Chapter 6

Synchronous Computations

Synchronous Computations

In a (fully) synchronous application, all the processes 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
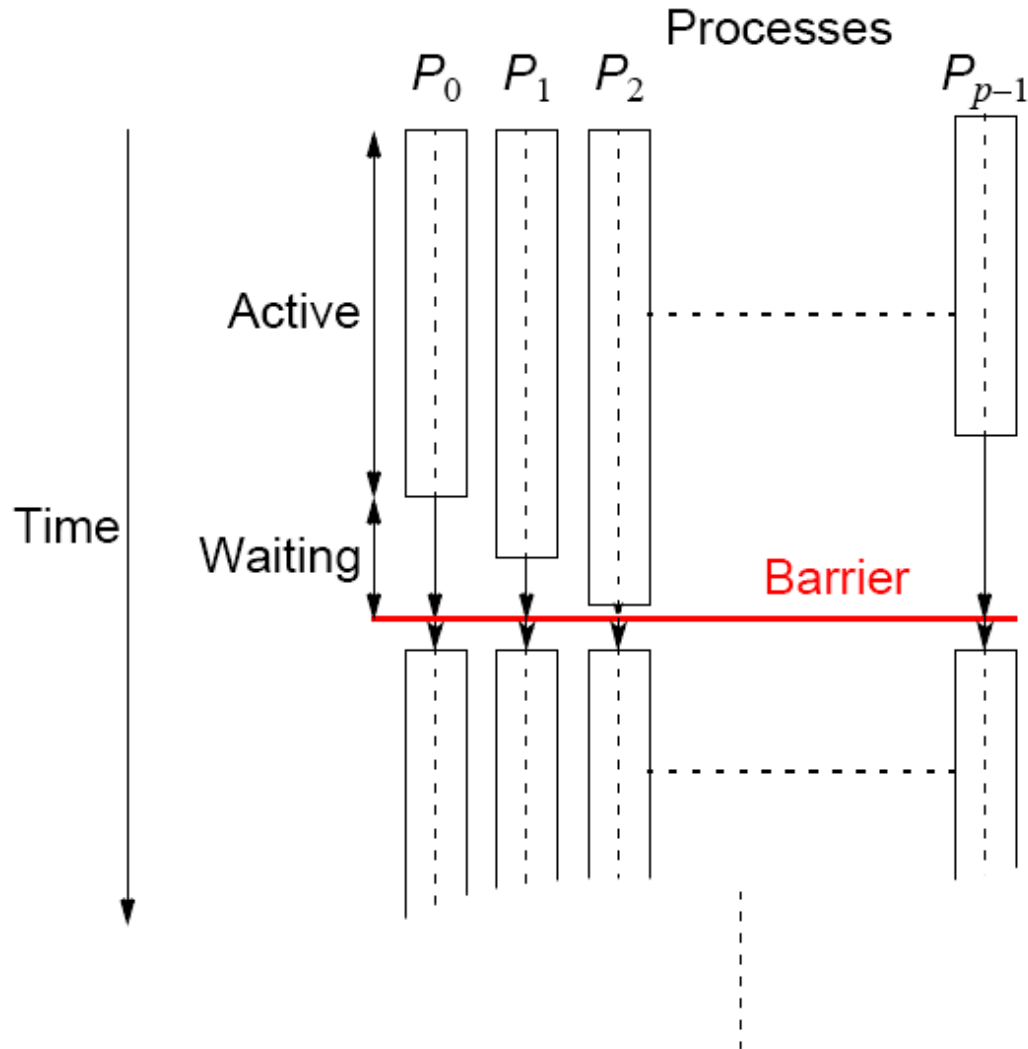
In message-passing systems, barriers provided with library routines:

Processes

P_0 P_1 P_{p-1}

```
Barrier();
```

```
Barrier();
```

```
Barrier();
```

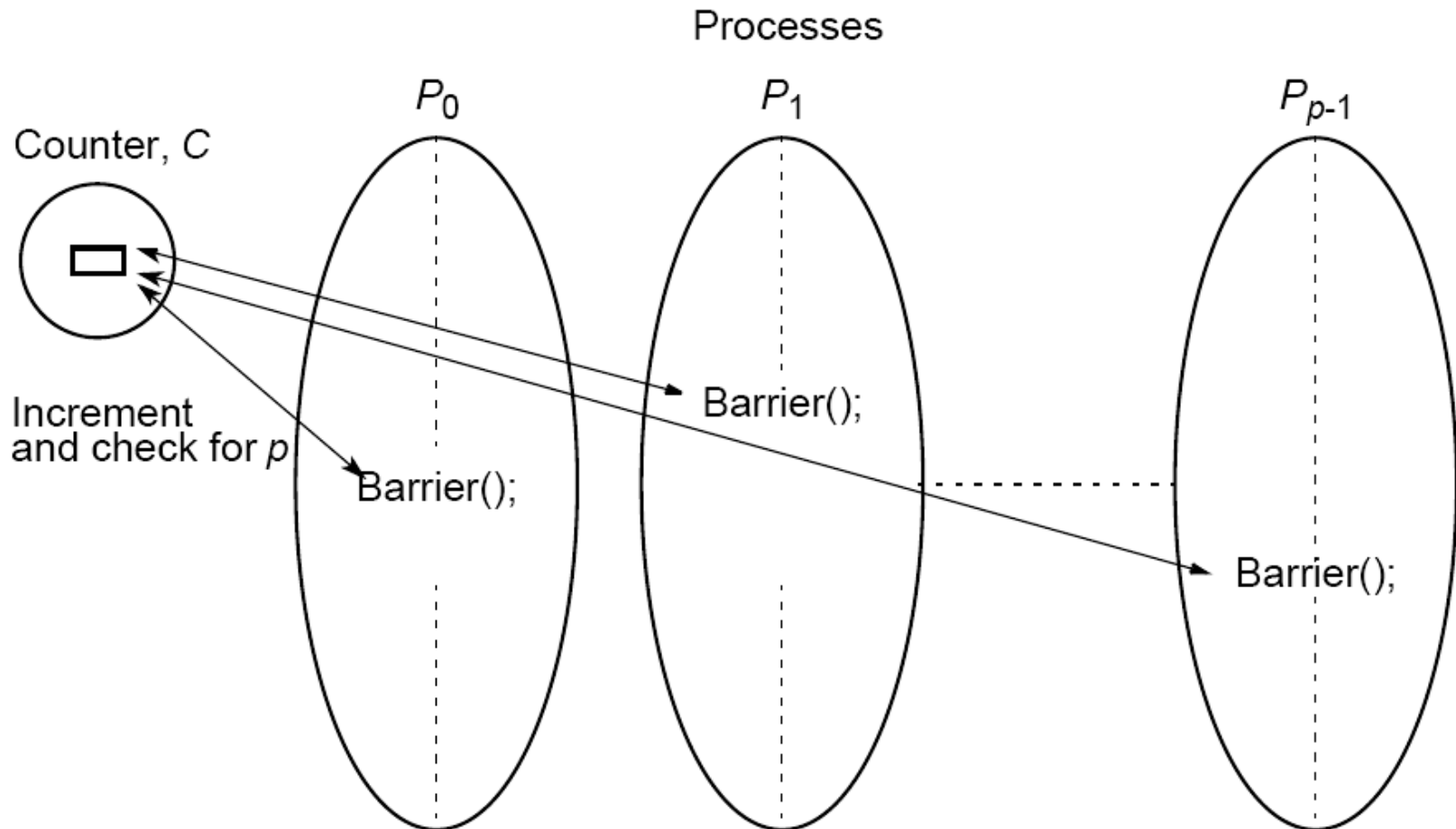Processes wait until all reach their barrier call

6.4

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*):

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.

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.

