Introduction to Parallel & Distributed Computing

Programming with MPI

Lecture 6, Spring 2024

Instructor: 罗国杰

gluo@pku.edu.cn

Correctly Time your Program

- **Wall clock time vs. CPU time**
- **Example**

```
> /usr/bin/time -p your-prog
...
real 0.04
user 0.04
sys 0.00
```

- **OpenMP APIs**

```
// Starting the time measurement
double start = omp_get_wtime();
// Computations to be measured
...
// Measuring the elapsed time
double end = omp_get_wtime();
// Time calculation (in seconds)
double time1 = end - start;
// print out the elapsed time
cout << "Time elpased: "<< time1
<< "  seconds." << endl;
```

Distributed Shared Memory

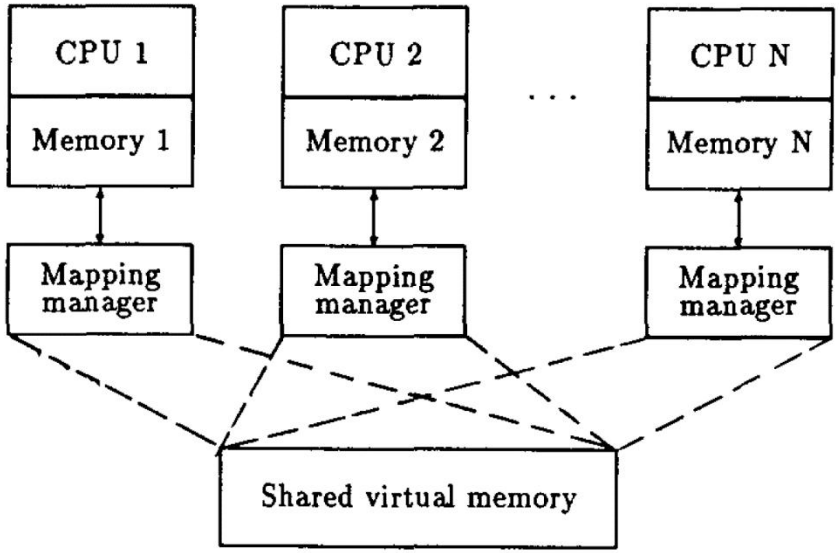


Table I. Spectrum of Solutions to the Memory Coherence Problem

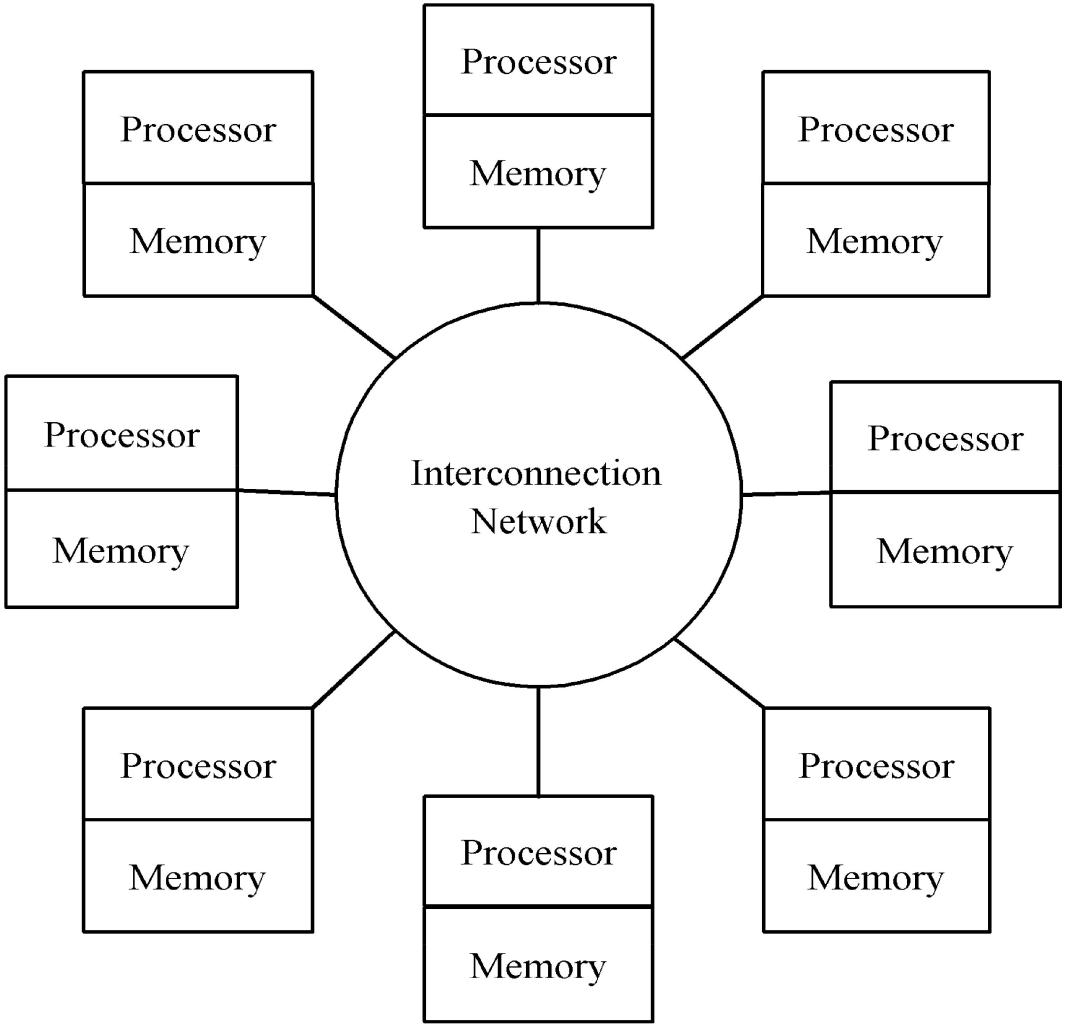
Page synchronization method	Page ownership strategy			
		Dynamic		
			Distributed manager	
	Fixed	Centralized manager	Fixed	Dynamic
Invalidation	Not allowed	Okay	Good	Good
Write-broadcast	Very expensive	Very expensive	Very expensive	Very expensive

K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," ACM Trans. Comput. Syst., vol. 7, no. 4, pp. 321–359, Nov. 1989.

In this Lecture: Topic Overview

- Principles of Message-Passing Programming
- The Building Blocks: Send and Receive Operations
- MPI: the Message Passing Interface
- A few examples of collective comm. (to-be cont.)

Message-Passing Model



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Principles of Message-Passing Programming

- **The logical view of the message-passing paradigm**
  - **a machine consists of *p* processors,**
  - **each with its own exclusive address space**
- **CONSTRAINTS…**
  - **data must be explicitly partitioned and placed**
  - **all interactions (read-only or read/write) require cooperation of two processors - the processor that has the data and the processor that wants to access the data**
- **These two constraints, while onerous, make underlying costs very explicit to the programmer**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Principles of Message-Passing Programming

- **Message-passing programs are often written using the *asynchronous* or *loosely synchronous* paradigms**
  - **In the asynchronous paradigm, all concurrent tasks execute asynchronously**
  - **In the loosely synchronous model, tasks or subsets of tasks synchronize to perform interactions. Between these interactions, tasks execute completely asynchronously**
- **Most message-passing programs are written using the *single program multiple data* (SPMD) model**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

The Message Passing Interface

- Late 1980s: vendors had unique libraries
- 1989: Parallel Virtual Machine (PVM) developed at Oak Ridge National Lab
- MPI Standards
  - 1992: Work on standard begun
  - 1994: Version 1.0
  - 1997: Version 2.0
  - 2008: Version 2.1
  - 2009: Version 2.2
  - 2012: Version 3.0
  - 2015: Version 3.1
  - 2021: Version 4.0
  - WIP: Version 4.1 & 5.0
- Today: MPI is dominant massage passing library standard

