

Lecture 8: Synchronous Computations

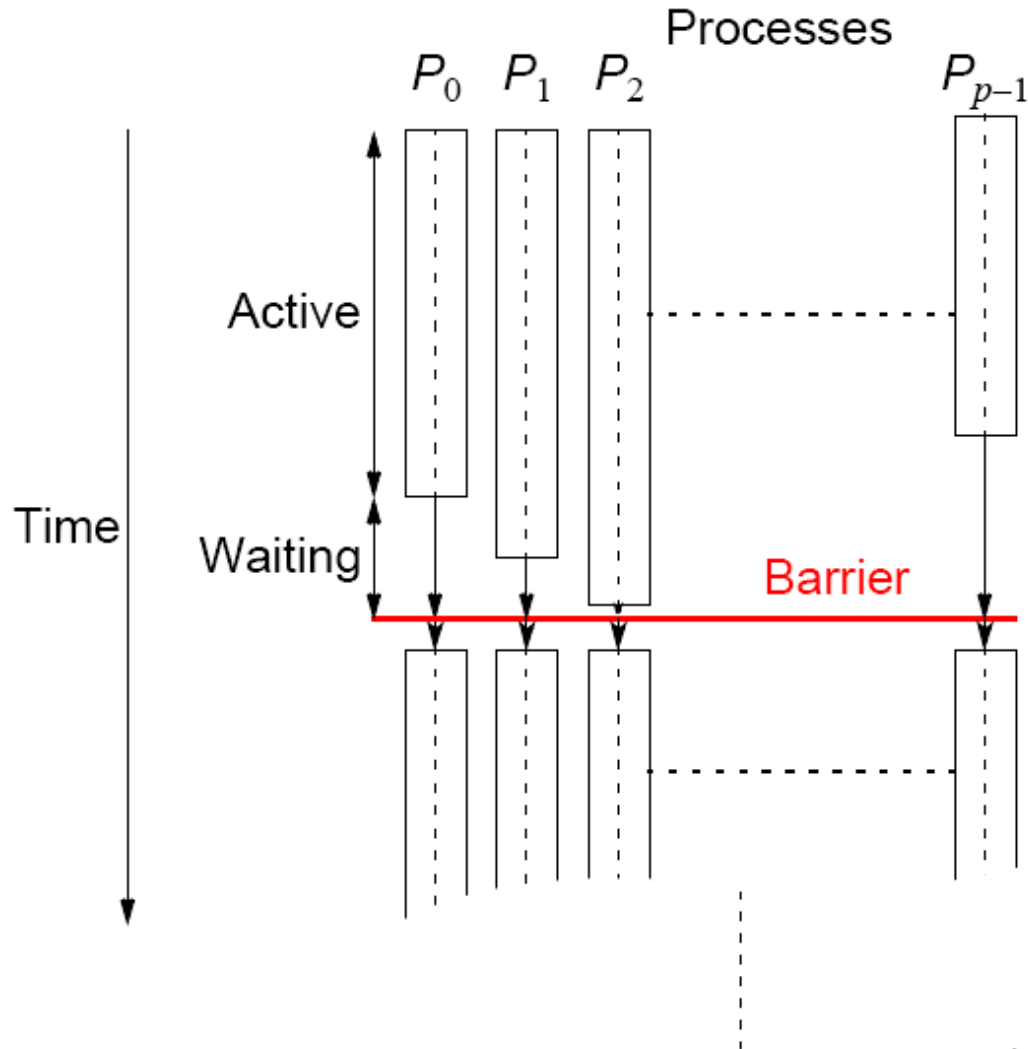
Parallell Programming (INF-3201)
Autumn 2013

John Markus Bjørndalen

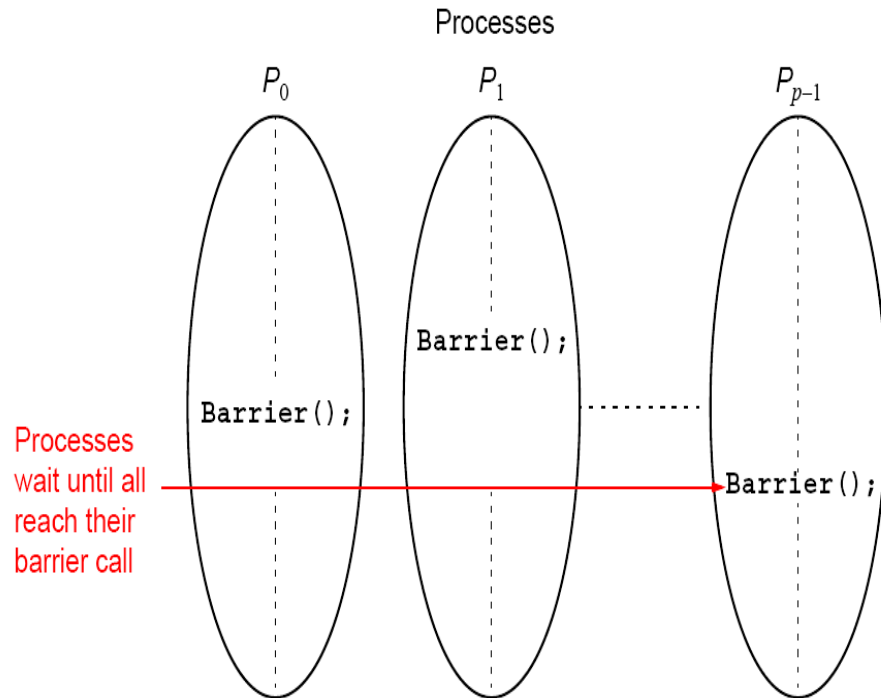
Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

Processes reaching barrier at different times



In message-passing systems, barriers provided with library routines:



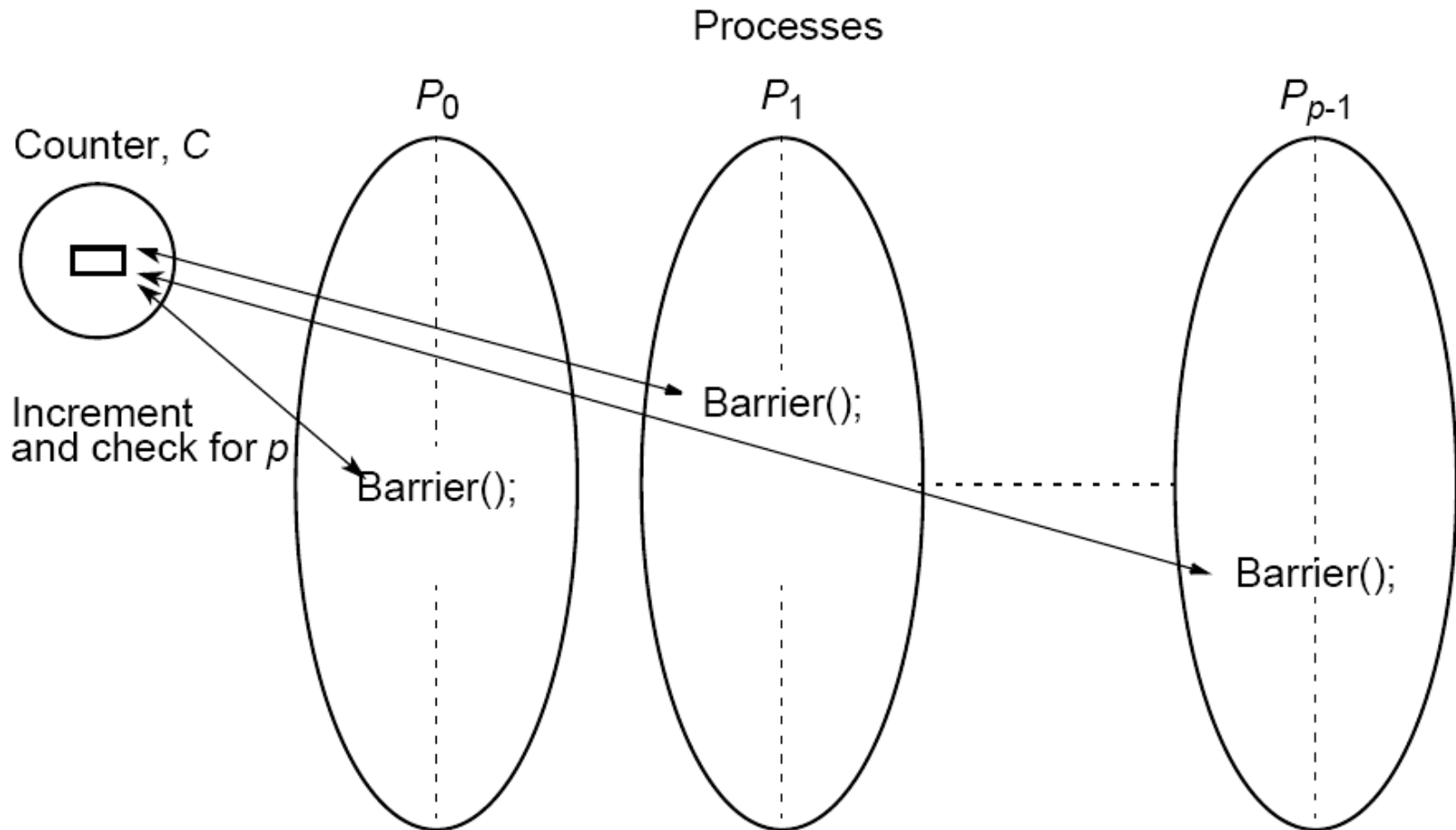
MPI **MPI_Barrier()**

Barrier with a named communicator being the only parameter.