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

Master   Slave processes

Arrival phase

Departure phase

```
for(i=0;i<n;i++)
    recv(P_any);
for(i=0;i<n;i++)
    send(P_i);
```

Barrier:
```
send(P_master);
recv(P_master);
```

Barrier:
```
send(P_master);
recv(P_master);
```
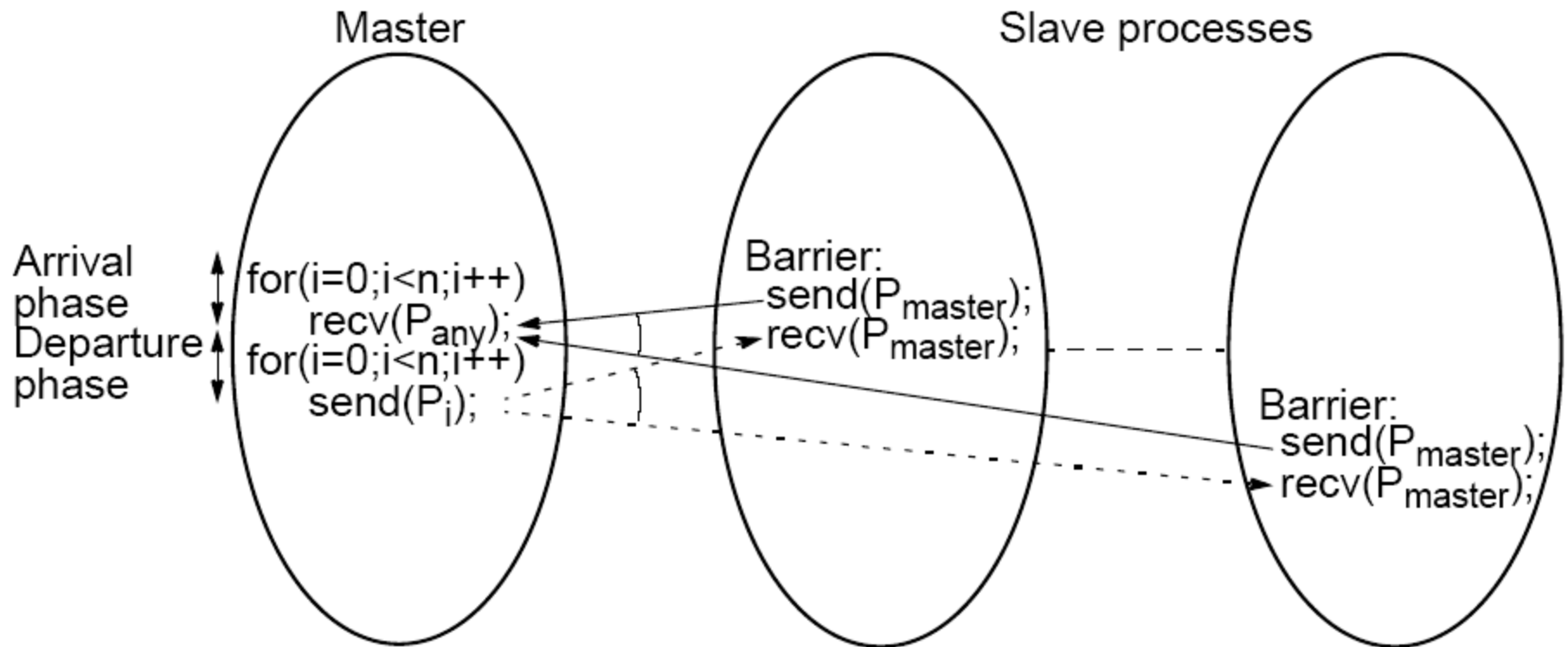
Tree Implementation

More efficient. O(log p) steps
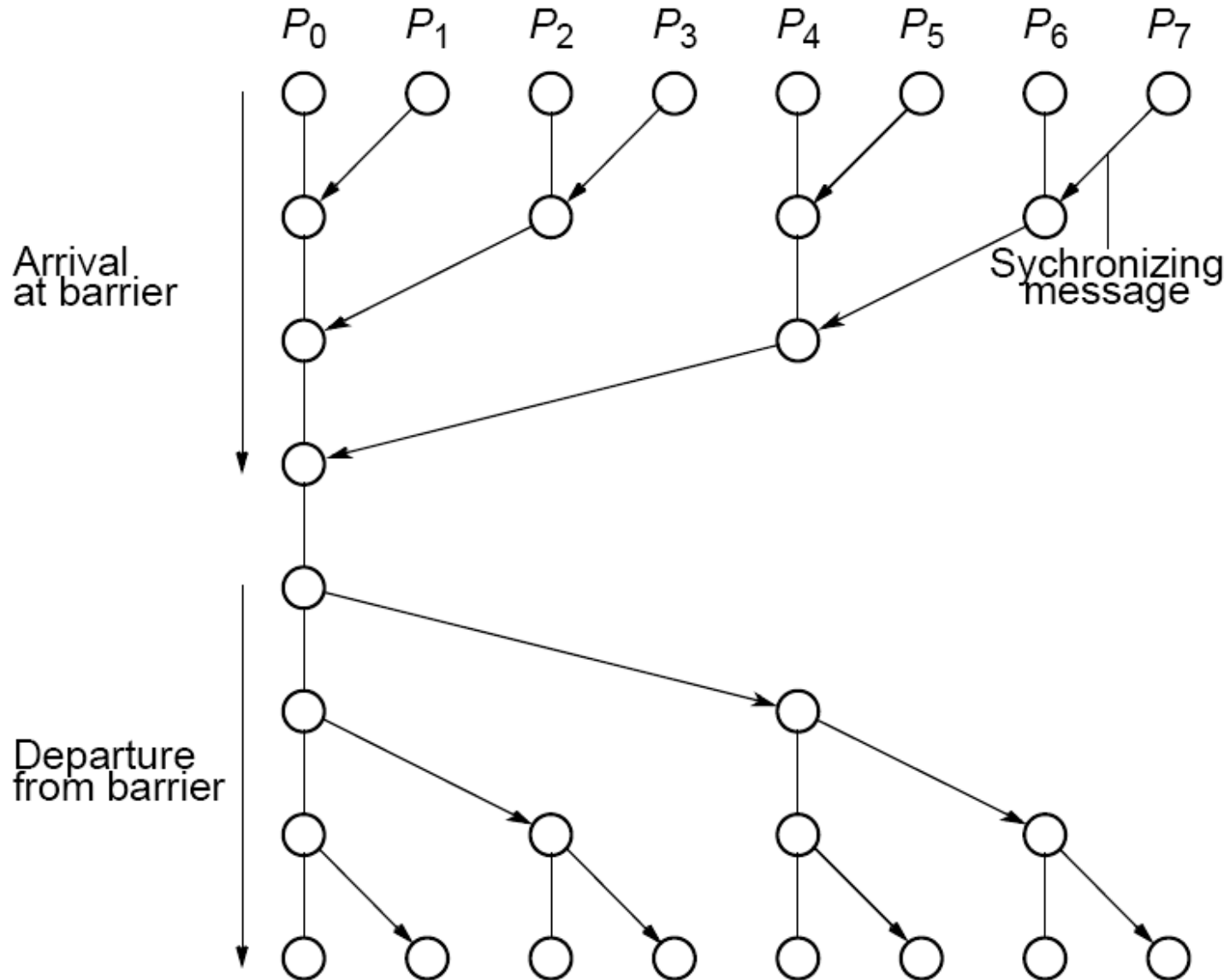Suppose 8 processes, $P0$, $P1$, $P2$, $P3$, $P4$, $P5$, $P6$, $P7$:

1st stage: $P1$ sends message to $P0$; (when $P1$ reaches its barrier)
 $P3$ sends message to $P2$; (when $P3$ reaches its barrier)
 $P5$ sends message to $P4$; (when $P5$ reaches its barrier)
 $P7$ sends message to $P6$; (when $P7$ reaches its barrier)

2nd stage: $P2$ sends message to $P0$; ($P2$ & $P3$ reached their barrier)
 $P6$ sends message to $P4$; ($P6$ & $P7$ reached their barrier)

3rd stage: $P4$ sends message to $P0$; ($P4$, $P5$, $P6$, & $P7$ reached barrier)
 $P0$ terminates arrival phase;
 (when $P0$ reaches barrier & received message from $P4$)

Release with a reverse tree construction.