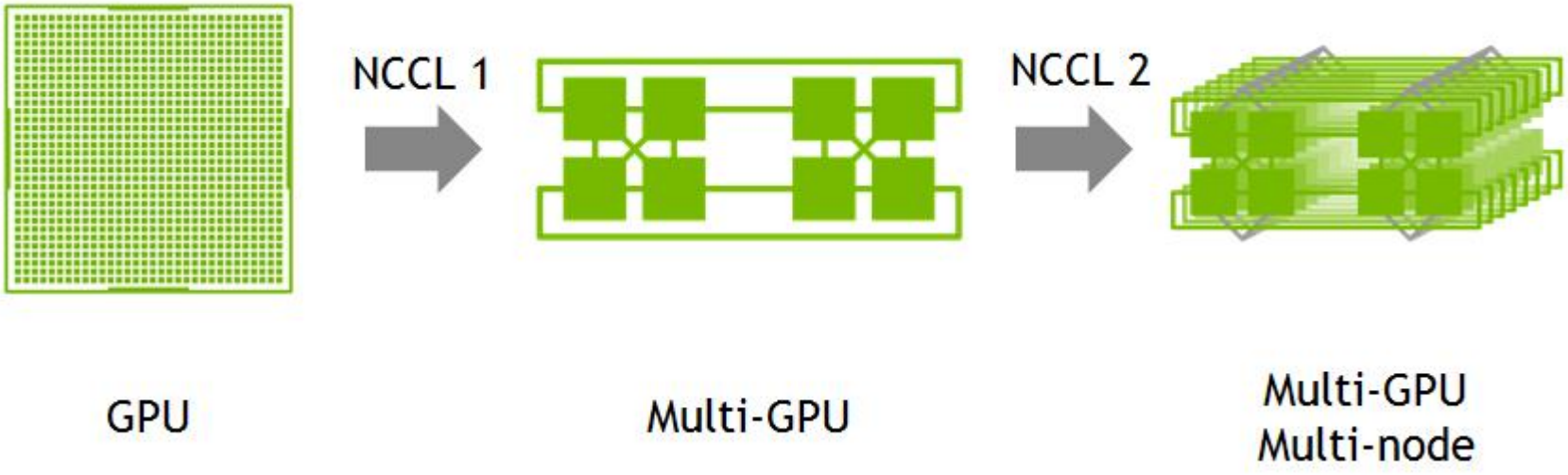
https://www.mpi-forum.org/

MPI Implementations and Tutorials

- **Basics**
  - http://www.mpi-forum.org
  - Gropp, W. (2011). MPI (Message Passing Interface). In: Padua, D. (eds) Encyclopedia of Parallel Computing. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-09766-4_222
- **Implementations**
  - MPICH2 http://www.mcs.anl.gov/research/projects/mpich2/
  - Open MPI http://www.open-mpi.org/
- **Tutorials**
  - https://computing.llnl.gov/tutorials/mpi/
  - http://www.mcs.anl.gov/research/projects/mpi/

What is the Point Learning the "Outdated" MPI Library?

- **NVIDIA Collective Communications Library (NCCL)**

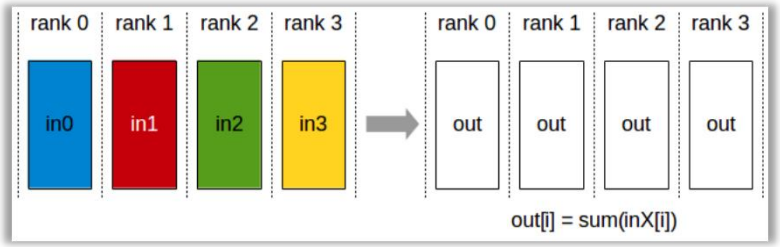


- **You will understand better some concepts in a modern library for collective communication**

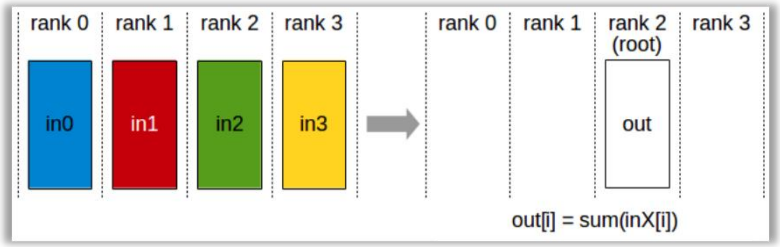
Example Collective Communications

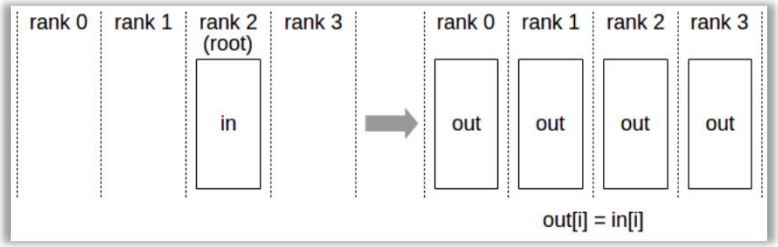
- **NVIDIA Collective Communications Library (NCCL)**

ncclAllReduce()

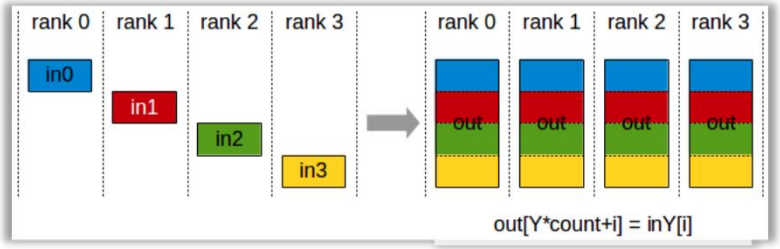


ncclReduce

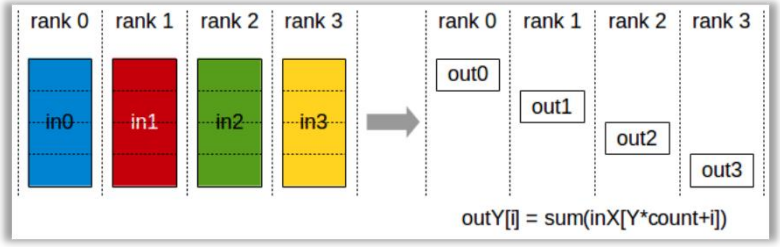




ncclBroadcast()



ncclAllGather()



ncclReduceScatter()

MPI: the Message Passing Interface

- **MPI defines a standard *library* for message-passing that can be used to develop portable message-passing programs using either C or Fortran.**
- **The MPI standard defines both the syntax as well as the semantics of a core set of library routines.**
- **Vendor implementations of MPI are available on almost all commercial parallel computers.**
- **It is possible to write fully-functional message-passing programs by using only the six routines.**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI: Hello World!

```
#include <mpi.h>

int main(int argc, char *argv[])
{
        int npes, myrank;
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &npes);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        printf("From process %d out of %d, Hello World!\n",
               myrank, npes);
        MPI_Finalize();
        return 0;
}
```

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

MPI: the Message Passing Interface

The minimal set of MPI routines.

`MPI_Init`	Initializes MPI.
`MPI_Finalize`	Terminates MPI.
`MPI_Comm_size`	Determines the number of processes.
`MPI_Comm_rank`	Determines the label of calling process.
`MPI_Send`	Sends a message.
`MPI_Recv`	Receives a message.

