Distributed-Memory Programming via MPI

Message Passing Interface (MPI) Overview

MPI is one of the oldest libraries, began in Supercomputing'92

Portable: minimum requirements on hardware platforms

Works on Distributed Memory, Shared Memory, and Hybrid systems

The programming model remains distributed-memory

Consists of *p* processes, each with its own exclusive address space.

All parallelism is explicit

Intellectually demanding,

Achieves high performance

Scales to a large number of processors

p processes, each with its own exclusive address space

Data must be explicitly partitioned and placed.

MPI Programming Structure

Single Program Multiple Data (SPMD)

Essentially MIMD: the program has branched instructions for different parts of data.

Interactions (read/write) require cooperation of two processes

The process that has the data and the process that wants to access the data.

Asynchronous mode

All concurrent tasks execute asynchronously

Loosely synchronous mode

Tasks synchronize to perform interactions.

Other than these interactions, tasks execute asynchronously.

Features of MPI

Communicator

Communicators combine context and group for message security

Point-to-Point communication

Modes: blocking, non-blocking, buffered

Collective communication

Large number of built-in data movement routines

Allow user-defined collective operations

Application-oriented process topologies

Built-in support for grids and graphs (uses groups)

Error handling

Six Golden MPI Functions

MPI has 125 functions

MPI has 6 most frequently used functions

`MPI_Init()`	initializes MPI
`MPI_Finalize()`	terminates MPI
`MPI_Comm_size()`	determines the number of processes
`MPI_Comm_rank()`	determines the label of the calling process
`MPI_Send()`	sends a message
`MPI_Recv()`	receives a message

MPI Programs

`int MPI_Init(int *argc, char ***argv)`

Initialize the MPI execution. May be used to pass the command line arguments to all processes

`int MPI_Finalize()`

Terminates MPI execution environment

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

A simple Hello World program

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    printf("Hello world\n");
    MPI_Finalize();
    return 0;
}
```

Compile (Build Script):

```
mpicc -g -Wall -o mpi_hello0 mpi_hello0.c
```

Run:

```
mpirun -np 4 ./mpi_hello0  //(number of processes = 4)
or mpiexec -np 4 ./mpi_hello0
```

Output:

```
Hello world
Hello world
Hello world
Hello world
```

Basic Communicator

`int MPI_Comm_size(MPI_Comm comm, int *size)`

Obtain the size of the communicator (e.g., MPI_COMM_WORLD)

Communicator: a collection of processes that talk to each other

`int MPI_Comm_rank(MPI_Comm comm, int *rank)`

Obtain the rank of this process in the communicator

```
#include "mpi.h"
#include <stdio.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("Hello from process %d out of %d\n”,myrank, npes);
    MPI_Finalize();
    return 0;
}
```

Send and Receive Messages

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

The first three arguments determine the content of the message

The remaining arguments determine the destination of the message

Tag is used to distinguish messages

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long 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	

Receive Message

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

Suppose process q calls `MPI_Send` with

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Also suppose that process r calls `MPI_Recv` with

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Then the message sent by q with the above call to `MPI_Send` can be received by r with the call to `MPI_Recv` if

- `recv_comm = send_comm`,
- `recv_tag = send_tag`,
- `dest = r`, and
- `src = q`.
- If `recv_type = send_type` and `recv_buf_sz` $\geq$ `send_buf_sz`, then the message sent by q can be successfully received by r.

Wildcard Arguments in MPI_Recv()

`MPI_ANY_SOURCE`

can receive from any source (in the order in which processes finish)

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

`MPI_ANY_TAG`

can receive messages of any tag in the order they were sent

Only a receiver can use wildcards: "push" rather than "pull"

No wildcard for communicator

both senders and receivers must always specify communicators

When wildcard arguments are used

Use the following to retrieve message information

```
MPI_Status status;

MPI_Recv(…, &status);


/*Get the source*/
status.MPI_SOURCE

/*Get the tag*/
status.MPI_TAG

/*Get the amount of data sent in the message*/
MPI_Get_count(&status, recv_type, &count)
```

A bit more complicated Hello World

See eClass MPI Program:

mpi_hello.c

Non-Buffered Blocking Message Passing

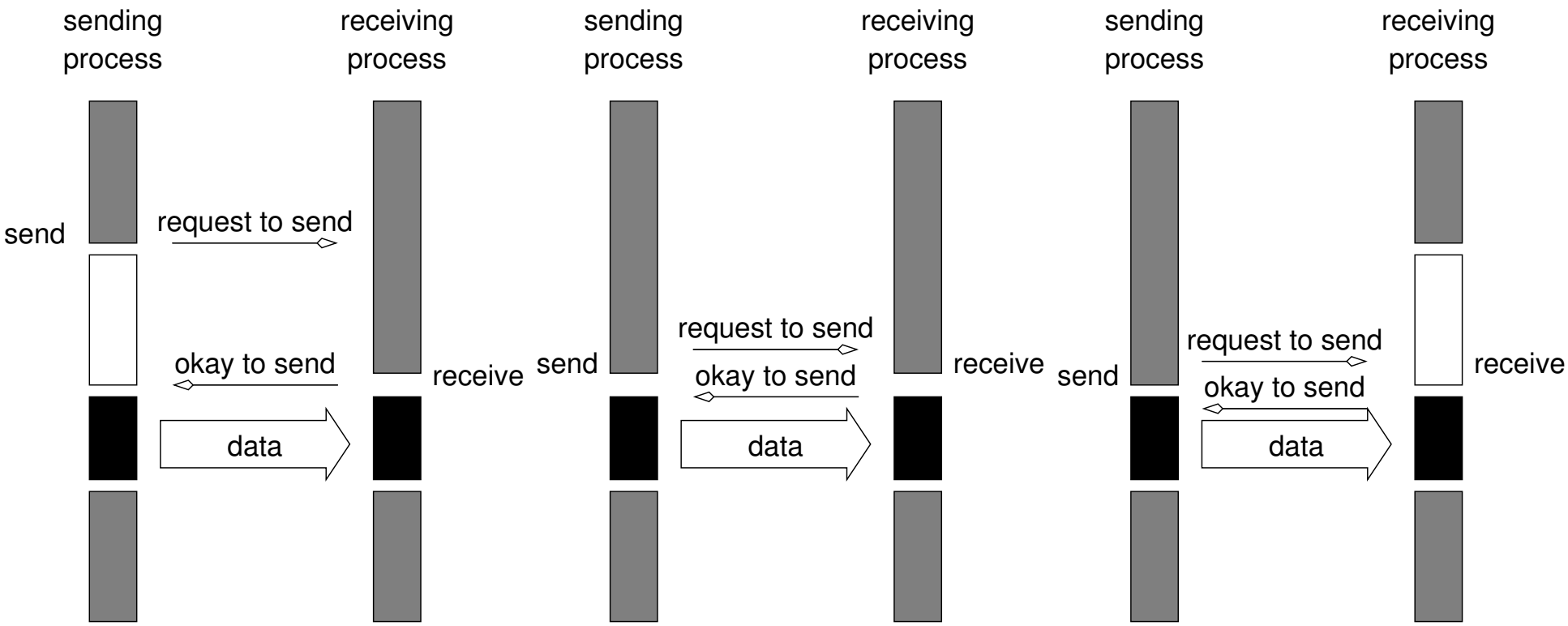
The first implementation of MPI_Send() and MPI_Recv()

Send blocks until the matching receive has been encountered at the receiving process.

Recv blocks if the send has not been performed.

Idling is a major issue with non-buffered blocking sends.

Non-Buffered Blocking Message Passing



(a) Sender comes first; idling at sender

(b) Sender and receiver come at about the same time; idling minimized

(c) Receiver comes first; idling at receiver

When sender and receiver do not reach the communication point at the same time, there can be lots of idling overheads.

Buffered Blocking Message Passing

The second implementation of MPI_Send () and MPI_Recv()

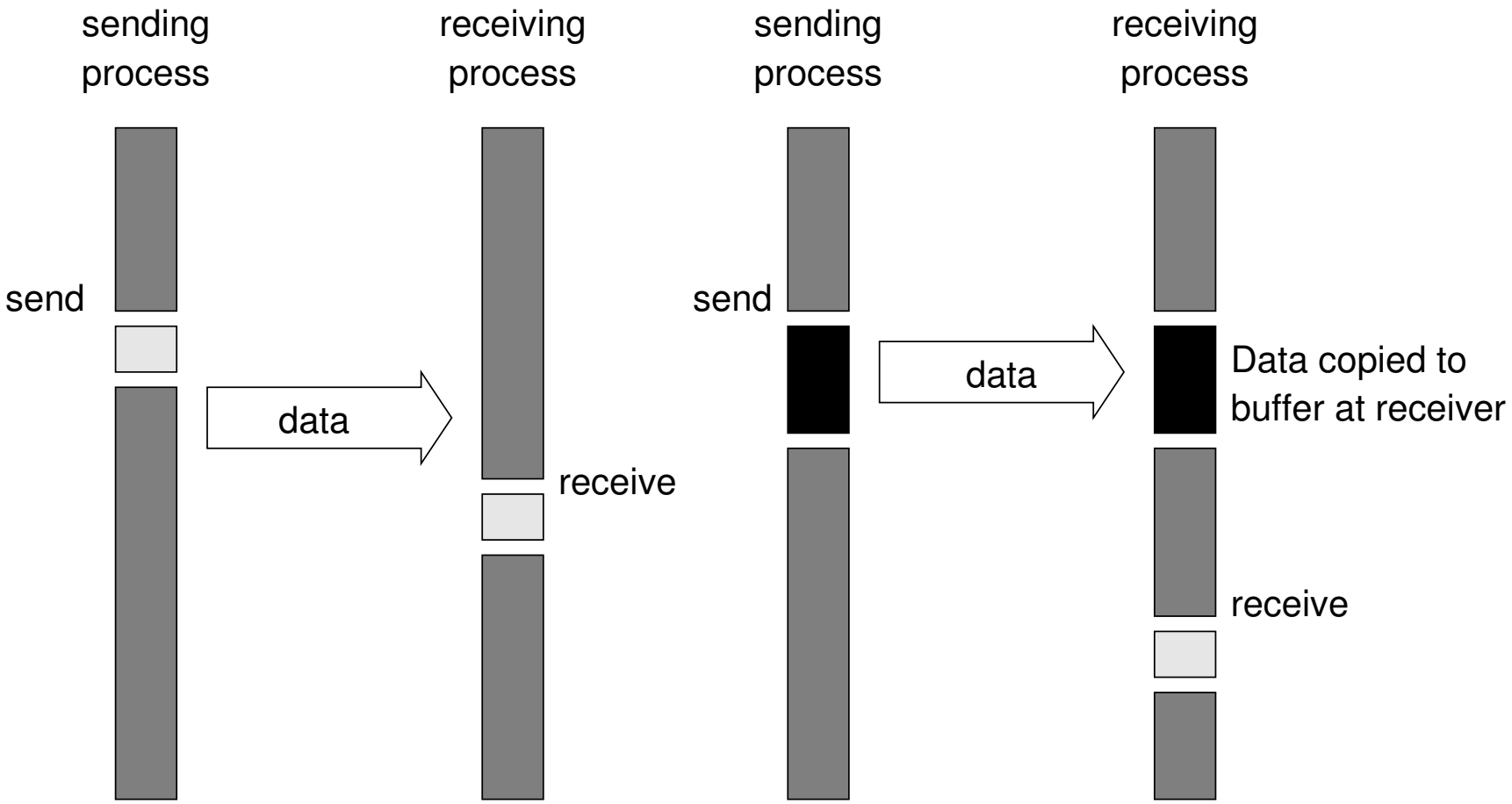
Rely on buffers at the sending and receiving ends to reduce idling overhead

If buffers are available at both the sender and receiver

- The sender copies the data into the designated buffer and returns after the copy is done.
- The data must be buffered at the receiving end.

Buffering trades off idling overhead for buffer copying overhead

Buffered Blocking Message Passing



a) Buffers present at both sender and receiver;

b) Buffer only present at receiver: sender interrupts receiver and places data in the buffer at receiver.

Buffered Blocking Message Passing

Bounded buffer size **can have significant impact on performance**

What if consumer was much slower than producer?

```
P0

for (i = 0; i < 1000; i++) {
  produce_data(&a);
  send(&a, 1, 1);
}
```

```
P1

for (i = 0; i < 1000; i++) {
  receive(&a, 1, 0);
  consume_data(&a);
}
```

Do not assume the buffer size is infinite in your program.

Deadlocks in Blocking Message Passing

Deadlock in Non-Buffered Block Message Passing

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Deadlock in Buffered Block Message Passing

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

```
P1

receive(&a, 1, 0);
send(&b, 1, 0);
```

Deadlocks in Blocking Message Passing

Is this program safe in buffered/non-buffered cases?