Tree barrier

P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7

Arrival at barrier

Sychronizing message

Departure from barrier
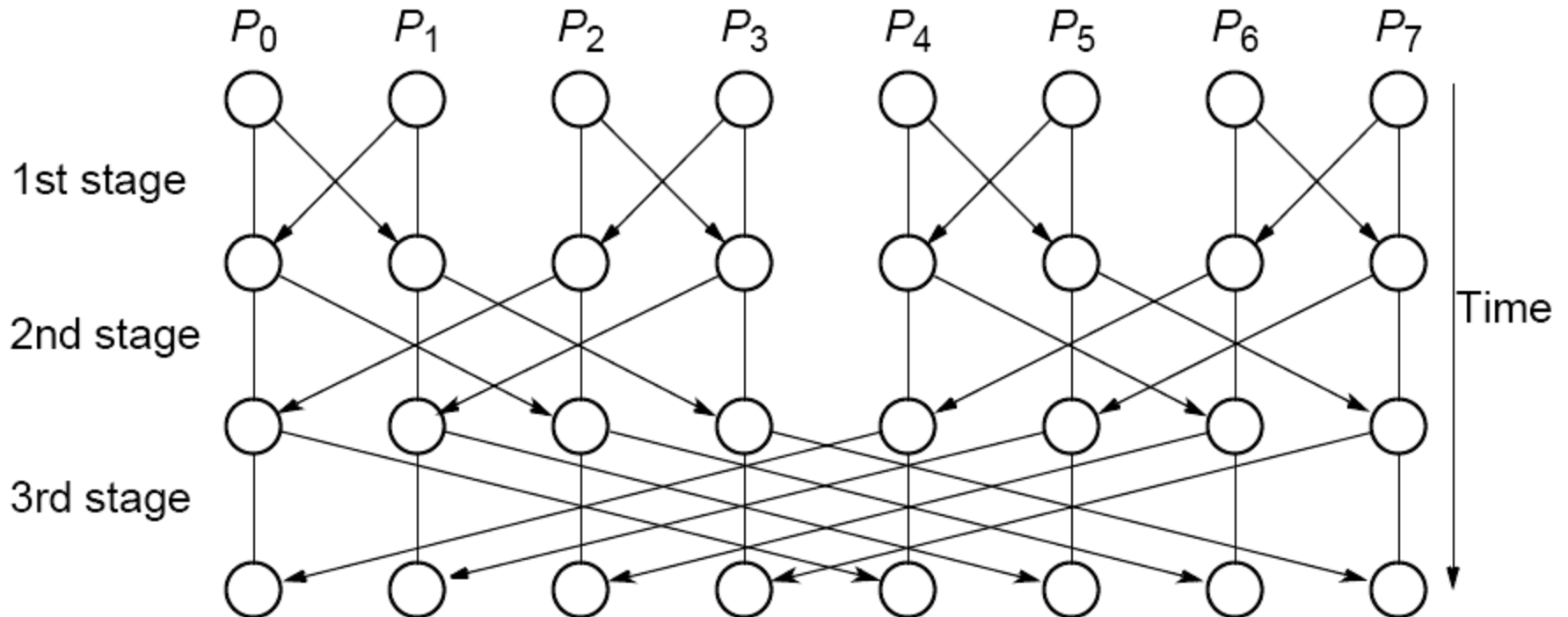
6.12

Butterfly Barrier

1st stage $\quad P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7$

2nd stage $\quad P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7$
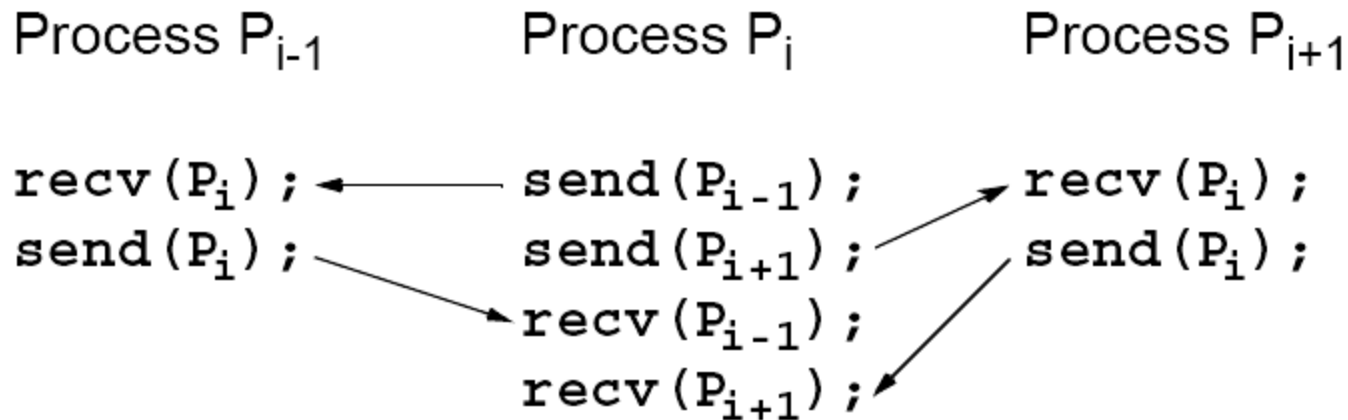
3rd stage $\quad P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7$

Local Synchronization

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

```
Process P_i-1              Process P_i              Process P_i+1

recv(P_i);     ◄────────  send(P_i-1);        ┌─► recv(P_i);
send(P_i);  ─┐            send(P_i+1);  ──┐    │   send(P_i);
             └─►          recv(P_i-1);    └──► │
                          recv(P_i+1);  ◄──────┘
```

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

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 sendrev()s actually has 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.

