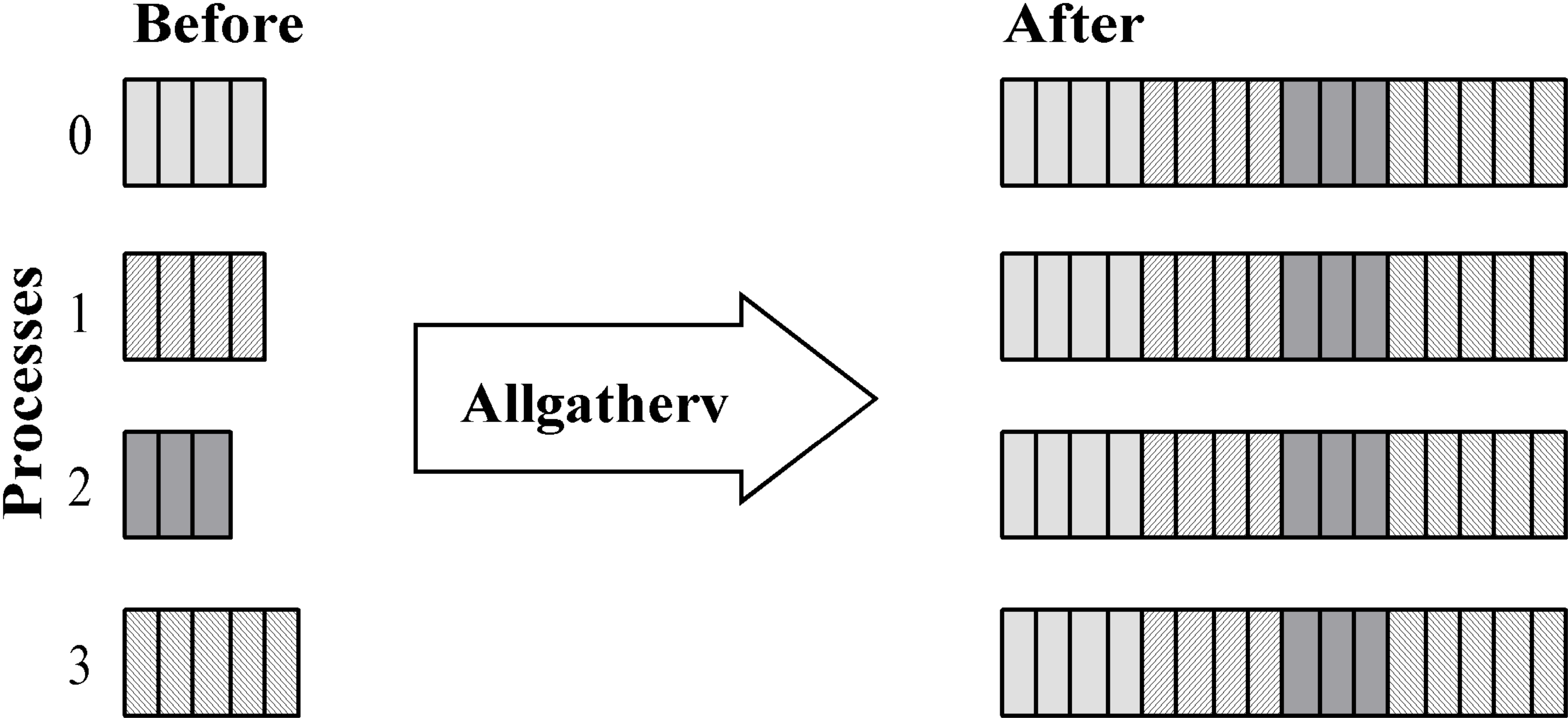
Is this a scaleable algorithm?

- Sequential time complexity: $\Theta(n^2)$
- Only parallel overhead is all-gather step
- When n is large, message transmission time dominates message latency
- Parallel communication time $\approx \Theta(n)$
- Assume that $n^2 \geq Cpn \Rightarrow n \geq Cp$
- Does not appear that the system is highly scalable

How are the elements calculated by each process correctly placed in the result vector?



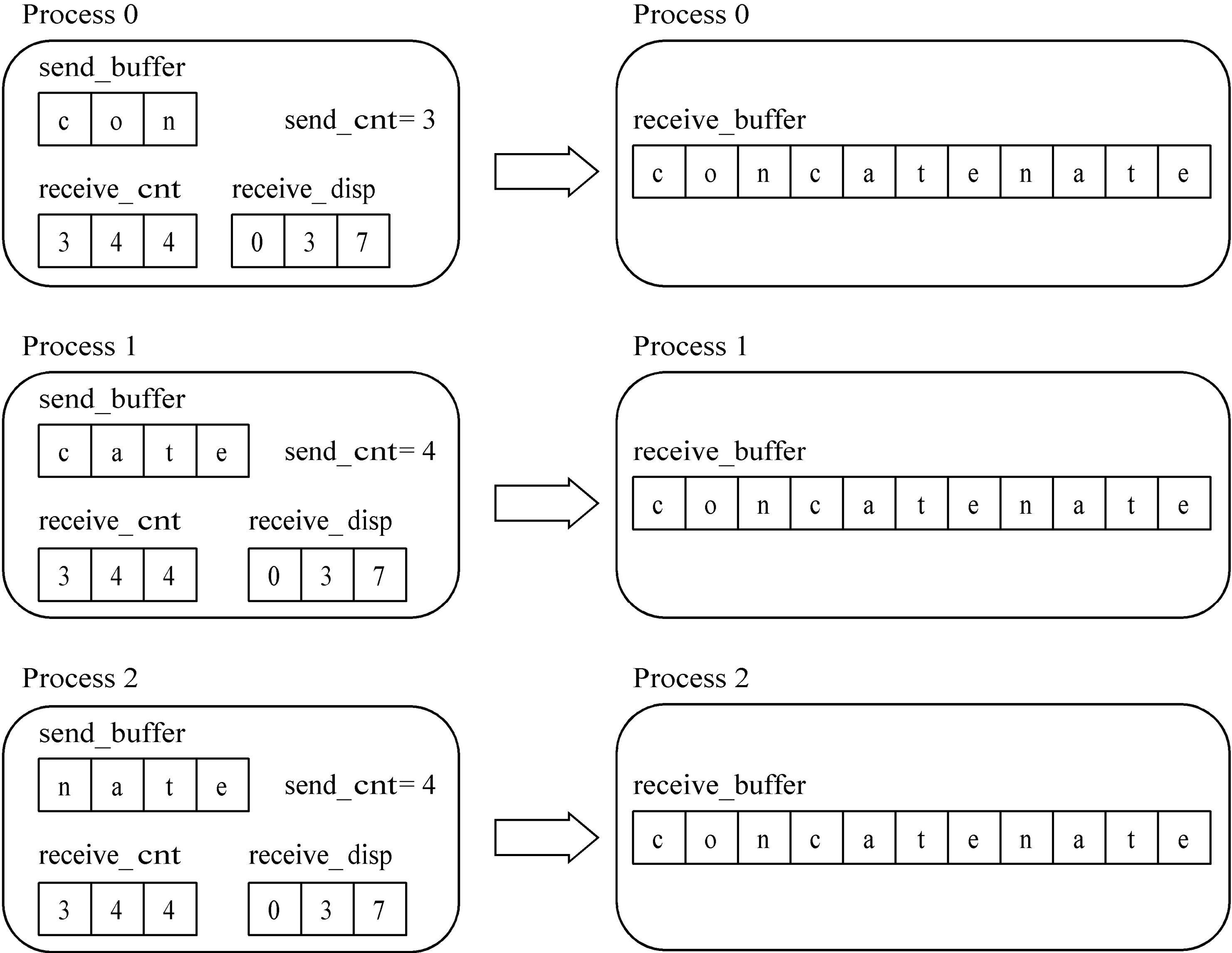
MPI_Allgatherv()