Called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then.

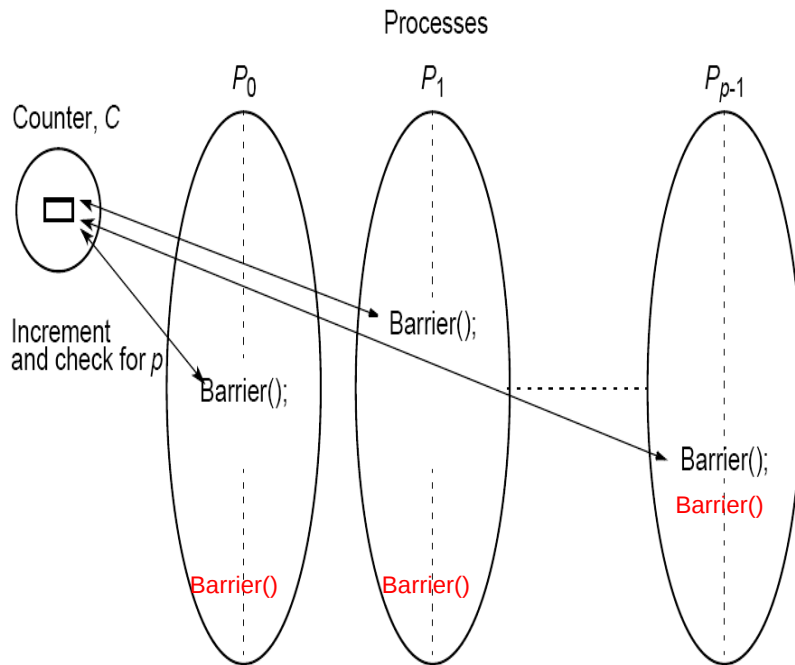
Barrier Implementation

Centralized counter implementation (a *linear barrier*):



Barrier Implementation

Centralized counter implementation (a *linear barrier*):

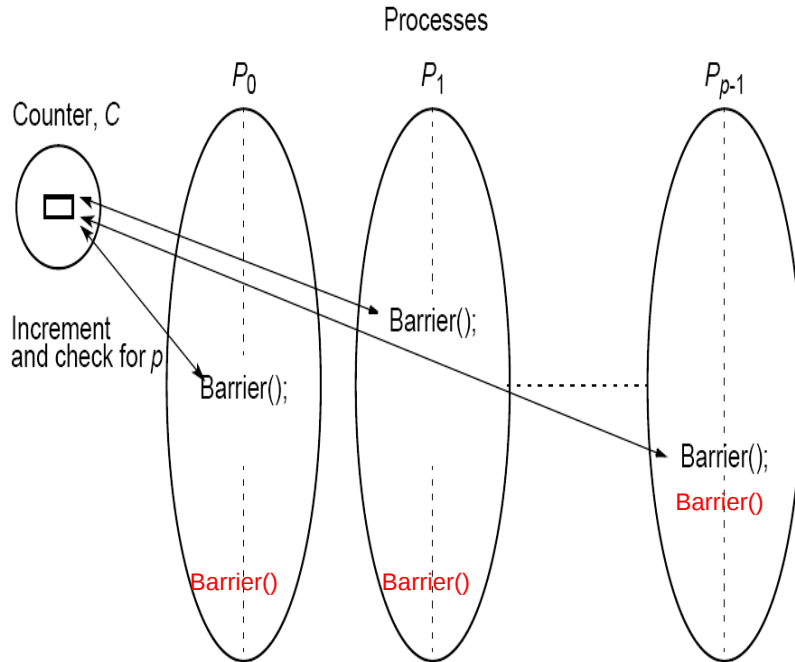


Good barrier implementations must take into account that a barrier might be used more than once in a process.

Might be possible for a process to enter the barrier for a second time before previous processes have left the barrier for the first time.

Barrier Implementation

Centralized counter implementation (a *linear barrier*):

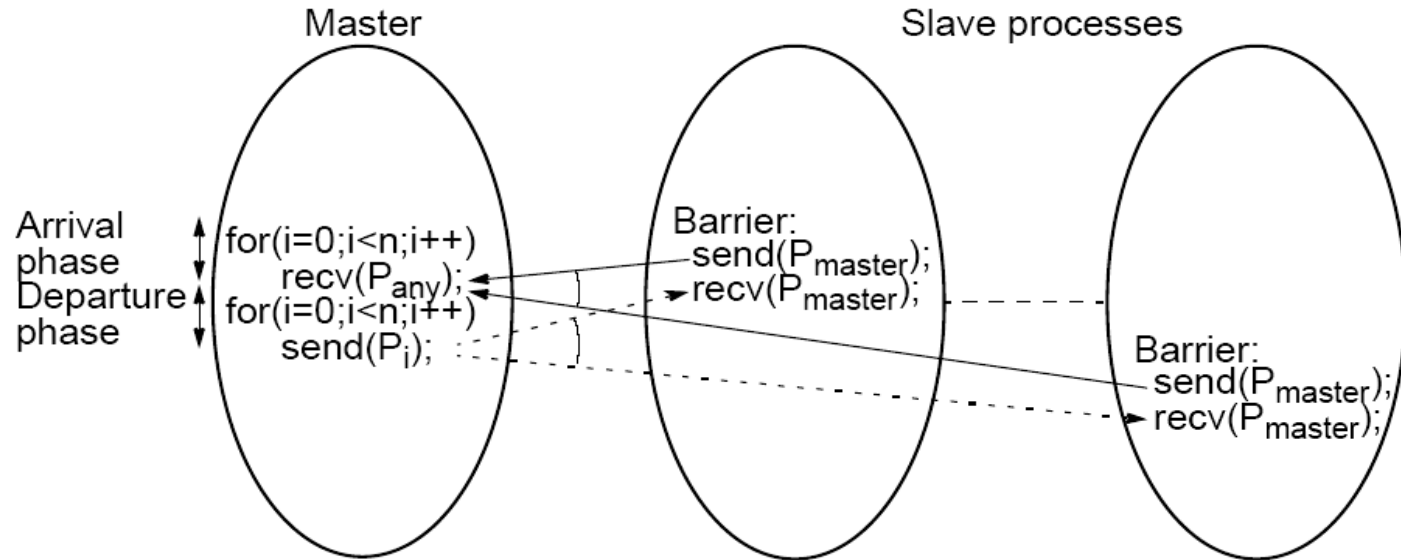


Counter-based barriers often have two phases:

- A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.
- Then processes move to departure phase and are released.

Two-phase handles the reentrant scenario.

Barrier implementation in a message-passing system



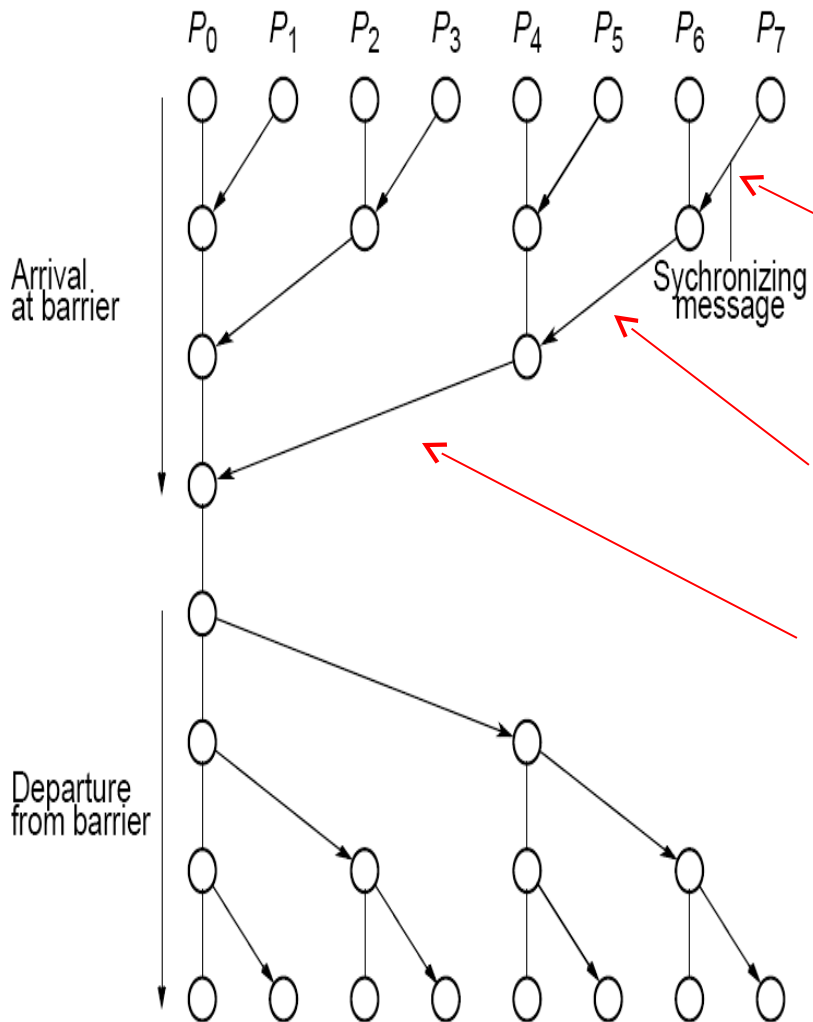
- Master:**

```
for (i = 0; i < n; i++) /* count slaves as they reach barrier*/
  rcv(Pany);
for (i = 0; i < n; i++) /* release slaves */
  send(Pi);
```