```
int a[10], b[10], myrank;
MPI_Status status;
...
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...
```

Safe in the buffered case, unsafe in non-buffered case

The code is implementation dependent!

Programmer needs to make sure code is correct on all implementations of blocking message passing

Deadlocks in Blocking Message Passing

Is this program (circular communication) safe in buffered/non-buffered cases?

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
...
```

Safe with buffer, unsafe without buffer

Deadlocks in Blocking Message Passing

Revised program: safe circular communication in both buffered/non-buffered cases

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
 MPI_Comm_size(MPI_COMM_WORLD, &npes);
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
}
else {
   MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1, MPI_COMM_WORLD);
   MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1, MPI_COMM_WORLD);
}
...
```

Send and Receive Simultaneously

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
        MPI_Datatype senddatatype, int dest, int sendtag,
        void *recvbuf, int recvcount, MPI_Datatype recvdatatype,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Send and recv buffers must be disjoint

Arguments combine those from MPI_Send() and MPI_Recv()

Thus, another safe version of circular communication is

```
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
             b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD, &status);
...
```

Send and Receive Simultaneously

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Execute a blocking send and receive.

The same buffer (void * buf) is used both for the send and for the receive

The message sent is replaced by the message received

Can further save the memory

Example: Trapezoidal Rule Calculation

See eClass for how to collect input arguments

mpi_trap1.c

mpi_trap2.c

Collective Communication and Computation

MPI provides an extensive set of functions for performing common collective communication operations.

Each of these operations is defined over a group of processes in the corresponding communicator

All processes in a communicator MUST call these operations

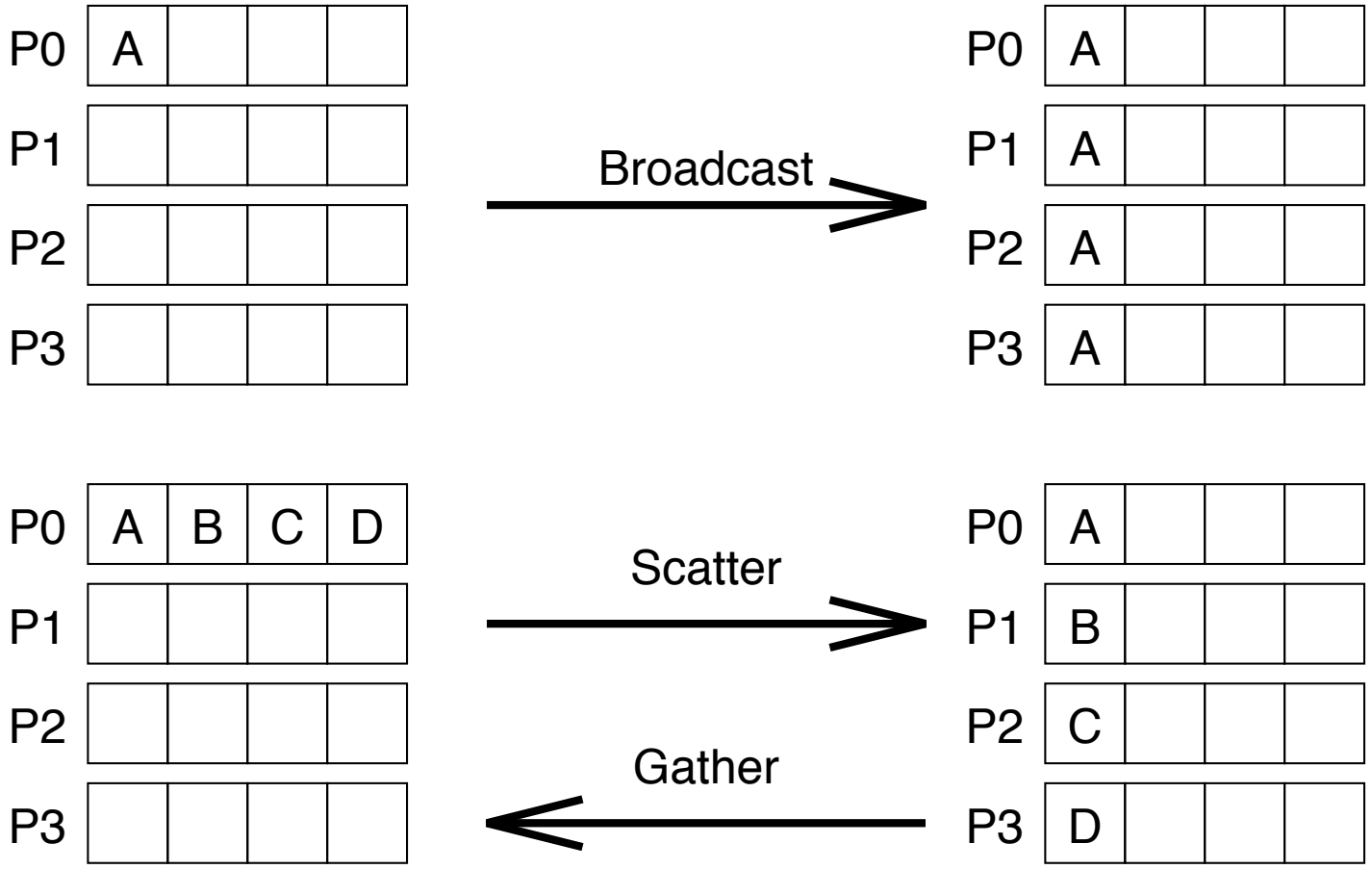
Virtual synchronization in collective communication

Optimized implementation and message scheduling

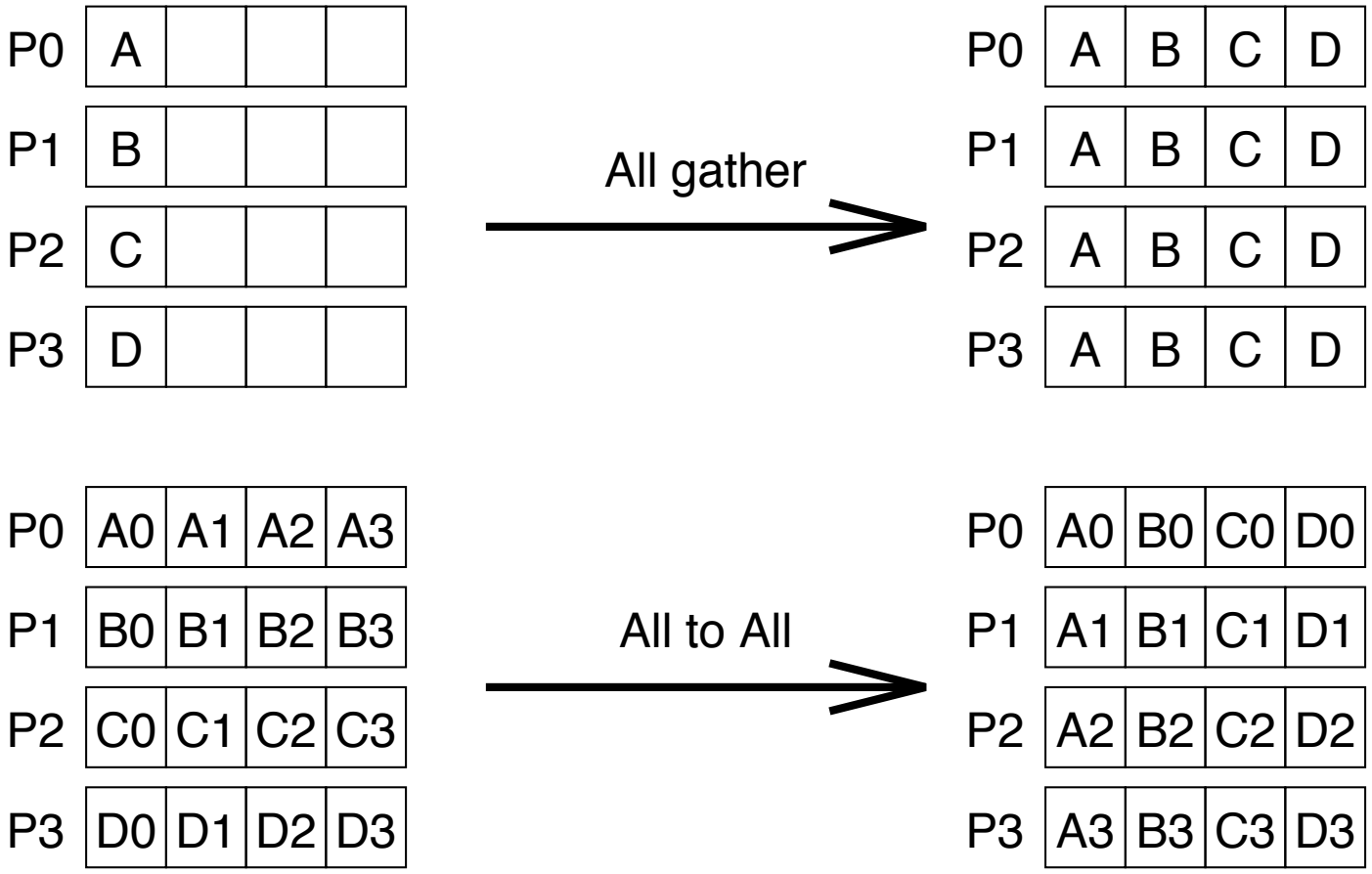
Three classes of collective operations

- Synchronization
- Data movement
- Collective computation

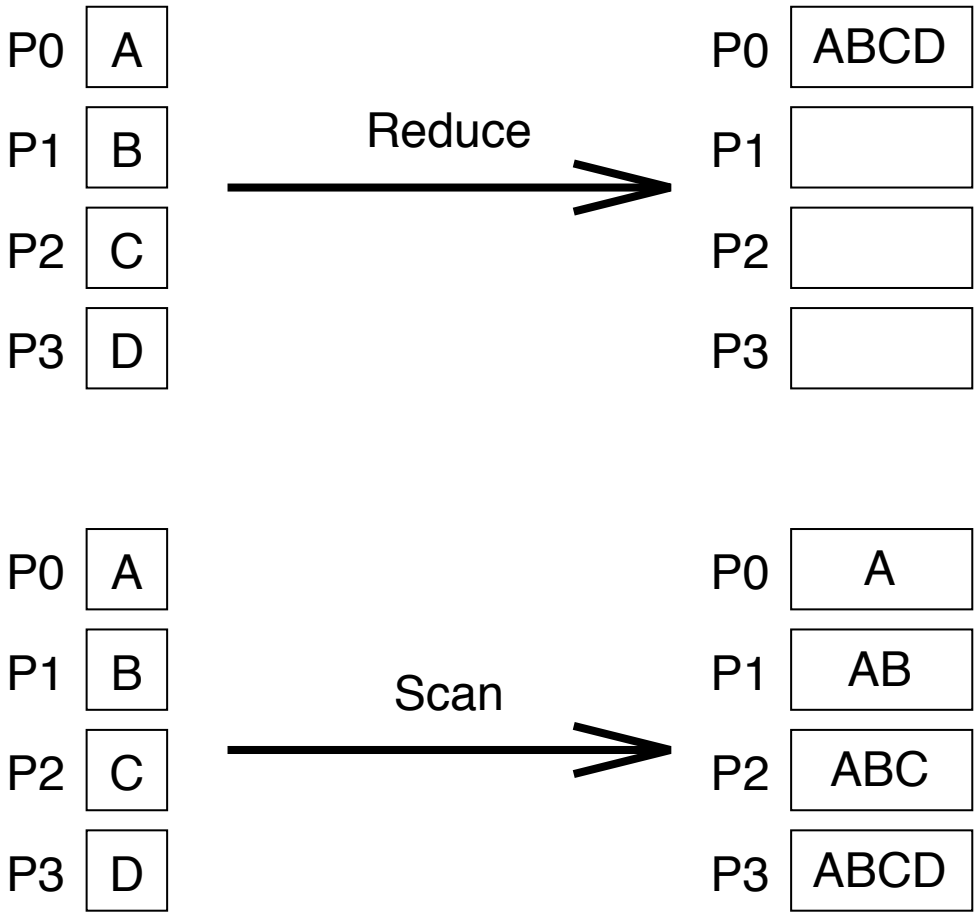
Schematic View of Collective Communication



Schematic View of Collective Communication



Schematic View of Collective Computation



Collective Communication

The barrier synchronization operation is performed in MPI