The function MPI_Allgatherv()

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

MPI_Allgatherv() in action



How to organize the elements of the result vector for transfer

Will use two arrays to set up transfer information:

- 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

Need to create space for the result vector

- Create space for the entire vector that will be filled
- Each process creates the “mixed transfer” arrays which contain that process's part of the result vector
- Each process calls `MPI_Allgatherv()` to initiate the transfer of data.

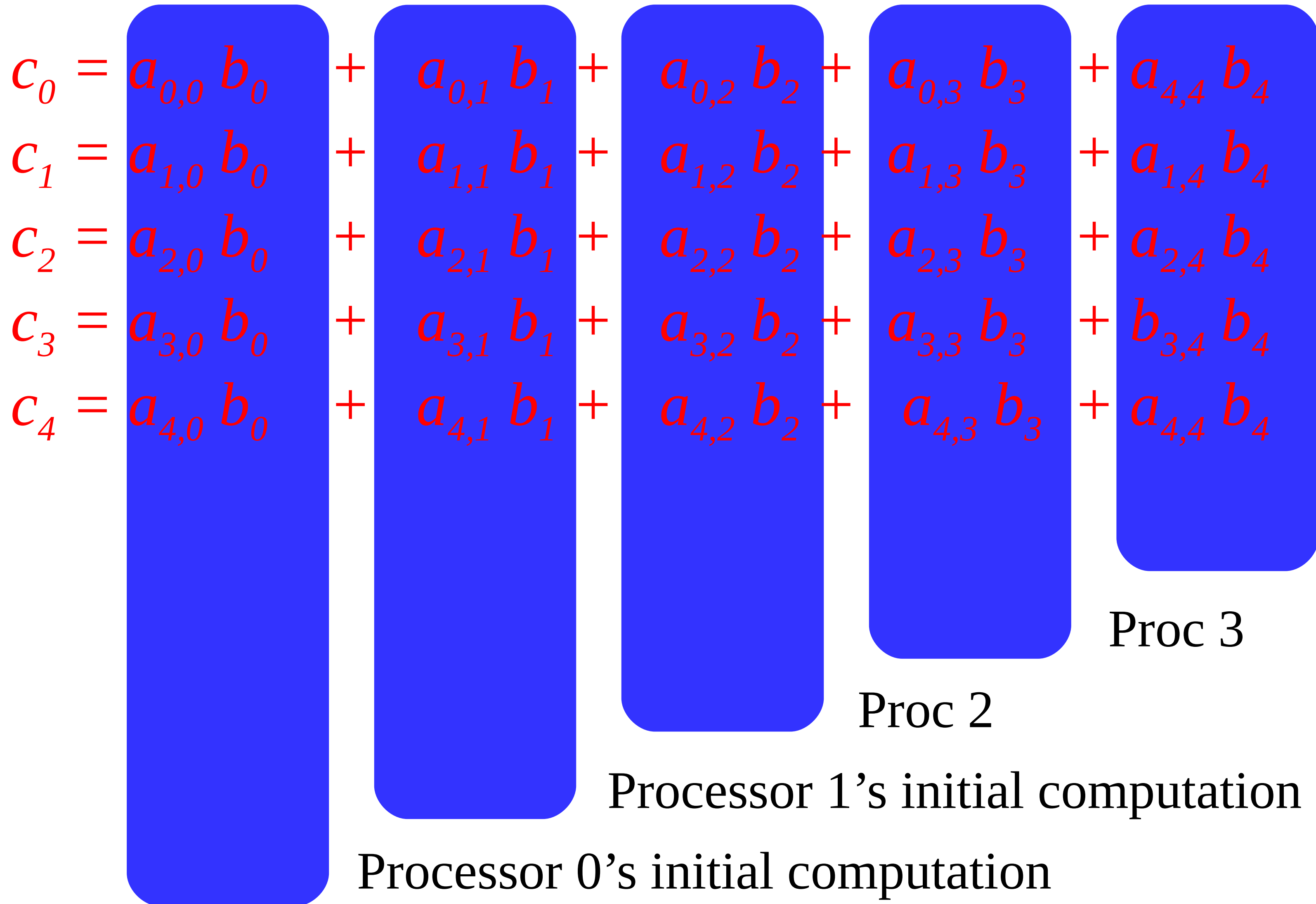
Reading the input vector

- Process *p*-1
  - Opens file
  - Reads vector length
- It broadcasts the vector length (root process = *p*-1)
- It allocates space for the vector
- Process *p*-1 reads the vector and the matrix, closes file
- Process p-1 broadcasts the vector
- Process 0 prints the vector
- Exact call to `printf()` depends on value of parameter `datatype`

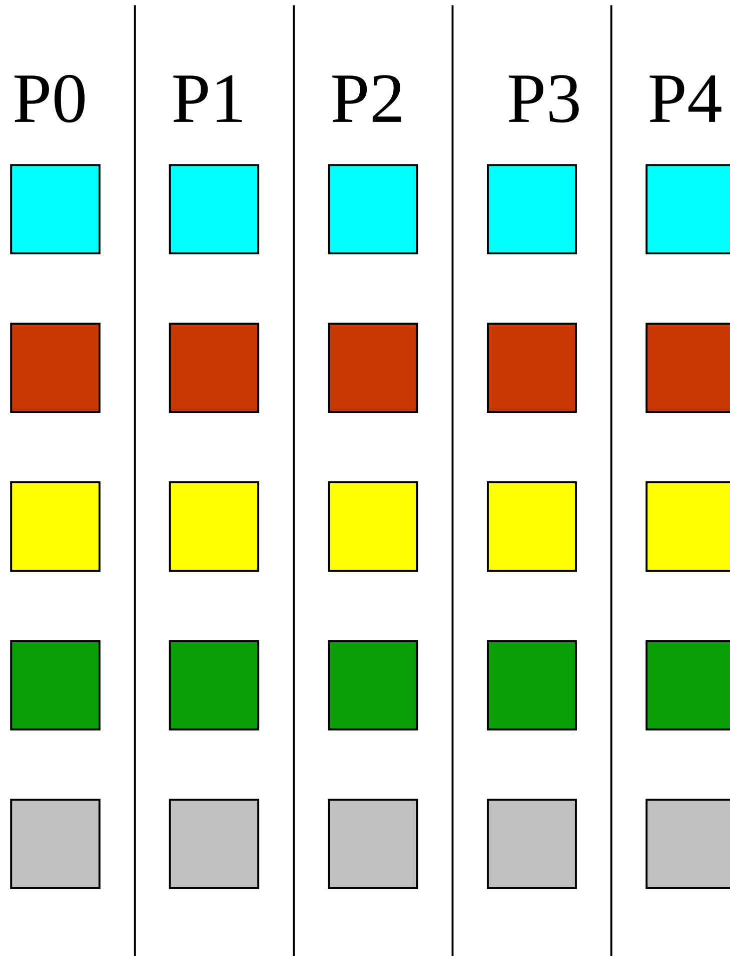
Can the calculation be improved by distributing the columns of the matrix?

