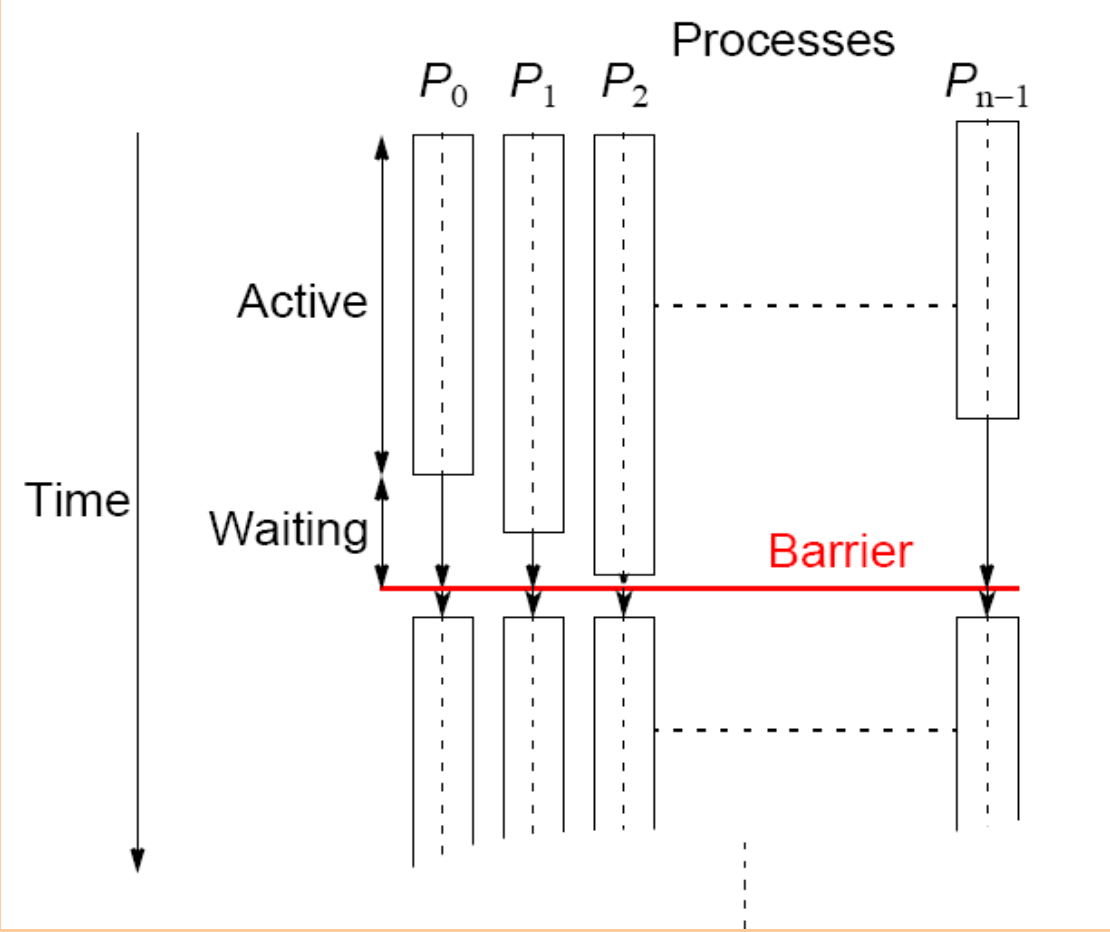
Synchronous Computations

Synchronous Computations

- In a (fully) synchronous application, all the processes are synchronized at regular points.
- **Barrier**
  - A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait.
  - All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

Processes reaching barrier at different times

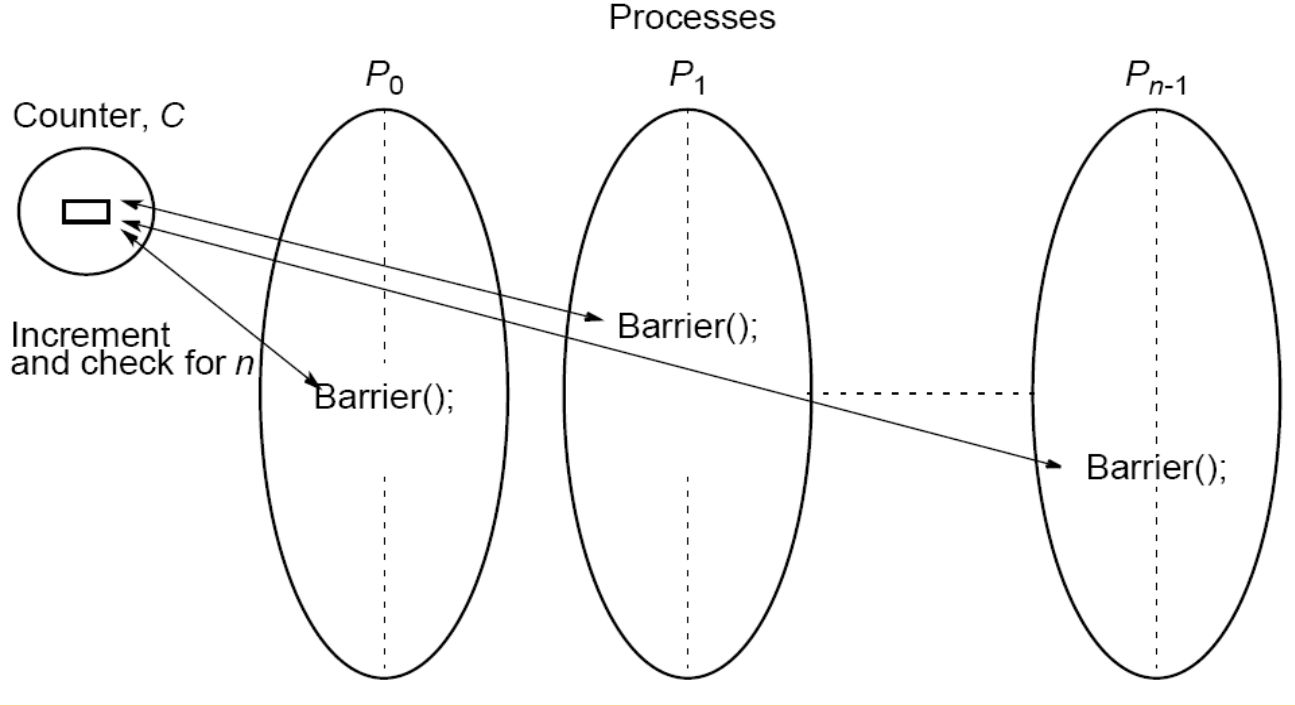


MPI

- **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 (*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.
- Good 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 barrier for a second time before previous processes have left barrier for the first time. Two-phase handles this scenario.