- Slave processes:**

```
send(Pmaster);
rcv(Pmaster);
```


Tree Implementation



More efficient. $O(\log p)$ steps

Suppose 8 processes, $P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7$:

1st stage:

- P_1 sends message to P_0 ; (when P_1 reaches its barrier)
- P_3 sends message to P_2 ; (when P_3 reaches its barrier)
- P_5 sends message to P_4 ; (when P_5 reaches its barrier)
- P_7 sends message to P_6 ; (when P_7 reaches its barrier)

2nd stage:

- P_2 sends message to P_0 ; (P_2 & P_3 reached their barrier)
- P_6 sends message to P_4 ; (P_6 & P_7 reached their barrier)

3rd stage:

- P_4 sends message to P_0 ; (P_4, P_5, P_6 , & P_7 reached barrier)
- P_0 terminates arrival phase; (when P_0 reaches barrier & received message from P_4)

Release with a reverse tree construction.

Butterfly Barrier

1st stage

$$P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$$

2nd stage

$$P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$$

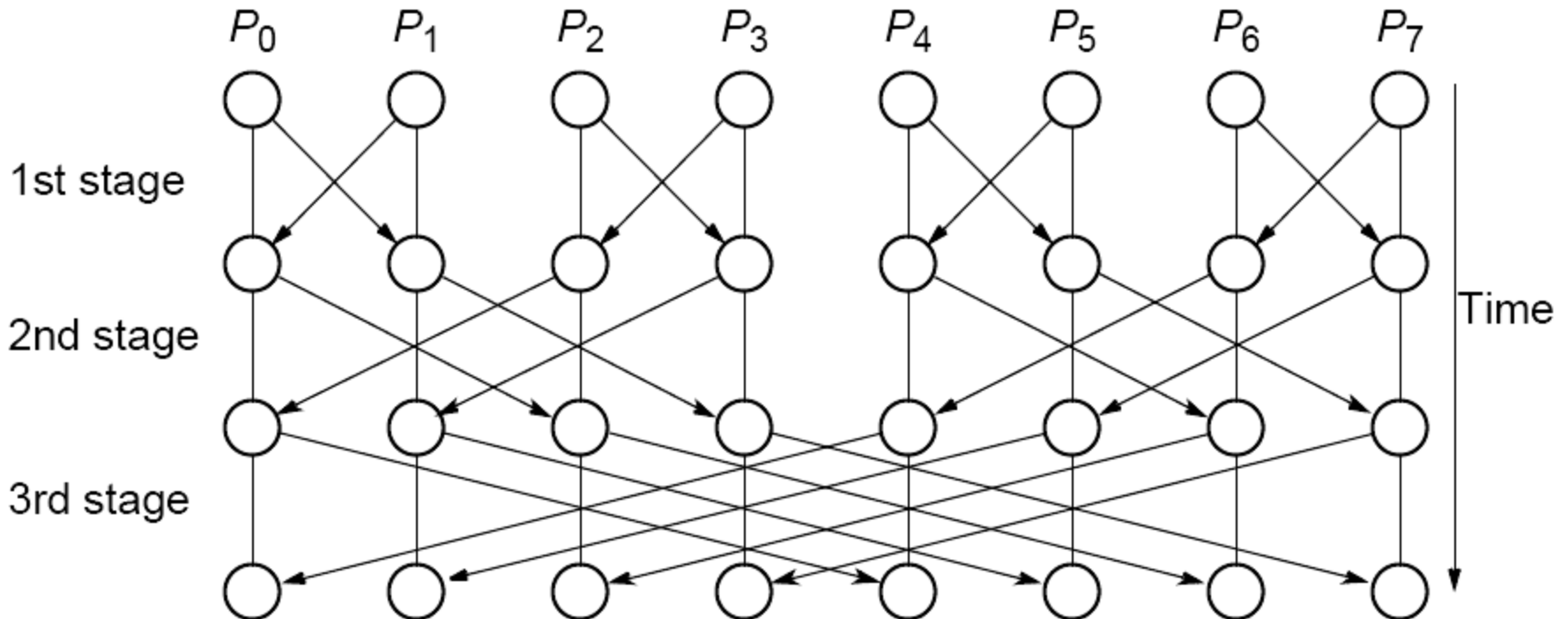
3rd stage

$$P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$$

Textbook: At stage s , p_i synchronizes with p_j , where $j = i + 2s - 1$. Wrong!

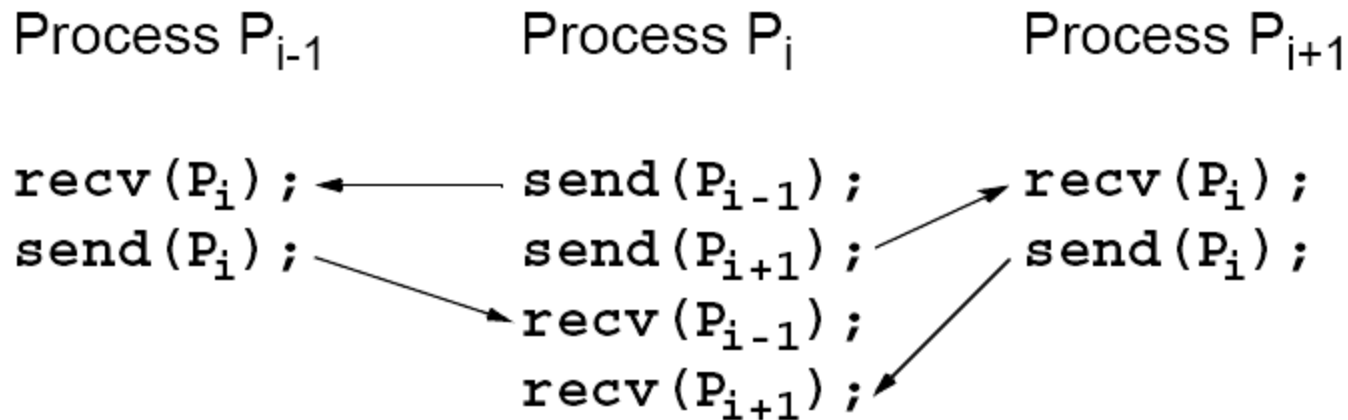
Correct: $j = i \text{ XOR } 2s - 1$

(cf. D. E. Culler et al., Parallel Computer Architecture – A Hardware/Software Approach). /Phuong



Local Synchronization

Suppose a process P_i needs to be synchronized and to exchange data with process P_{i-1} and process P_{i+1} before continuing:



Not a perfect three-process barrier because process P_{i-1} will only synchronize with P_i and continue as soon as P_i allows. Similarly, process P_{i+1} only synchronizes with P_i .

Deadlock

When a pair of processes each send and receive from each other, deadlock may occur.

Deadlock will occur if both processes perform the **send**, using **synchronous routines** first (or blocking routines without sufficient buffering). This is because neither will return; they will wait for matching receives that are never reached.

Deadlock conditions

Necessary conditions for deadlocks:

- **Mutual exclusion:** At least two resources must be non-shareable
- **Hold and Wait / Resource holding :** processes can hold a resource while requesting another
- **No Preemption:** resources must be released voluntarily – cannot force a process to release it (or take it away)
- **Circular wait:** a set of processes form a circle, where each wait for a resource held by the next in the circle.

A Solution

Arrange for one process to receive first and then send and the other process to send first and then receive.

Example

Linear pipeline, deadlock can be avoided by arranging so the even-numbered processes perform their sends first and the odd-numbered processes perform their receives first.



Combined deadlock-free blocking sendrecv() routines

Example

Process P_{i-1}

Process P_i

Process P_{i+1}

```
sendrecv( $P_i$ ) ;  sendrecv( $P_{i-1}$ ) ;  
sendrecv( $P_{i+1}$ ) ;  sendrecv( $P_i$ ) ;
```

MPI provides **MPI_Sendrecv()** and **MPI_Sendrecv_replace()**.
MPI sendrecv()s actually has 12 parameters!

Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

Synchronized Computations

Can be classified as:

- Fully synchronous

or

- Locally synchronous

In fully synchronous, all processes involved in the computation must be synchronized.