- Partitioning through column-wise domain decomposition of the data
- The tasks are associated with
  - A column of the matrix
  - A single vector element
- A process will compute a partial sum associated with several elements of c[n]. The partial sums will need to be communicated to other processes.

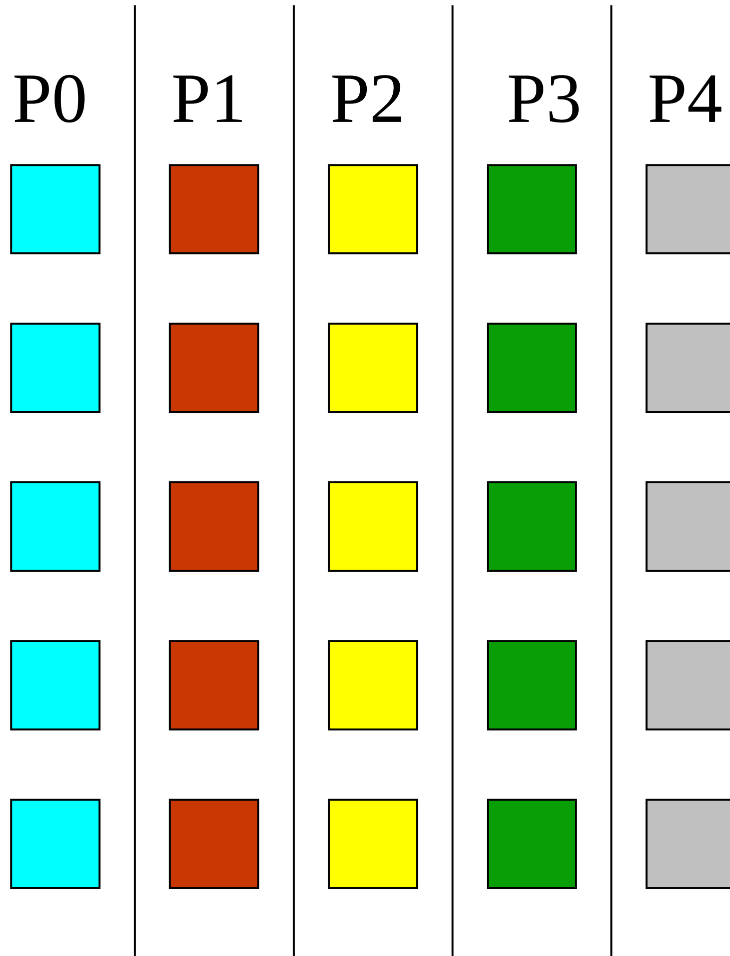
Matrix-Vector multiplication



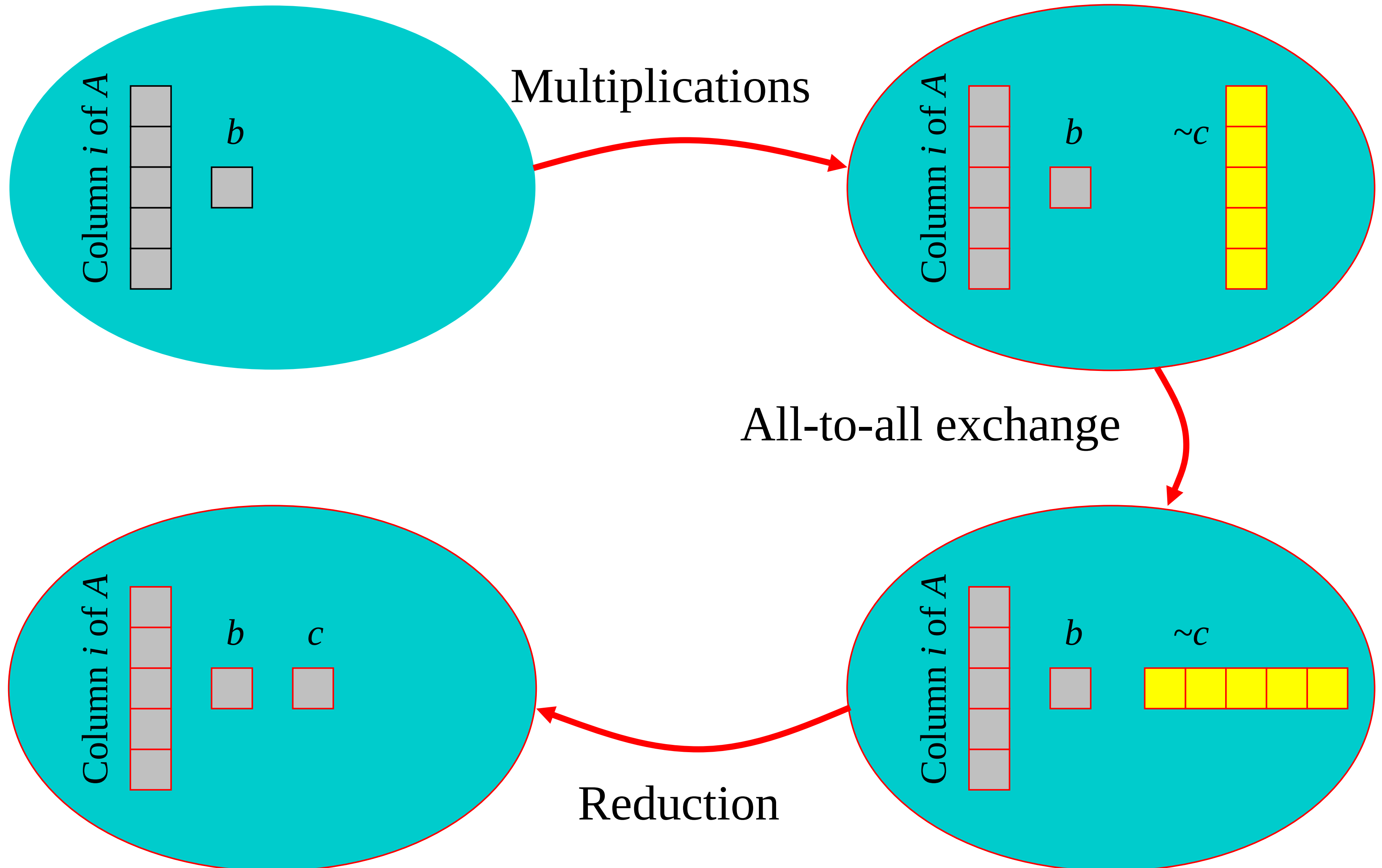
All-to-all exchange (before)