Example code:

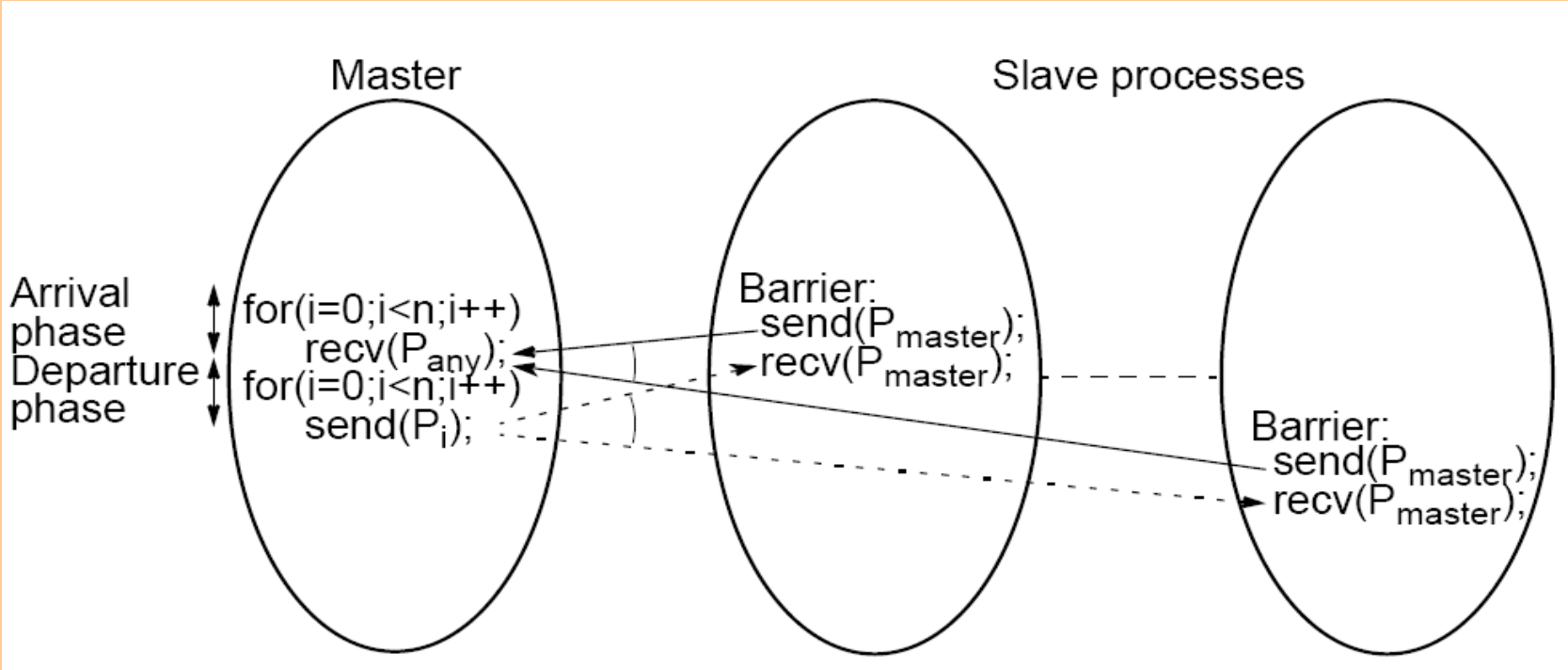
- **Master:**

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

- **Slave processes:**

```
send(Pmaster);
Recv(Pmaster);
```

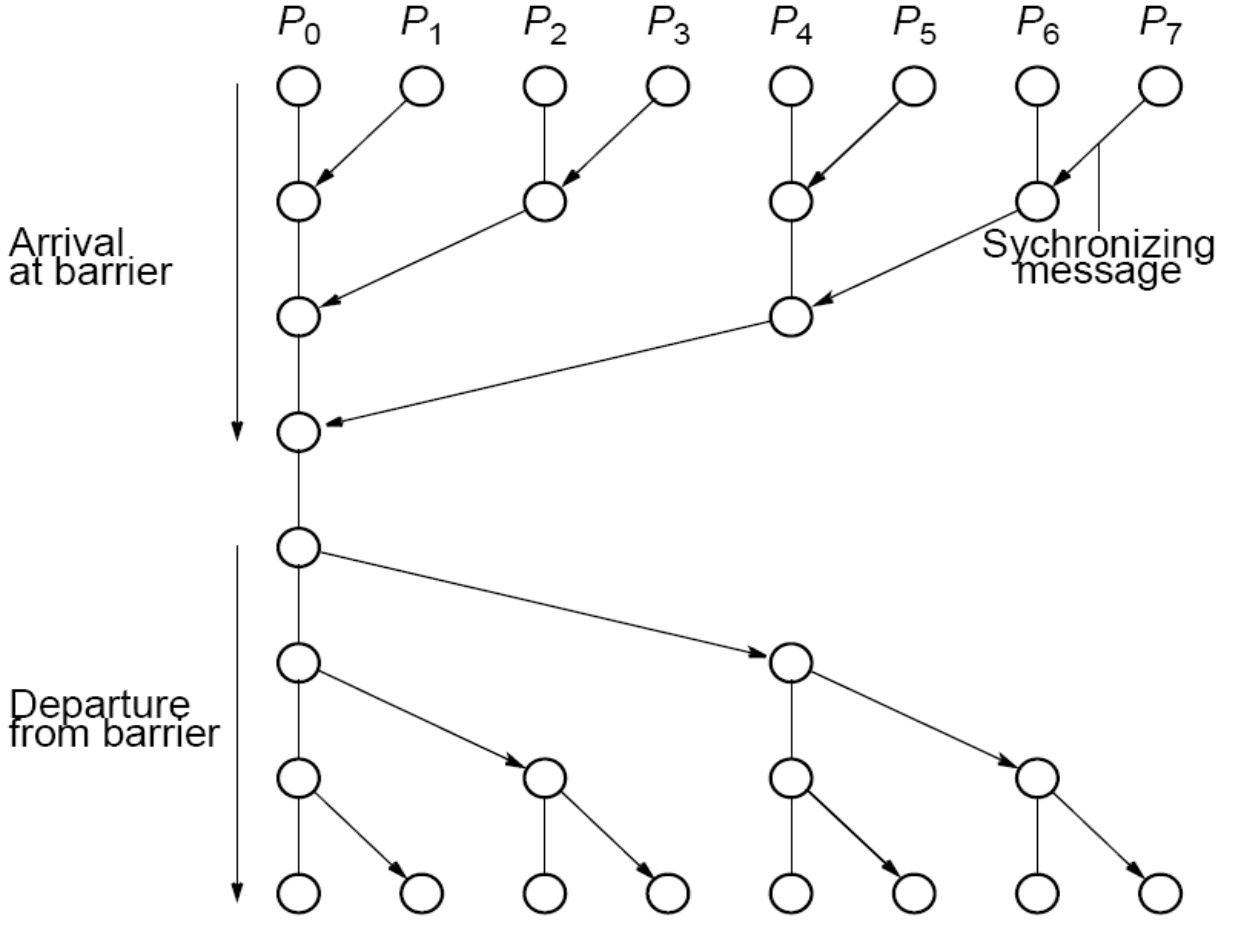
Barrier implementation in a message-passing system



Tree Implementation

- More efficient. Suppose 8 processes, *P*0, *P*1, *P*2, *P*3, *P*4, *P*5, *P*6, *P*7:
- First 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)
- Second 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)
- Third 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.