```
int MPI_Barrier(MPI_Comm comm)
```

The one-to-all broadcast operation is:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype
    datatype,int source, MPI_Comm comm)
```

The all-to-one reduction operation is:

```
int MPI_Reduce(void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    int target, MPI_Comm comm)
```

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value-location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

Collective Communication

If the result of the reduction operation is needed by all processes, MPI provides:

```
int MPI_Allreduce(void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op,
    MPI_Comm comm)
```

To compute prefix-sums, MPI provides:

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

The gather operation is performed in MPI using:

```
int MPI_Gather(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf, int
    recvcount, MPI_Datatype recvdatatype, int target,
    MPI_Comm comm)
```

Collective Communication

MPI also provides the MPI_Allgather function in which the data are gathered at all the processes

```
int MPI_Allgather(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf, int
    recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
```

The corresponding scatter operation is:

```
int MPI_Scatter(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf,
    int recvcount, MPI_Datatype recvdatatype, int source,
    MPI_Comm comm)
```

The all-to-all personalized communication operation is performed by:

```
int MPI_Alltoall(void *sendbuf, int sendcount,
    MPI_Datatype senddatatype, void *recvbuf, int
    recvcount, MPI_Datatype recvdatatype, MPI_Comm comm)
```

Many More Collective Routines

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
ReduceScatter	Scan	Scatter
Scatterv		

All versions: deliver results to all participating processes

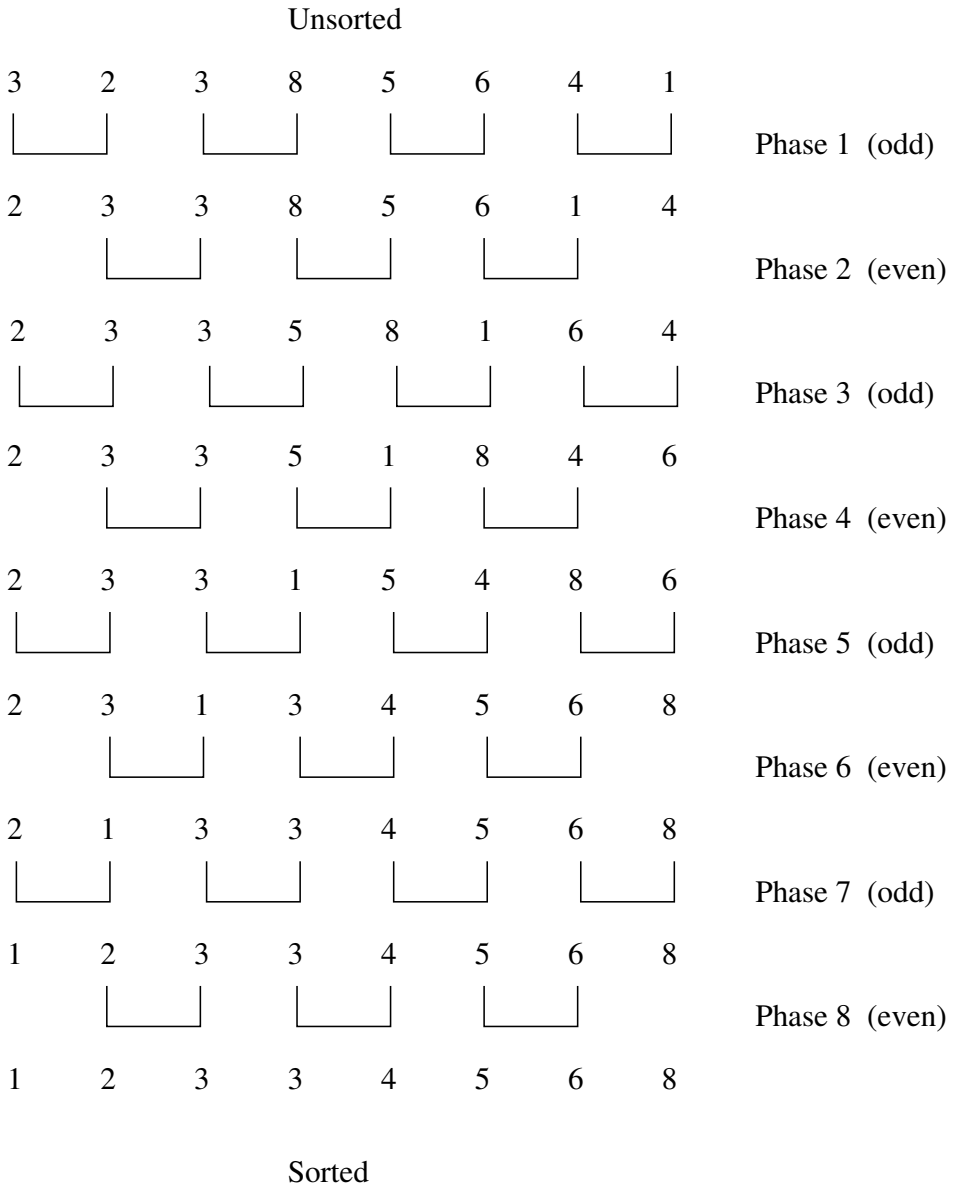
V: vector versions allow the chunks to have different sizes

See eClass for Trapezoidal integral with collective communication

- mpi_trap3.c
- mpi_trap4.c (optional)

Granularity and Cost Analysis

Example: Odd-Even Transposition Sort



Odd-Even Transposition Sort

The Sequential Algorithm for *n* items

```
procedure ODD-EVEN(n)
begin
   for i := 1 to n do
   begin
      if i is odd then
           for j := 0 to n/2 - 1 do
               compare-exchange(a2j + 1, a2j + 2);
      if i is even then
           for j := 1 to n/2 - 1 do
               compare-exchange(a2j, a2j + 1);
   end for
end ODD-EVEN
```

After *n* phases, the sorting is completed (proved).

Parallel Odd-Even Transposition

First version:

consider *n* processes on a ring topology,

each element is on a different process

```
procedure ODD-EVEN_PAR(n)
begin
    id := process's label
    for i := 1 to n do
    begin
        if i is odd then
            if id is odd then
                compare-exchange_min(id + 1);
            else
                compare-exchange_max(id − 1);
        if i is even then
            if id is even then
                compare-exchange_min(id + 1);
            else
                compare-exchange_max(id − 1);
    end for
end ODD-EVEN_PAR
```

Parallel Odd-Even Transposition

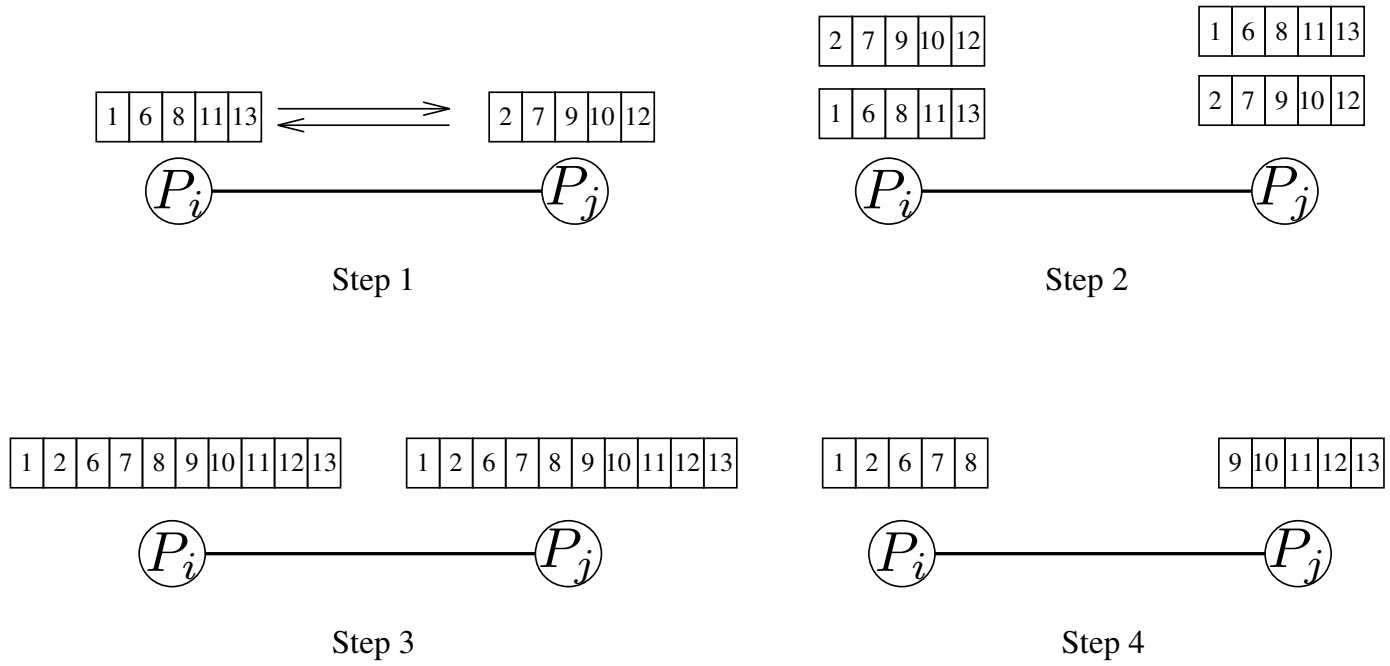
Second version:

Consider *n/p* elements per process

The first step is a local sort of *n/p* elements in each process

In each phase, do a compare-split operation with a neighbour process

Sorted guarantee after *p* phases



A Compare-Split Operation

Parallel Odd-Even Transposition

See the MPI source code on eClass

Which version is faster on a computer of 4 cores?

Given a problem at hand (given input size *n*)

- How to determine the best number of processes to be used?
- How to determine how many processing elements (e.g., cores) to employ?

In other words, what's the best "Granularity" in this program?

Cost Analysis of Parallel Programs

Sources of overhead in parallel programs

- Interprocess interaction/communication
- Idling (due to load imbalance, synchronization and serial parts)
- Excess Computation Overhead
  - The fastest known sequential algorithm may be impossible to parallelize
  - Forces us to use another algorithm for easy parallelization
  - The difference in computation performed by the parallel program and the best serial program is the *excess computation overhead* incurred by the parallel program
  - The parallelization of a serial algorithm may also incur more aggregate computation than the serial algorithm itself.

Speedup and Efficiency (Review)

Let T_1 be the time to solve the problem sequentially.

Let T_p be the time to solve the problem in parallel using p processing elements

Then, speedup $S(p)$ for problem size n is defined as:

$$S(p) = T_1/T_p$$

The overhead is defined by

$$T_p = T_1/p + T_{\text{overhead}}$$

T_1 represents the amount of work

Efficiency: $E(p) = S(p)/p$

Total Parallel Overhead

Let T_S be the optimal serial time to solve the problem sequentially.

Let T_P be the time to solve the problem in parallel (probably using a different algorithm) with p processing elements

The Total Parallel Overhead is

$$T_O = pT_P - T_S$$

which is the total time spent in solving a problem summed over all processing elements minus the necessary total time spent on useful work.

Cost of a Parallel System

The cost of solving a problem on a parallel system using *p* processing elements (processes) is

$$\text{cost} = pT_P$$

Cost is also referred to as ***work*** or ***processor-time product***. But cost has nothing to do with the number of processors used. (Why?)

The *serial cost* is the execution time of the fastest known sequential algorithm on a single processing element.

Cost-Optimal

A parallel system is said to be *cost-optimal* if the cost of solving a problem on this parallel system (as a function of the input size *n*) is *asymptotically* identical to the serial cost

Since efficiency $E = T_S / pT_P$, for cost-optimal systems $E = \Theta(1)$

Cost of the Odd-Even Transposition Sorts

The first version (one element per process)

Θ(1) run time per phase for parallel compare-exchange

There are n phases in total

Thus, the total parallel run time is $\Theta(n)$

The cost is $\Theta(n^2) = \Theta(n) \times n$

As compared to the best sequential sort $\Theta(n \log n)$ (MergeSort), the first version is NOT cost-optimal!

How the cost analysis is useful?