Example

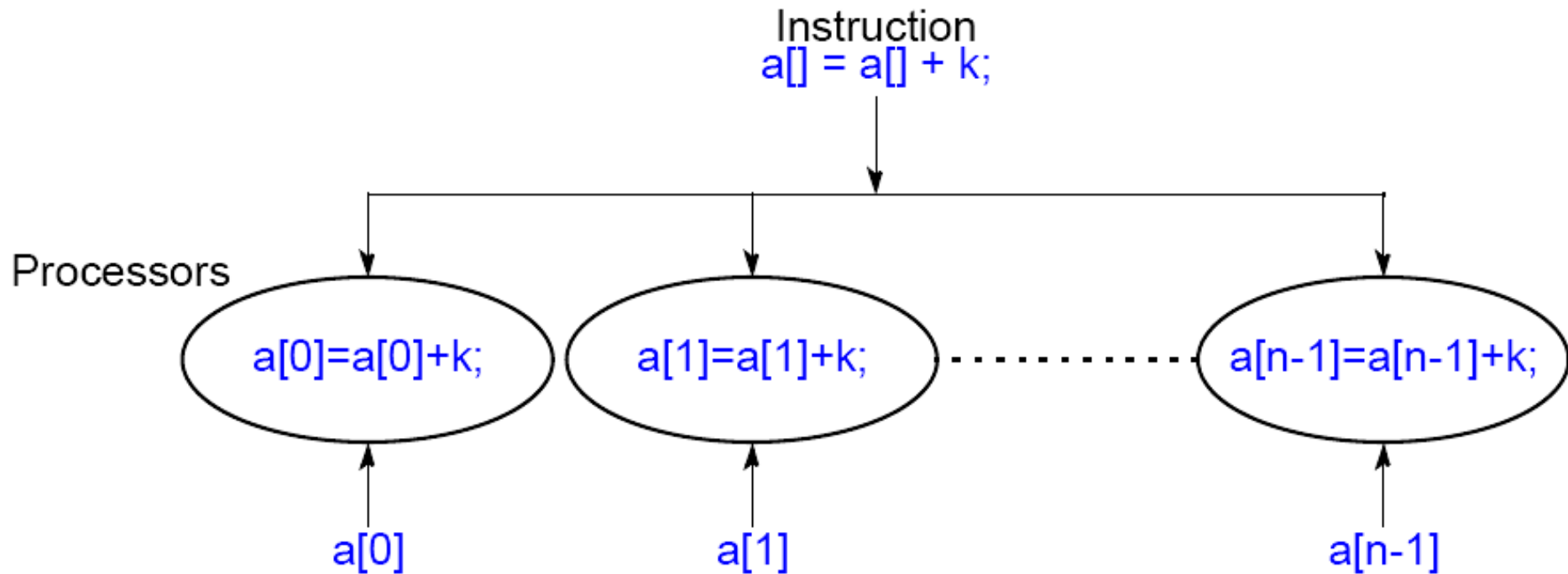To add the same constant to each element of an array:

```
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 $(0 < i <= n)$.

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.

Data Parallel Example
Prefix Sum Problem

Given a list of numbers, x_0, …, x_{n-1}, compute all the partial summations, i.e.:
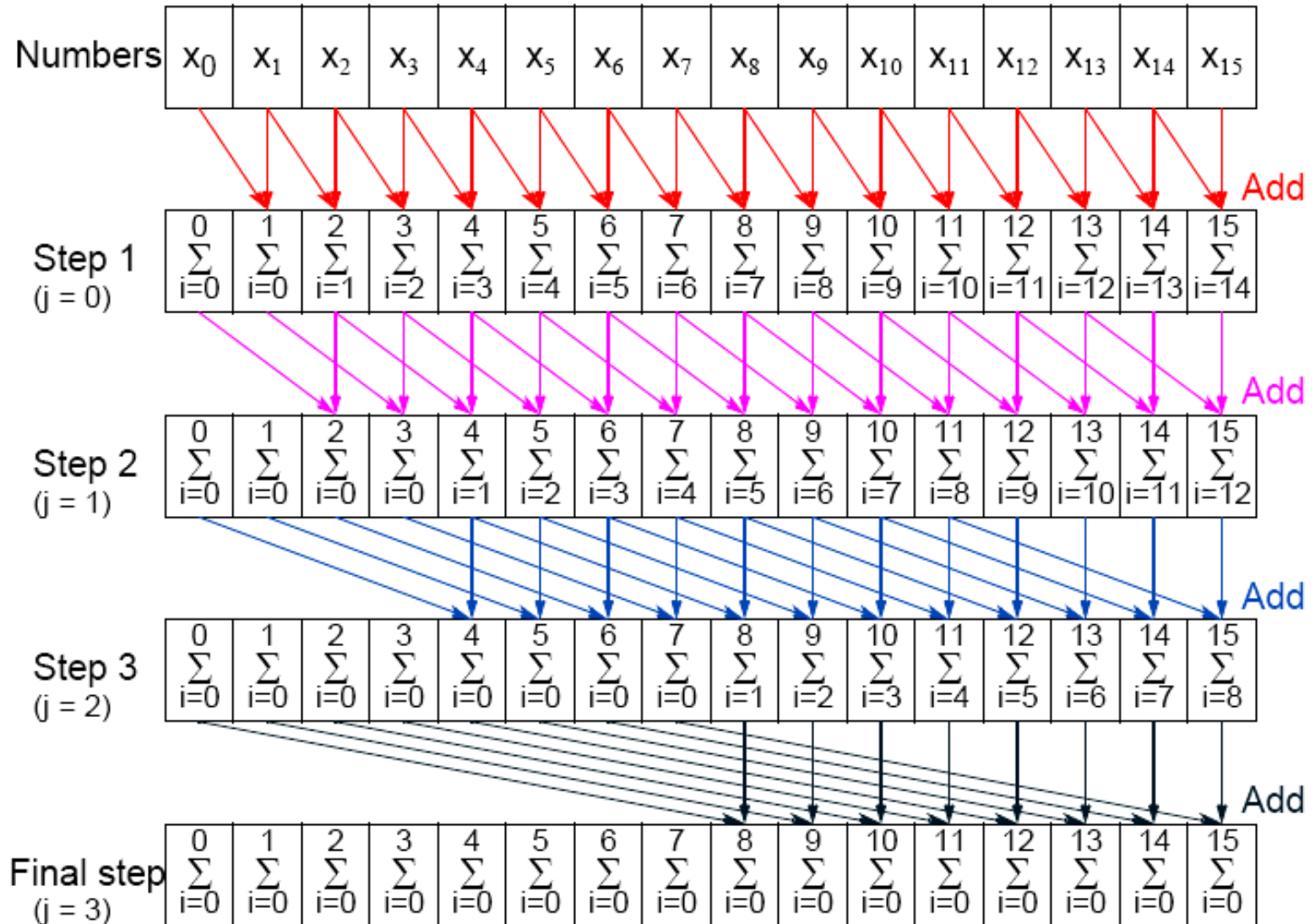
$$x_0 + x_1; x_0 + x_1 + x_2; x_0 + x_1 + x_2 + x_3; …).$$

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 method of adding all partial sums of 16 numbers

Data parallel prefix sum operation

Sequential code

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

Parallel code

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

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

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

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 \ldots \qquad + a_{2,n-1}x_{n-1} = b_2$$
$$a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 \ldots \qquad + a_{1,n-1}x_{n-1} = b_1$$
$$a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 \ldots \qquad + a_{0,n-1}x_{n-1} = b_0$$

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

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. 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 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

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

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