Tree Barrier

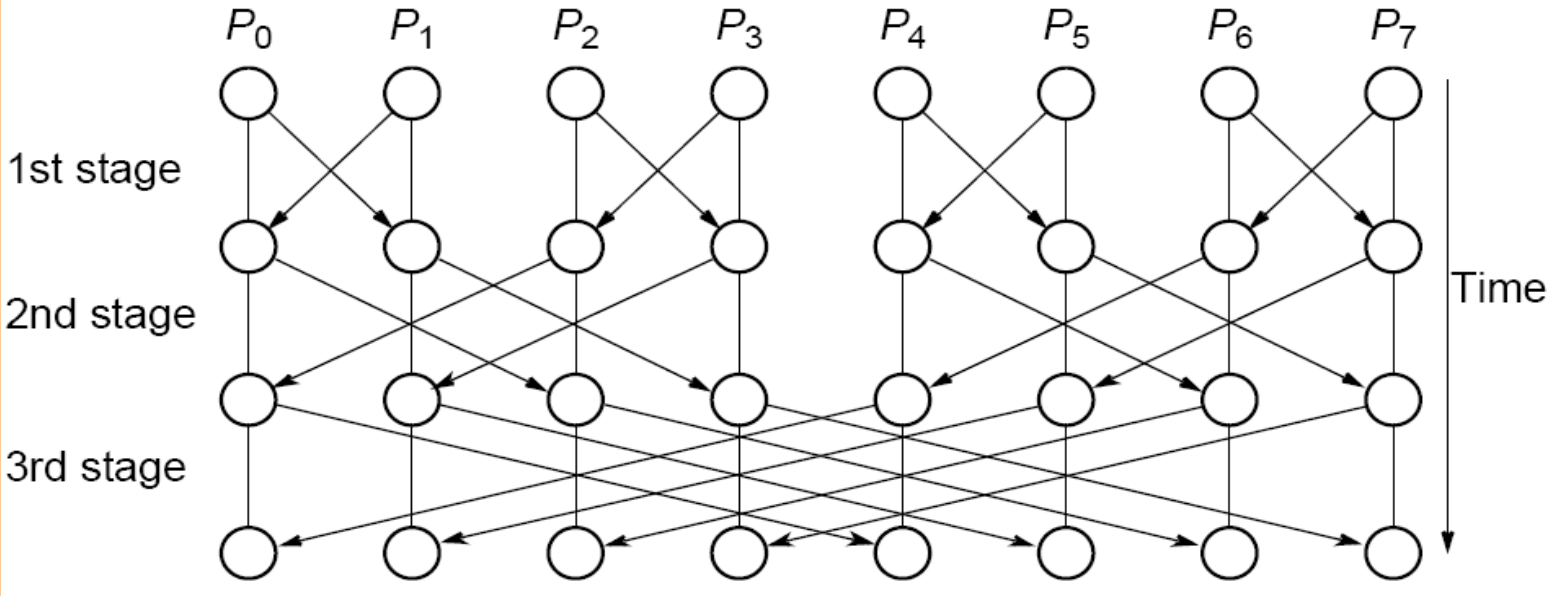


Butterfly Barrier

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

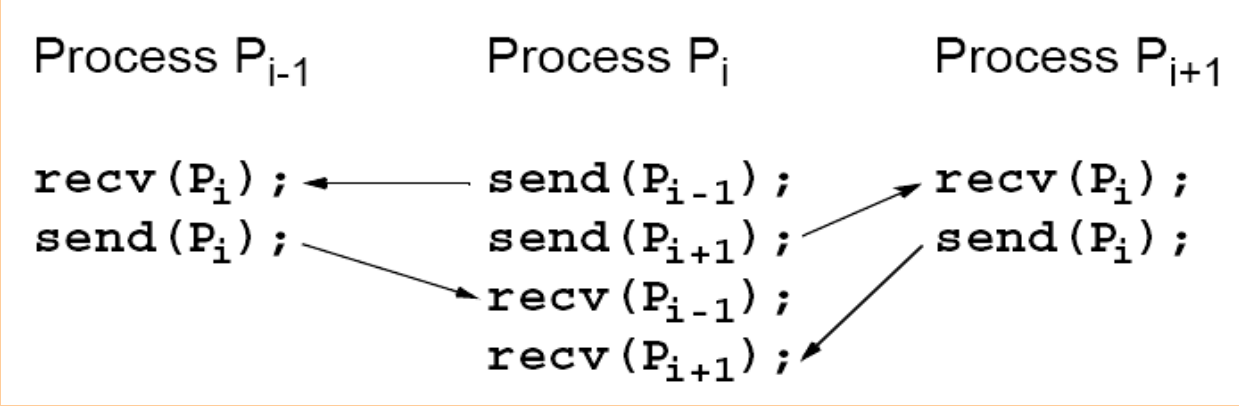
Second stage $P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$

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



Local Synchronization

- Suppose a process *Pi* needs to be synchronized and to exchange data with process *Pi*-1 and process *Pi*+1 before continuing:



- Not a perfect three-process barrier because process *Pi*-1 will only synchronize with *Pi* and continue as soon as *Pi* allows.
- Similarly, process *Pi*+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.

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

- MPI provides **MPI_Sendrecv()** and **MPI_Sendrecv_replace().**

Example

Process P_{i-1}	Process P_i	Process P_{i+1}
`sendrecv(Pi);` ↔	`sendrecv(Pi-1);`	
	`sendrecv(Pi+1);` ↔	`sendrecv(Pi);`

sendrev()s have 12 parameters!

Synchronized Computations

- Can be classififed 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

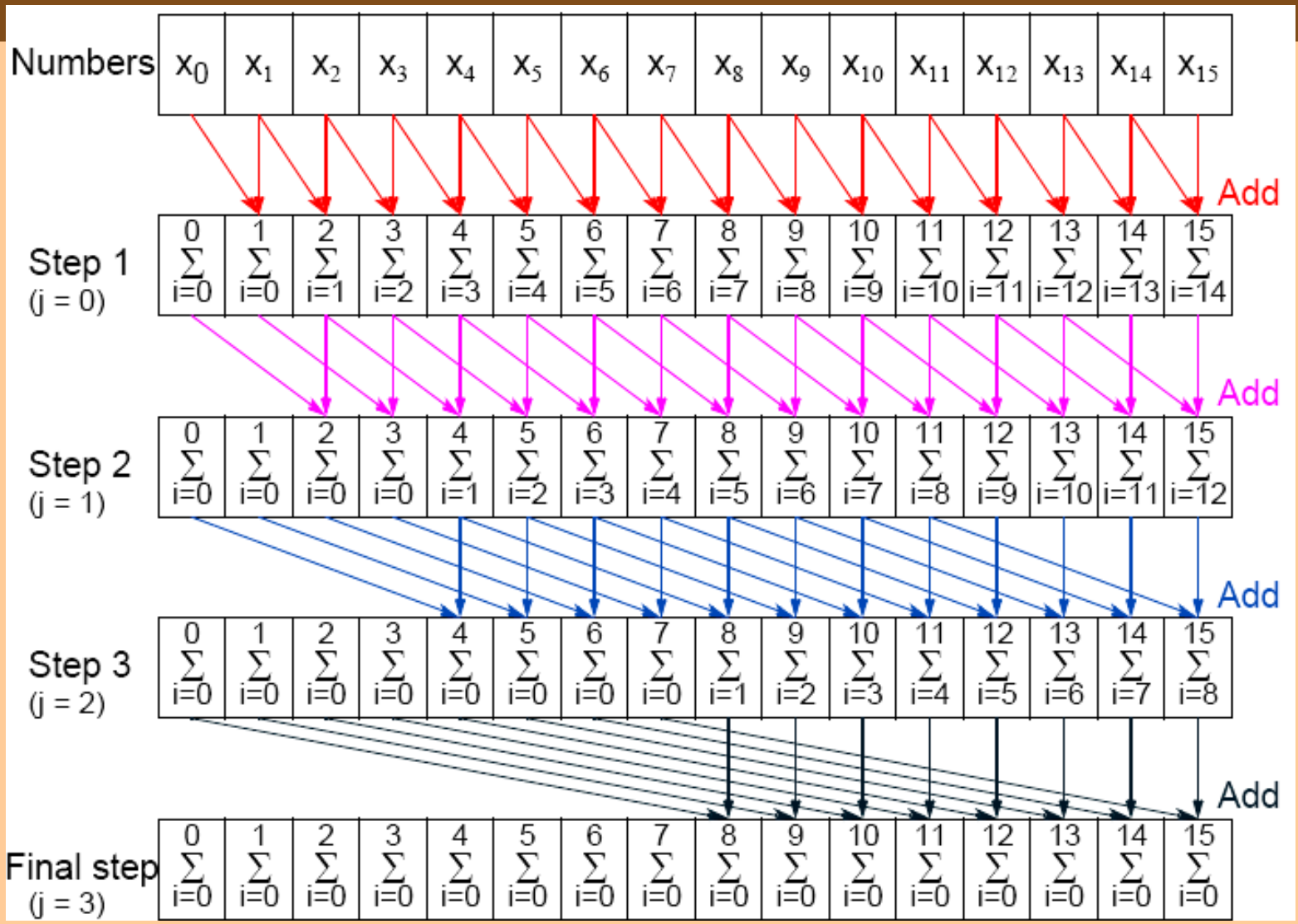
Fully Synchronized Computation Examples

- 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 larger problem sizes.
  - Many numeric and some non-numeric problems can be cast in a data parallel form.