In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation

Fully Synchronized Computation Examples

Data Parallel Computations

Same operation performed on different data elements simultaneously; i.e., in parallel.

Particularly convenient because:

- Ease of programming (essentially only one program).
- Can scale easily to large problem sizes. How large?
- Many numeric and some non-numeric problems can be cast in a data parallel form.

Example

To add the same constant to each element of an array:

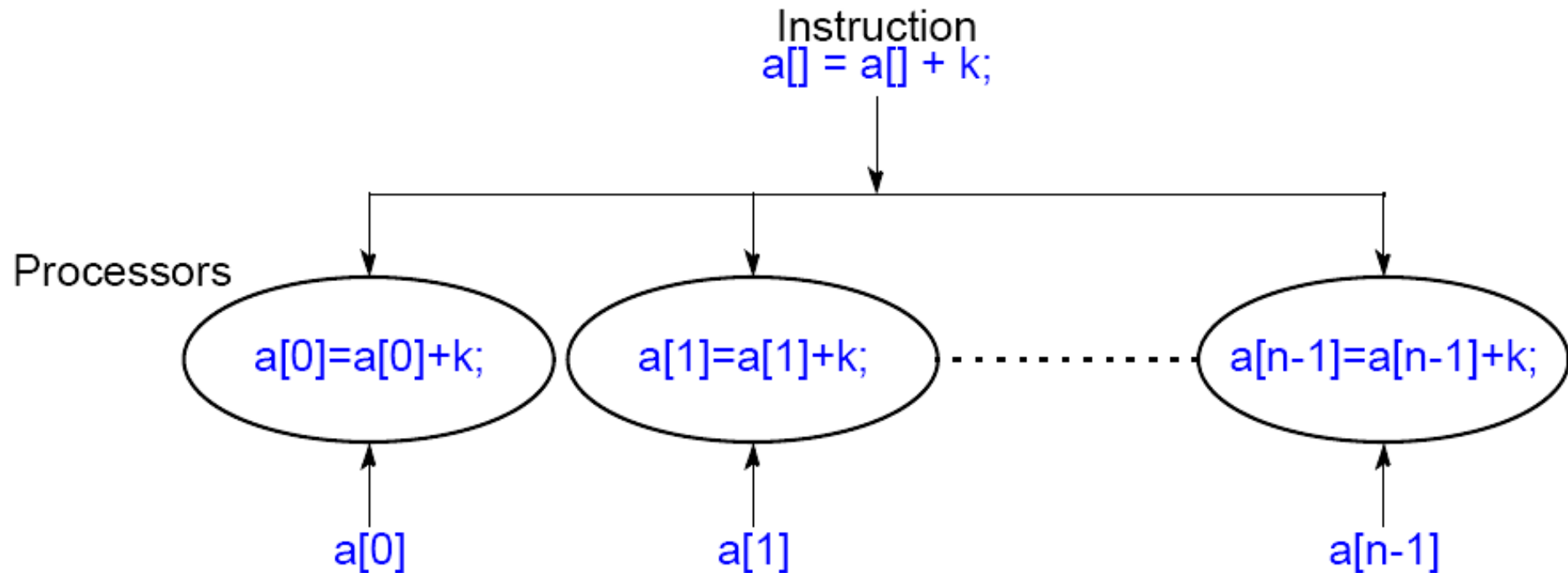
```
for (i = 0; i < n; i++)  
    a[i] = a[i] + k;
```

The statement:

```
a[i] = a[i] + k;
```

could be executed simultaneously by multiple processors,
each using a different index i ($0 < i \leq n$).

Data Parallel Computation



forall construct

Special “parallel” construct in parallel programming languages to specify data parallel operations

Example

```
forall (i = 0; i < n; i++) {  
    body  
}
```

states that n instances of the statements of the body can be executed simultaneously.

One value of the loop variable i is valid in each instance of the body, the first instance has $i = 0$, the next $i = 1$, and so on.

To add **k** to each element of an array, **a**, we can write

```
forall (i = 0; i < n; i++)  
    a[i] = a[i] + k;
```

Data parallel technique applied to multiprocessors and multicomputers

Example

To add **k** to the elements of an array:

```
i = myrank;  
a[i] = a[i] + k;          /* body */  
barrier(mygroup);
```

where **myrank** is a process rank between 0 and $n - 1$.

Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

Data Parallel Example

Prefix Sum Problem

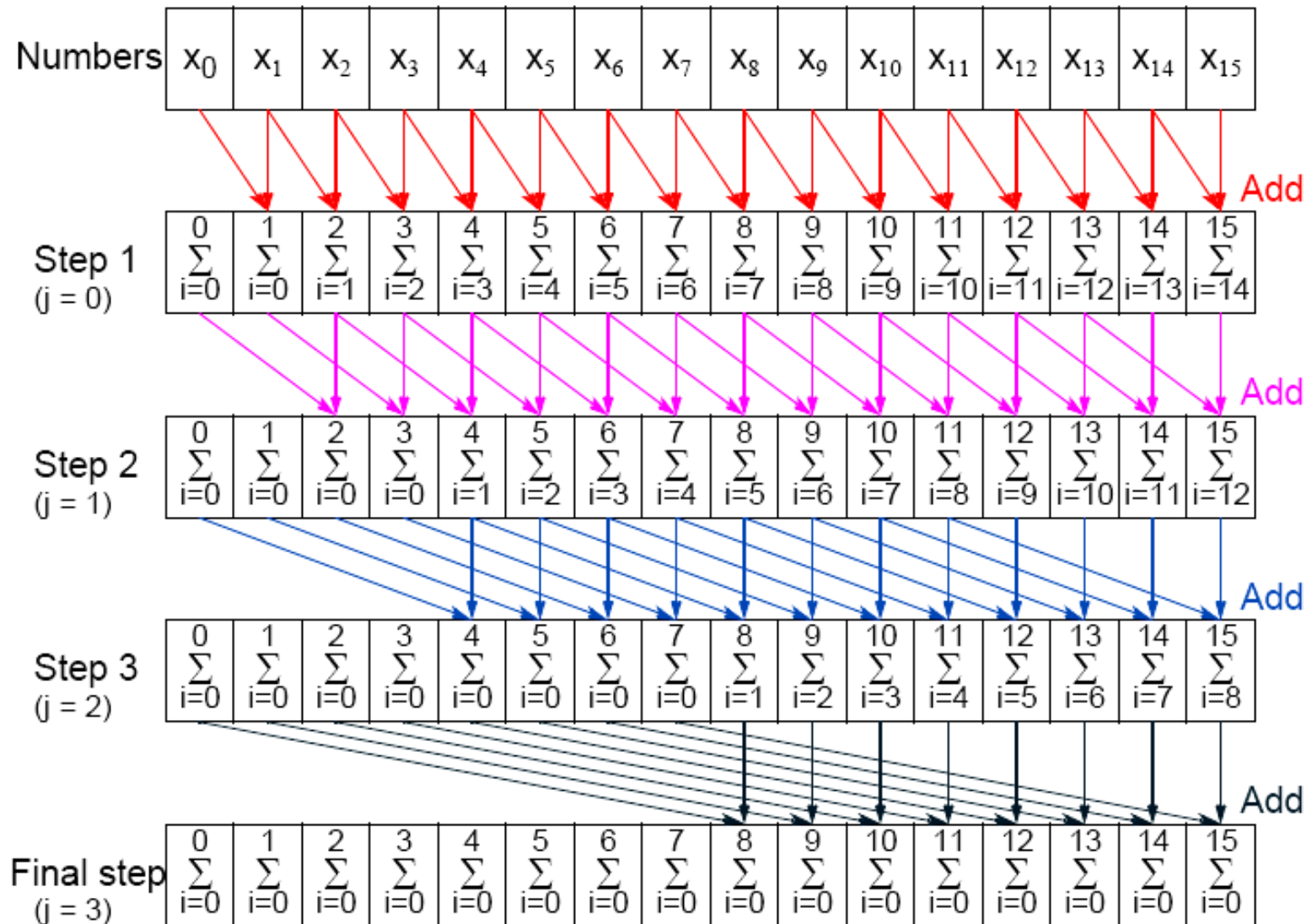
Given a list of numbers, x_0, \dots, x_{n-1} , compute all the partial summations, i.e.:

$$x_0 + x_1; x_0 + x_1 + x_2; x_0 + x_1 + x_2 + x_3; \dots$$

Can also be defined with associative operations other than addition.

Widely studied. Practical applications in areas such as processor allocation, data compaction, sorting, and polynomial evaluation.

Data parallel prefix sum operation



Sequential code

```
for (j = 0; j < log(n); j++) /* at each step, add */  
    for (i = 2j; i < n; i++) /* to accumulating sum */  
        x[i] = x[i] + x[i - 2j];
```

Parallel code

```
for (j = 0; j < log(n); j++) /* at each step, add */  
    forall (i = 0; i < n; i++) /* to sum */  
        if (i >= 2j) x[i] = x[i] + x[i - 2j];
```

Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

Synchronous Iteration

(Synchronous Parallelism)

Each iteration composed of several processes that start together at beginning of iteration. Next iteration cannot begin until all processes have finished previous iteration.