All-to-all exchange (after)



Steps in the 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

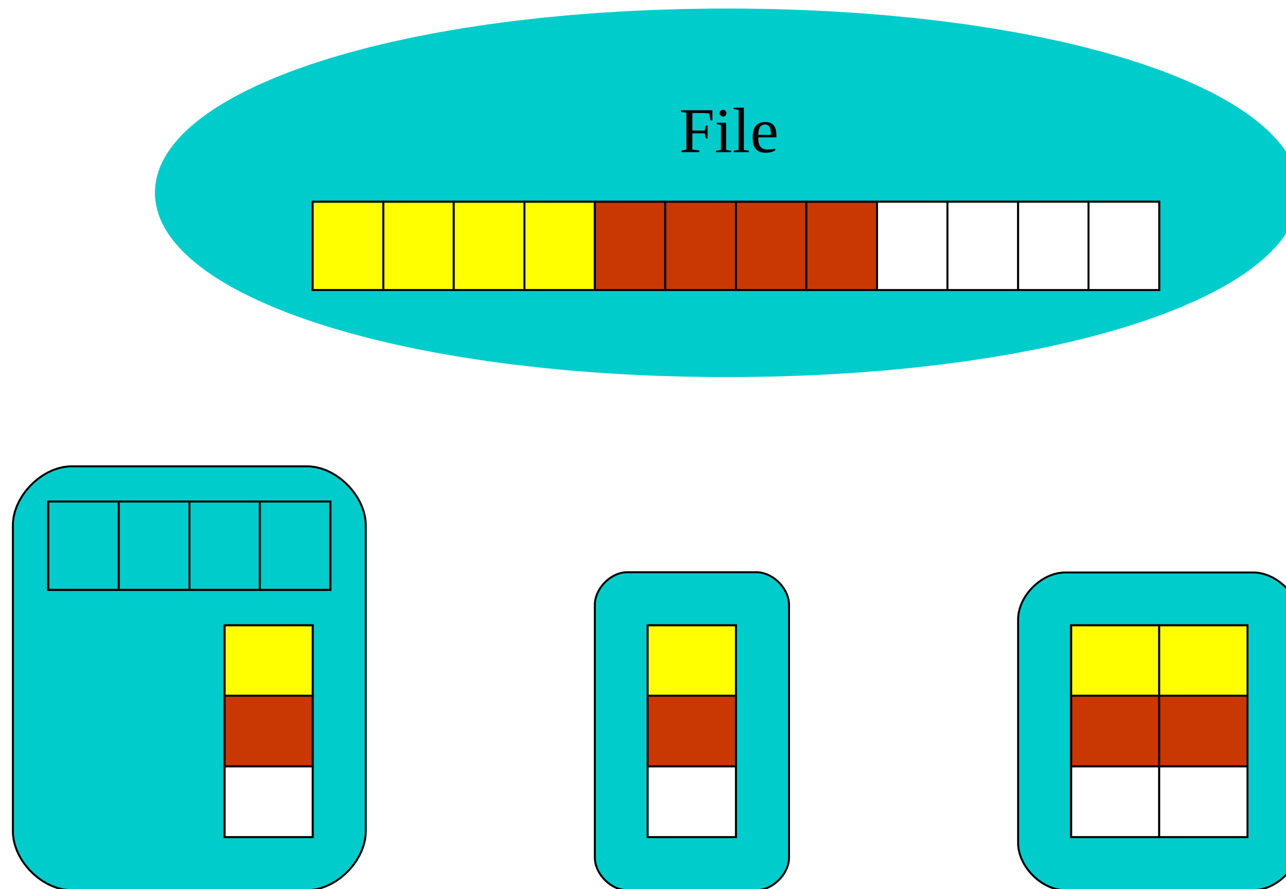
Complexity analysis

- Sequential algorithm complexity: $\Theta(n^2)$
- Parallel algorithm computational complexity: $\Theta(n^2/p)$
- Communication complexity of all-to-all: $\Theta(p + n)$ – each process sends at most n elements to p-1 processes
- Overall complexity: $\Theta(n^2/p + n + p)$

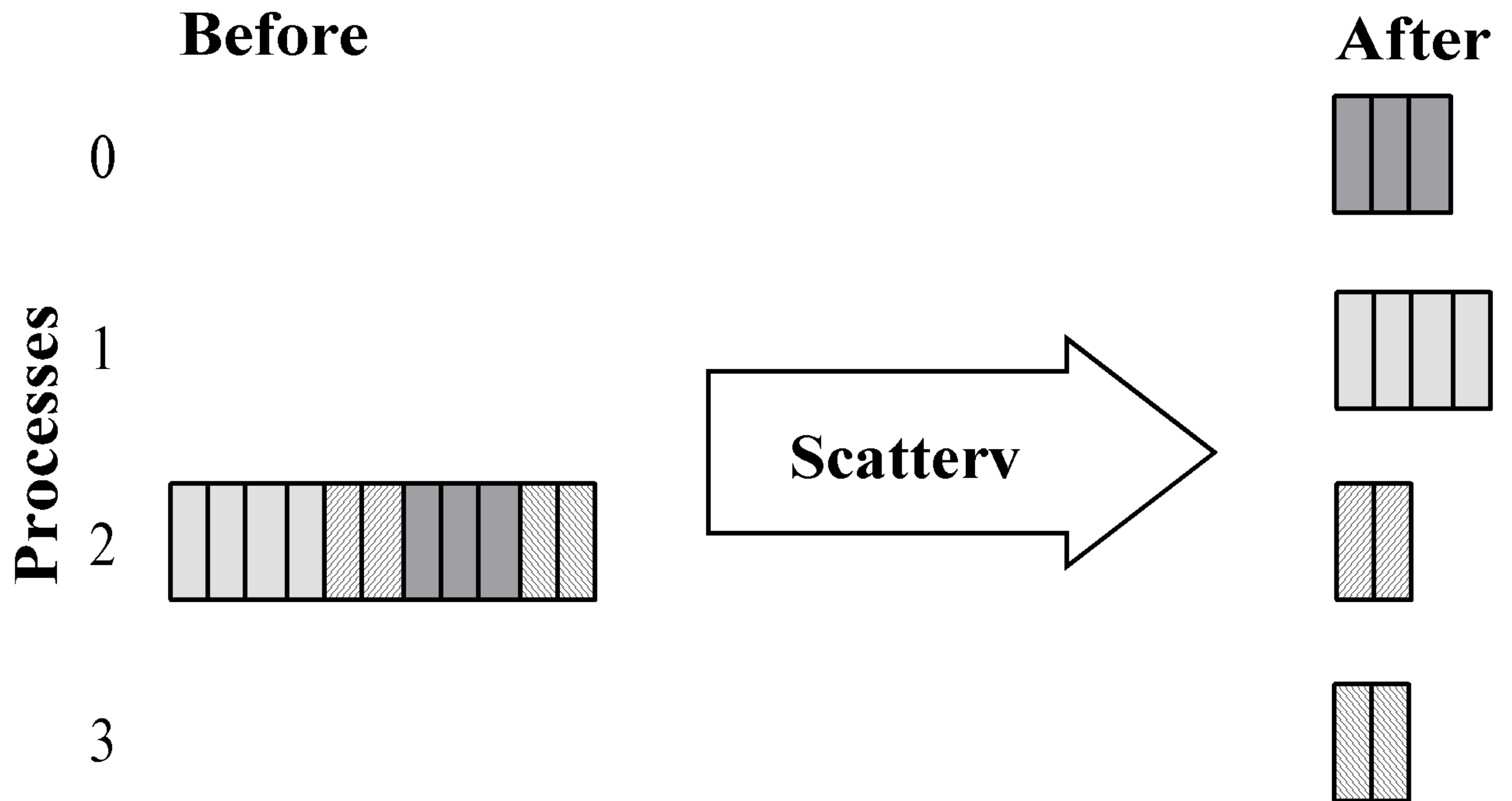
Does this version scale better?