Data Parallel Example - Prefix Sum Problem

- Given a list of numbers, $x0$, …, xn-1, compute all the partial
- summations (i.e., $x0 + x1$; $x0 + x1 + x2$; $x0 + x1 + x2 + x3$; …).
- 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



Synchronous Iteration

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 */
        }
```

Synchronous Iteration

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

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \cdots \quad & + a_{n-1,n-1}x_{n-1} = b_{n-1} \\
& \vdots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \cdots \quad & + a_{2,n-1}x_{n-1} = b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \cdots \quad & + a_{1,n-1}x_{n-1} = b_1 \\
a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \cdots \quad & + a_{0,n-1}x_{n-1} = b_0
\end{aligned}
$$

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

- By rearranging the *i*th equation:

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

to

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

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 and can be 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 the Jacobi method will converge if the diagonal values of *a* have an absolute value greater than the 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

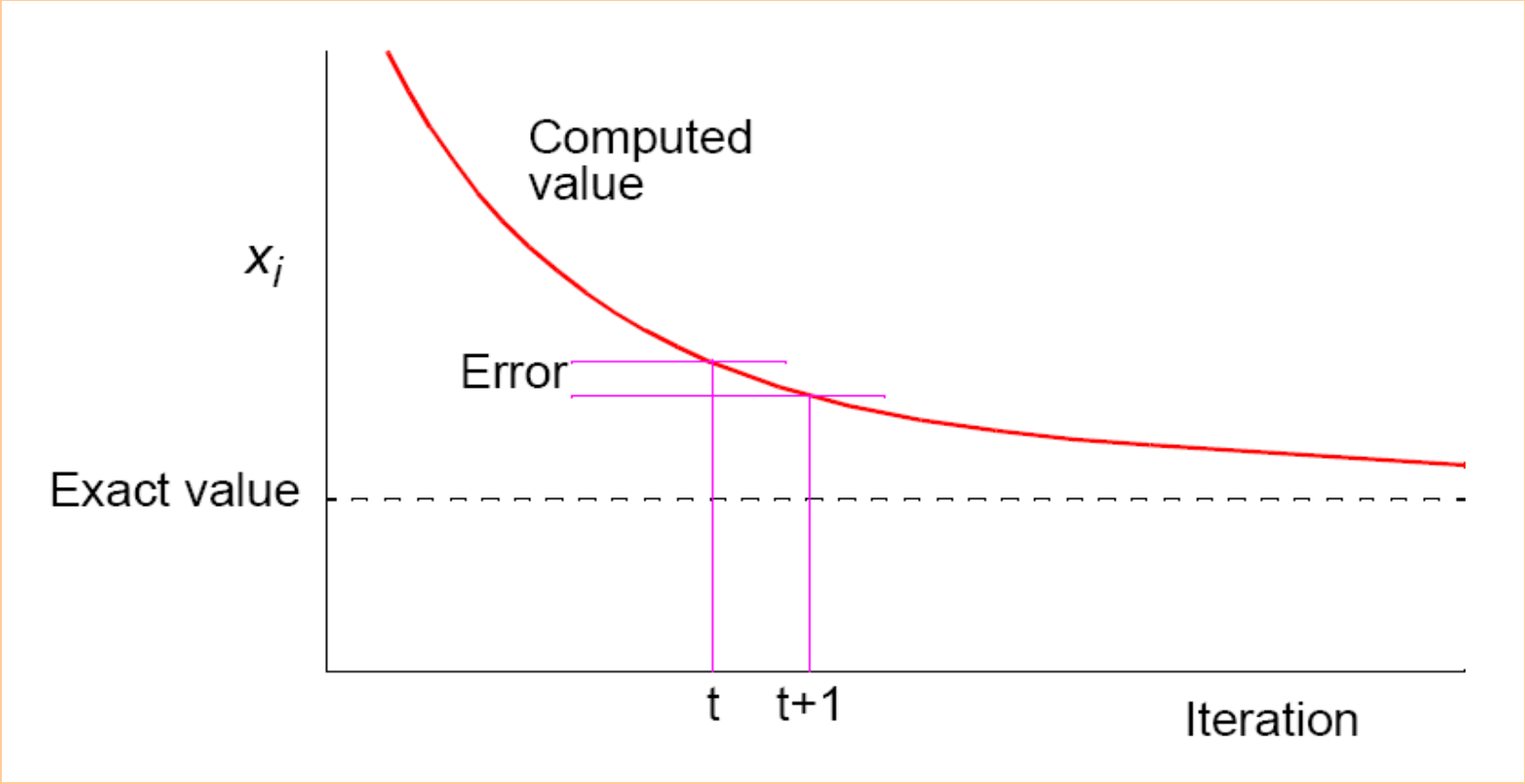
- A simple, common approach. Compare values computed in one iteration to values obtained from the previous iteration.
- Terminate computation when all values are within given tolerance; i.e., when

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

for all i, where x_i^t is the value of x_i after the tth 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

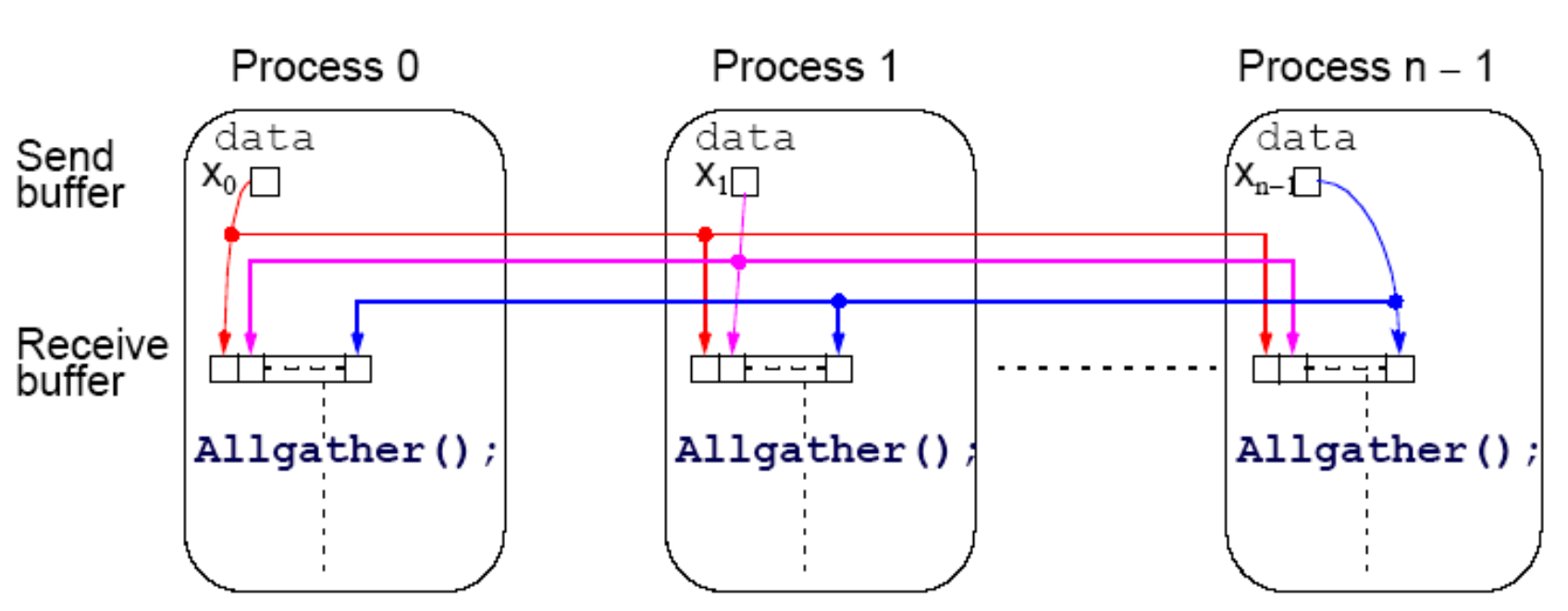
- Process *Pi* could be of the form