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

Starting and Terminating the MPI Library

- `int MPI_Init(int *argc, char ***argv);`
  - **to initialize the MPI environment**
  - **called prior to any calls to other MPI routines**
  - **also strips off any MPI related command-line arguments**
- `int MPI_Finalize();`
  - **is called at the end of the computation**
  - **performs various clean-up tasks to terminate the MPI environment**
- **All MPI routines, data-types, and constants are prefixed by "MPI_". The return code for successful completion is** MPI_SUCCESS

Querying Information

- `int MPI_Comm_size(MPI_Comm comm, int *size);`
  - **the number of processes**
- `int MPI_Comm_rank(MPI_Comm comm, int *rank);`
  - **the label of the calling process**
- **The rank of a process is an integer that ranges from zero up to the size of the communicator minus one**

Compiling and Running MPI Programs

- **Compiling examples**
  - mpicc –o foo foo.c
  - mpic++ -o bar bar.cpp
- **Running examples**
  - mpirun –np 4 foo
  - mpirun –np 2 foo : -np 4 bar
- **Specifying host processors**
  - see “--hostfile” and “--host” options

Better Understanding of MPI

- **Compilation**
  - mpicc **or** mpic++ **is just a wrapper**
  - **Try "**mpicc --show**" in Open MPI or MPICH. For example,**

    gcc -I/usr/lib64/mpi/gcc/openmpi/include/openmpi
    -I/usr/lib64/mpi/gcc/openmpi/include –pthread
    -L/usr/lib64/mpi/gcc/openmpi/lib64 -lmpi -lopen-rte -lopen-pal
    -ldl -Wl,--export-dynamic -lnsl -lutil -lm –ldl

- **Execution**
  - mpirun **or** mpiexec**. For example,**

> mpirun -np 8 ./a.out

Better Understanding of MPI Processes

```
/* hello.c */
#include  <stdio.h>
#include  <mpi.h>

void main(int argc, char *argv[]) {
  MPI_Status status;
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(
    MPI_COMM_WORLD, &rank);
  MPI_Comm_size(
    MPI_COMM_WORLD, &size);

  /* see code segments on right */

  MPI_Finalize();
}
```

```
char message[20];
int tag=11;

if (rank == 0) {
  strcpy(message, "Hello,World!");
  for (i=1; i<size; i++)
    MPI_Send(message, 13, MPI_CHAR,
      i, tag, MPI_COMM_WORLD);
}

else {
  MPI_Recv(message, 20, MPI_CHAR,
    0, tag, MPI_COMM_WORLD,
    &status);
  printf( "Process %d : %.13s\n",
    rank, message);
}
```

Better Understanding of MPI Processes

```
/* master.c */

#include  <stdio.h>
#include  <mpi.h>

void main(int argc, char *argv[]) {
  MPI_Status status;
  MPI_Init(&argc, &argv);

  char message[20];
  int i, tag=11;

  strcpy(message, "Hello,World!");
    for (i=1; i<size; i++)
      MPI_Send(message, 13, MPI_CHAR,
        i, tag, MPI_COMM_WORLD);

  MPI_Finalize();
}
```

```
/* slave.c */

#include  <stdio.h>
#include  <mpi.h>

void main(int argc, char *argv[]) {
  MPI_Status status;
  MPI_Init(&argc, &argv);

  char message[20];
  int i, tag=11;

 MPI_Recv(message, 20, MPI_CHAR,
      0, tag, MPI_COMM_WORLD,
      &status);
    printf( "Process %d : %.13s\n",
      rank,message);

  MPI_Finalize();
}
```

Better Understanding of MPI Processes

- **Single program multiple data (SPMD) mode**

  > mpirun -np 8 ./hello

- **Multiple program multiple data (MPMD) mode**

  > mpirun -np 1 ./master : -np 7 ./slave

Better Understanding of MPI

- **Example hostfle**

  host_x slots=2 max_slots=8

  host_y slots=2 max_slots=8

- **mpirun -hostfile <file> -np 3 ./hello**
  - **2 processes on host_x, and 1 process on host_y**
- **mpirun -hostfile <file> -np 4 ./hello**
  - **2 processes on host_x, and 2 process on host_y**
- **mpirun -hostfile <file> -np 5 ./hello**
  - **3 processes on host_x, and 2 process on host_y**
- **mpirun -hostfile <file> -np 17 ./hello**
  - **There are not enough slots available in the system**

MPI Messages

- **data : (address, count, datatype)**
  - **Datatype is either a predefined type, or a custom type**
- **message : (data, tag)**
  - **Tag is an integer to assist the receiving process in identifying the message**

Send and Receive Operations

- Non-buffered blocking
- Buffered blocking
- Non-blocking

Sending and Receiving Messages

- **The basic functions for sending and receiving messages in MPI are the *MPI_Send* and *MPI_Recv*, respectively**
- **The calling sequences of these routines are as follows:**

```
int MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

- **MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons**
- **The datatype *MPI_BYTE* corresponds to a byte (8 bits) and *MPI_PACKED* corresponds to a collection of data items that has been created by packing non-contiguous data**
- **The message-tag can take values ranging from zero up to the MPI defined constant *MPI_TAG_UB***

MPI's Send Modes (1/2)

- **MPI_Send**
  - Will not return until you can use the send buffer
- **MPI_Bsend**
  - Returns immediately and you can use the send buffer
  - Related: MPI_buffer_attach(), MPI_buffer_detach()
- **MPI_Ssend**
  - Will not return until matching receive posted
  - Send + synchronous communication semantics
- **MPI_Rsend**
  - May be used ONLY if matching receive already posted
  - The sender provides additional information to the system that can save some overhead

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

MPI's Send Modes (2/2)

- **MPI_Isend**
  - **Nonblocking send, but you can NOT reuse the send buffer immediately**
  - **Related: MPI_Wait(), MPI_Test()**
- **MPI_Ibsend**
- **MPI_Issend**
- **MPI_Irsend**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

MPI Datatypes

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Sending and Receiving Messages

- **MPI allows specification of wildcard arguments for both source and tag.**
- **If source is set to *MPI_ANY_SOURCE*, then any process of the communication domain can be the source of the message.**
- **If tag is set to *MPI_ANY_TAG*, then messages with any tag are accepted.**
- **On the receive side, the message must be of length equal to or less than the length field specified.**

Sending and Receiving Messages

- **On the receiving end, the status variable can be used to get information about the *MPI_Recv* operation**
- **The corresponding data structure contains:**

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR; };
```

- **The *MPI_Get_count* function returns the precise count of data items received**