For example, on a single machine with p=4 cores

The time it takes to run this parallel algorithm is $\Theta(n^2)/4$

The first version is a special case of the second version

Cost of the Odd-Even Transposition Sorts

The second version (*n*/*p* elements per process)

$$T_P = \overbrace{\Theta\left(\frac{n}{p}\log\frac{n}{p}\right)}^{\text{local sort}} + \overbrace{\Theta(n)}^{\text{comparisons}} + \overbrace{\Theta(n).}^{\text{communication}}$$

Cost is $\Theta\left(n\log\frac{n}{p}\right) + \Theta(np)$

Efficiency is $E = \frac{1}{1 - \Theta(\frac{\log p}{\log n}) + \Theta(\frac{p}{\log n})}$

Odd-even transposition sort is cost-optimal when *p* = *O*(log*n*)!

e.g., on 4 cores, the times to run the parallel program is Θ(*n* log*n*)/4

Poor scalability: can only benefit from a small number of processors

Partitioning

Partitioning

Example: Matrix-Vector Multiplication

Due to their regular structure, parallel computations involving matrices and vectors readily lend themselves to data decomposition.

Typical algorithms rely on input, output, or intermediate data decomposition

Most algorithms use one- and two-dimensional block, cyclic, and block-cyclic partitionings

We aim to multiply a dense $n \times n$ matrix A with an $n \times 1$ vector x to yield the $n \times 1$ result vector y.

The serial algorithm requires n^2 multiplications and additions.

$$T_s = n^2$$

Rowwise 1-D Partitioning

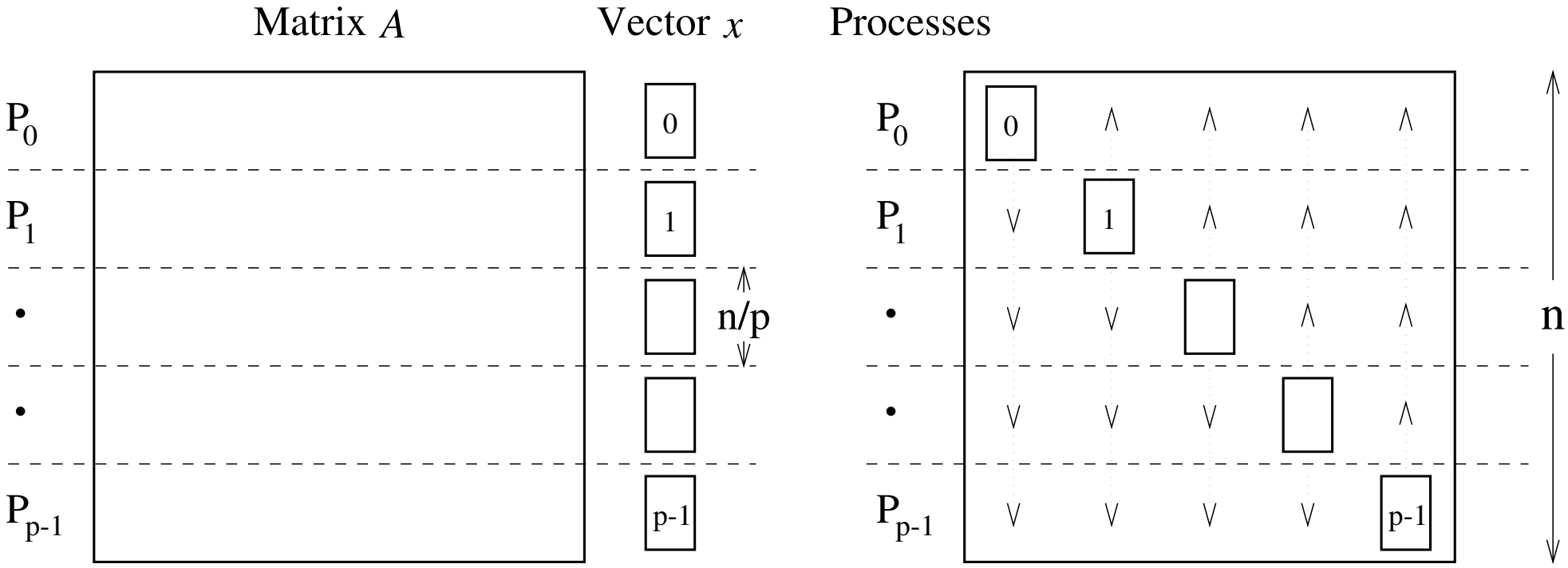
The $n \times n$ matrix A is partitioned among p processes, with each process storing a n/p rows of the matrix.

The $n \times 1$ vector x is distributed such that each process owns n/p elements.

For the special case of one row per process, $n = p$.

p adjusts the granularity

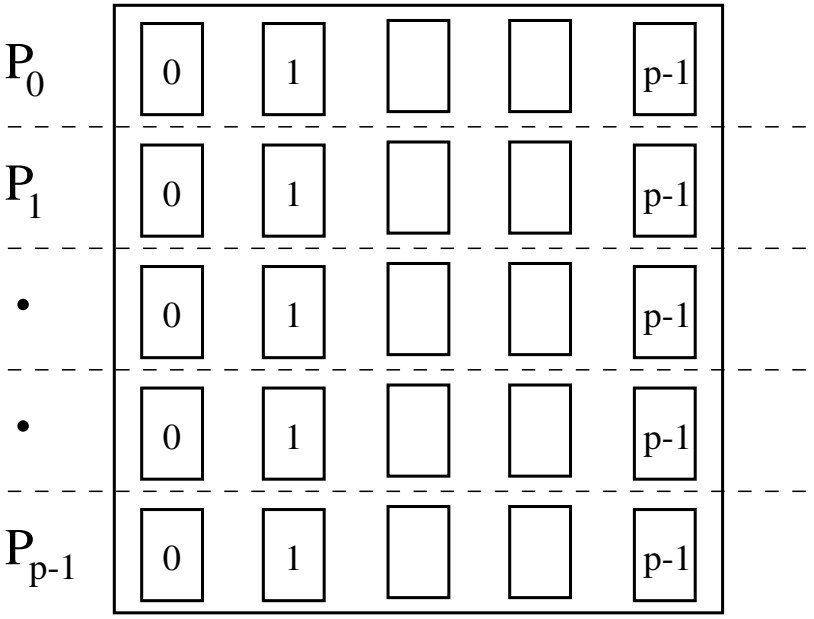
Rowwise 1-D Partitioning



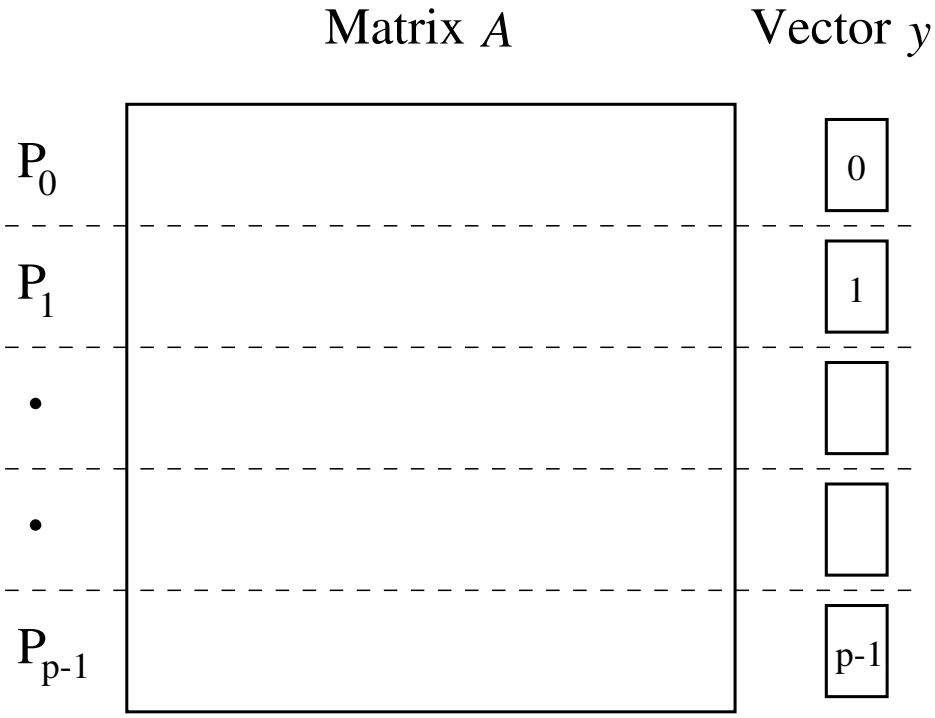
(a) Initial partitioning of the matrix and the starting vector x

(b) Distribution of the full vector among all the processes by all-to-all broadcast

Rowwise 1-D Partitioning



(c) Entire vector distributed to each process after the broadcast



(d) Final distribution of the matrix and the result vector y

Rowwise 1-D Partitioning

When $p = n$ (one row per process)

Since each process starts with only one element of x, an all-to-all broadcast is required to distribute all the elements to all the processes.

Process P_i now computes $y[i] = \Sigma_j A[i, j] \times x[j]$

The all-to-all broadcast (using circular shift) takes time $\Theta(n)$. Why?

The computation of $y[i]$ also takes time $\Theta(n)$.

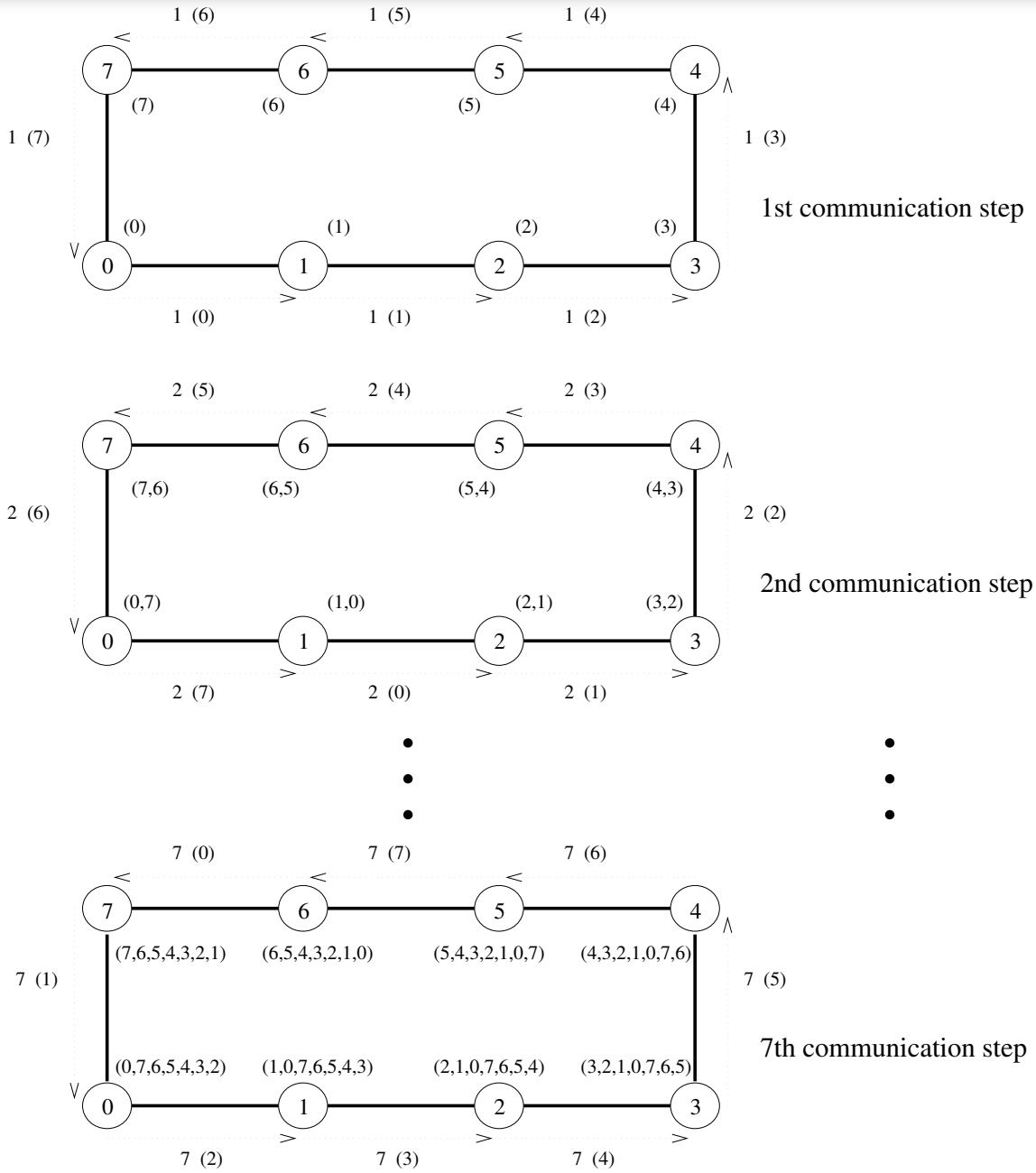
Therefore, the parallel execution time is $\Theta(n)$.