```
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]);                      /*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.

Allgather

- Broadcast and gather values in one composite construction.

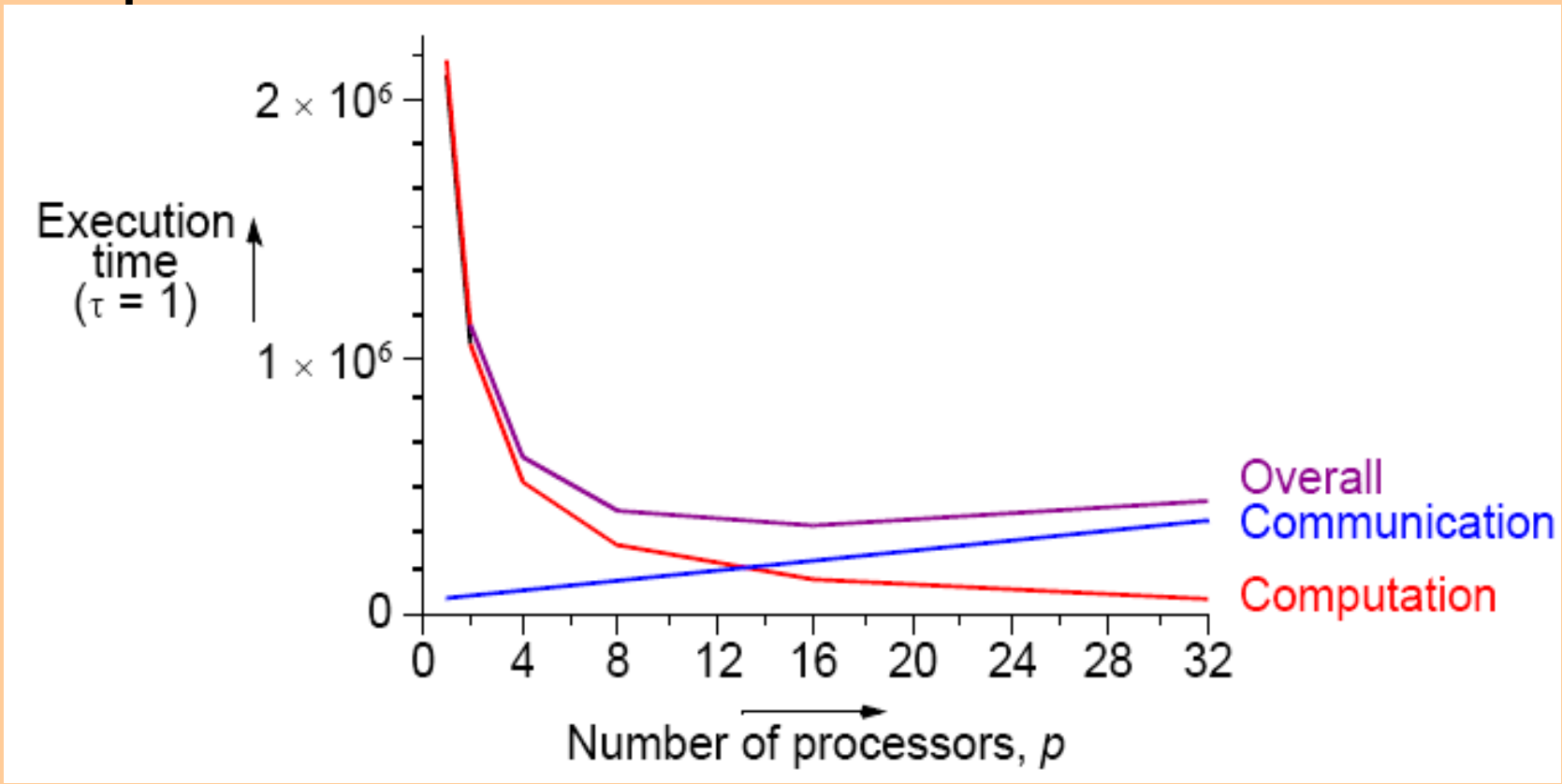


Partitioning

- 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.
  - *cyclic* allocation – processors are allocated one unknown in order; i.e., processor $P0$ is allocated $x0$, xp, $x2p$, …, $x((n/p)-1)p$, processor $P1$ is allocated $x1$, $xp+1$, $x2p+1$, …, $x((n/p)-1)p+1$, and so on.
- Cyclic allocation no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).

Effects of computation and communication in Jacobi iteration

- Consequences of different numbers of processors done in textbook.