Using **forall** construct:

```
for (j = 0; j < n; j++) {           // for each synch. iteration
    forall (i = 0; i < N; i++) {     // N procs each using
        body(i);                   // specific value of i
    }
}
```

Using message passing barrier:

```
for (j = 0; j < n; j++) {           /*for each synchr.iteration */
    i = myrank;                     /*find value of i to be used */
    body(i);
    barrier(mygroup);
}
```

Another fully synchronous computation example

Solving a General System of Linear Equations **by Iteration**

Suppose the equations are of a general form with n equations and n unknowns

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \dots + a_{2,n-1}x_{n-1} = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \dots + a_{1,n-1}x_{n-1} = b_1$$

$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \dots + a_{0,n-1}x_{n-1} = b_0$$

where the unknowns are $x_0, x_1, x_2, \dots, x_{n-1}$ ($0 \leq i < n$).

By rearranging the i th equation:

$$a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots + a_{i,n-1}x_{n-1} = b_i$$

to

$$x_i = (1/a_{i,i})[b_i - (a_{i,0}x_0 + a_{i,1}x_1 + a_{i,2}x_2 \dots a_{i,i-1}x_{i-1} + a_{i,i+1}x_{i+1} \dots + a_{i,n-1}x_{n-1})]$$

or

$$x_i = \frac{1}{a_{i,i}} \left[b_i - \sum_{j \neq i} a_{i,j} x_j \right]$$

This equation gives x_i in terms of the other unknowns.

Can be used as an iteration formula for each of the unknowns to obtain better **approximations**.

Jacobi Iteration

All values of x are updated **together**.

Can be proven that Jacobi method will converge if diagonal values of a have an absolute value greater than sum of the absolute values of the other a 's on the row (the array of a 's is *diagonally dominant*) i.e. if

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}|$$

This condition is a sufficient but not a necessary condition.

Termination

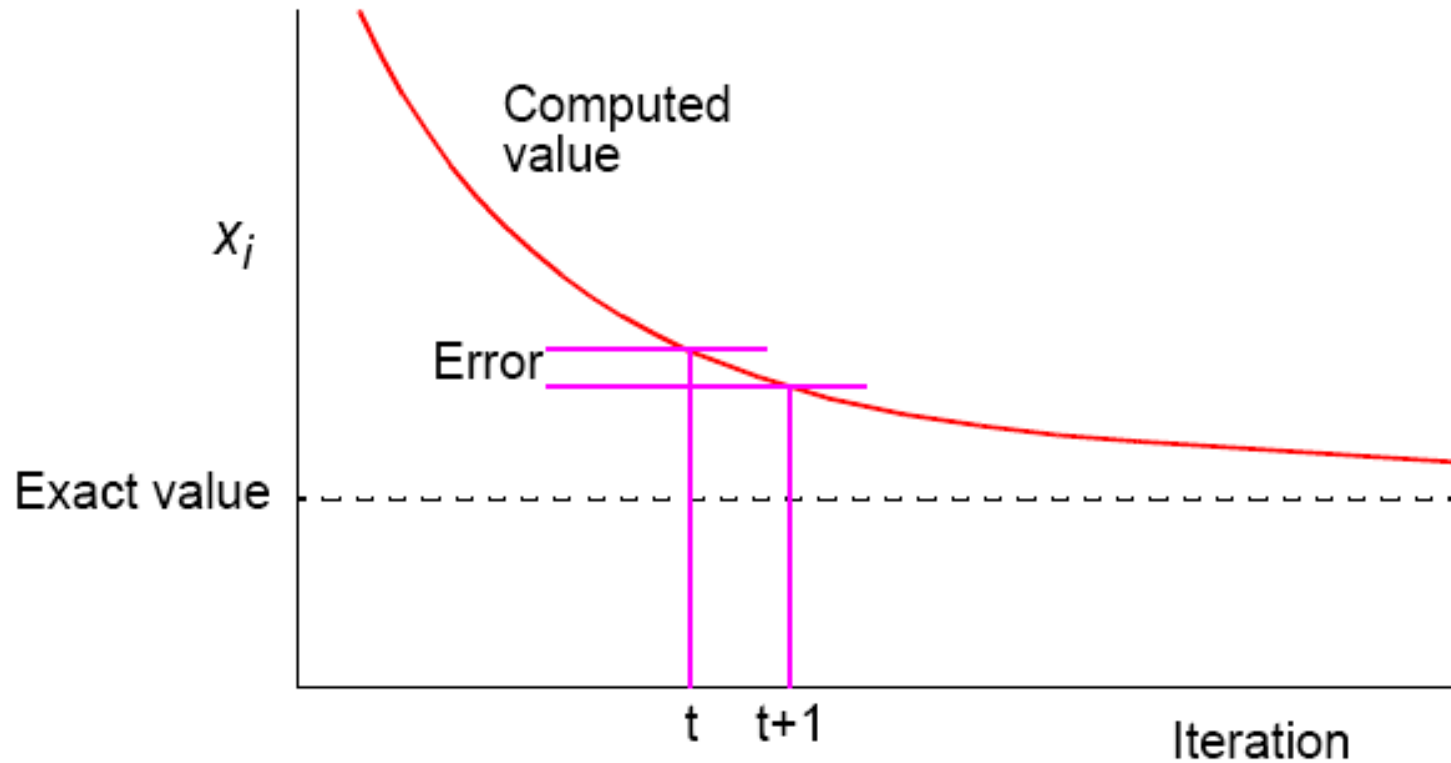
A simple, common approach. Compare values computed in one iteration to values obtained from the previous iteration. Terminate computation when all values are within given tolerance; i.e., when

$$\left| x_i^t - x_i^{t-1} \right| < \text{error tolerance}$$

for all i , where x_i^t is the value of x_i after the t th iteration and x_i^{t-1} is the value of x_i after the $(t - 1)$ th iteration.

However, this does not guarantee the solution to that accuracy.

Convergence Rate



Parallel Code

Process P_i could be of the form

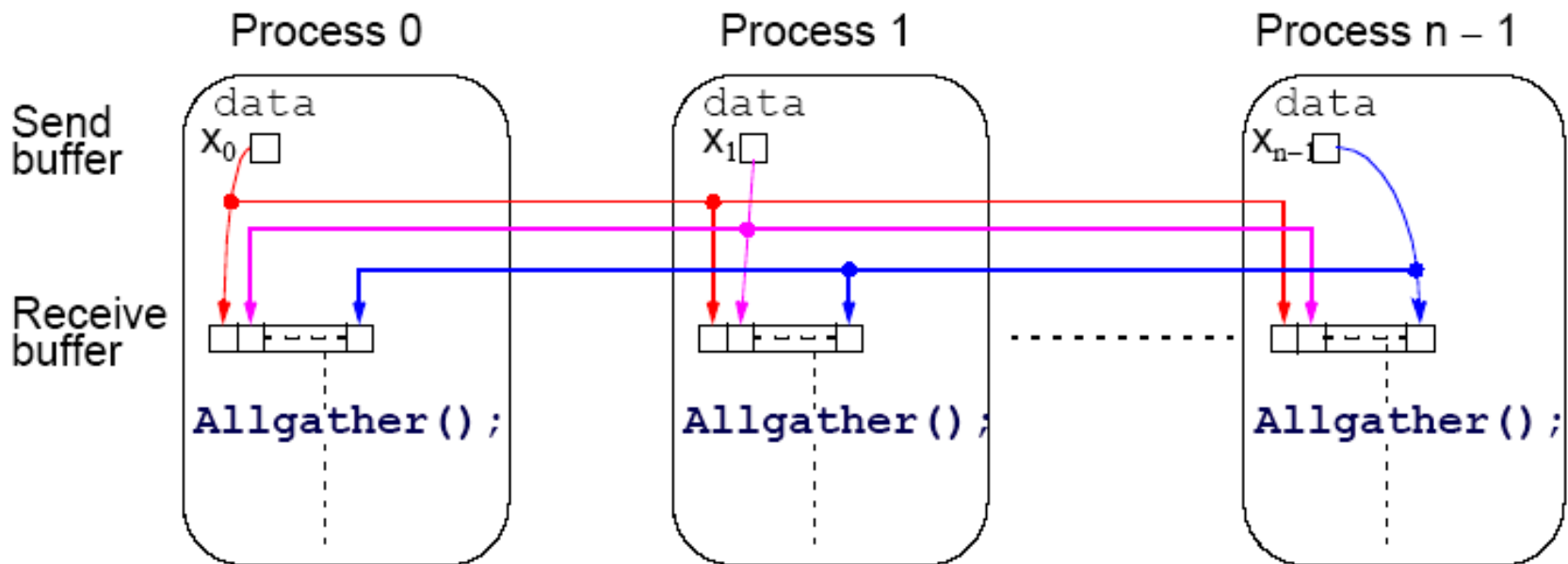
```
x[i] = b[i]; //initialize unknown
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++) // compute summation
        sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i]; // compute unknown
    allgather(&new_x[i], &x[]); // bcast/rec values
    global_barrier(); // wait for all procs
}
```

allgather() sends the newly computed value of **x[i]** from process i to every other process and collects data broadcast from the other processes.