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype
                  datatype, int *count)
```

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Collective Communications

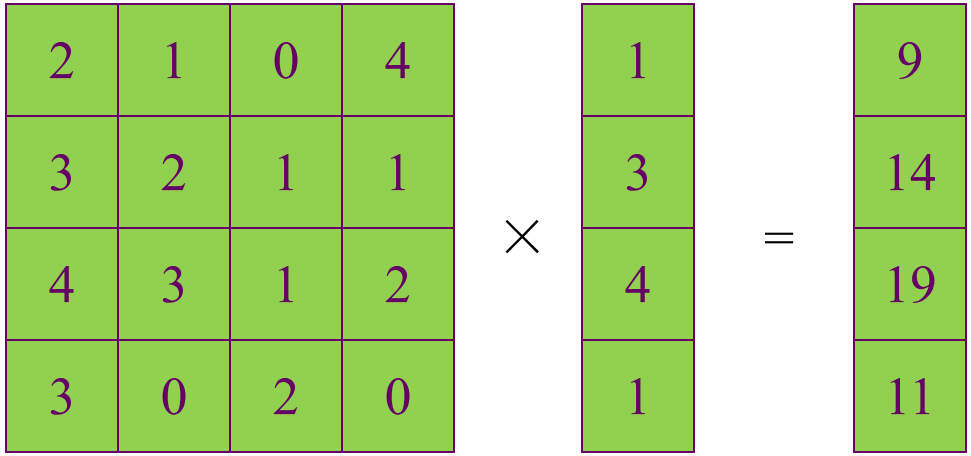
- one-to-all broadcast & all-to-one reduction
- all-to-all broadcast & all-to-all reduction
- scatter & gather
- all-to-all personalized communications

Example: Matrix-Vector Multiplication

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

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Sequential Algorithm



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Storing Vectors

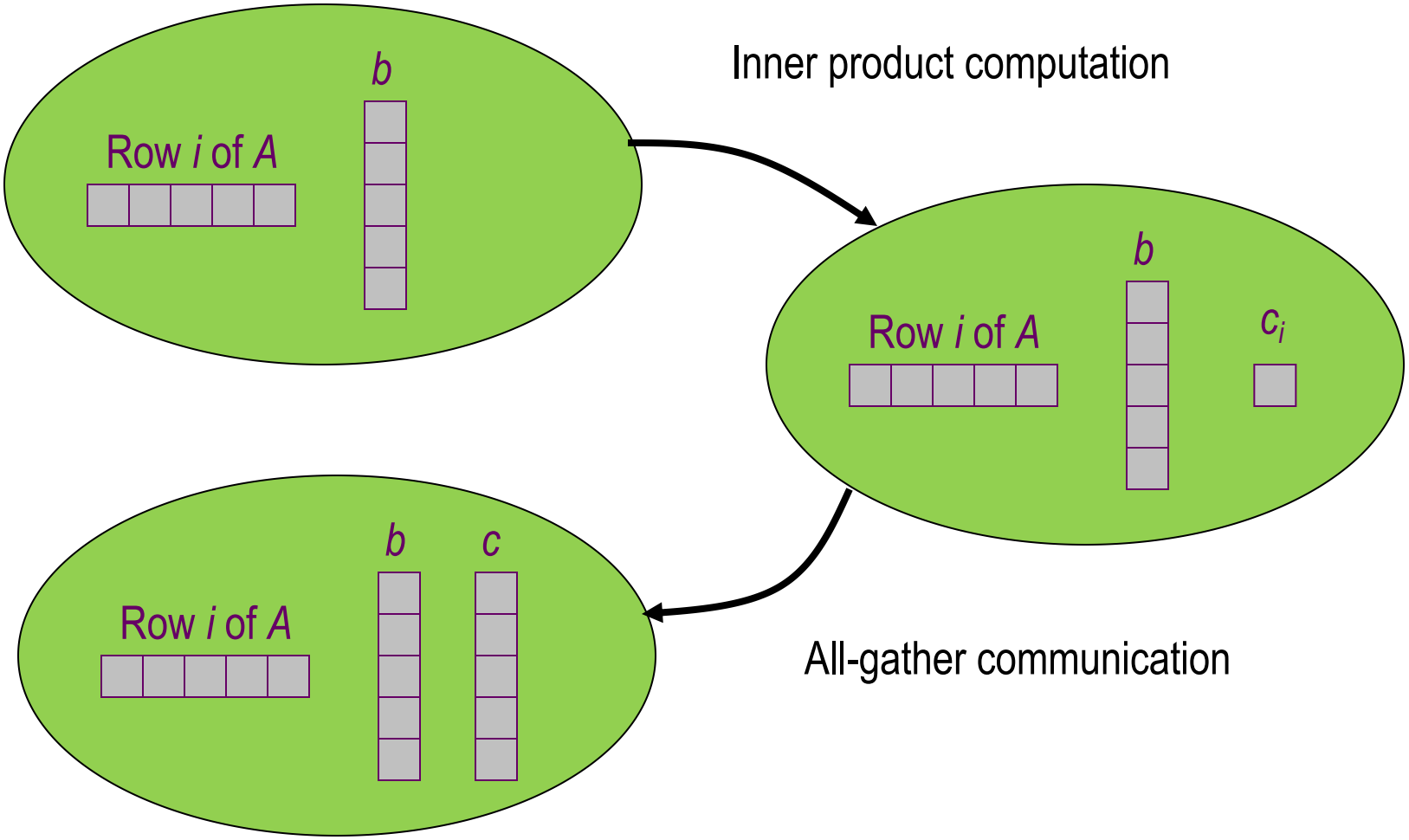
- Divide vector elements among processes, or
- Replicate vector elements
  - Vector replication acceptable because vectors have only n elements, versus n^2 elements in matrices

Rowwise Block Striped Matrix

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

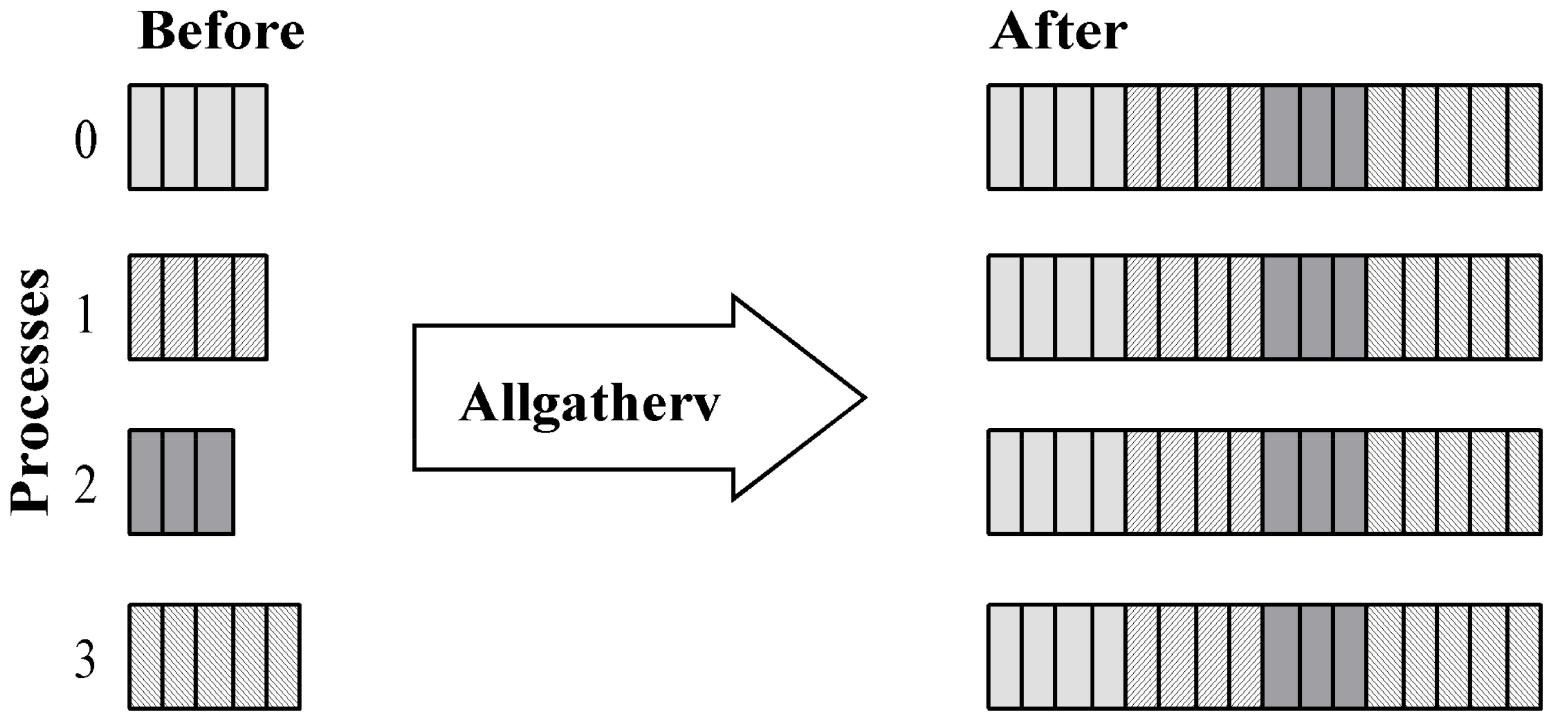
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Phases of Parallel Algorithm



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI_Allgatherv



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

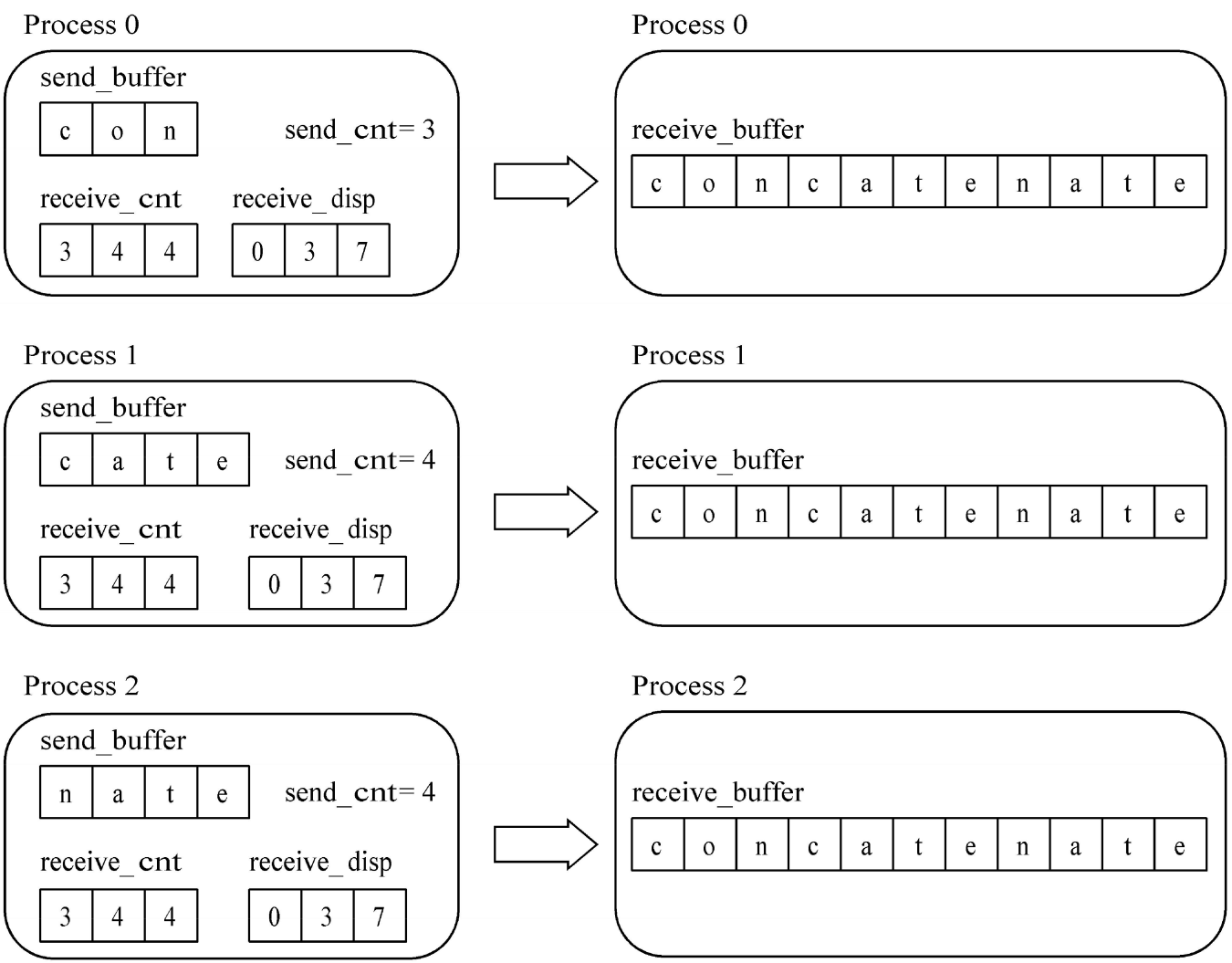
MPI_Allgatherv