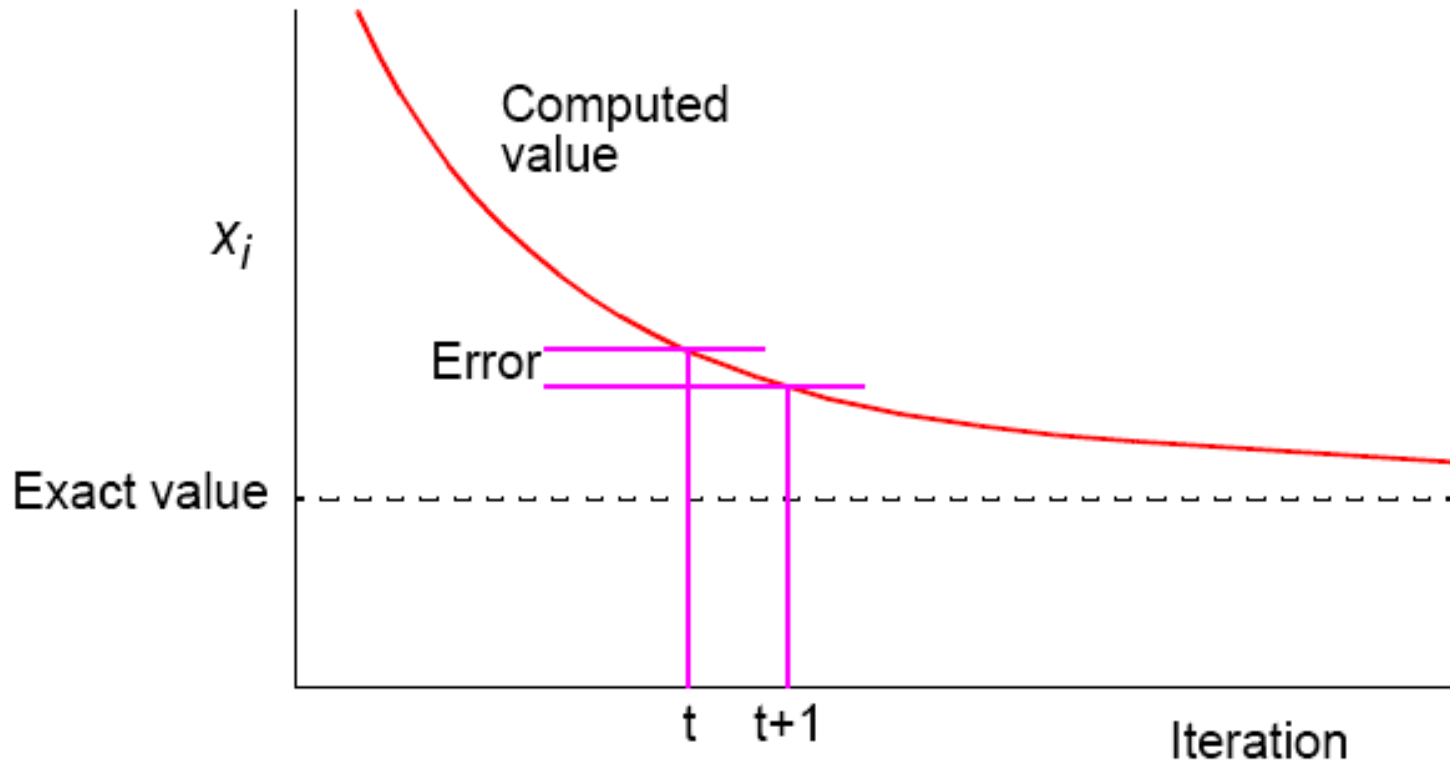
Parallel Code

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.

Introduce a new message-passing operation - Allgather.

Allgather

Broadcast and gather values in one composite construction.
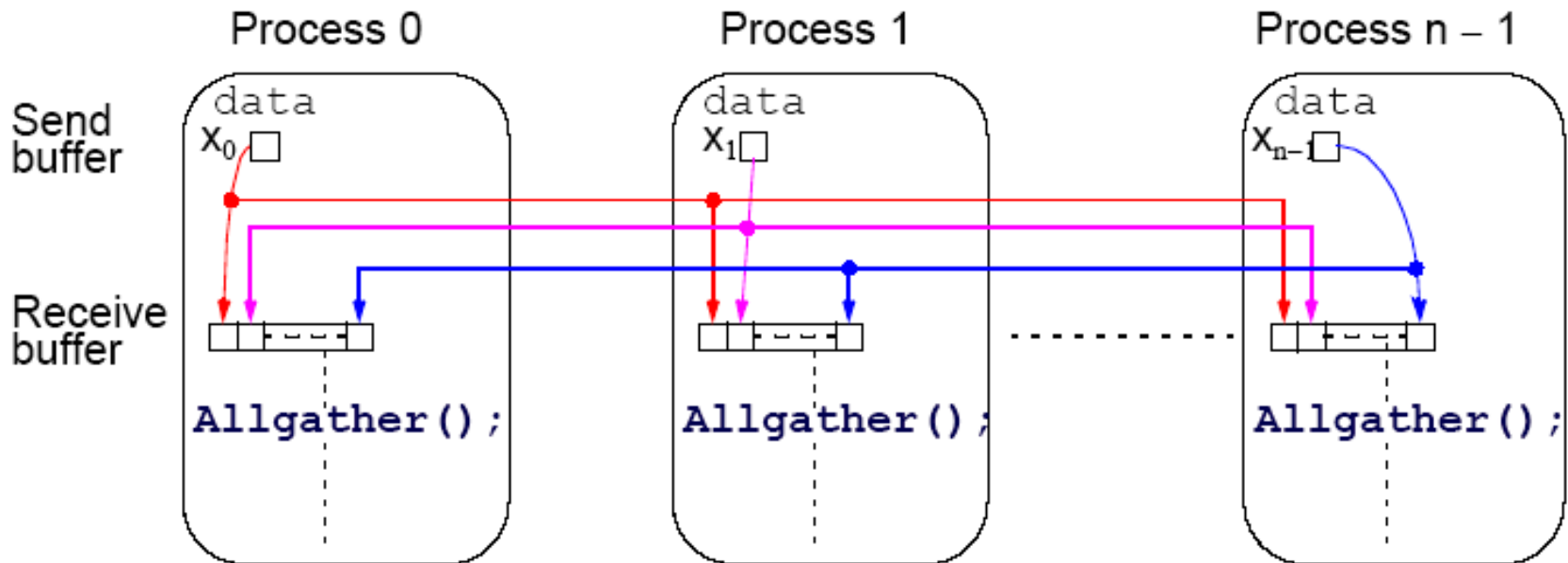
Partitioning

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.

cyclic allocation – processors are allocated one unknown in order; i.e., processor P_0 is allocated x_0, x_p, x_{2p}, …, $x_{((n/p)-1)p}$, processor P_1 is allocated x_1, x_{p+1}, x_{2p+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.
Get:
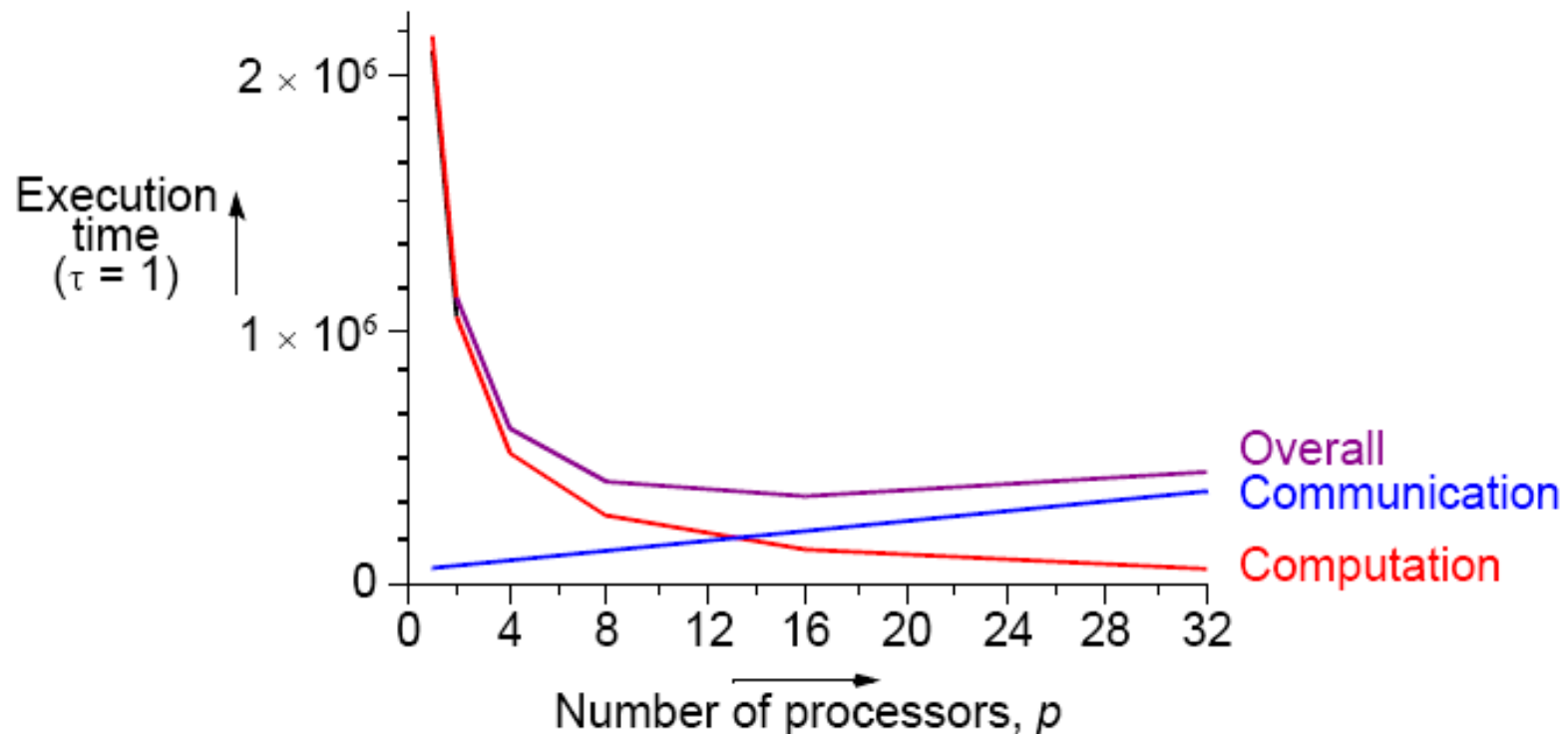
Locally Synchronous Computation
Heat Distribution Problem

An area has known temperatures along each of its edges.

Find the temperature distribution within.

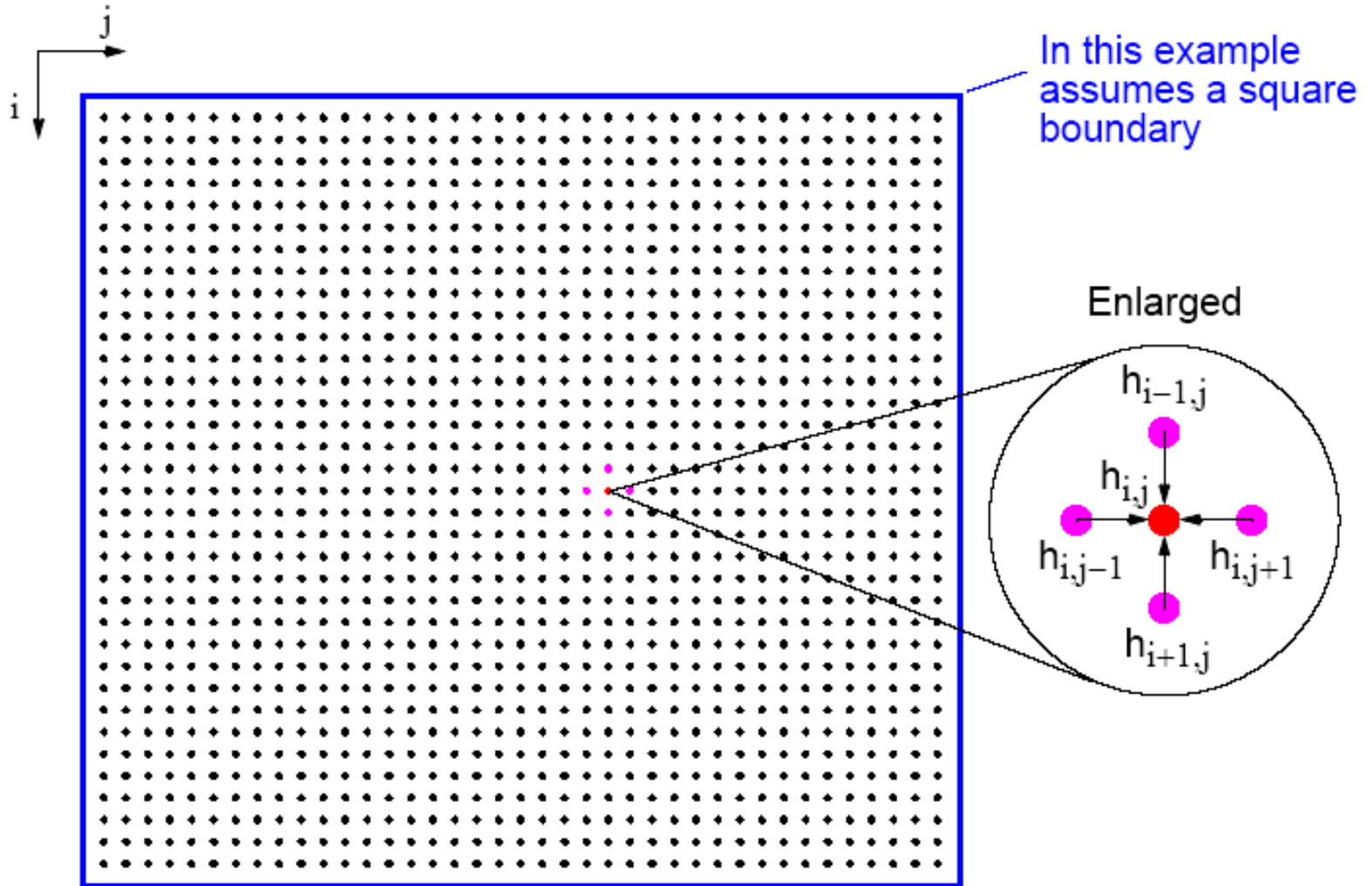Divide area into fine mesh of points, $h_{i,j}$.

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

In this example assumes a square boundary

Enlarged

$$h_{i-1,j}$$

$$h_{i,j}$$

$$h_{i,j-1} \qquad h_{i,j+1}$$

$$h_{i+1,j}$$

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

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

Could be written as a linear equation containing the unknowns x_{i-m}, x_i-1, x_i+1, and x_i+m:

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

Notice: solving a (sparse) system
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];
}
```