Introduce a new message-passing operation - Allgather.

Allgather

Broadcast and gather values in one composite construction.



Partitioning

Usually number of processors much fewer than number of data items to be processed. Partition the problem so that processors take on more than one data item.

block allocation – allocate groups of consecutive unknowns to processors in increasing order.

- processor P_0 is allocated $x_0, x_1, x_2, \dots, x_{(n/p)-1}$,
- processor P_1 is allocated $x_{n/p}, x_{n/p+1}, \dots, x_{(2n/p)-1}$, and so on.

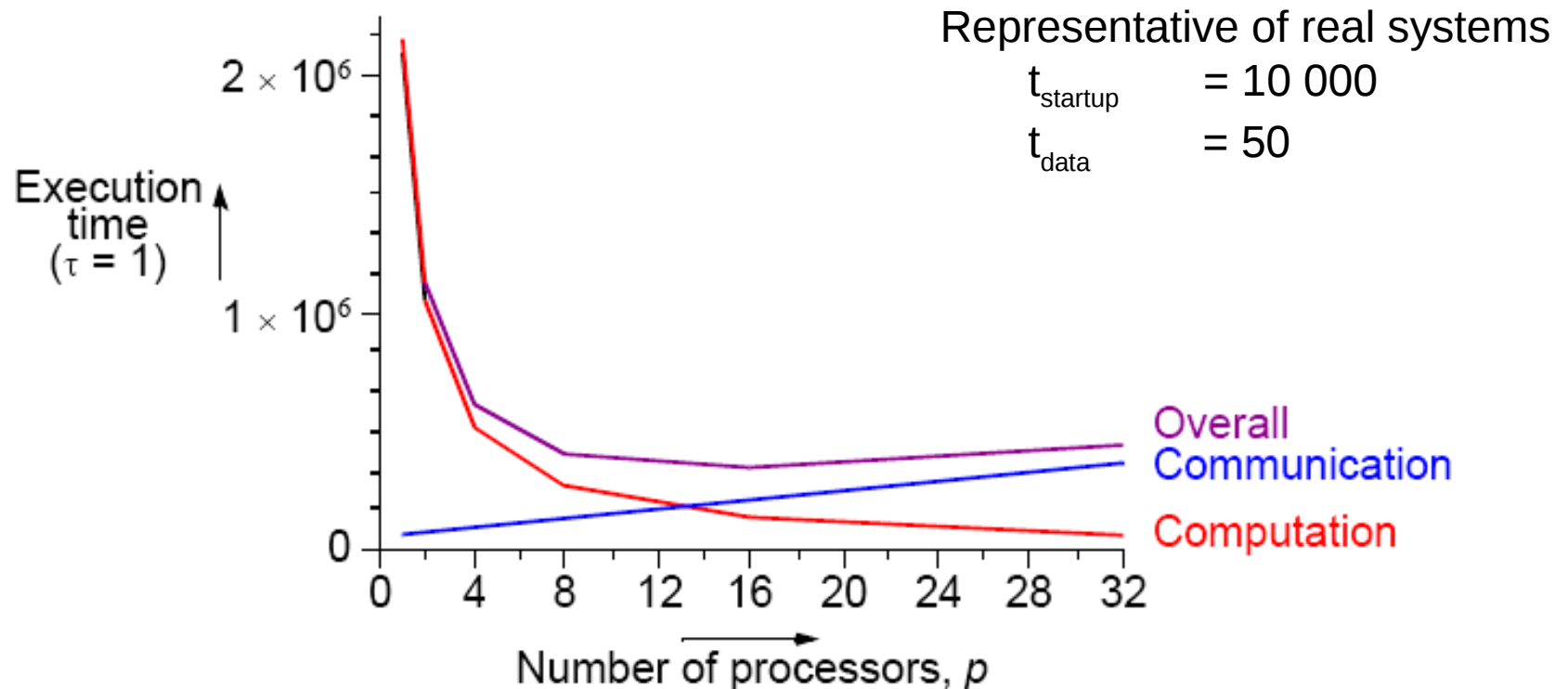
cyclic allocation – processors are allocated one unknown in order:

- processor P_0 is allocated $x_0, x_p, x_{2p}, \dots, x_{((n/p)-1)p}$,
- processor P_1 is allocated $x_1, x_{p+1}, x_{2p+1}, \dots, x_{((n/p)-1)p+1}$, and so on.
-

Effects of computation and communication in Jacobi iteration

Consequences of different numbers of processors done in textbook.

Get:



Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

Locally Synchronous Computation

Heat Distribution Problem

An area has known temperatures along each of its edges.

Find the temperature distribution within.

Heat Distribution Problem

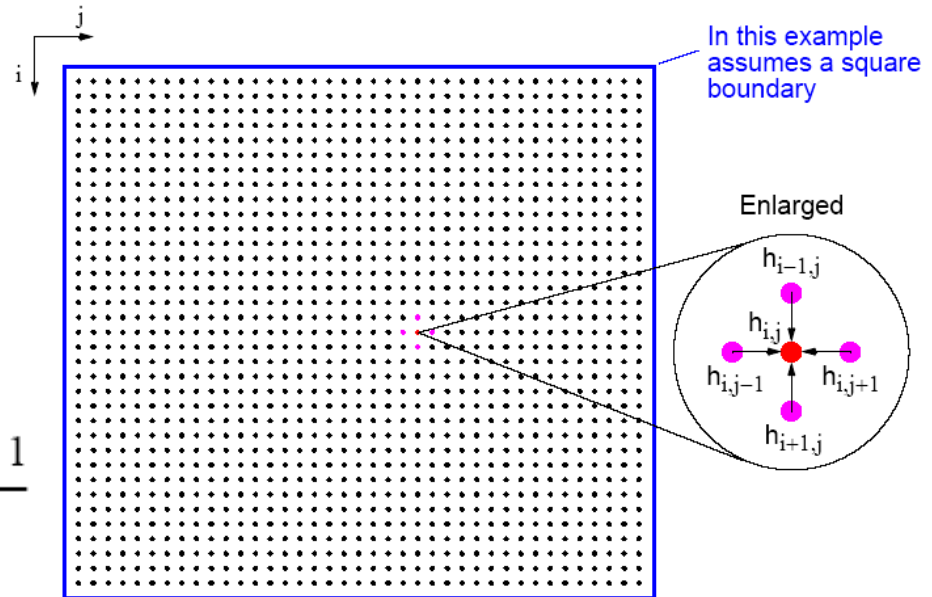
Divide area into fine mesh of points, $h_{i,j}$, $0 \leq i, j \leq n$

Temperature at an inside point taken to be average of temperatures of four neighboring points. Convenient to describe edges by points.

Temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

($0 < i < n$, $0 < j < n$) for a fixed number of iterations or until the difference between iterations less than some very small amount.



Sequential Code

Using a fixed number of iterations

```
for (iteration = 0; iteration < limit; iteration++) {  
    for (i = 1; i < n; i++)  
        for (j = 1; j < n; j++)  
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);  
    for (i = 1; i < n; i++)                                /* update points */  
        for (j = 1; j < n; j++)  
            h[i][j] = g[i][j];  
}
```

using original numbering system ($n \times n$ array).

To stop at some precision:

```
do {
    for (i = 1; i < n; i++)
        for (j = 1; j < n; j++)
            g[i][j] = 0.25*(h[i-1][j]+h[i+1][j]+h[i][j-1]+h[i][j+1]);

    continue = FALSE;                                /* indicates whether to continue */
    for (i = 1; i < n; i++)                            /* check each point for convergence */
        for (j = 1; j < n; j++)
            if (!converged(i,j) {                      /* point found not converged */
                continue = TRUE;
                break;
            }

    for (i = 1; i < n; i++)                            /* update points */
        for (j = 1; j < n; j++)
            h[i][j] = g[i][j];

} while (continue == TRUE);
```


Parallel Code

With fixed number of iterations, P_{ij}

(except for the boundary points):