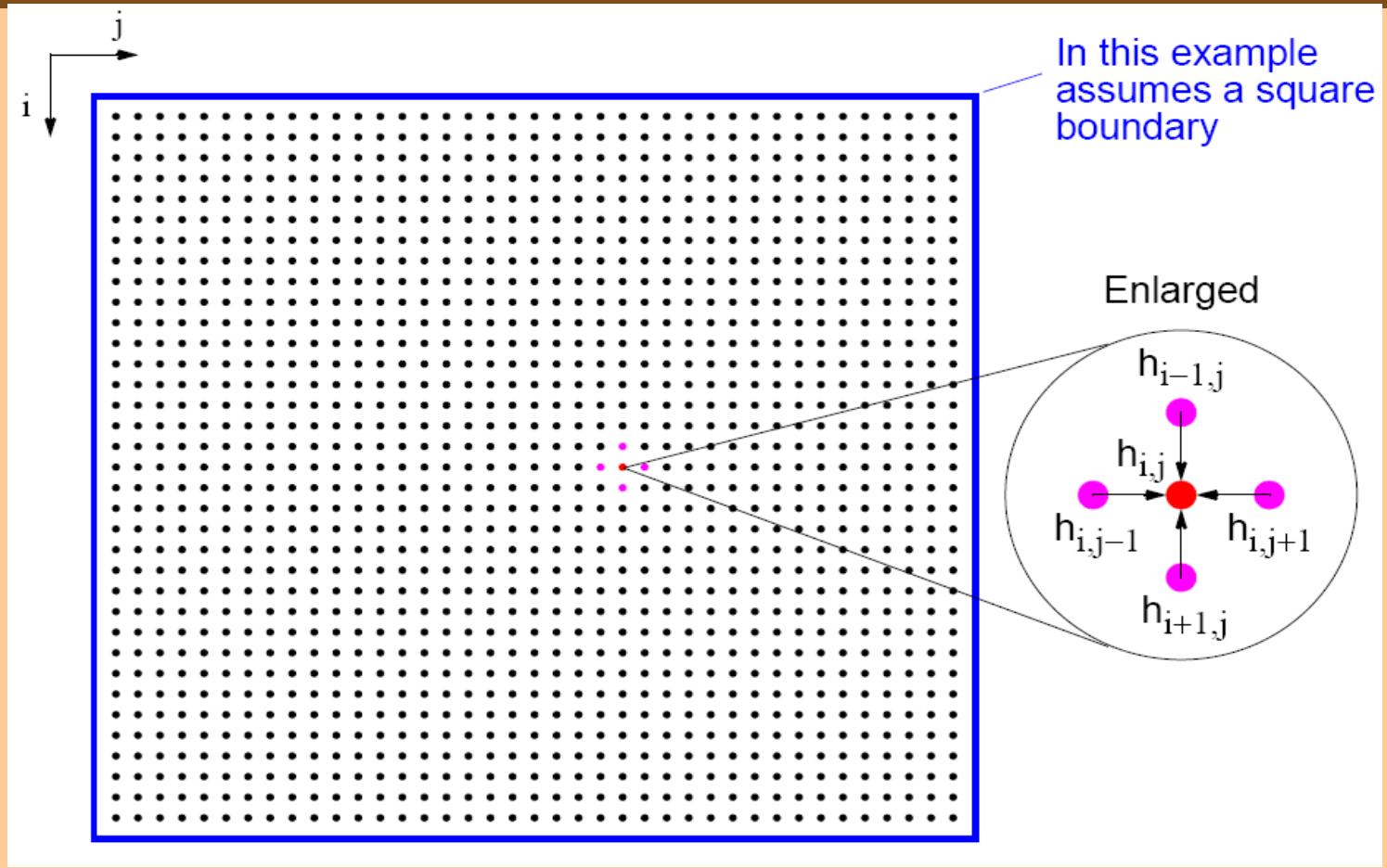
Locally Synchronous Computation

- **Heat Distribution Problem**
  - An area has known temperatures along each of its edges. Find the temperature distribution within.
  - Divide area into fine mesh of points, hi,j. 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:

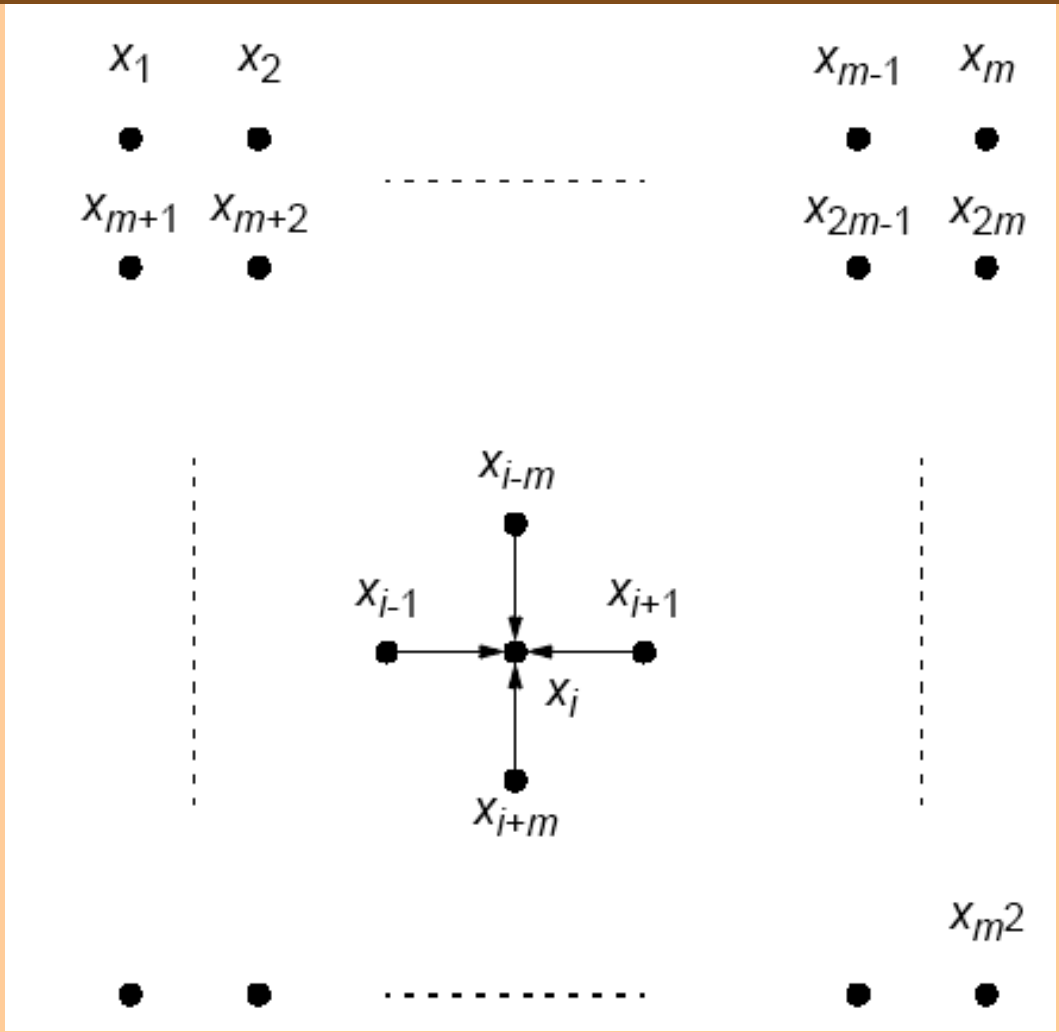
$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

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

Heat Distribution Problem



Natural ordering of heat distribution problem



- Number points from 1 for convenience and include those representing the edges. Each point will then use the equation

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4}$$

- Could be written as a linear equation containing the unknowns *xi-k*, *xi*-1, *xi*+1, and *xi+k*:

$$x_{i-k} + x_{i-1} - 4x_i + x_{i+1} + x_{i-k} = 0$$

- Notice: solving a (sparse) system of linear equations.
- Also solving Laplace's equation.