using original numbering system (*n* x *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]);

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

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

Parallel Code

With fixed number of iterations, $P_{i,j}$
(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});
}
```
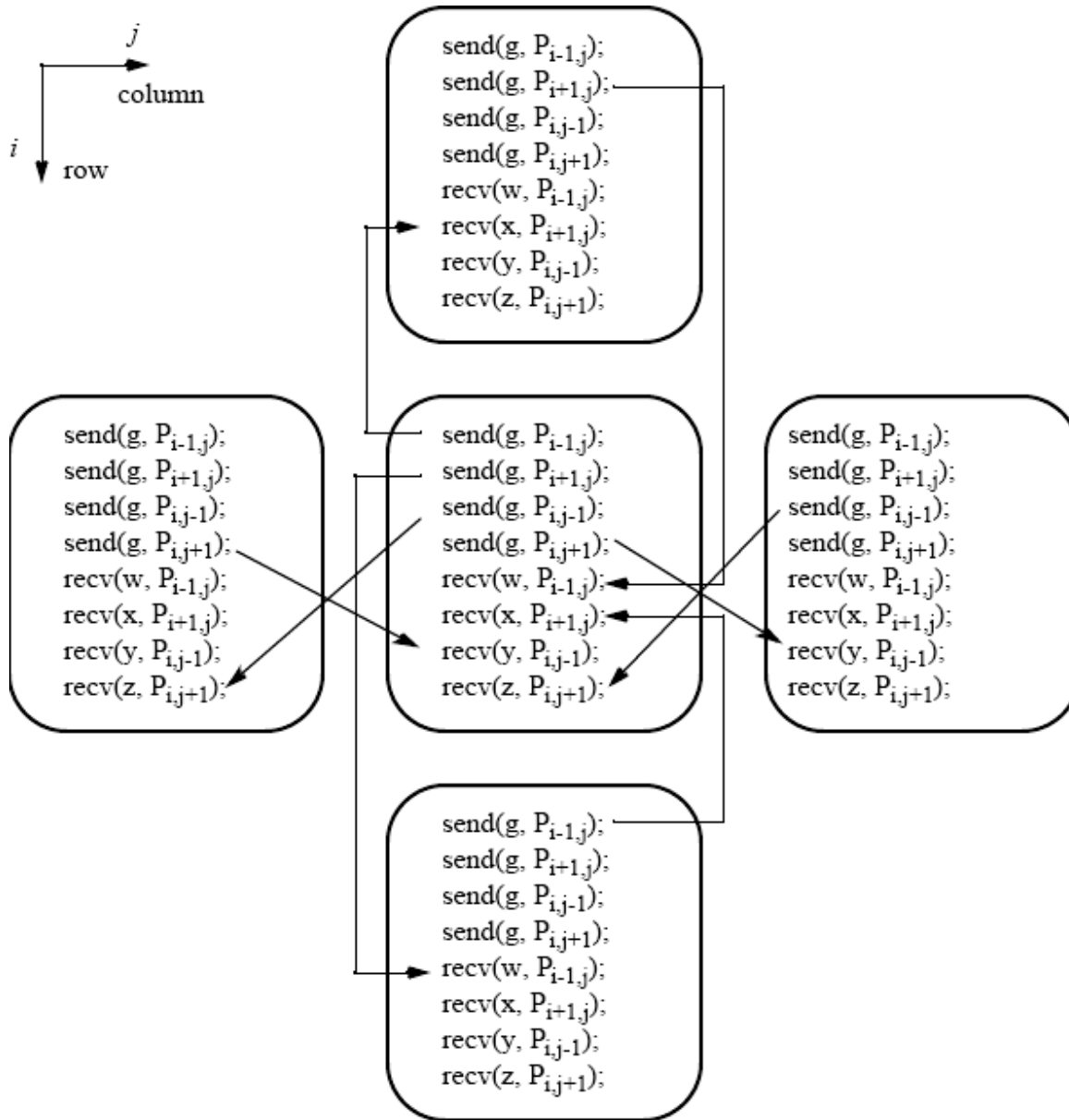
Local barrier

Important to use **send()**s that do not block while waiting for **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, P_{i-1,j});          /* locally 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});          /* locally blocking receives */
    recv(&x, P_{i+1,j});
    recv(&y, P_{i,j-1});
    recv(&z, P_{i,j+1});
} while((!converged(i, j)) || (iteration < limit));
send(&g, &i, &j, &iteration, P_master);
```

To handle the processes operating at the edges:

MPI has a construct to help here

```
if (last_row) w = bottom_value;
if (first_row) x = 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, P_{i-1,j});
  if !(last_row) send(&g, P_{i+1,j});
  if !(first_column) send(&g, P_{i,j-1});
  if !(last_column) send(&g, P_{i,j+1});
  if !(last_row) recv(&w, P_{i-1,j});
  if !(first_row) recv(&x, P_{i+1,j});
  if !(first_column) recv(&y, P_{i,j-1});
  if !(last_column) recv(&z, P_{i,j+1});
} while((!converged) || (iteration < limit));
send(&g, &i, &j, iteration, P_{master});
```

Example

A room has four walls and a fireplace. Temperature of wall is 20°C, and temperature of fireplace is 100°C. Write a parallel program using Jacobi iteration to compute the temperature inside the room and plot (preferably in color) temperature contours at 10°C intervals using Xlib calls or similar graphics calls as available on your system.
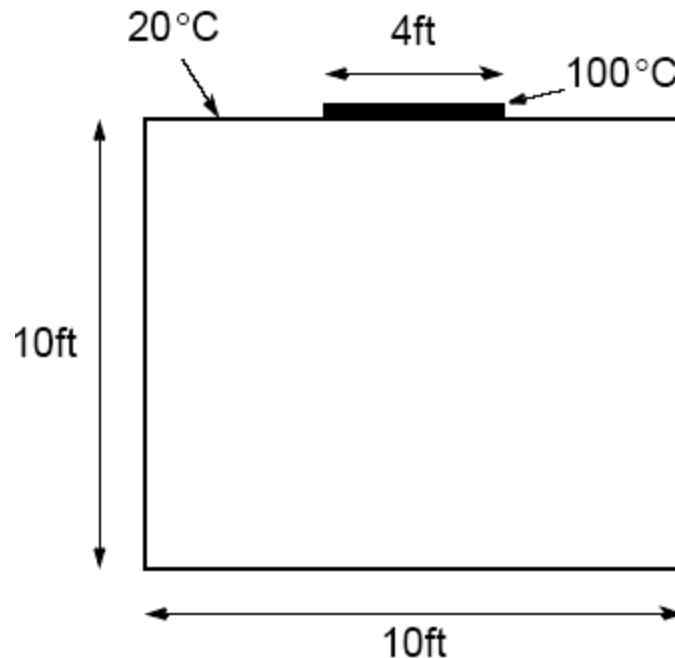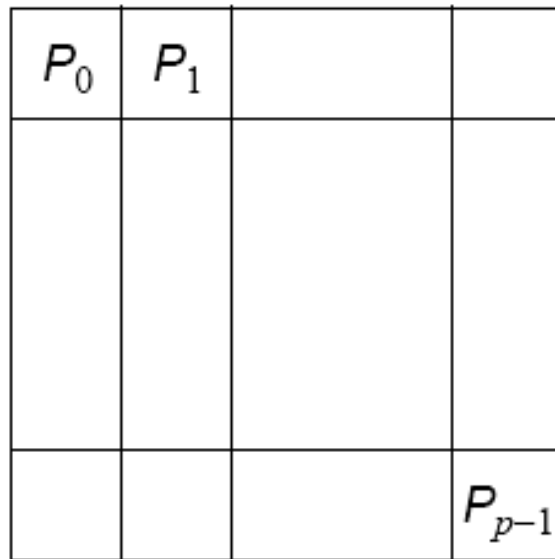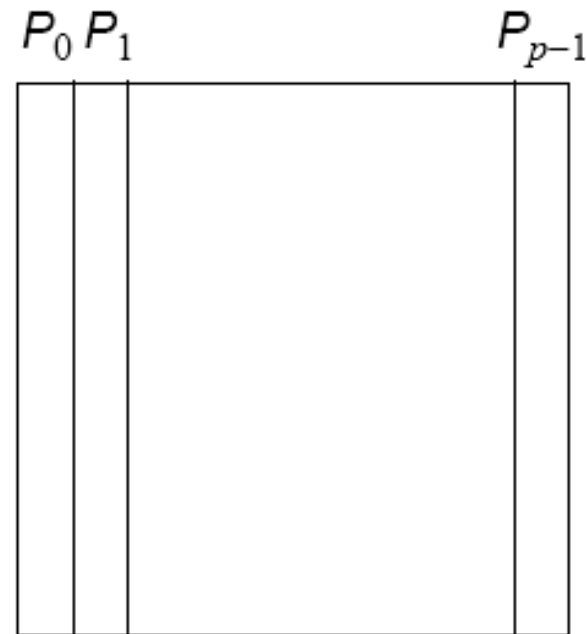
Sample student output