The cost is $\Theta(n^2)$.

Is it cost optimal?

All-to-All Broadcast: First Algorithm

All-to-all broadcast on a ring (takes *p*-1 steps)



Rowwise 1-D Partitioning

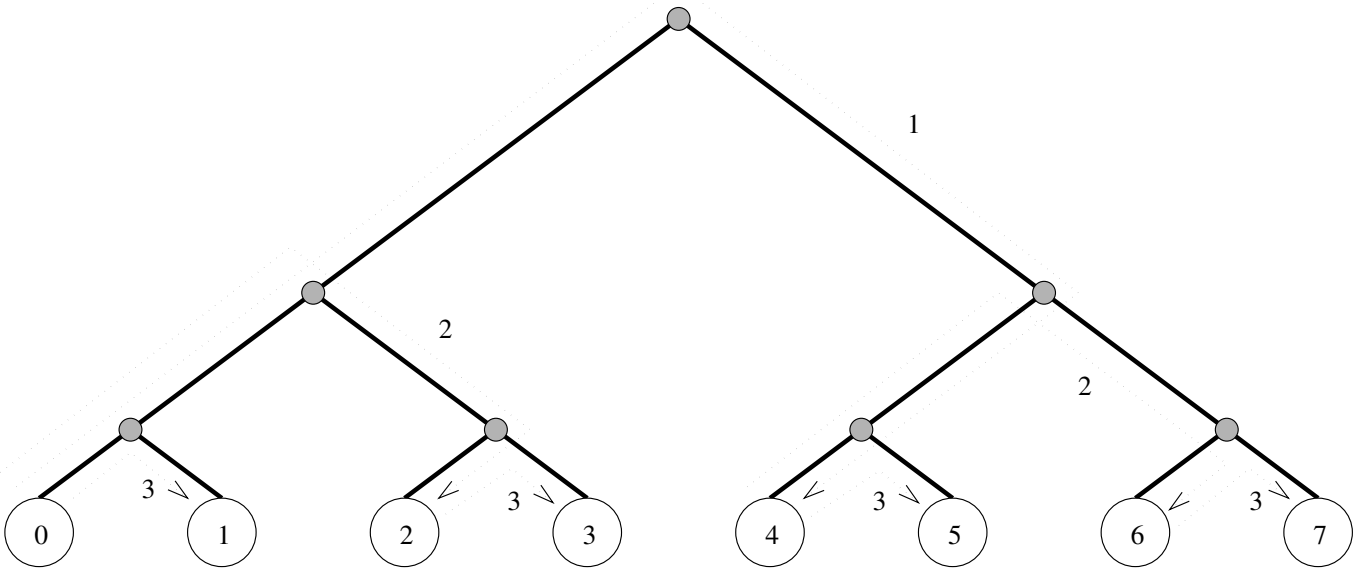
Consider now the case when $p < n$ (multiple rows per process)

Each process initially stores n/p rows of the matrix and a portion of the vector of size n/p

The all-to-all broadcast takes place among p processes and involves messages of size n/p

What is the optimal all-to-all broadcast time here?

One-to-All Broadcast



One-to-all broadcast on an eight-node tree.

The broadcast or reduction procedure involves $\log p$ point-to-point simple message transfers, each taking time $t_s + t_w m$

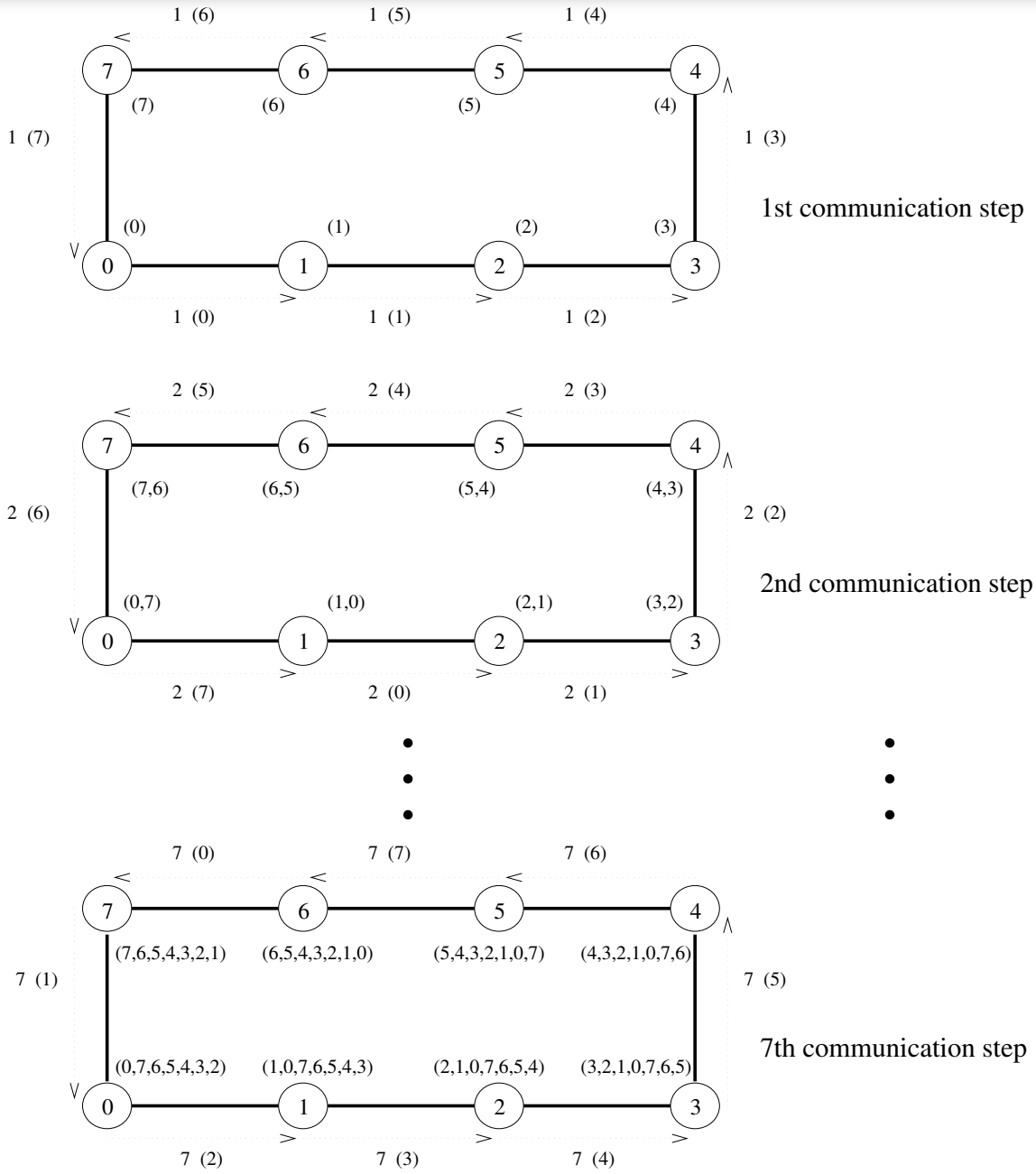
t_s the transfer startup time

t_w the per-word transfer time

m the message size in terms of words

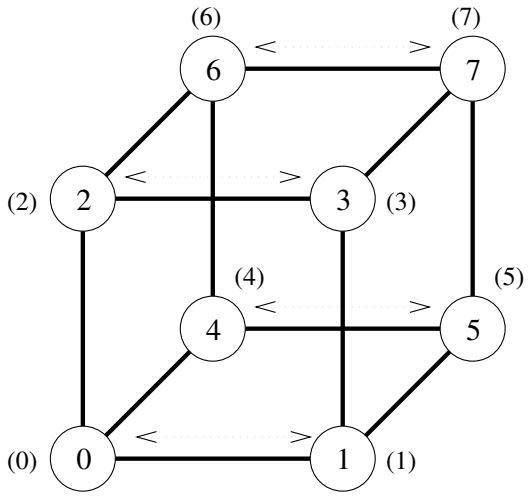
All-to-All Broadcast: First Algorithm

All-to-all broadcast on a ring (takes *p*-1 steps)

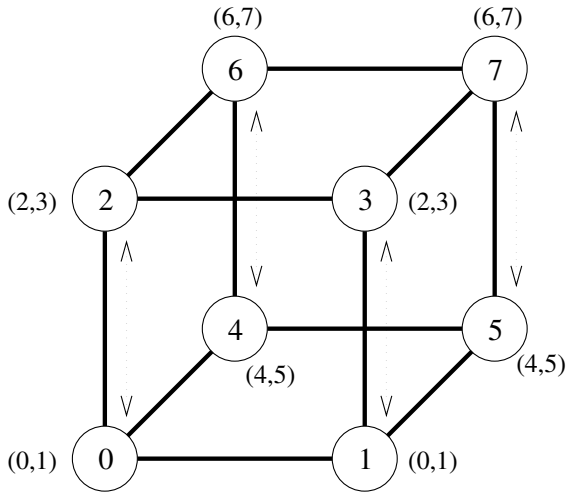


All-to-All Broadcast: Optimal Algorithm

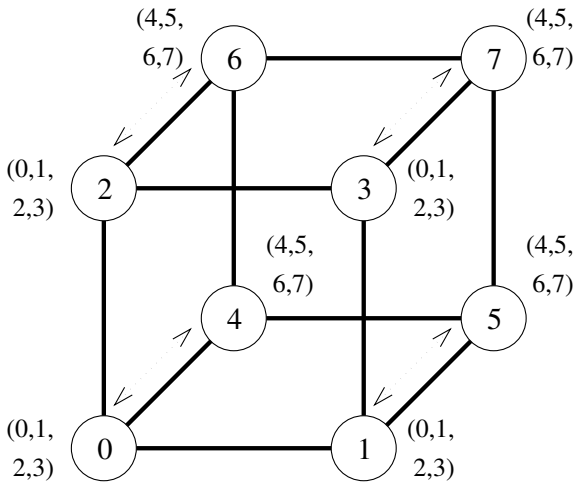
All-to-all broadcast on a hypercube



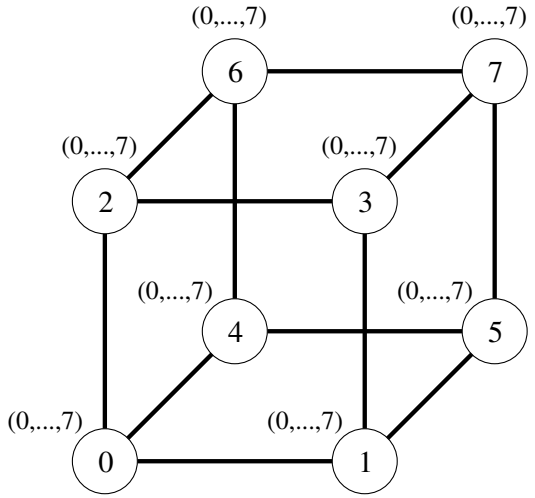
(a) Initial distribution of messages



(b) Distribution before the second step



(c) Distribution before the third step



(d) Final distribution of messages

All-to-All Broadcast: Cost Analysis

On a ring, the time is given by: $(t_s + t_w m)(p-1)$.

On a hypercube, we have:

$$\begin{aligned} T &= \sum_{i=1}^{\log p} (t_s + 2^{i-1} t_w m) \\ &= t_s \log p + t_w m (p-1). \end{aligned}$$

Rowwise 1-D Partitioning

Continue with the case when $p < n$ (multiple rows per process)

Each process initially stores n/p rows of the matrix and a portion of the vector of size n/p

The **all-to-all broadcast** of messages of size n/p among p processes takes time

$$t_s \log p + t_w(n/p)(p-1) \approx t_s \log p + t_w n$$

This is followed by n^2/p multiplication-additions at each process

Therefore, the parallel run time of this procedure is

$$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

Is it cost-optimal?

It is cost-optimal for $p = O(n)$!