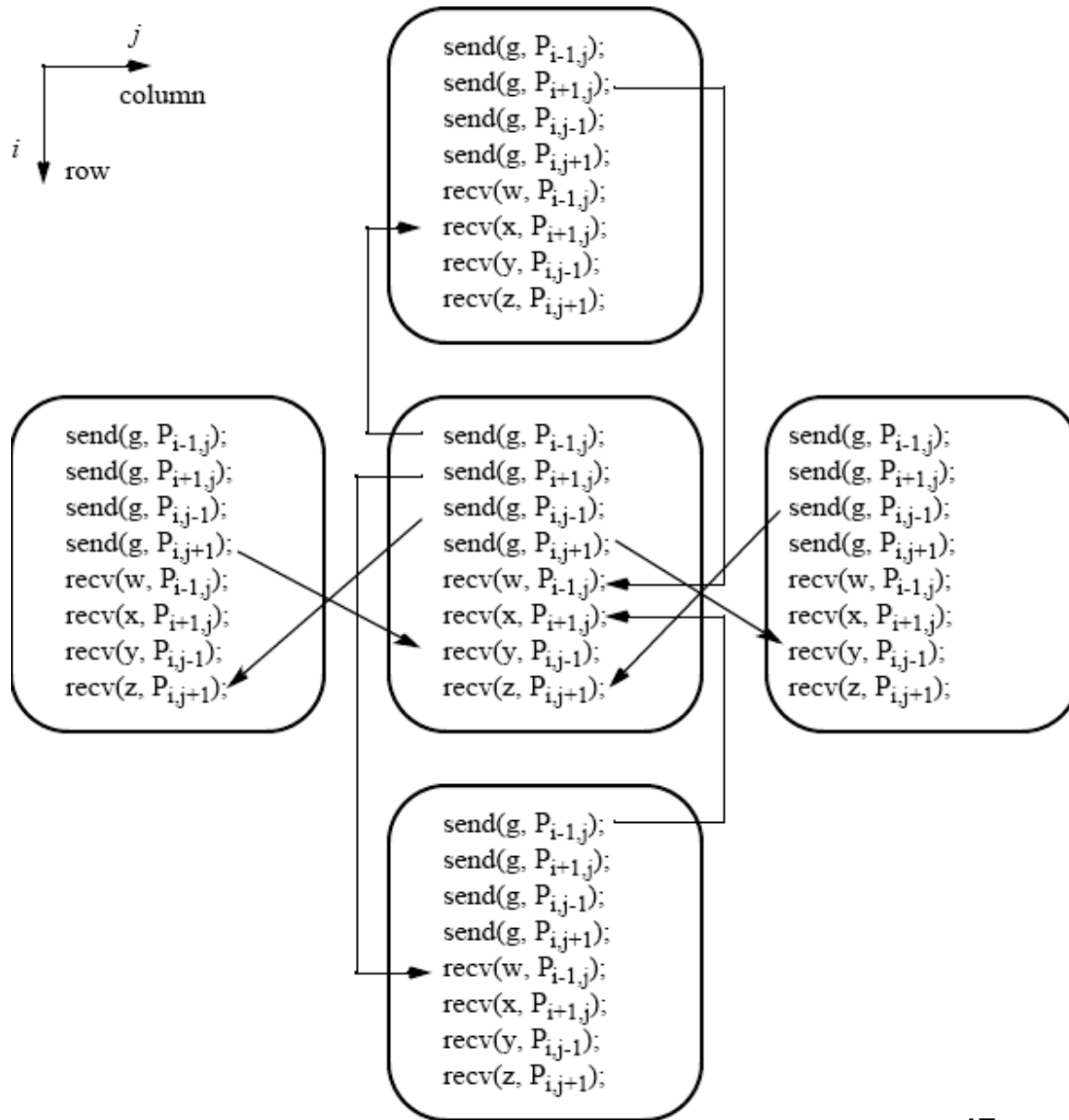
```
for (iteration = 0; iteration < limit; iteration++) {  
    g = 0.25 * (w + x + y + z);  
    send(&g, Pi-1,j);    /* non-blocking sends */  
    send(&g, Pi+1,j);  
    send(&g, Pi,j-1);  
    send(&g, Pi,j+1);  
    recv(&w, Pi-1,j);    /* synchronous receives */  
    recv(&x, Pi+1,j);  
    recv(&y, Pi,j-1);  
    recv(&z, Pi,j+1);  
}
```

Local
barrier



Important to use **send()**s that do not block while waiting for **recv()**s; otherwise processes would deadlock, each waiting for a **recv()** before moving on - **recv()**s must be synchronous and wait for **send()**s.

Message passing for heat distribution problem



Version where processes stop when they reach their required precision:

```
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    send(&g, Pi-1,j);          /* locally blocking sends */
    send(&g, Pi+1,j);
    send(&g, Pi,j-1);
    send(&g, Pi,j+1);
    recv(&w, Pi-1,j);          /* locally blocking receives */
    recv(&x, Pi+1,j);
    recv(&y, Pi,j-1);
    recv(&z, Pi,j+1);
} while ((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, Pmaster);
```


To handle the processes operating at the edges:

MPI has a construct to help here

```
if (last_row) x = bottom_value;
if (first_row) w = top_value;
if (first_column) y = left_value;
if (last_column) z = right_value;
iteration = 0;
do {
    iteration++;
    g = 0.25 * (w + x + y + z);
    if !(first_row) send(&g, Pi-1,j);
    if !(last_row) send(&g, Pi+1,j);
    if !(first_column) send(&g, Pi,j-1);
    if !(last_column) send(&g, Pi,j+1);
    if !( first_row ) recv(&w, Pi-1,j);
    if !( last_row ) recv(&x, Pi+1,j);
    if !(first_column) recv(&y, Pi,j-1);
    if !(last_column) recv(&z, Pi,j+1);
} while ((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, Pmaster);
```

MPI provides Virtual Topologies to map processes into
a mesh. /Phuong

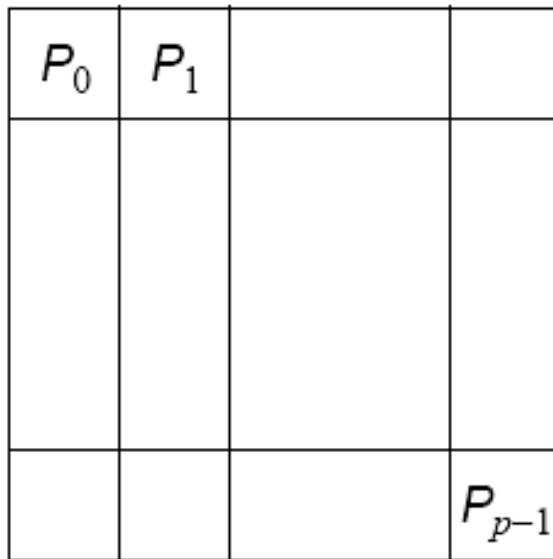
Outline

- Synchronization
 - Barrier implementation
- Synchronized computations
 - Data parallel computations
 - Prefix Sum Problem
 - Synchronous iterations
 - General system of linear equations
 - Locally synchronous computations
 - Heat distribution
 - Implementation issues
 - Partitioning, deadlock

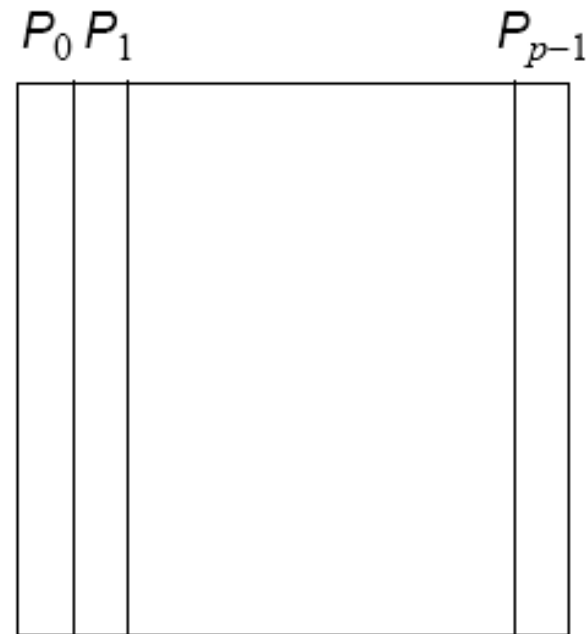
Partitioning

Normally allocate more than one point to each processor, because many more points than processors.

Points could be partitioned into square blocks or strips:



Blocks

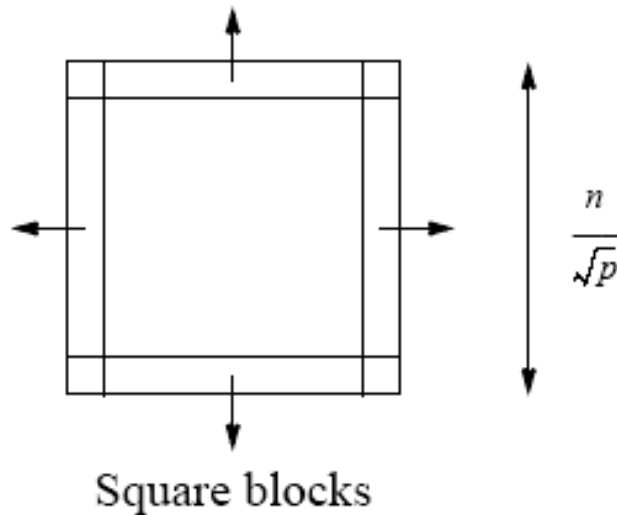


Strips (columns)

Block partition

Four edges where data points exchanged.
Communication time given by

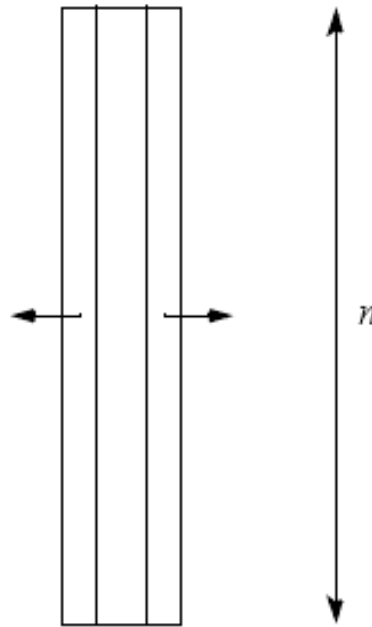
$$t_{\text{commsq}} = 8 \left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}} \right)$$