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:
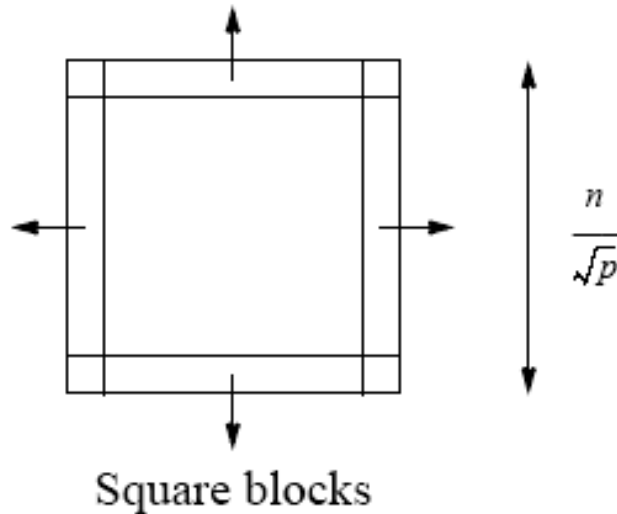

Blocks   Strips (columns)

Block partition

Four edges where data points exchanged.
Communication time given by

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

$\dfrac{n}{\sqrt{p}}$

Square blocks

Strip partition

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

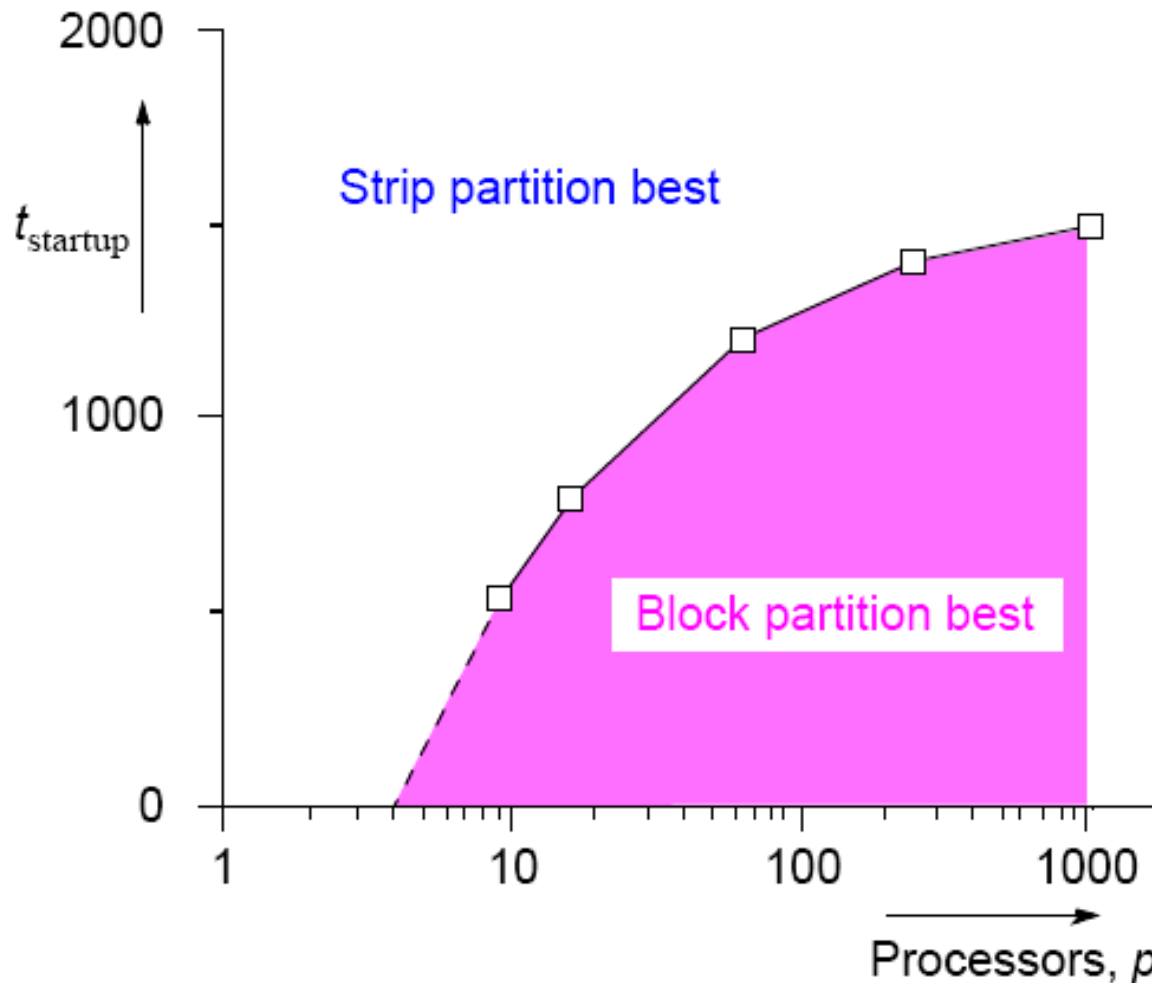$$t_{commcol} = 4(t_{startup} + nt_{data})$$
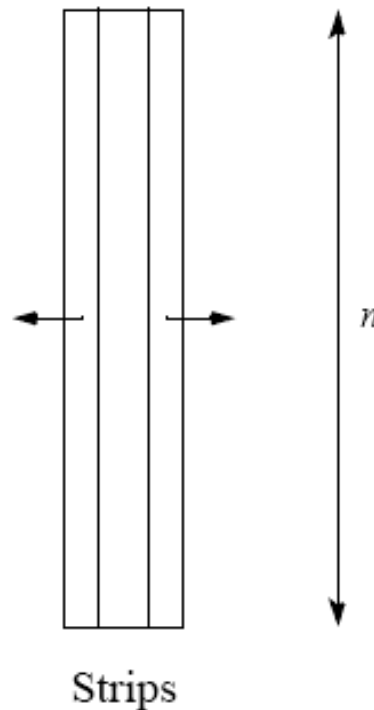
Strips

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

$$t_{\text{startup}} > n \left(1 - \frac{2}{\sqrt{p}} \right) t_{\text{data}}$$
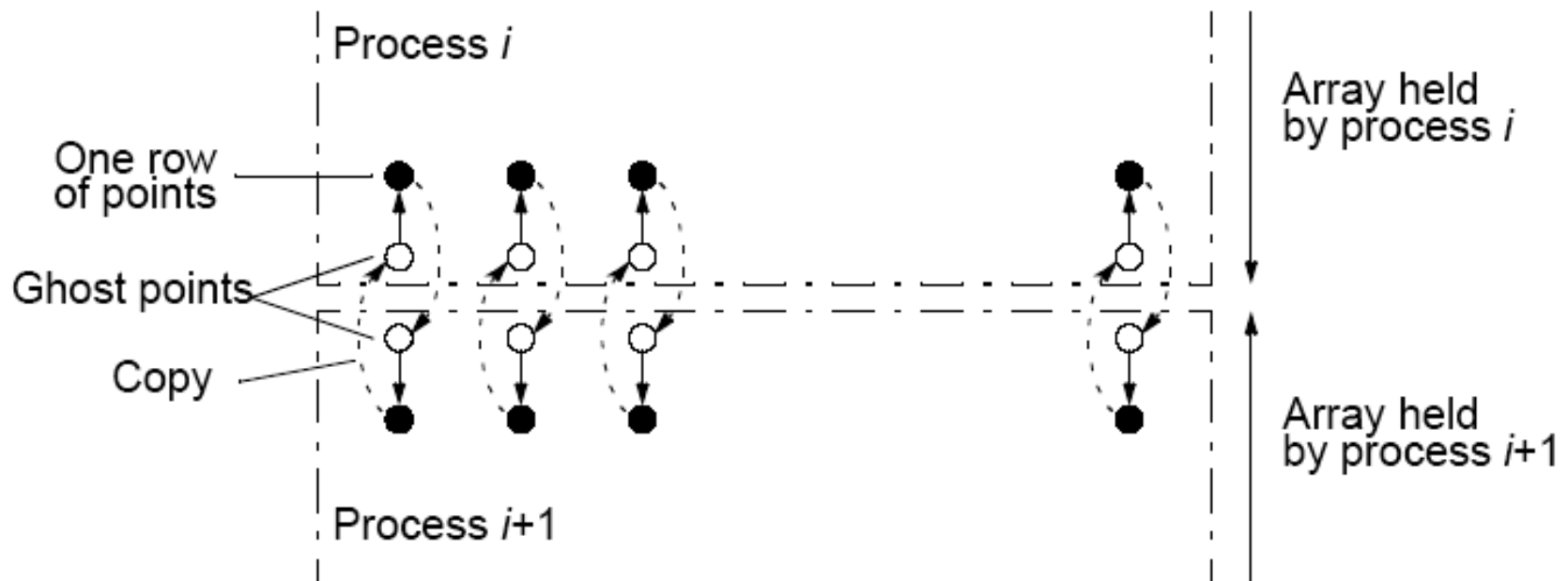
$(p \geq 9)$.