1-D Partitioning Implementation

```
1   RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
2                           MPI_Comm comm)
3   {
4     int i, j;
5     int nlocal;         /* Number of locally stored rows of A */
6     double *fb;         /* Will point to a buffer that stores the entire vector b
7     int npes, myrank;
8     MPI_Status status;
9
10    /* Get information about the communicator */
11    MPI_Comm_size(comm, &npes);
12    MPI_Comm_rank(comm, &myrank);
13
14    /* Allocate the memory that will store the entire vector b */
15    fb = (double *)malloc(n*sizeof(double));
16
17    nlocal = n/npes;
18
19    /* Gather the entire vector b on each processor using MPI's ALLGATHER operat
20    MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
21        comm);
22
23    /* Perform the matrix-vector multiplication involving the locally stored sub
24    for (i=0; i<nlocal; i++) {
25      x[i] = 0.0;
26      for (j=0; j<n; j++)
27        x[i] += a[i*n+j]*fb[j];
28    }
29
30    free(fb);
31  }
```

Matrix-Vector Multiplication: 2-D Partitioning

The Special Case

The $n \times n$ matrix is partitioned among n^2 processes such that each process owns a single element.

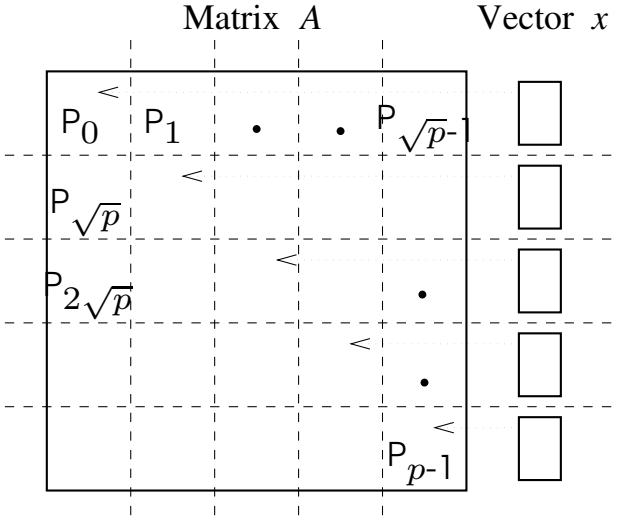
The $n \times 1$ vector x is distributed only in the last column of n processes.

The General Case

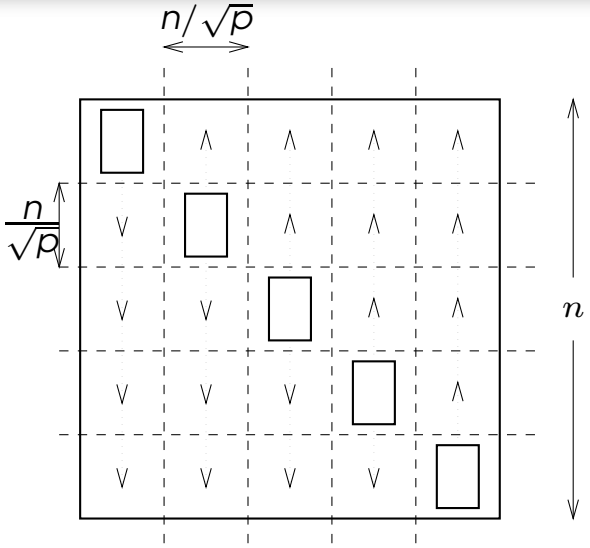
The $n \times n$ matrix is partitioned among p processes such that each process owns a sub matrix of $n \times n/p$ elements.

The $n \times 1$ vector x is distributed only in the last column of processes.

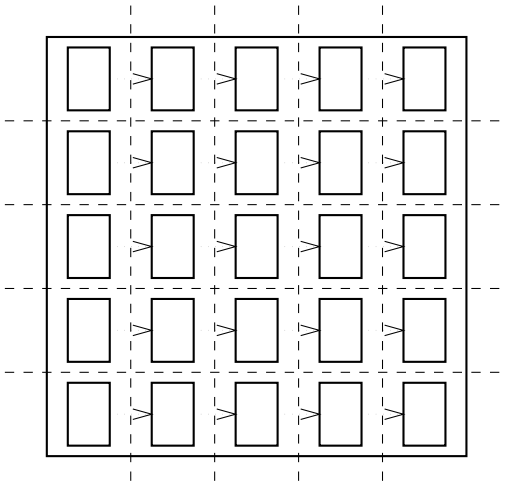
Matrix-Vector Multiplication: 2-D Partitioning



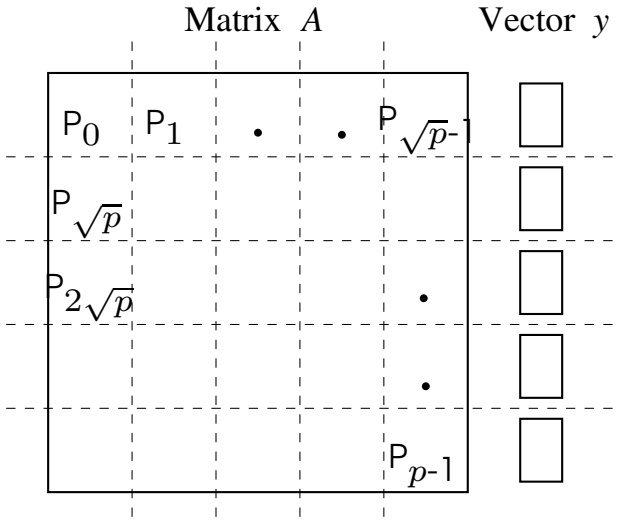
(a) Initial data distribution and communication steps to align the vector along the diagonal



(b) One-to-all broadcast of portions of the vector along process columns



(c) All-to-one reduction of partial results



(d) Final distribution of the result vector

Matrix-Vector Multiplication: 2-D Partitioning

We must first aling the vector with the matrix appropriately.

The first communication step for the 2-D partitioning aligns the vector x along the principal diagonal of the matrix.

The second step copies the vector elements from each diagonal process to all the processes in the corresponding column using n simultaneous broadcasts among all processors in the column.

Finally, the result vector is computed by performing an all-to-one reduction along the columns.

Matrix-Vector Multiplication: 2-D Partitioning

When using fewer than n^2 processors, each process owns an $(n/\sqrt{p}) \times (n/\sqrt{p})$ block of the matrix.

The vector is distributed in portions of $n/\sqrt{p}$ elements in the last process-column only.

In this case, the message sizes for the alignment, broadcast, and reduction are all $(n/\sqrt{p})$.

The computation is a product of an $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrix with a vector of length $(n/\sqrt{p})$.

Matrix-Vector Multiplication: 2-D Partitioning

The first alignment step takes time $t_s + t_w n/\sqrt{p}$.

The broadcast and reductions take time $(t_s + t_w n/\sqrt{p}) \log(\sqrt{p})$.

Local matrix-vector products take time $t_c n^2/p$.

Total time is

$$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

Comparison of 1-D and 2-D Partitionings

1-D $$T_P = \frac{n^2}{p} + t_s \log p + t_w n.$$

2-D $$T_P \approx \frac{n^2}{p} + t_s \log p + t_w \frac{n}{\sqrt{p}} \log p$$

Take-Away Messages:

2-D partitioning is faster than 1-D for the same number of processes.

If the number of processes is less than or equal to *n*, 2-D is preferable due to better efficiency

If the number of processes is greater than *n*, then 1-D cannot be used, but 2-D can still be used.

Topologies and Embeddings

MPI allows a programmer to organize processes into logical *k*-dimensional meshes.

The process ids in `MPI_COMM_WORLD` can be mapped to other communicators (corresponding to higher-dimensional meshes) in many ways.

The goodness of any such mapping is determined by the interaction pattern of the underlying program and the topology of the machines.

MPI can optimize these mappings.

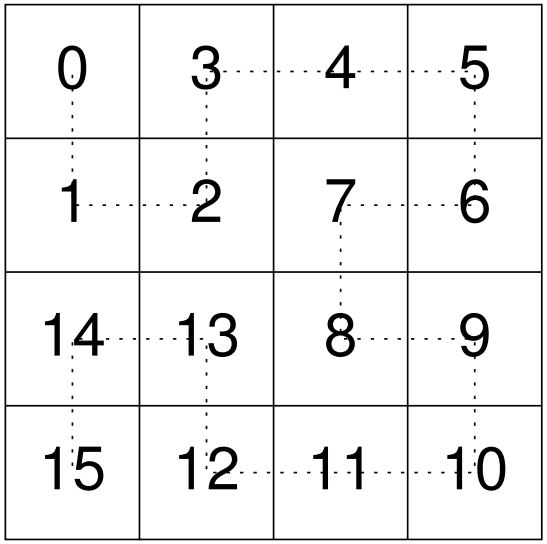
Topologies and Embeddings

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a) Row–major mapping

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(b) Column–major mapping



(c) Space–filling curve mapping

Different ways to map a set of processes to a grid. (a) and (b) show a row- and column-wise mapping, (c) shows a mapping that follows a space-filling curve (dotted line)

Creating and Using Cartesian Topologies

We can create cartesian topologies using the function:

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims,
int *dims, int *periods, int reorder, MPI_Comm
  *comm_cart)
```

Takes the processes in the old communicator and creates a new communicator `comm_cart` with `ndims` dimensions

Each process can now be identified in this new cartesian topology `comm_cart` of a size `dims`

`reorder:` **if true, the system re-determines the rank of the process in the new communicator**

Creating and Using Cartesian Topologies

Since sending and receiving messages still require (one-dimensional) ranks, MPI provides routines to convert ranks to cartesian coordinates and vice-versa.

```
int MPI_Cart_coords(MPI_Comm comm_cart, int rank, int maxdims,
  int *coords)
```

```
int MPI_Cart_rank(MPI_Comm comm_cart, int *coords, int *rank)
```

The most common operation on cartesian topologies is a shift.

To determine the rank of source and destination of such shifts, MPI provides the following function:

```
int MPI_Cart_shift(MPI_Comm comm_cart, int dir, int
s_step, int *rank_source, int *rank_dest)
```

Splitting Cartesian Topologies

Partition a Cartesian topology into sub-topologies of lower-dimensional grids.

```
int MPI_Cart_sub(MPI_Comm comm_cart, int *keep_dims, MPI_Comm *comm_subcart)
```

`keep_dims[i]` is true, then the *i*th dimension is retained in the new sub-topology

`comm_cart` is the communicator of the original topology

`comm_subcart` stores information about the created sub-topology

Only a single communicator is returned to each process

For processes that do not belong to the same sub-topology, the groups specified by the returned communicators are different

2-D Matrix-Vector Multiplication Implementation

```
 1   MatrixVectorMultiply_2D(int n, double *a, double *b, double *x,
 2                           MPI_Comm comm)
 3   {
 4     int ROW=0, COL=1; /* Improve readability */
 5     int i, j, nlocal;
 6     double *px; /* Will store partial dot products */
 7     int npes, dims[2], periods[2], keep_dims[2];
 8     int myrank, my2drank, mycoords[2];
 9     int other_rank, coords[2];
10     MPI_Status status;
11     MPI_Comm comm_2d, comm_row, comm_col;
12
13     /* Get information about the communicator */
14     MPI_Comm_size(comm, &npes);
15     MPI_Comm_rank(comm, &myrank);
16
17     /* Compute the size of the square grid */
18     dims[ROW] = dims[COL] = sqrt(npes);
19
20     nlocal = n/dims[ROW];
21
22     /* Allocate memory for the array that will hold the partial dot-products */
23     px = malloc(nlocal*sizeof(double));
24
```

2-D Matrix-Vector Multiplication Implementation

```
25      /* Set up the Cartesian topology and get the rank & coordinates of the proce
this topology */
26      periods[ROW] = periods[COL] = 1; /* Set the periods for wrap-around connecti
27
28      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 1, &comm_2d);
29
30      MPI_Comm_rank(comm_2d, &my2drank); /* Get my rank in the new topology */
31      MPI_Cart_coords(comm_2d, my2drank, 2, mycoords); /* Get my coordinates */
32
33     /* Create the row-based sub-topology */
34      keep_dims[ROW] = 0;
35      keep_dims[COL] = 1;
36      MPI_Cart_sub(comm_2d, keep_dims, &comm_row);
37
38     /* Create the column-based sub-topology */
39      keep_dims[ROW] = 1;
40      keep_dims[COL] = 0;
41      MPI_Cart_sub(comm_2d, keep_dims, &comm_col);
```

2-D Matrix-Vector Multiplication Implementation