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];
}
```

Parallel Code

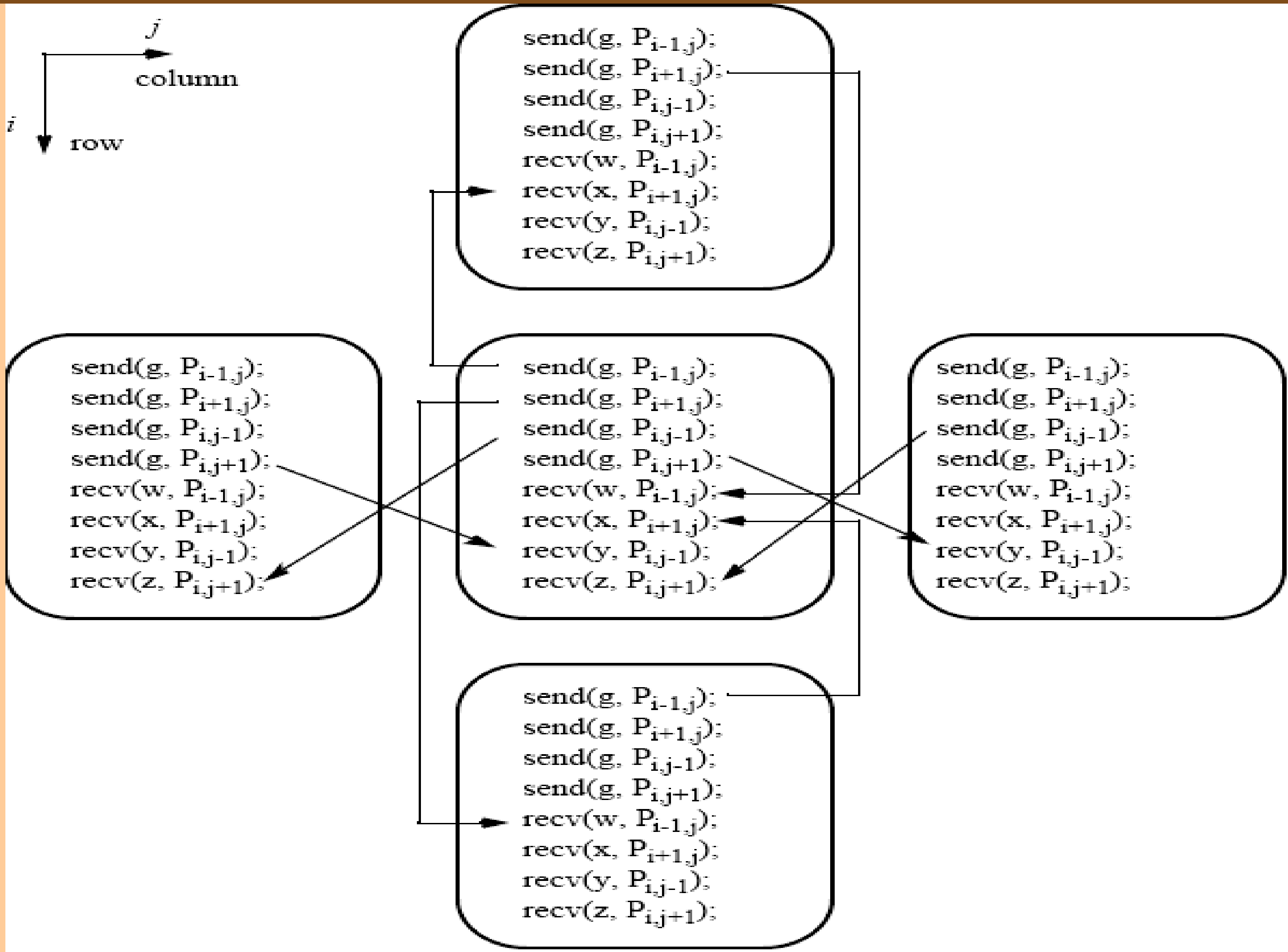
- With fixed number of iterations, *Pi,j* (except for the boundary points):

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

Local barrier

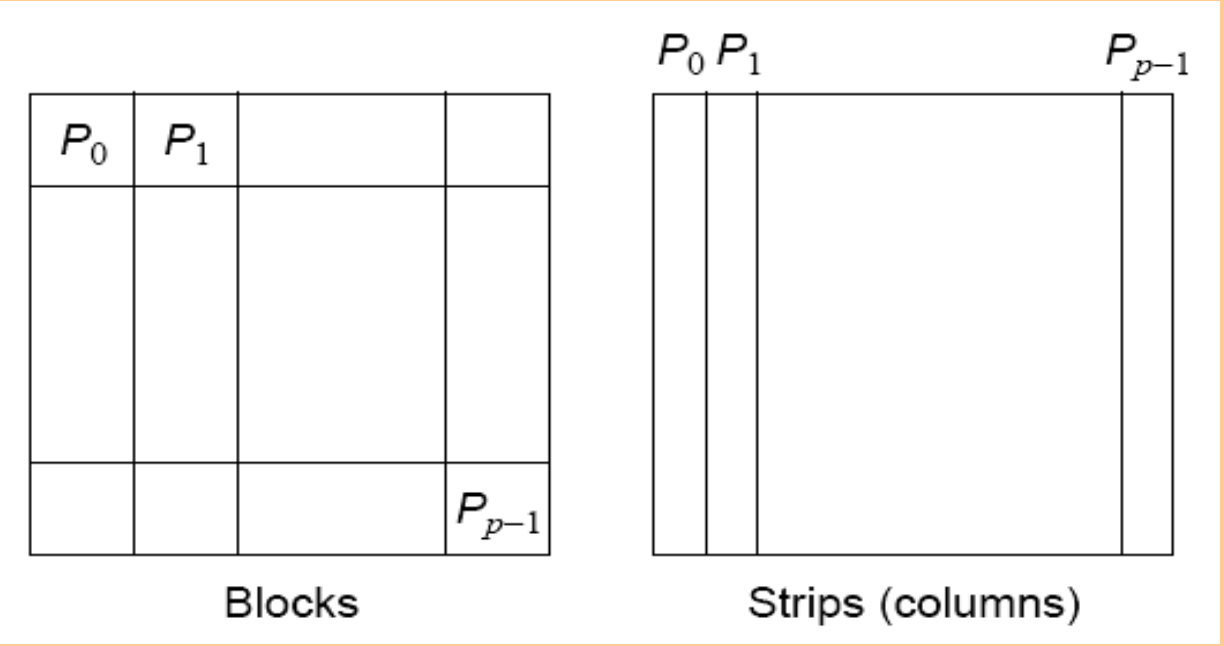
- Important to use **send()**s that do not block while waiting for the **recv()**s; otherwise the processes would deadlock, each waiting for a **recv()** before moving on - **recv()**s must be synchronous and wait for the **send()**s.

Message passing for heat distribution problem



Partitioning

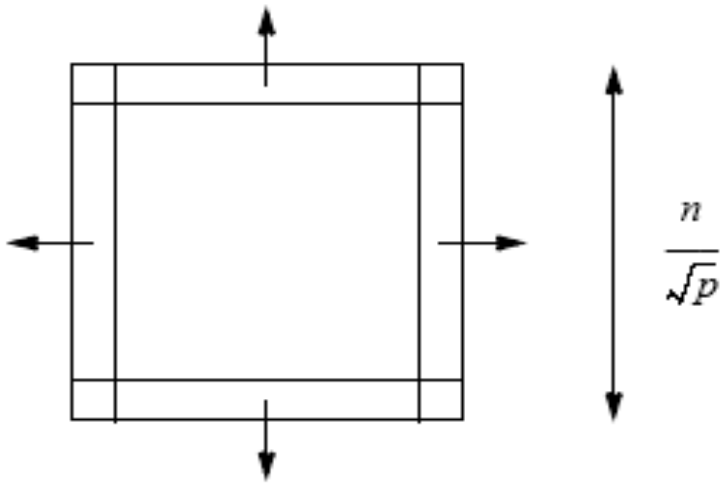
- Normally allocate more than one point to each processor, because many more points than processors.
- Points could be partitioned into square blocks or strips:



Block partition

- Four edges where data points exchanged. Communication time given by

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

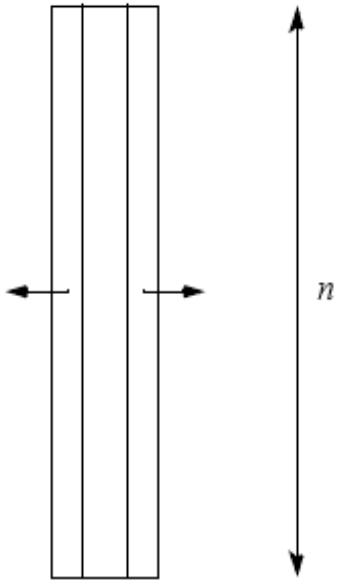


Square blocks

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