- 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 same as rowwise algorithm

Reading and transferring one or more columns to each process



The function MPI_Scatterv()



Function prototype for MPI_Scatterv()

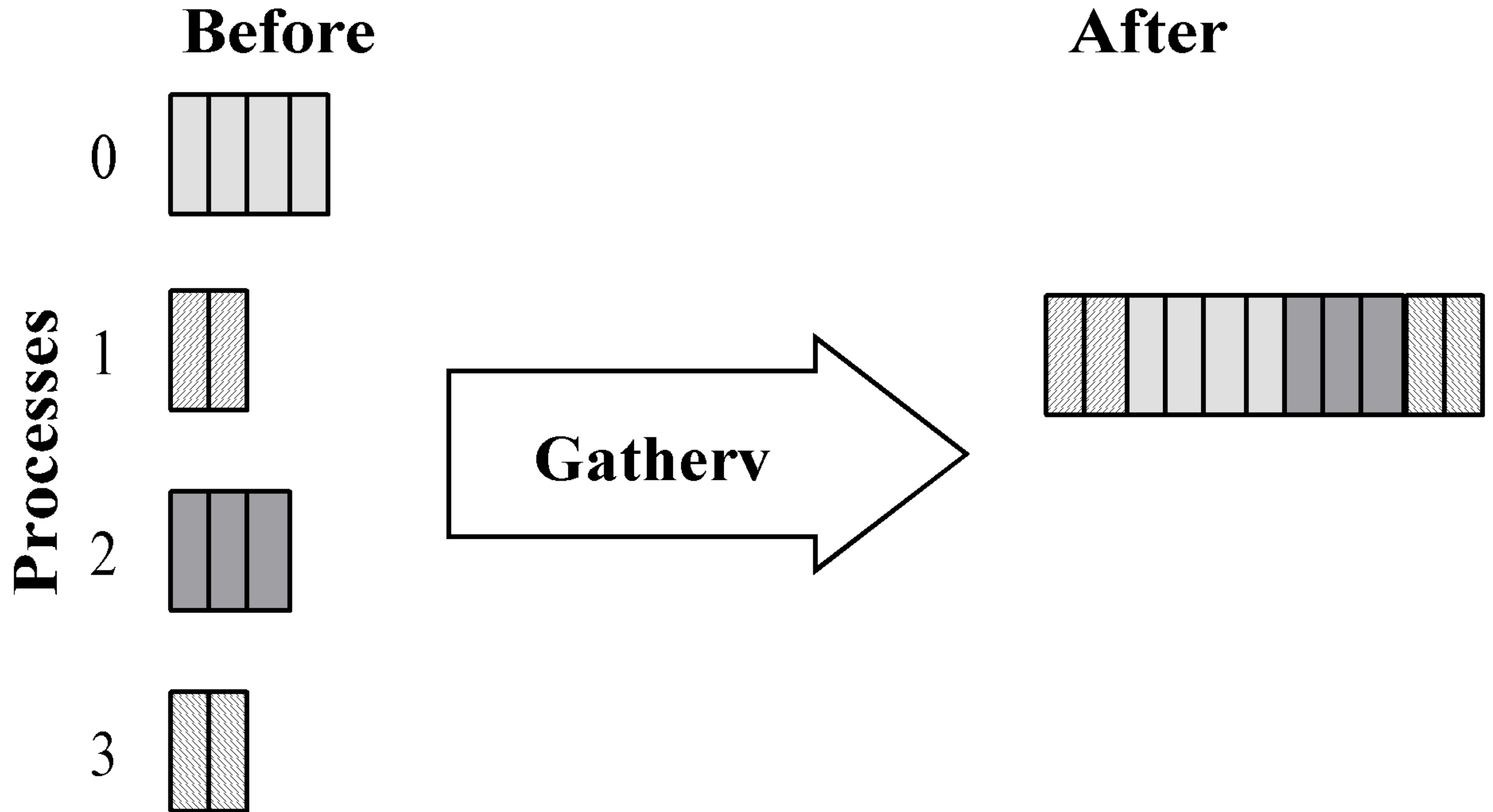
```
int MPI_Scatterv (
    void             *send_buffer,
    int              *send_cnt,
    int              *send_disp,
    MPI_Datatype      send_type,
    void             *receive_buffer,
    int               receive_cnt,
    MPI_Datatype      receive_type,
    int               root,
    MPI_Comm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

Printing a matrix with columns distributed to multiple processes

Data motion opposite to that we did when reading the matrix:

- Each process has part of several columns
- Replace a “scatter” function with a “gather” function
- Use the “v” variant because different processes contribute different numbers of elements

Action of the function MPI_Gatherv()



The function MPI_Gatherv()

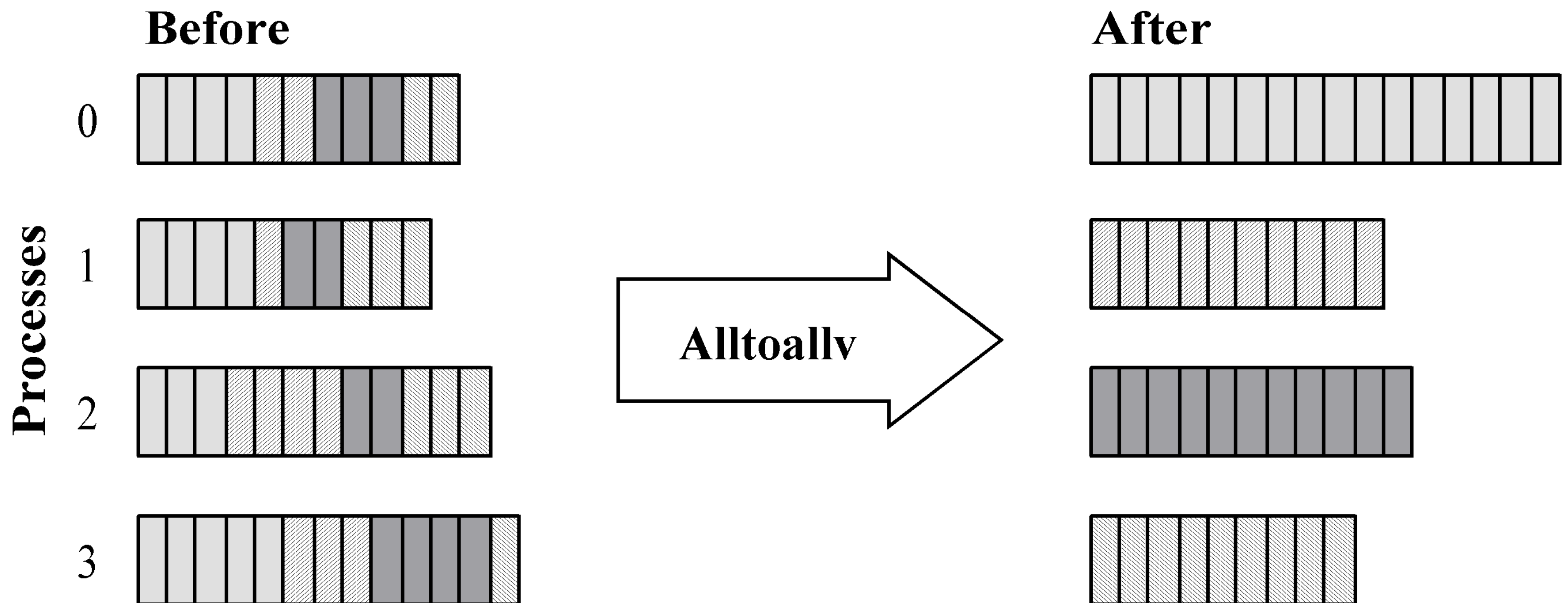
```
int MPI_Gatherv (
   void            *send_buffer,
   int              send_cnt,
   MPI_Datatype     send_type,
   void            *receive_buffer,
   int             *receive_cnt,
   int             *receive_disp,
   MPI_Datatype     receive_type,
   int              root,
   MPI_Comm         communicator)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