```
43      /* Redistribute the b vector. */
44      /* Step 1. The processors along the 0th column send their data to the diagon
processors */
45      if (mycoords[COL] == 0 && mycoords[ROW] != 0) { /* I'm in the first column *
46        coords[ROW] = mycoords[ROW];
47        coords[COL] = mycoords[ROW];
48        MPI_Cart_rank(comm_2d, coords, &other_rank);
49        MPI_Send(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d);
50      }
51      if (mycoords[ROW] == mycoords[COL] && mycoords[ROW] != 0) {
52        coords[ROW] = mycoords[ROW];
53        coords[COL] = 0;
54        MPI_Cart_rank(comm_2d, coords, &other_rank);
55        MPI_Recv(b, nlocal, MPI_DOUBLE, other_rank, 1, comm_2d,
56            &status);
57      }
58
```

2-D Matrix-Vector Multiplication Implementation

```
59      /* Step 2. The diagonal processors perform a column-wise broadcast */
60      coords[0] = mycoords[COL];
61      MPI_Cart_rank(comm_col, coords, &other_rank);
62      MPI_Bcast(b, nlocal, MPI_DOUBLE, other_rank, comm_col);
63
64      /* Get into the main computational loop */
65      for (i=0; i<nlocal; i++) {
66        px[i] = 0.0;
67        for (j=0; j<nlocal; j++)
68          px[i] += a[i*nlocal+j]*b[j];
69      }
70
71      /* Perform the sum-reduction along the rows to add up the partial dot-produc
72      coords[0] = 0;
73      MPI_Cart_rank(comm_row, coords, &other_rank);
74      MPI_Reduce(px, x, nlocal, MPI_DOUBLE, MPI_SUM, other_rank,
75          comm_row);
76
77      MPI_Comm_free(&comm_2d); /* Free up communicator */
78      MPI_Comm_free(&comm_row); /* Free up communicator */
79      MPI_Comm_free(&comm_col); /* Free up communicator */
80
81      free(px);
82    }
```

Nonblocking communications

Nonblocking communications are useful for overlapping communication with computation

That is, compute while communicating data

A nonblocking operation requests the MPI library to perform an operation (when it can)

Nonblocking operations do not wait for any communication events to complete

Nonblocking send and receive: return almost immediately

The user can modify a send [resp. receive] buffer only after send [resp. receive] is completed

There are "wait" routines to figure out when a nonblocking operation is done

MPI_Isend()

Performs a nonblocking send

```
int MPI_Isend(void* buf, int count, MPI Datatype datatype,
    int dest, int tag, MPI Comm comm, MPI Request *request)
```

buf: starting address of buffer

count: number of entries in buffer datatype data type of buffer

dest: rank of destination

tag: message tag

comm: communicator

request: communication request (out)

MPI_Irecv()

Performs a nonblocking receive

```
int MPI_Irecv (void* buf , int count , MPI Datatype datatype ,
    int source, int tag ,MPI Comm comm, MPI Request *request)
```

buf: starting address of buffer (out)

count: number of entries in buffer

datatype: data type of buffer

source: rank of source

tag: message tag

comm: communicator

request: communication request (out)

Wait Routines

Waits for MPI_Isend or MPI_Irecv to complete

```
int MPI_Wait (MPI Request *request, MPI Status *status)
```

request: request (in), which is out parameter in MPI Isend and MPI Irecv status
status: output

Other routines include

MPI Waitall waits for all given communications to complete

MPI Waitany waits for any of given communications to complete

MPI Test tests for completion of send or receive

MPI Testany tests for completion of any previously initiated communication

Example: Nonblocking communication

```
/* nonb.c */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
   int numtasks, rank, next, prev,
     buf[2], tag1=1, tag2=2;

   tag1=tag2=0;
   MPI_Request reqs[4];
   MPI_Status stats[4];

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Example: Nonblocking communication

```
    prev = rank-1;
    next = rank+1;

    if (rank == 0)               prev = numtasks - 1;
    if (rank == numtasks - 1)  next = 0;

    MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1,
              MPI_COMM_WORLD, &reqs[0]);
    MPI_Irecv(&buf[1], 1, MPI_INT, prev, tag2,
              MPI_COMM_WORLD, &reqs[1]);

    MPI_Isend(&rank, 1, MPI_INT, next, tag2,
              MPI_COMM_WORLD, &reqs[2]);
    MPI_Isend(&rank, 1, MPI_INT, next, tag1,
              MPI_COMM_WORLD, &reqs[3]);

    MPI_Waitall(4, reqs, stats);
    printf("Task %d communicated with tasks %d & %d\n",
           rank,prev,next);

    MPI_Finalize();
    return 0;
}
```

Nonblocking communication

Nonblocking send can be posted whether a matching receive has been posted or not

Send is completed when data has been copied out of send buffer

Nonblocking send can be matched with blocking receive and vice versa

Communications are initiated by sender

Avoiding Deadlocks with Nonblocking Communication

Is the following code safe?

```
int a[10], b[10], myrank;
MPI_Status status;
...
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Safe in the buffered case, unsafe in non-buffered case

The code is implementation dependent!

Avoiding Deadlocks with Nonblocking Communication

If we replace *either* the send *or* receive with their non-blocking counterparts, then the code will be safe, e.g.,

```
int a[10], b[10], myrank;
MPI_Status status;
MPI_Request requests[2];
...
 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Irecv(b, 10, MPI_INT, 0, 2, &requests[0], MPI_COMM_WORLD);
   MPI_Irecv(a, 10, MPI_INT, 0, 1, &requests[1], MPI_COMM_WORLD);
}
...
```

Matrix Multiplication

Consider the problem of multiplying two $n \times n$ dense, square matrices A and B to yield the product matrix $C = A \times B$.

The serial complexity is $O(n^3)$

A useful concept in this case is called *block* operations. In this view, an $n \times n$ matrix A can be regarded as a $q \times q$ array of blocks $A_{i,j}$ $(0 \leq i, j < q)$ such that each block is an $(n/q) \times (n/q)$ submatrix.

In this view, we perform q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices.

Matrix Multiplication: Simple 2D Partitioning

Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ ($0 \leq i, j < p$) of size $(n/p^{0.5}) \times (n/p^{0.5})$ each.

Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix.

Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $0 \leq k < p^{0.5}$.

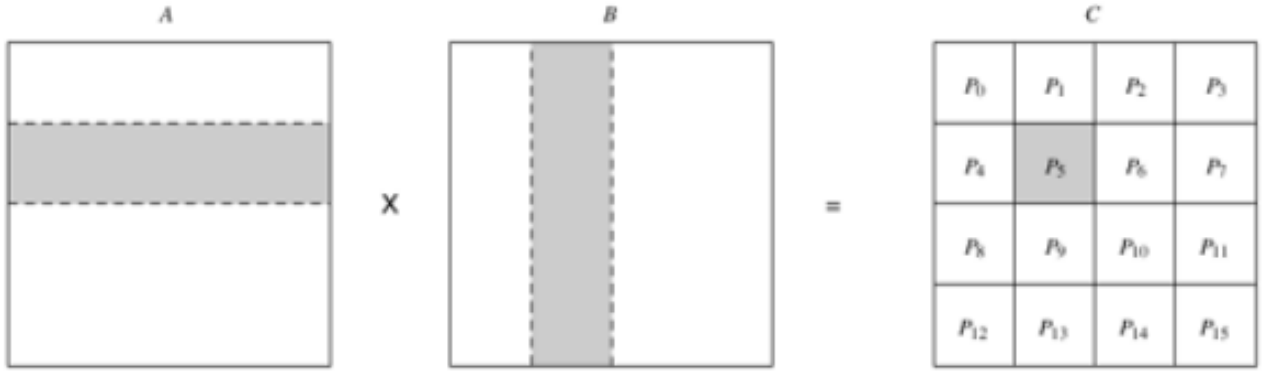
All-to-all broadcast blocks of A along rows and B along columns.

Perform local submatrix multiplication.

Major drawback of the algorithm is that it is not memory optimal!

Matrix Multiplication: Simple 2D Partitioning