```
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)
```

Use a simpler function *MPI_Allgather(…)* when the number of elements per processor is a constant

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI_Allgatherv in Action



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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

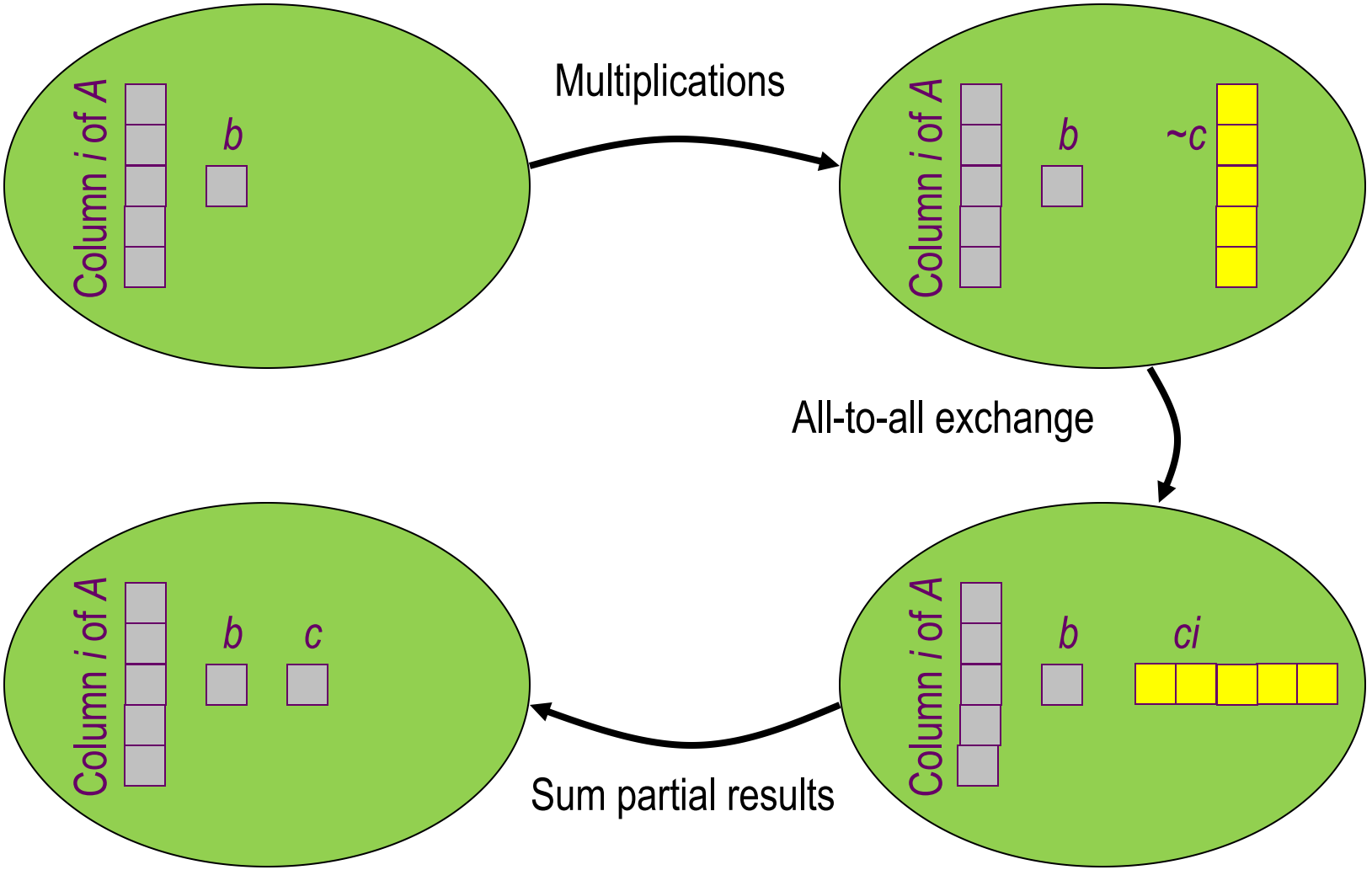
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Columnwise Block Striped Matrix

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

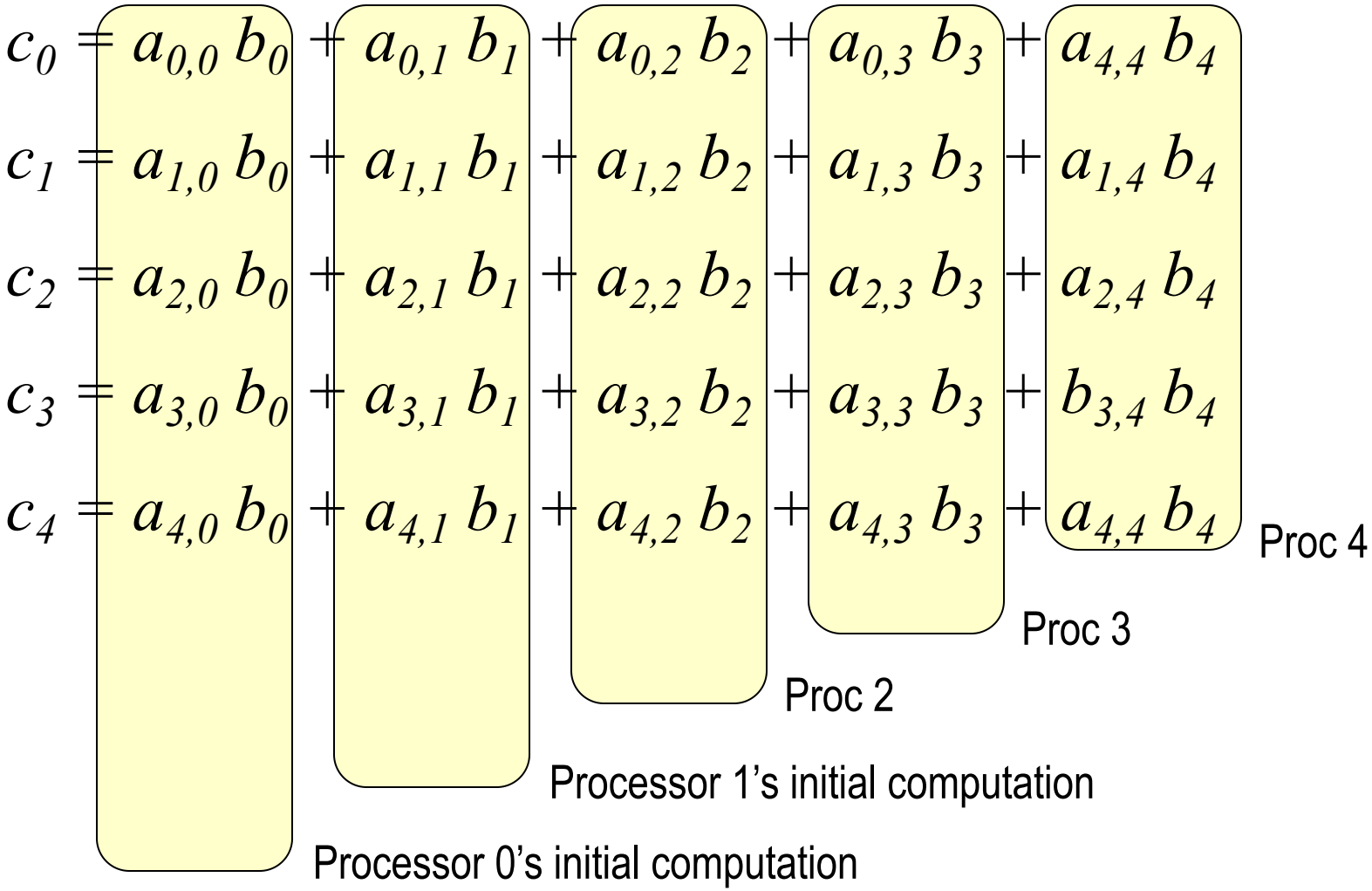
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Phases of Parallel Algorithm



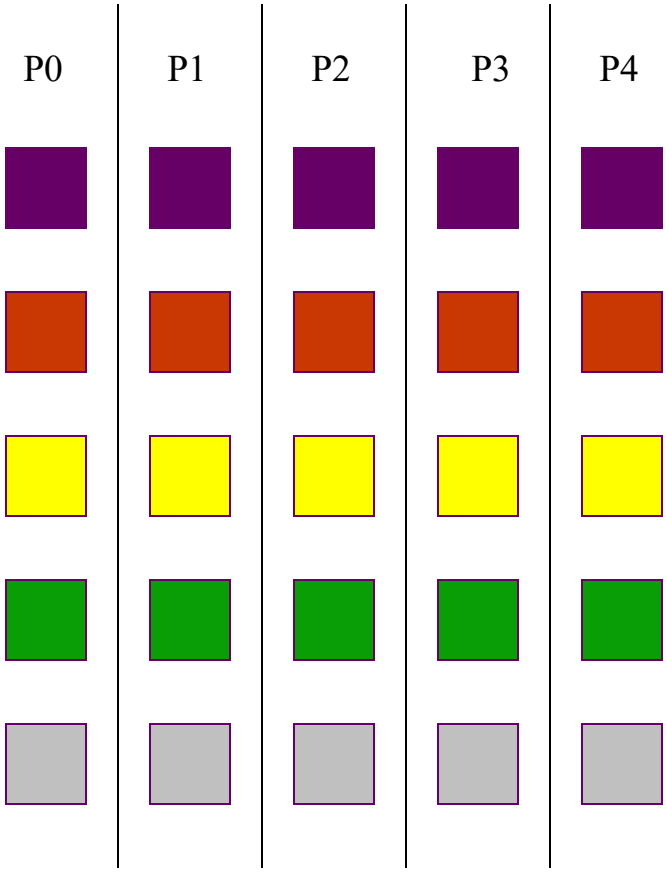
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Matrix-Vector Multiplication



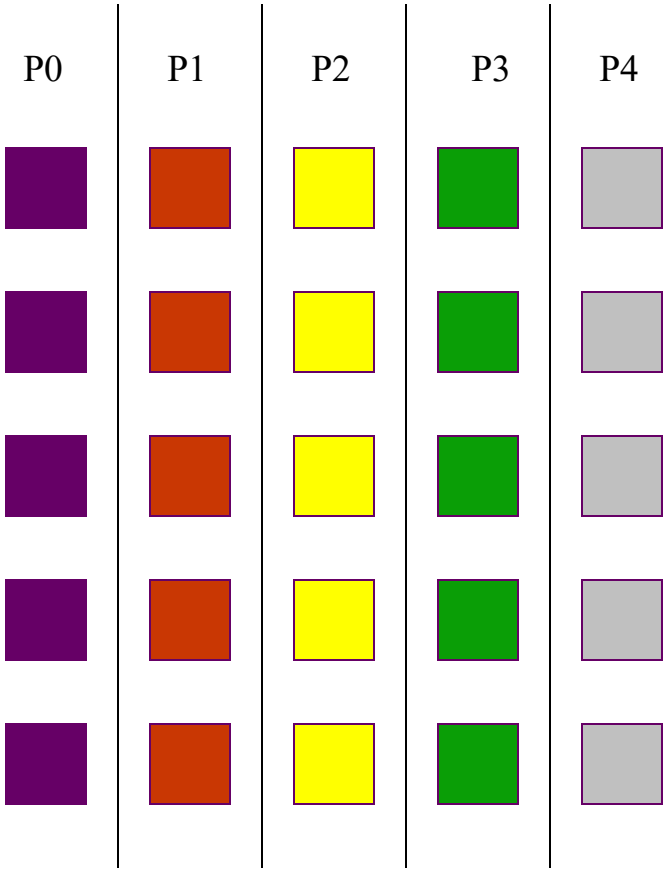
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

All-to-all Exchange (Before)



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

All-to-all Exchange (After)



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

All-to-All Exchange (All-to-All Personalized Communication)

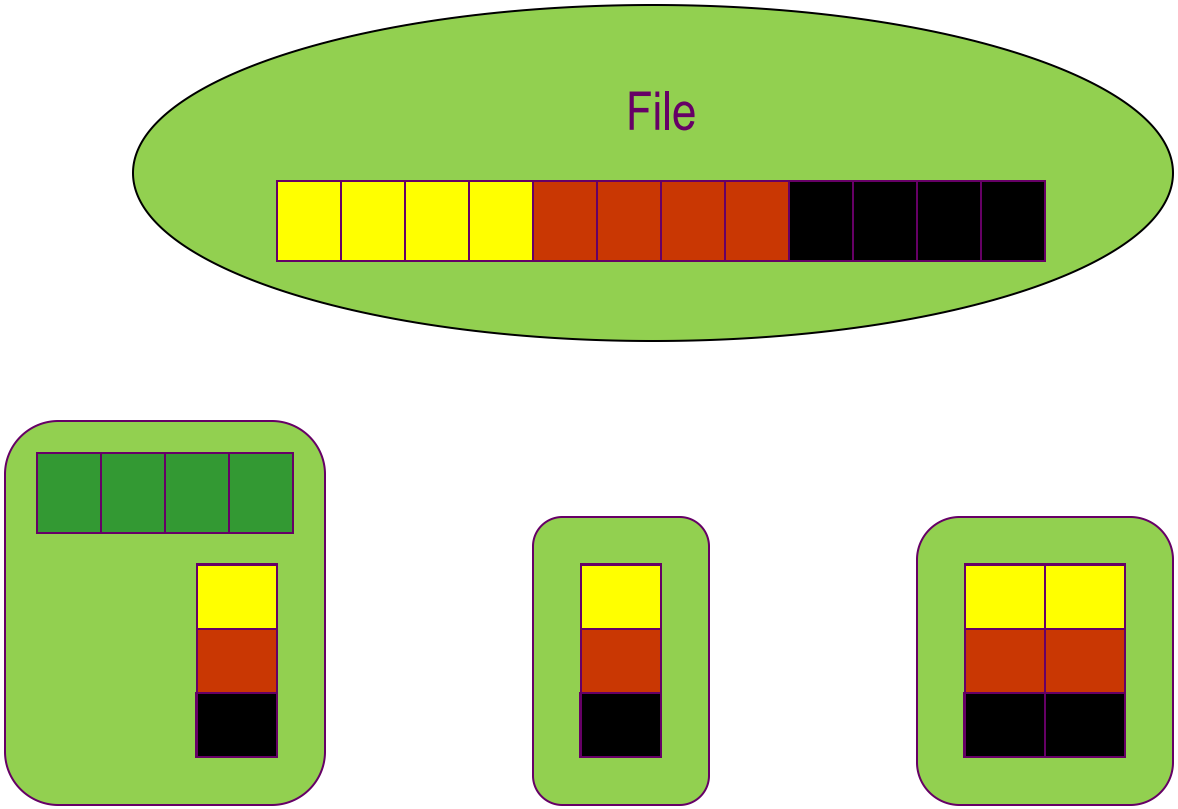
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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

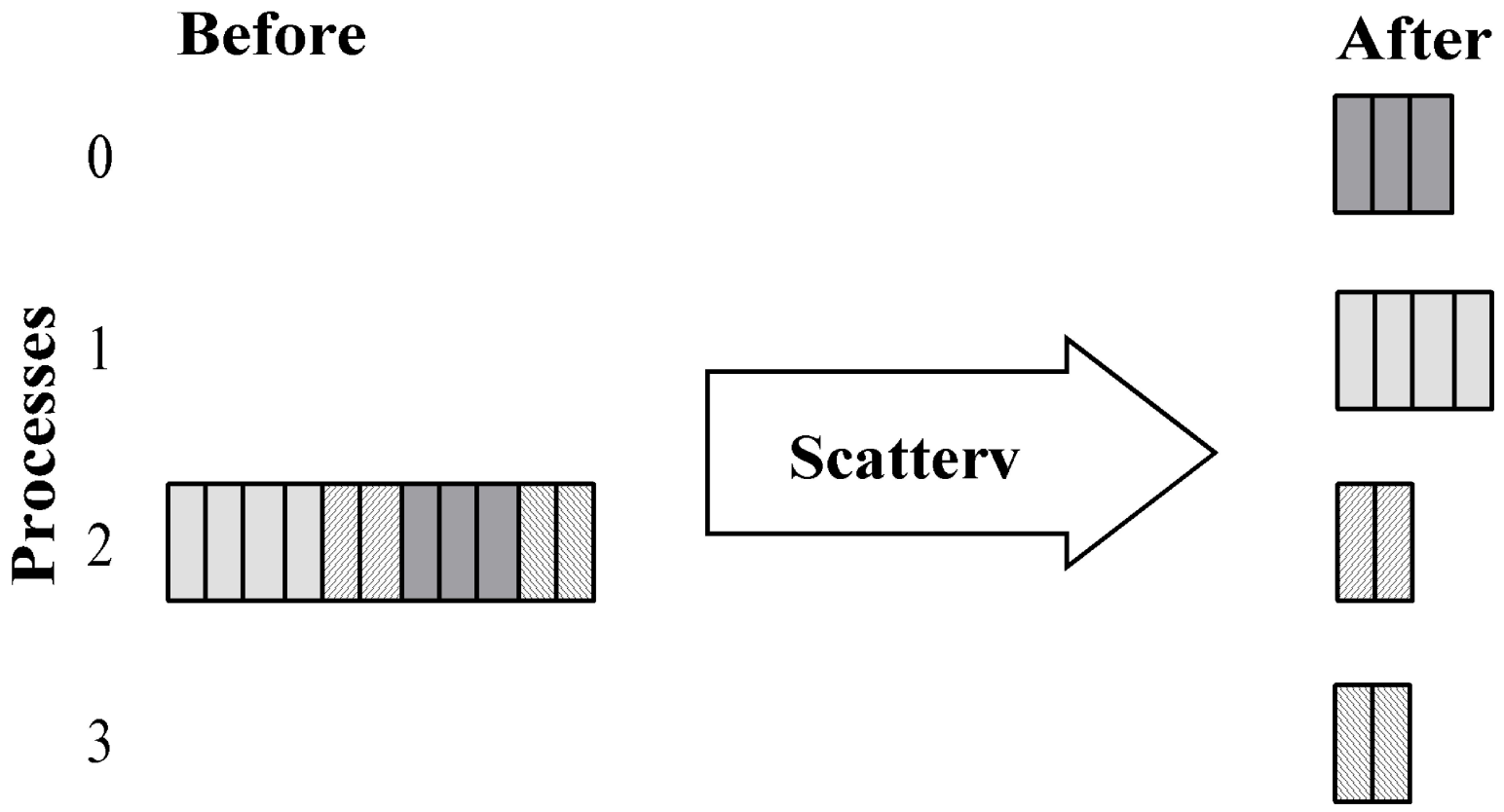
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Reading a Block-Column Matrix



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

MPI_Scatterv



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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)
```