Assembling the result vector

The last action in the algorithm is for all processes to exchange needed partial sums.

- Each process receives BLOCK_SIZE() elements from every other process.
- After the exchange, process I has p subarrays that it adds to form the elements of the result vector.
- Will use the “v” version of the function since the number of elements in the exchanged arrays differ from process to process.

Action of the function MPI_Alltoallv()



The function `MPI_Alltoallv()`

```
int MPI_Alltoallv (
    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)
```

Usage of MPI_Alltoallv()

- MPI_Alltoallv() requires two pairs of 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

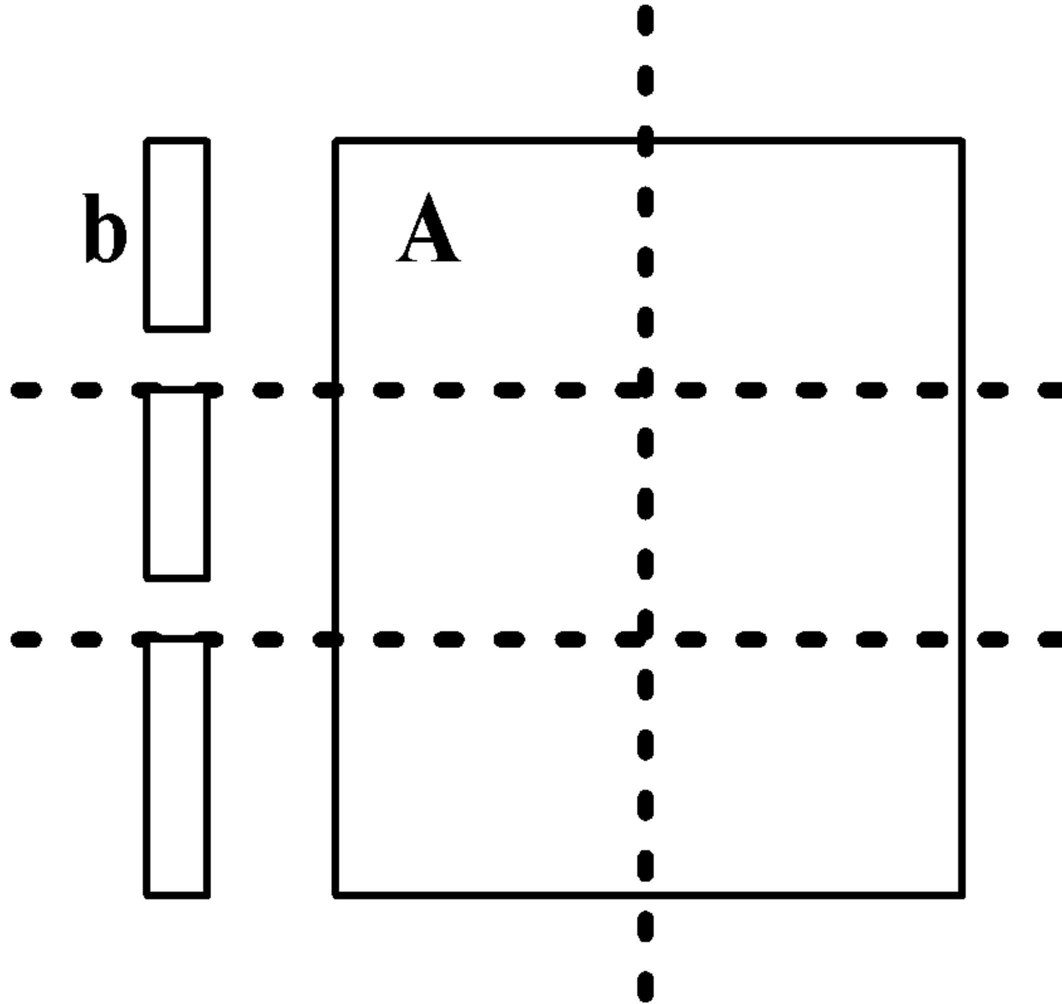
Details of the arrays in MPI_Alltoall()

- The first array:
  - How many elements received from each process (always the same value)
  - Uses the process ID and utility macro BLOCK_SIZE
- The second array:
  - Starting position of each process' block
  - Assume blocks in process rank order