```
procedure BLOCK_MAT_MULT (A, B, C)
begin
    for i := 0 to q - 1 do
        for j := 0 to q - 1 do
            begin
                Initialize all elements of C_{i,j} to zero;
                for k := 0 to q - 1 do
                    C_{i,j} := C_{i,j} + A_{i,k} x B_{k,j};
            endfor;
end BLOCK_MAT_MULT
```



Cannon's Algorithm: the Idea

We schedule the computations of the $p^{0.5}$ processes of the ith row such that, at any given time, each process is using a different block $A_{i,k}$.

These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh $A_{i,k}$ after each rotation.

Align the blocks of A and B in such a way that each process multiplies its local submatrices. This is done by shifting all submatrices $A_{i,j}$ to the left (with wraparound) by i steps and all submatrices $B_{i,j}$ up (with wraparound) by j steps.

Perform local multiplication of blocks you have now and add on to $C_{i,j}$

Each block of A moves one step left and each block of B moves one step up (again with wraparound).

Perform next block multiplication, add to partial result, repeat until all blocks have been multiplied.

Cannon's Algorithm

// make initial alignment

for i, j :=0 **to** $\sqrt{p} - 1$ **do**

Send block $A_{i,j}$ to process $\left(i, \left(j - i + \sqrt{p}\right) mod \sqrt{p}\right)$ and block $B_{i,j}$ to process $\left(\left(i - j + \sqrt{p}\right) mod \sqrt{p}, j\right)$;

endfor;

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

// compute-and-shift. A sequence of one-step shifts pairs up $A_{i,k}$ and $B_{k,j}$

// on process $P_{i,j}$. $C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}$

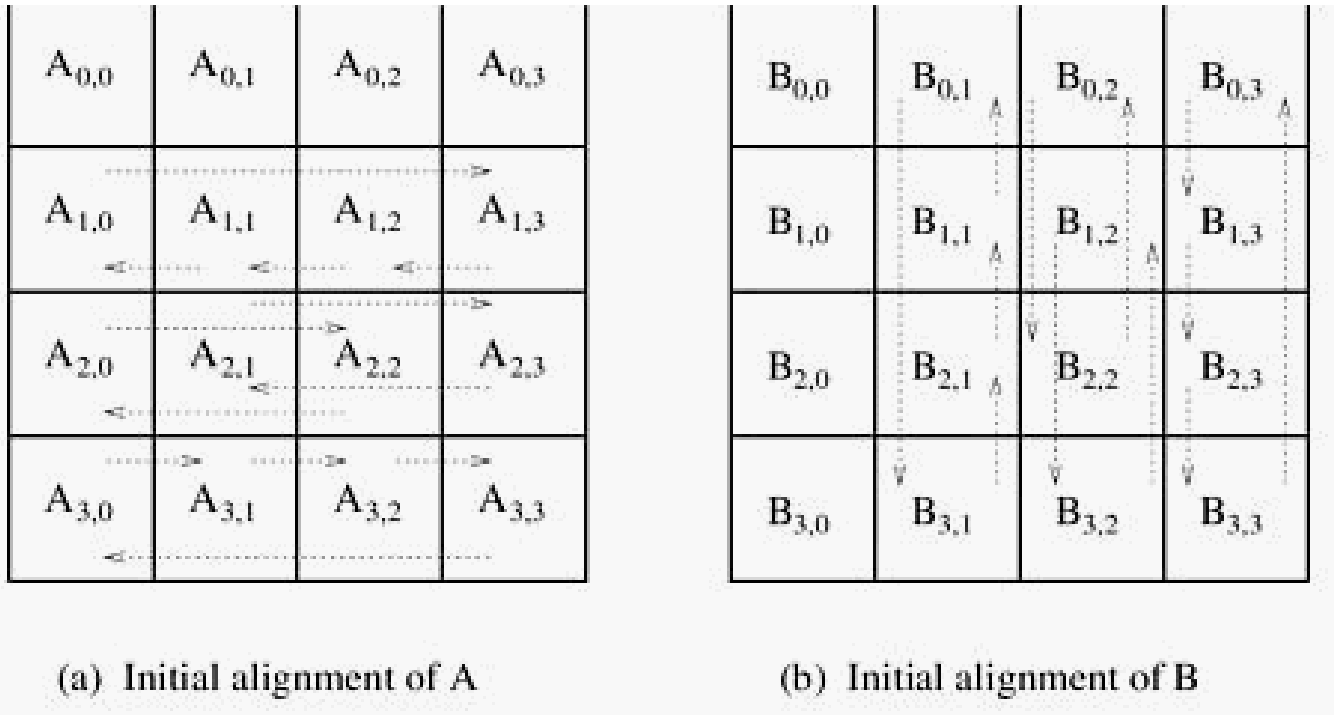
for step :=1 **to** $\sqrt{p} - 1$ **do**

Shift $A_{i,j}$ one step left (with wraparound) and $B_{i,j}$ one step up (with wraparound);

Process $P_{i,j}$ multiply received submatrices together and add the result to $C_{i,j}$;

Endfor;

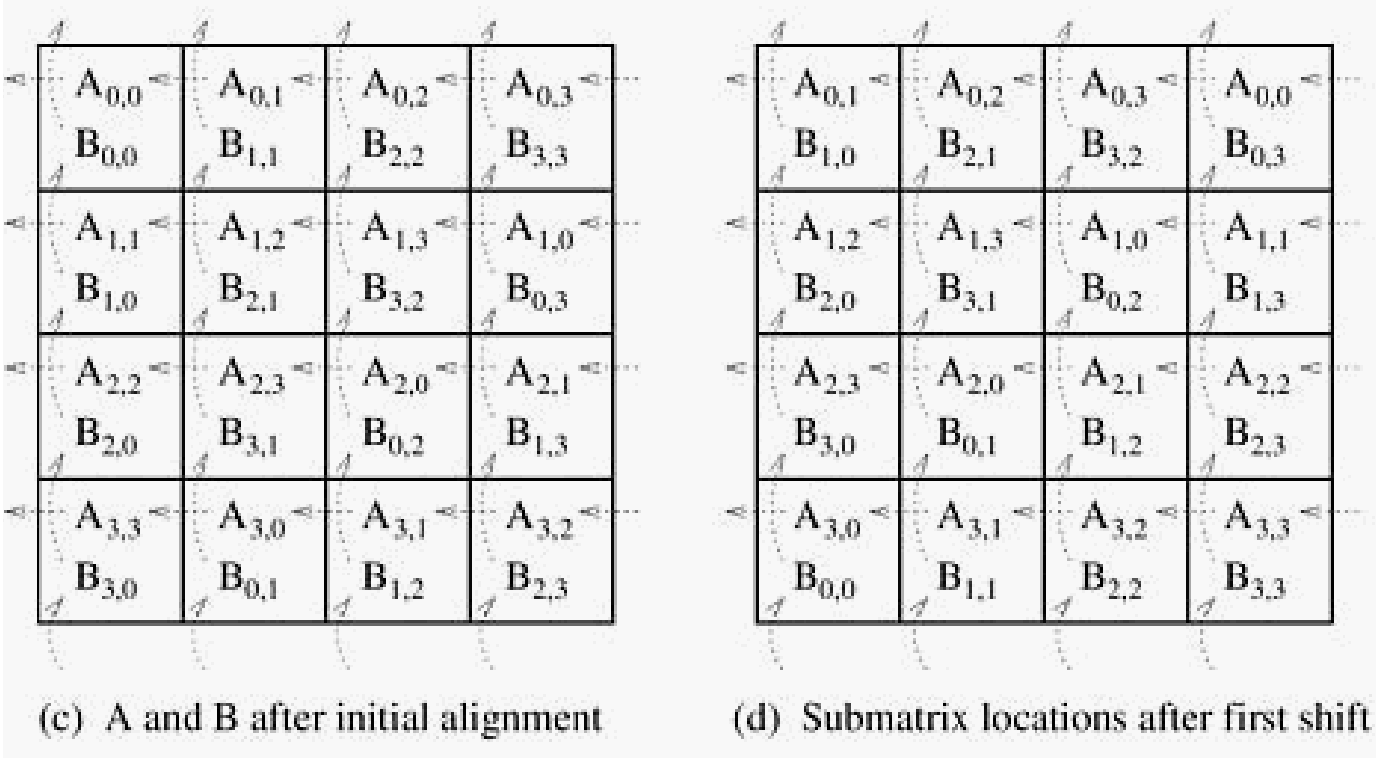
Matrix Multiplication: Cannon's Algorithm



(a) Initial alignment of A

(b) Initial alignment of B

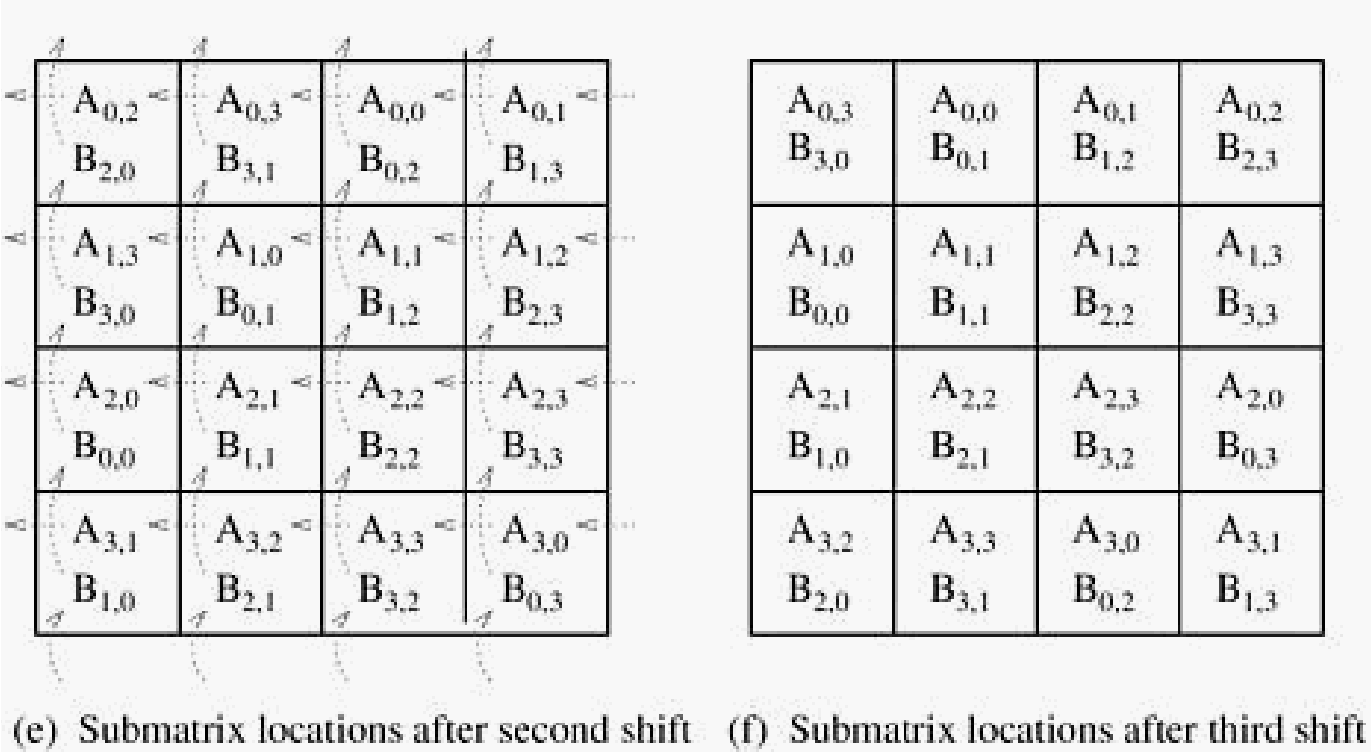
Matrix Multiplication: Cannon's Algorithm



(c) A and B after initial alignment

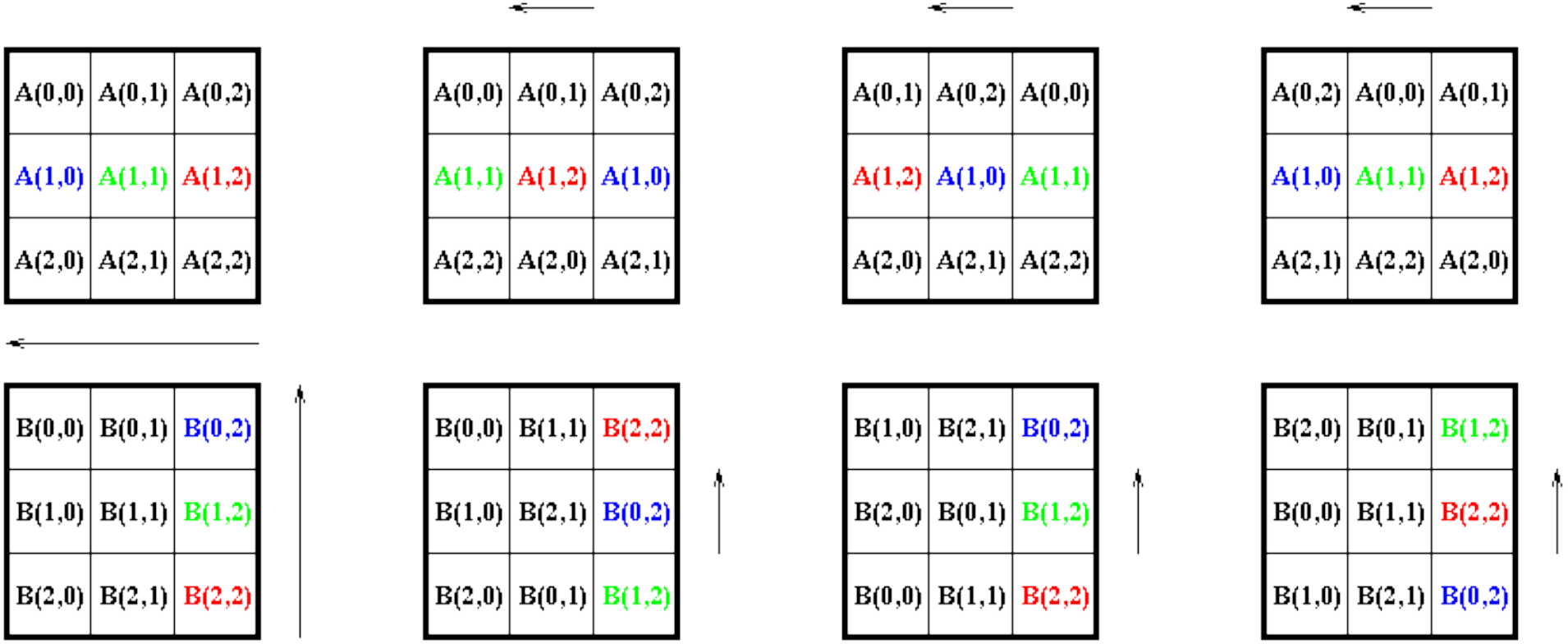
(d) Submatrix locations after first shift

Matrix Multiplication: Cannon's Algorithm



(e) Submatrix locations after second shift (f) Submatrix locations after third shift

Matrix Multiplication: Cannon's Algorithm



Initial A, B

A, B initial alignment

A, B after shift step 1

A, B after shift step 2

Communication Cost

t_s the latency or the startup time for the data transfer

t_w the per-word transfer time

An exchange of an m-word message between two processes running on different nodes

takes $t_s + m\,t_w$ time

One-to-All Broadcast (message: m words, among p nodes)

takes $(t_s + m\,t_w) \log p$ time

All-to-All Broadcast (message: m words, among p nodes)

Circular shift: takes $(t_s + m\,t_w)\,(p\text{-}1)$ time

A hypercube algorithm: takes $t_s \log p + m\,t_w\,(p\text{-}1)$ time

Performance Analysis: Simple 2D Partitioning

$\sqrt{p}$ rows of all-to-all broadcasts, each is among a group of $\sqrt{p}$ processes. A message size is $\frac{n^2}{p}$, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$

$\sqrt{p}$ columns of all-to-all broadcasts, communication time: $t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)$

Computation time: $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$

Parallel time: $T_p = \frac{n^3}{p} + 2\left(t_s log\sqrt{p} + t_w \frac{n^2}{p}\left(\sqrt{p}-1\right)\right)$

Performance Analysis: Cannon's Algorithm

In the initial alignment step, the maximum distance over which block shifts is $\sqrt{p} - 1$

- The circular shift operations in row and column directions take time: $t_{comm} = 2(t_s + \frac{t_w n^2}{p})$

Each of the $\sqrt{p}$ single-step shifts in the compute-and-shift phase takes time: $t_s + \frac{t_w n^2}{p}$.

Multiplying $\sqrt{p}$ submatrices of size $(\frac{\mathrm{n}}{\sqrt{p}}) \times (\frac{\mathrm{n}}{\sqrt{p}})$ takes time: n^3/p.

Parallel time: $T_p = \frac{n^3}{p} + 2\sqrt{p}\left(t_s + \frac{t_w n^2}{p}\right) + 2(t_s + \frac{t_w n^2}{p})$

Implementation of Cannon's Algorithm

Use Cartesian Topology

to model the topology of submatrices on a 2D grid

Blocking Communication

Use sendrecv to implement the "shift" operations

Nonblocking Communciation

Overlap the computation with communication

Use Isend and Irecv

When receiving the next submatrices, perform the multiplication on current submatrices.