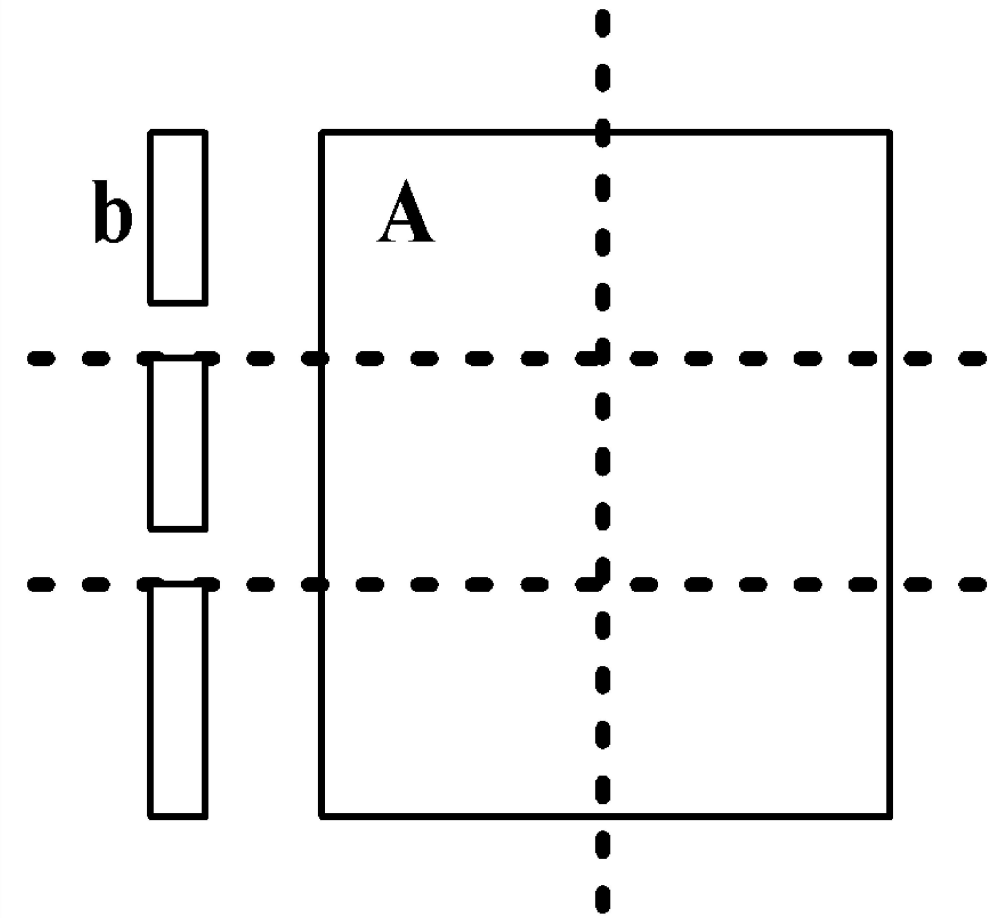
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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
- Processes form a 2-D grid
- Vector b distributed by blocks among processes in first column of grid