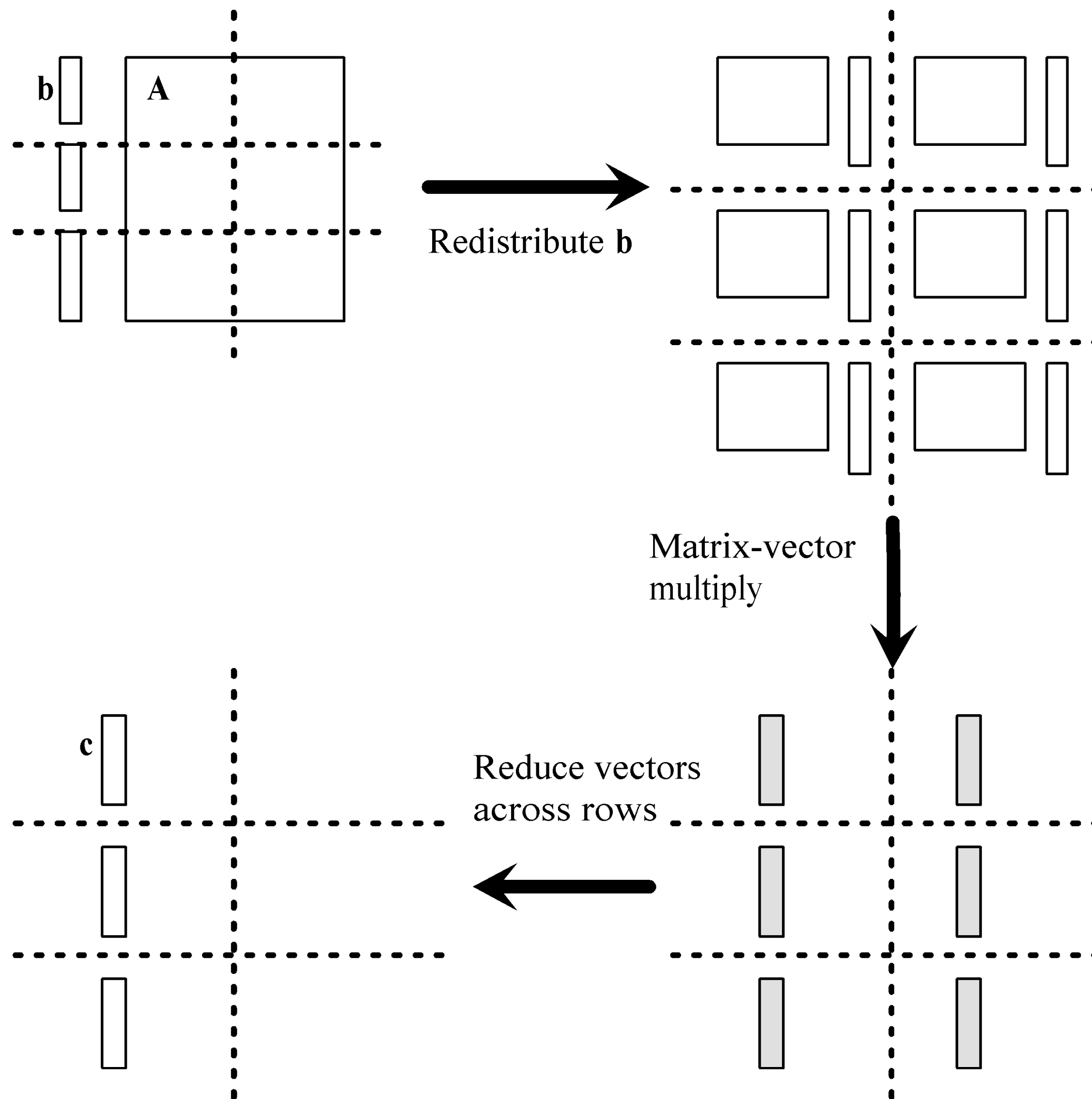
Third version – checkerboard block decomposition

- Associate primitive task with each element of the matrix **a**
- Each primitive task performs one multiply
- Agglomerate primitive tasks into rectangular blocks for each process
- Processes form a 2-D grid
- Vector **b** distributed by blocks among processes in first column of grid
- All processes do a sum reduction so each process has parts of the vector **c**
- All the parts have to be gathered together.

Tasks after agglomeration



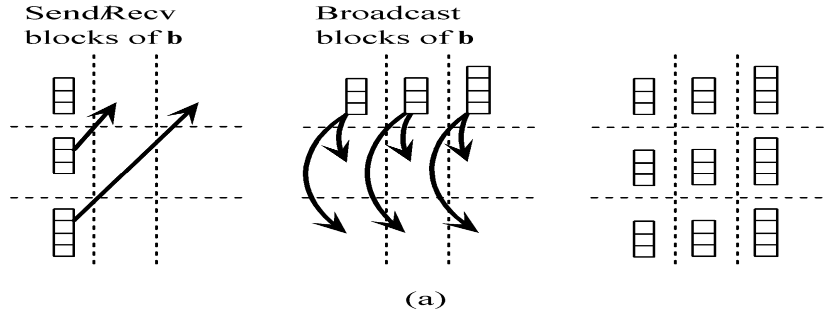
Steps in the algorithm



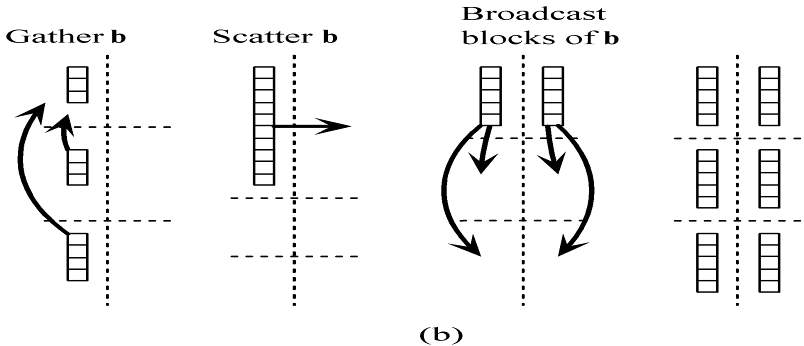
Redistributing vector **b**

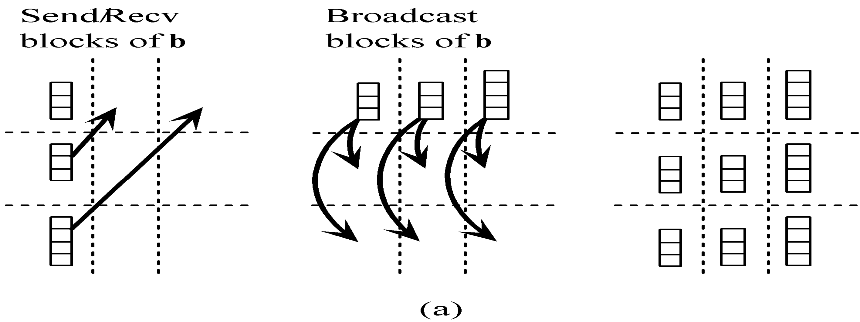
- Step 1: Move **b** from processes in first row to processes in first column
  - If *p* square
    - First column/first row processes send/receive portions of **b**
  - If *p* not square
    - Gather **b** on process 0, 0
    - Process 0, 0 broadcasts to first row procs
- Step 2: First row processes scatter **b** within columns

Redistributing Vector **b** (cont.)