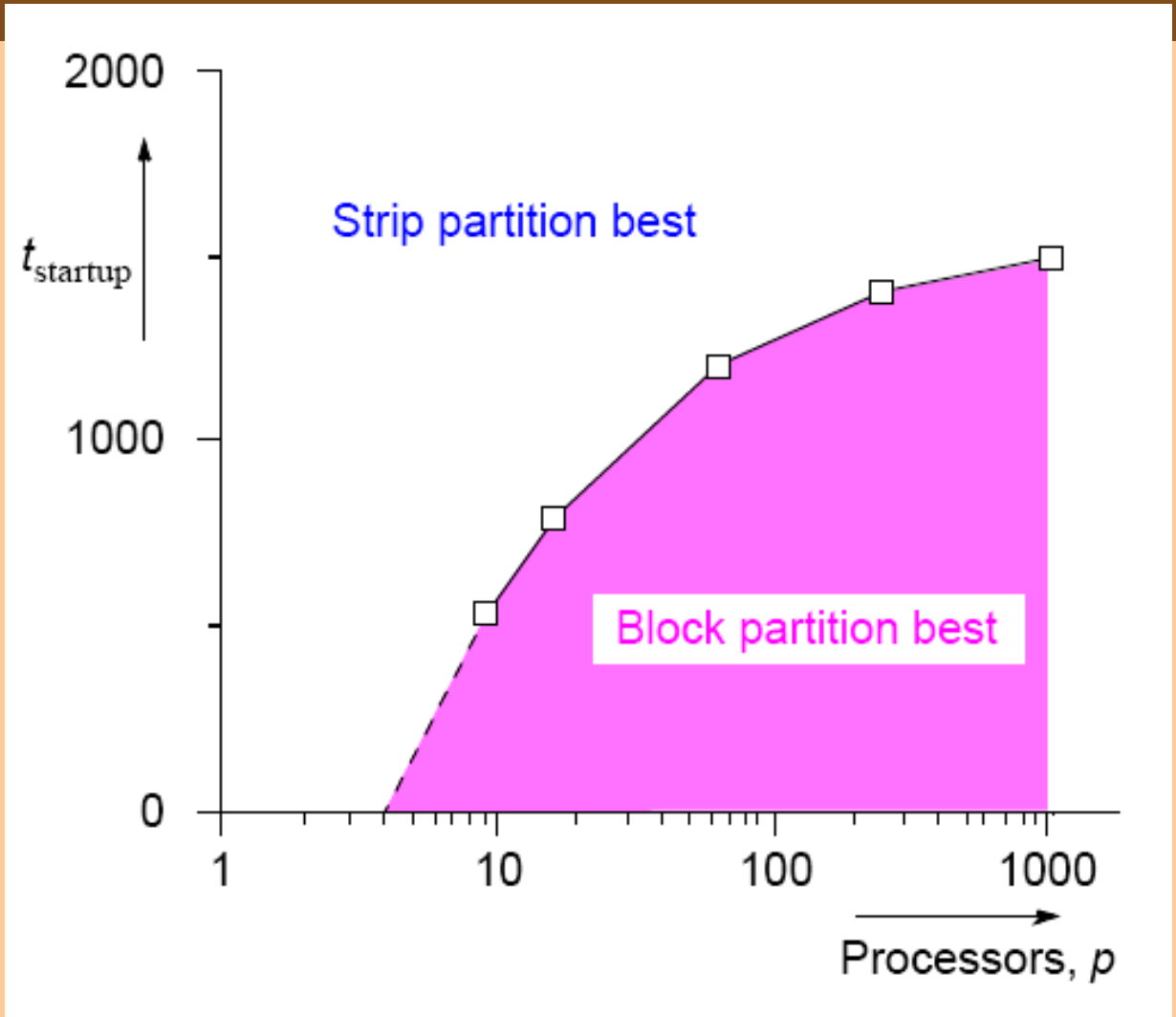
- In general, the strip partition is best for a large startup time, and a block partition is best for a small startup time.
- With the previous equations, the block partition has a larger communication time than the strip partition if

$$8\left(t_{\text{startup}} + \frac{n}{\sqrt{p}} t_{\text{data}}\right) > 4(t_{\text{startup}} + nt_{\text{data}})$$

Or

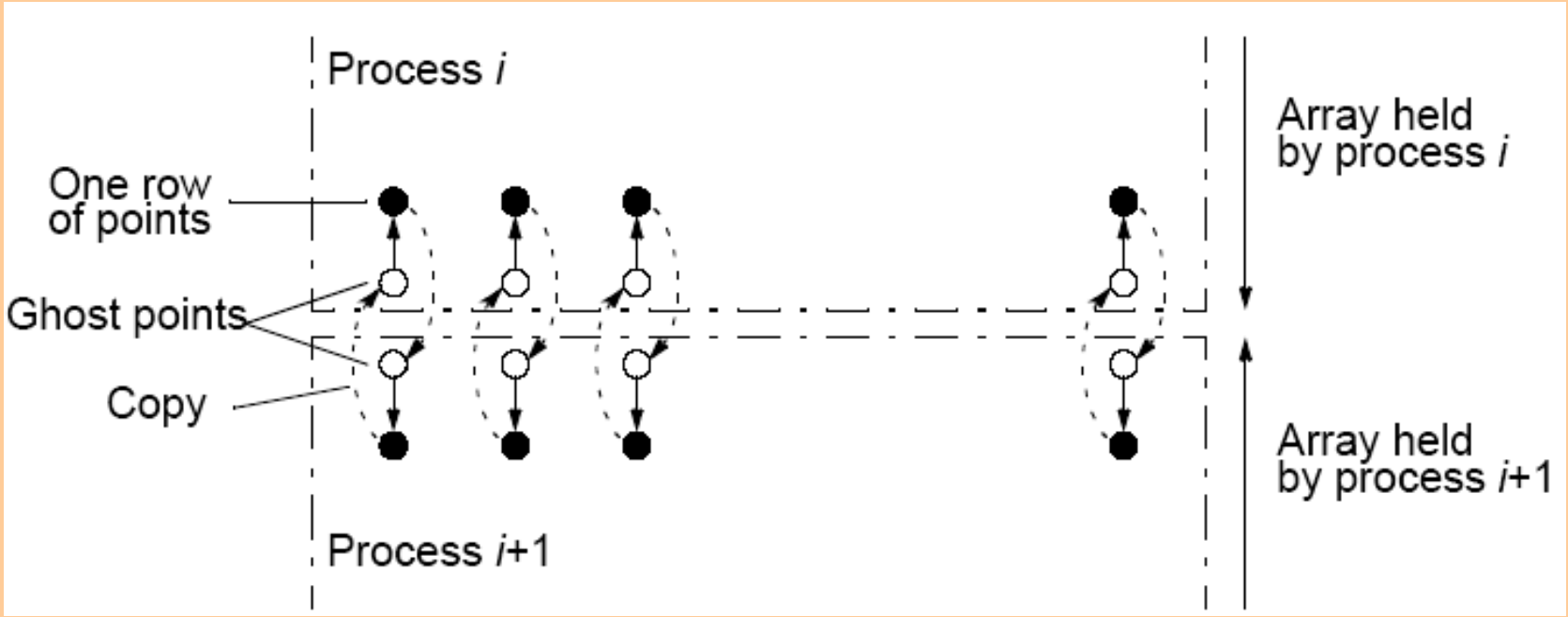
$$t_{\text{startup}} > n\left(1 - \frac{2}{\sqrt{p}}\right) t_{\text{data}}$$

Startup times for block and strip partitions



Ghost Points

- An additional row of points at each edge that hold the values from the adjacent edge. Each array of points is increased to accommodate the ghost rows.



Safety and Deadlock

- If all processes send their messages first and then receive all of their messages this 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 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 alternative 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()**