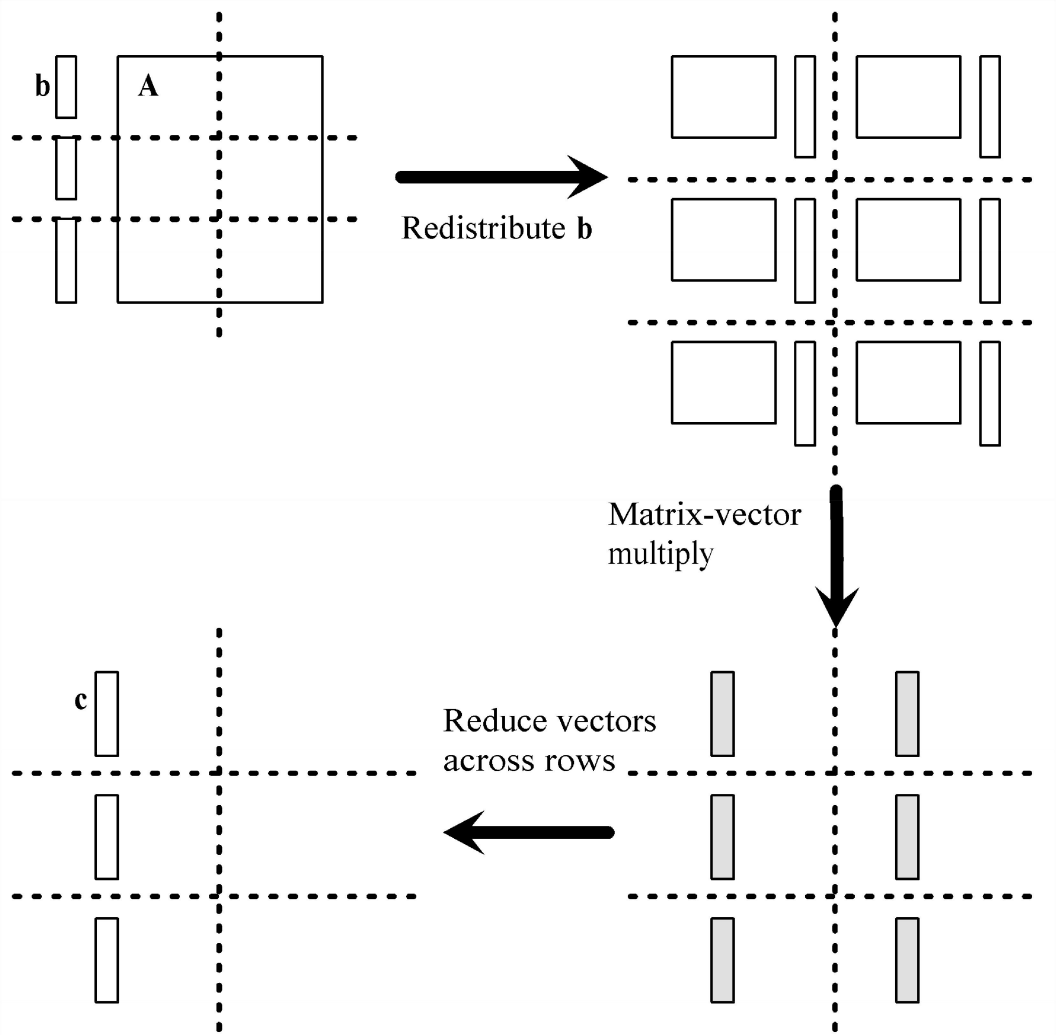
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Tasks after Agglomeration