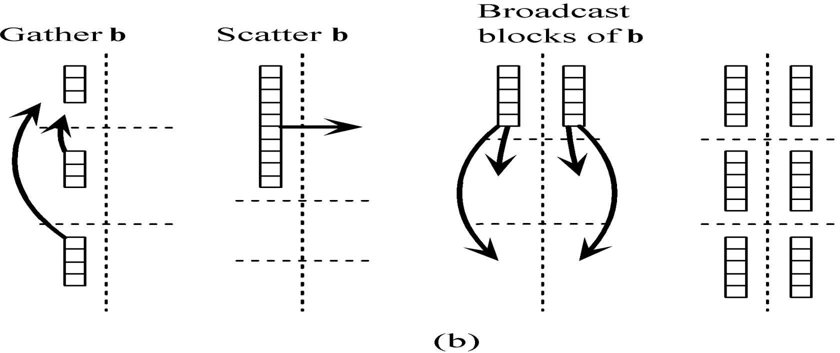


When p is a square number





When p is not a square number



Complexity analysis of checkerboard decomposition

Assume p is a square number:

If grid is $1 \times p$, the problem becomes a columnwise block striped one.

If grid is $p \times 1$, the problem becomes a rowwise block striped one.

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: $\Theta(n^3/p + n \log p / \sqrt{p})$

Efficiency 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 scaleable than the previous two implementations

Creating new communicators

- Processes need to restrict the broadcast in Step 2) above to a specific column.
- Have to arrange the processes used in a virtual 2-D grid.
- Create a custom communicator to do this.
- Collective communications involve all the processes in a communicator.
- Need to broadcast data and do reductions among the subset of processes.
- We will create communicators for processes in same row or same column.

What does a communicator contain?

- Process group identity
- Context information
- Attributes
  - Topology – allows a rearrangement of the processes within the calculation. For example,
    - a cartesian or grid arrangement
    - a binary or other graph relationship
  - Plus other attributes

The functions used to create a cartesian grid of processes

- MPI_Dims_create()
  - The input parameters
    - Total number of processes in desired grid
    - Number of grid dimensions
  - Returns number of processes in each dimension

  The function MPI_Dims_create() helps the user select a balanced distribution of processes per coordinate direction.

MPI_Dims_create()

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

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

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

MPI_Cart_create() - creates a communicator with the cartesian topology

```
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 */
```

An example of creating a new communicator

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

Other useful grid-related functions

MPI_Cart_rank()

Given coordinates of process in cartesian communicator, returns process 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 */
```

MPI_Cart_coords()

Given rank of process in cartesian communicator, returns process's coordinates

```
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 a columnwise scatter and a rowwise reduce

```
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

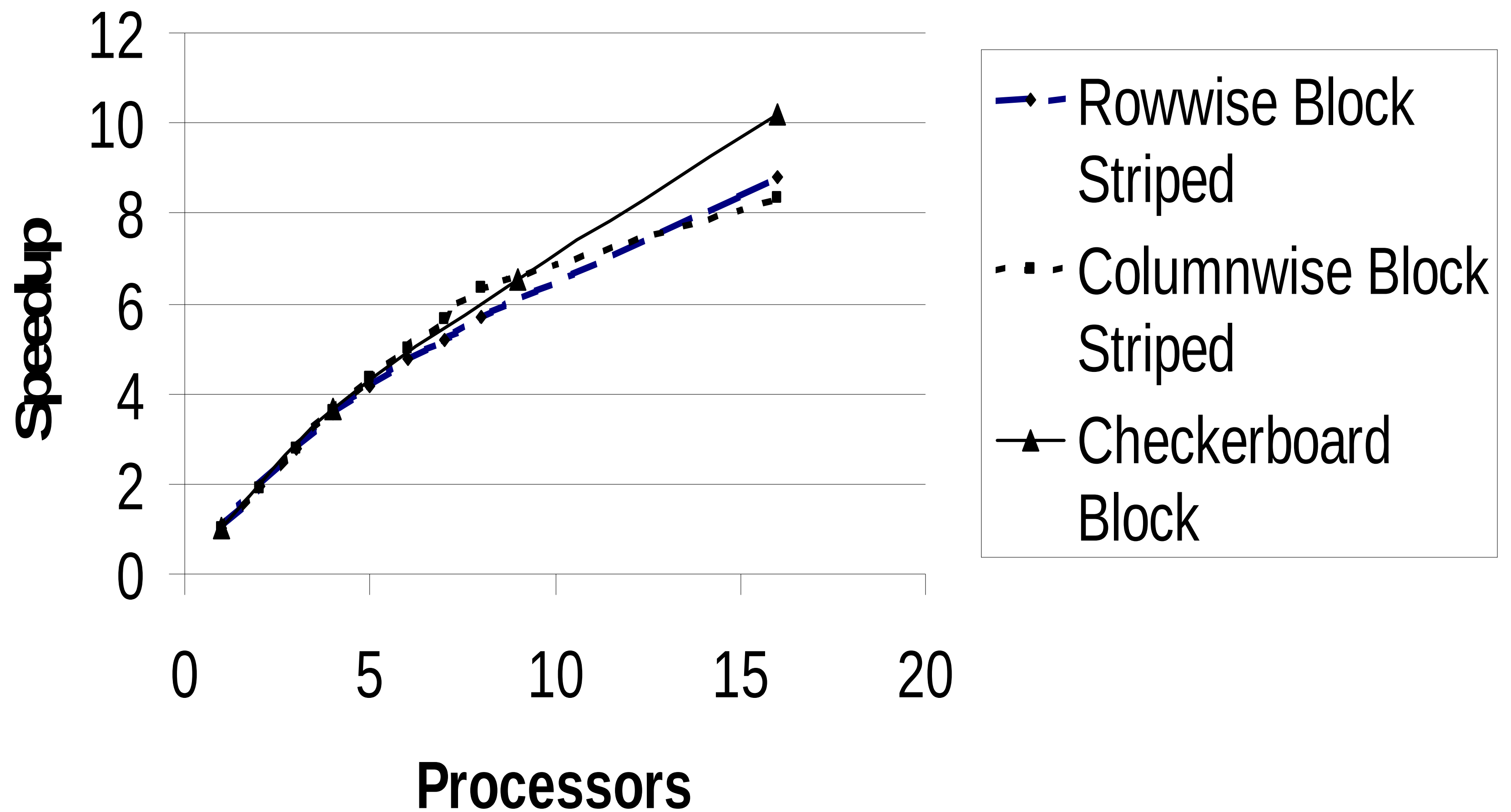
```
MPI_Comm grid_comm; /* 2-D process grid */

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);
```

Comparison of Three Algorithms



Summary - 1

- 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

- Communicators with Cartesian topology
  - How to create new communicators
  - Identifying the processes by rank or coordinates
- Subdividing communicators
  - Allows collective operations among subsets of processes