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()`.

Other fully synchronous problems
Cellular Automata

The problem space is divided into cells.

Each cell can be in one of a finite number of states.

Cells affected by their neighbors according to certain rules, and all cells are affected simultaneously in a "generation."

Rules re-applied in subsequent generations so that cells evolve, or change state, from generation to generation.

Most famous cellular automata is the "Game of Life" devised by John Horton Conway, a Cambridge mathematician.

The Game of Life

Board game - theoretically infinite two-dimensional array of cells. Each cell can hold one "organism" and has eight neighboring cells, including those diagonally adjacent. Initially, some cells occupied. The following rules apply:

1. Every organism with two or three neighboring organisms survives for the next generation.
2. Every organism with four or more neighbors dies from overpopulation.
3. Every organism with one neighbor or none dies from isolation.
4. Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

These rules were derived by Conway "after a long period of experimentation."

Simple Fun Examples of Cellular Automata
"Sharks and Fishes"

An ocean could be modeled as a three-dimensional array of cells.

Each cell can hold one fish or one shark (but not both).

Fish and sharks follow "rules."

Fish

Might move around according to these rules:

1. If there is one empty adjacent cell, the fish moves to this cell.

2. If there is more than one empty adjacent cell, the fish moves to one cell chosen at random.

3. If there are no empty adjacent cells, the fish stays where it is.

4. If the fish moves and has reached its breeding age, it gives birth to a baby fish, which is left in the vacating cell.

5. Fish die after x generations.

Sharks

Might be governed by the following rules:

1. If one adjacent cell is occupied by a fish, the shark moves to
   this cell and eats the fish.
2. If more than one adjacent cell is occupied by a fish, the shark chooses one fish at random, moves to the cell occupied by the fish, and eats the fish.
3. If no fish are in adjacent cells, the shark chooses an unoccupied adjacent cell to move to in a similar manner as fish move.
4. If the shark moves and has reached its breeding age, it gives birth to a baby shark, which is left in the vacating cell.
5. If a shark has not eaten for y generations, it dies.

Sample Student Output
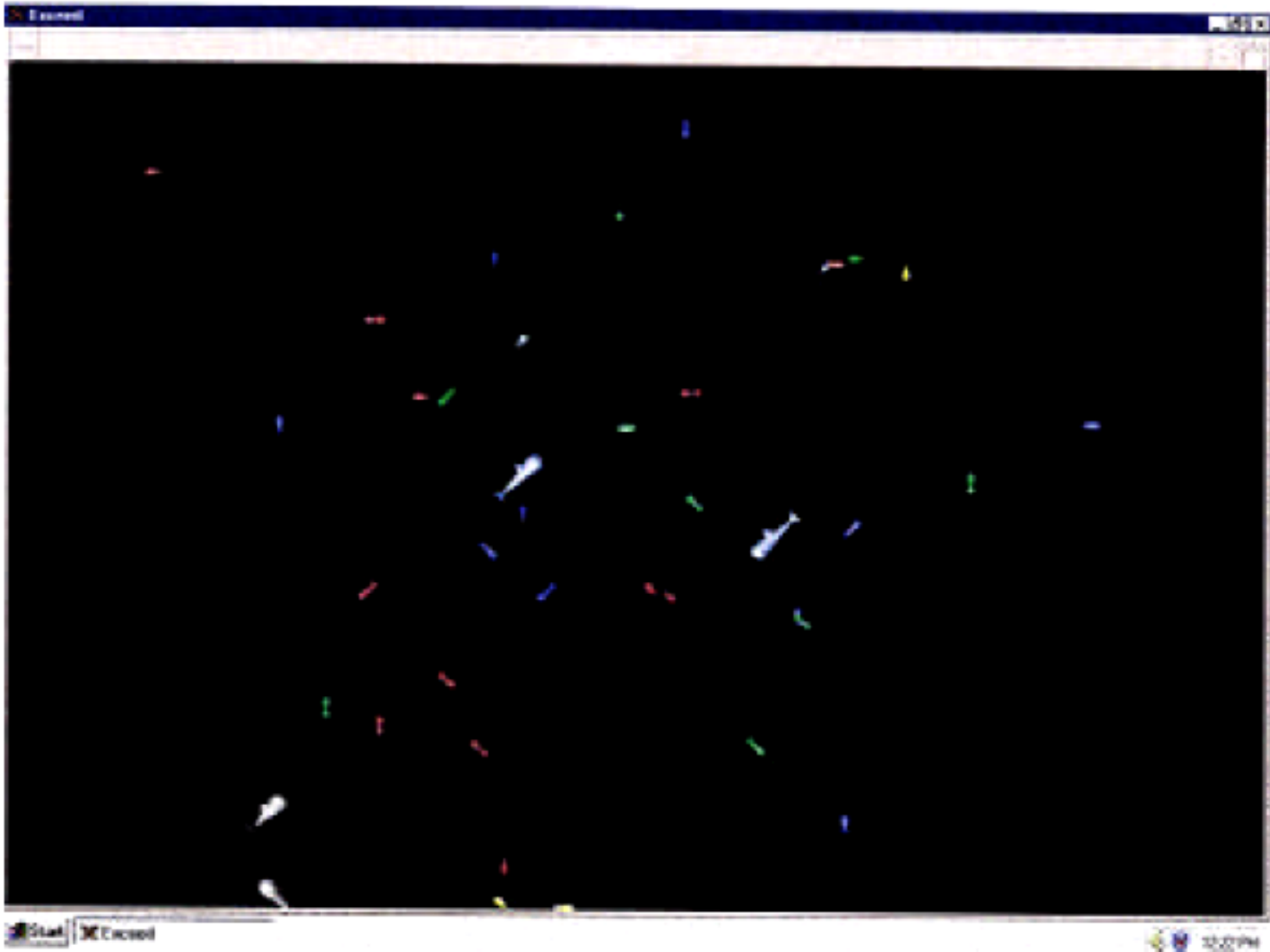
Similar examples:

"foxes and rabbits" - Behavior of rabbits to move around happily whereas behavior of foxes is to eat any rabbits they come across.

Serious Applications for Cellular Automata

Examples

• fluid/gas dynamics

• the movement of fluids and gases around objects

• diffusion of gases

• biological growth

• airflow across an airplane wing

• erosion/movement of sand at a beach or riverbank.

Partially Synchronous Computations

Computations in which individual processes operate without needing to synchronize with other processes on every iteration.

Important idea because synchronizing processes is an expensive operation which very significantly slows the computation and a major cause for reduced performance of parallel programs is due to the use of synchronization.

Global synchronization done with barrier routines. Barriers cause processor to wait sometimes needlessly.

Heat Distribution Problem Re-visited

To solve heat distribution problem, solution space divided into a two dimensional array of points. The value of each point computed by taking average of four points around it repeatedly until values converge on the solution to a sufficient accuracy.

The waiting can be reduced by not forcing synchronization at each iteration.

Sequential code

```
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]);

  for (i = 1; i < n; i++)        /* find max divergence/update pts */
    for (j = 1; j < n; j++) {
      dif = h[i][j] - g[i][j];
      if (dif < 0) dif = -dif;
      if (dif < max_dif) max_dif = dif;
      h[i][j] = g[i][j];
    }
} while (max_dif > tolerance);    /* test convergence */
```

First section of code computing the next iteration values based on the immediate previous iteration values is traditional Jacobi iteration method.

Suppose however, processes are to continue with the next iteration before other processes have completed.

Then, the processes moving forward would use values computed from not only the previous iteration but maybe from earlier iterations.

Method then becomes an *asynchronous iterative method*.

Asynchronous Iterative Method – Convergence

Mathematical conditions for convergence may be more strict.

Each process may not be allowed to use any previous iteration values if the method is to converge.

Chaotic Relaxation

A form of asynchronous iterative method introduced by Chazan and Miranker (1969) in which the conditions are stated as:

"there must be a fixed positive integer s such that, in carrying out the evaluation of the ith iterate, a process cannot make use of any value of the components of the jth iterate if $j < i - s$" (Baudet, 1978).

The final part of the code, checking for convergence of every iteration can also be reduced. It may be better to allow iterations to continue for several iterations before checking for convergence.

Overall Parallel Code

Each process allowed to perform s iterations before being synchronized and also to update the array as it goes. At s iterations, maximum divergence recorded. Convergence is checked then.

The actual iteration corresponding to the elements of the array being used at any time may be from an earlier iteration but only up to s iterations previously. May be a mixture of values of different iterations as array is updated without synchronizing with other processes - truly a chaotic situation.