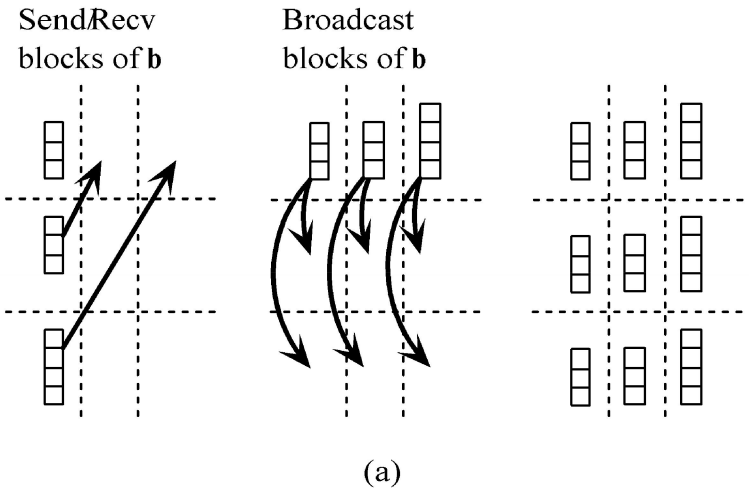
Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Algorithm's Phases



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

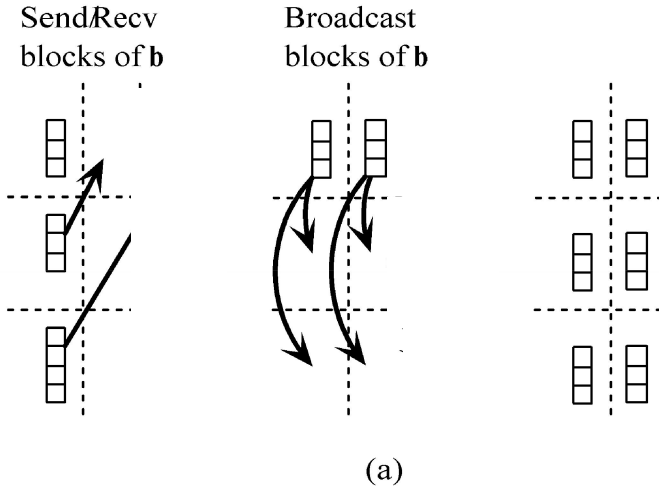
Redistributing Vector *b*



When *p* is a square number

When *p* is not a square number

Carefully



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

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**

Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

Summary

- Introduction to MPI
- Send and receive
- Communications in the DMVP example
  - Row-wise partition
  - Column-wise partition
  - Checkerboard partition