Strip partition

Two edges where data points are exchanged.
Communication time is given by

$$t_{\text{commcol}} = 4(t_{\text{startup}} + nt_{\text{data}})$$



Strips

Optimum

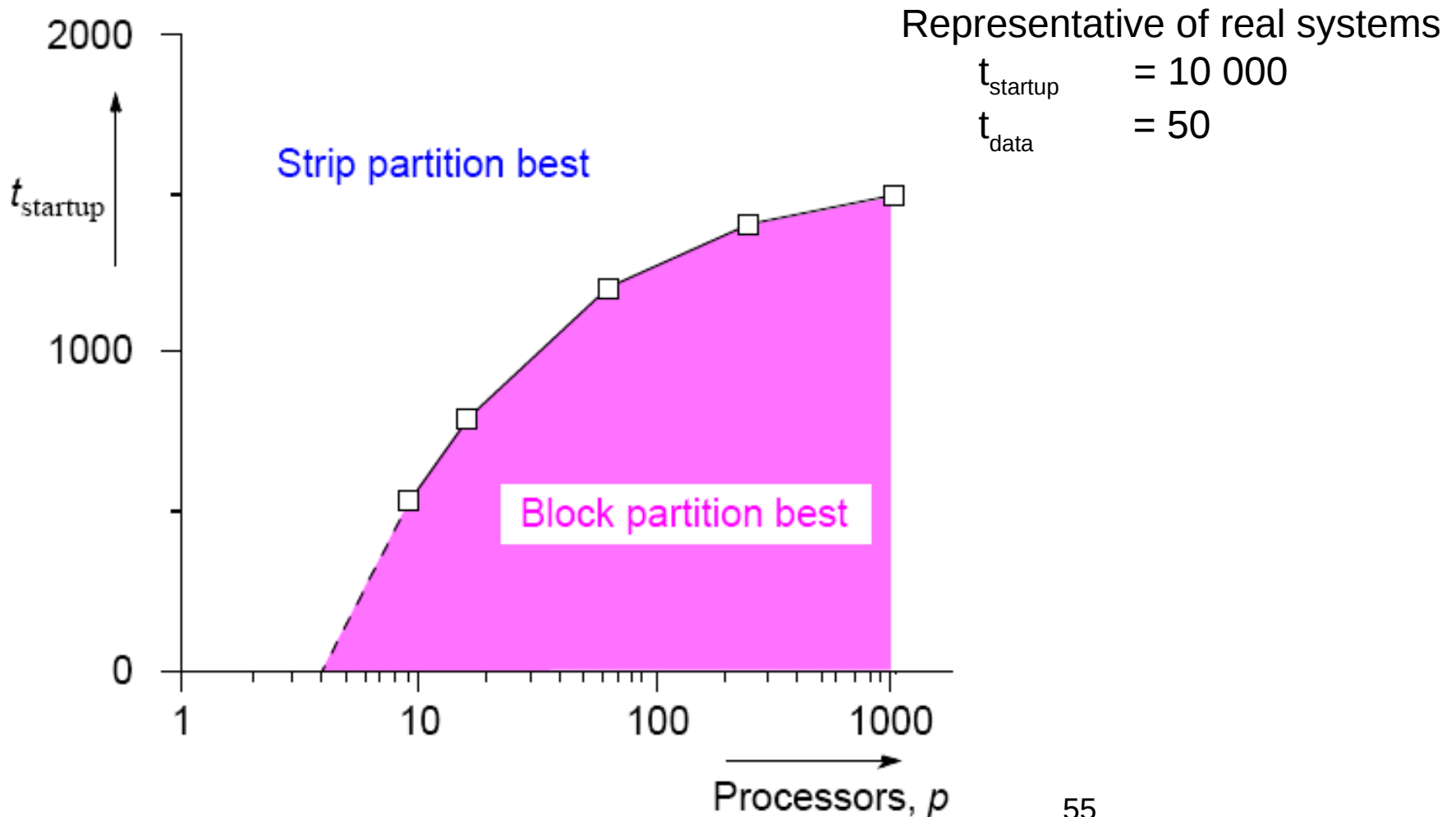
In general, strip partition best for large startup time, and block partition best for small startup time.

With the previous equations, block partition has a larger communication time than strip partition if

$$t_{\text{startup}} > n \left(1 - \frac{2}{\sqrt{p}} \right) t_{\text{data}}$$

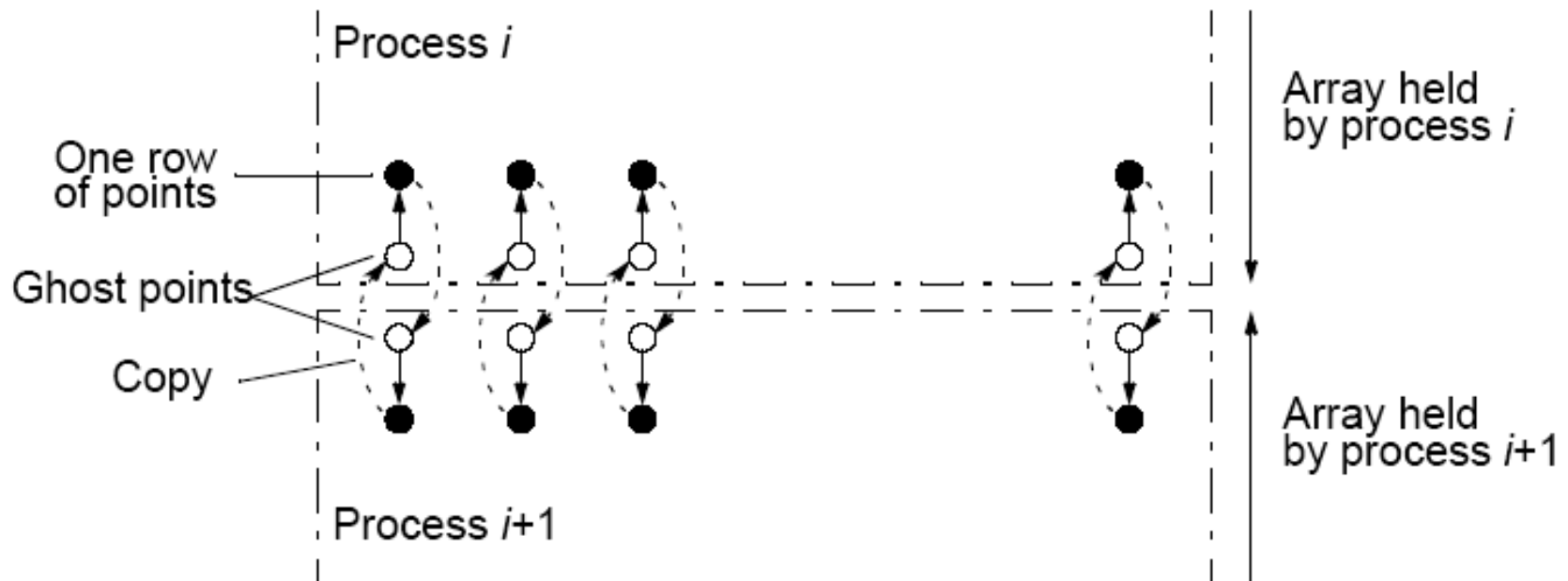
$(p \geq 9)$.

Startup times for block and strip partitions



Ghost Points

Additional row of points at each edge that hold values from adjacent edge. Each array of points increased to accommodate ghost rows.



Safety and Deadlock

When all processes send their messages **first** and then receive all of their messages is “**unsafe**” because it relies upon buffering in the **send()**s. The amount of buffering is not specified in MPI.

If insufficient storage available, send routine may be delayed from returning until storage becomes available or until the message can be sent without buffering.

Then, a locally blocking **send()** could behave as a **synchronous send()**, only returning when the matching **recv()** is executed. Since a matching **recv()** would never be executed if all the **send()**s are synchronous, **deadlock would occur**.

Making the code safe

Alternate the order of the **send()**s and **recv()**s in adjacent processes so that only one process performs the **send()**s first.

Then even synchronous **send()**s would not cause deadlock.

Good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

MPI Safe Message Passing Routines

MPI offers several methods for safe communication:

- Combined send and receive routines:

`MPI_Sendrecv()`

which is guaranteed not to deadlock

- Buffered send(s):

`MPI_Bsend()`

here the user provides explicit storage space

- Nonblocking routines:

`MPI_Isend()` and `MPI_Irecv()`

which return immediately.

Separate routine used to establish whether message has been received:

`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`,
`MPI_Testall()`, or `MPI_Testany()`.

References

- Barry Wilkinson & Michael Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.
- D. E. Culler et al.. Parallel Computer Architecture – A Hardware/Software Approach