Synchronous Computations

Thoai Nam

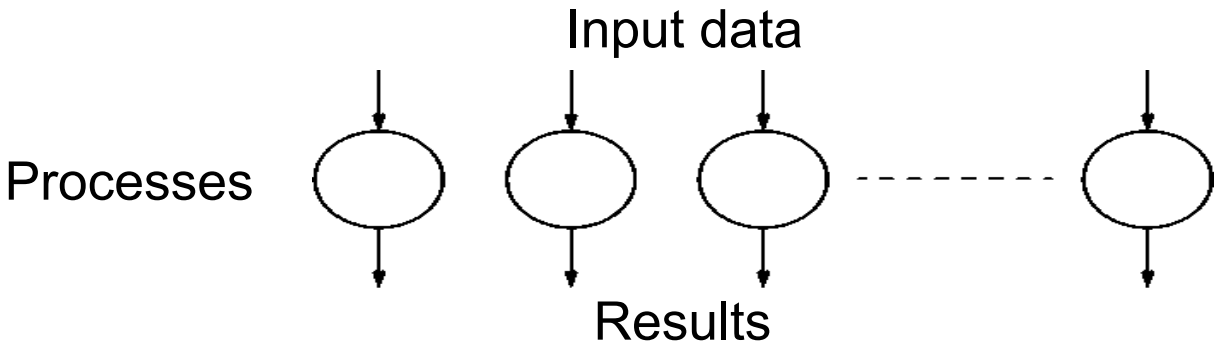
High Performance Computing Lab (HPC Lab)

Faculty of Computer Science and Technology

HCMC University of Technology

EPC

- A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or)



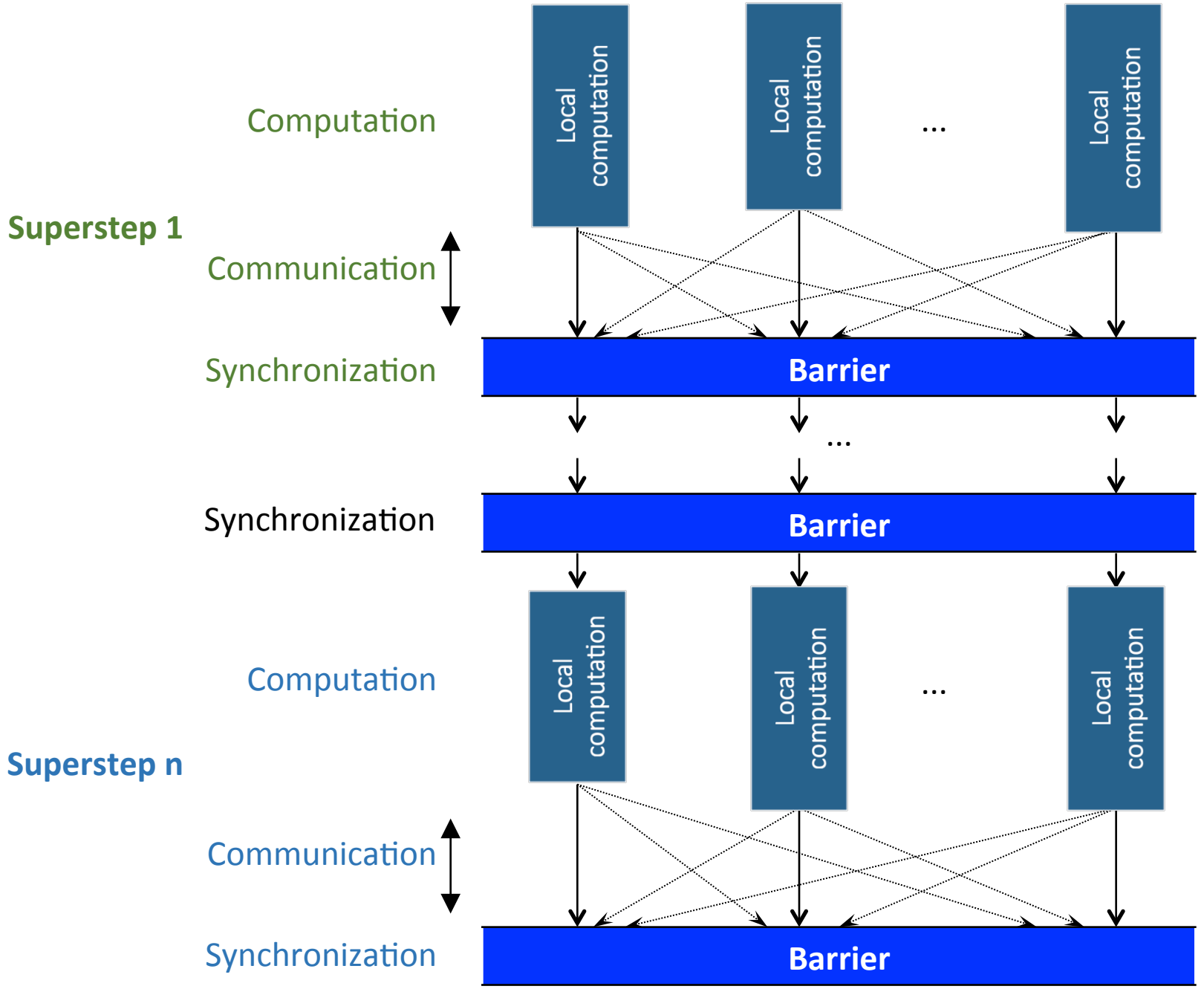
Communication???

Bulk Synchronous Parallel (BSP)

No communication or very little communication between processes;
Each process can do its tasks without any interaction with other processes.

- Data parallel computation

BSP


Computation
Local computation
Local computation
...
Local computation
Superstep 1
Communication
Synchronization
Barrier
...
Synchronization
Barrier
Computation
Local computation
Local computation
...
Local computation
Superstep n
Communication
Synchronization
Barrier

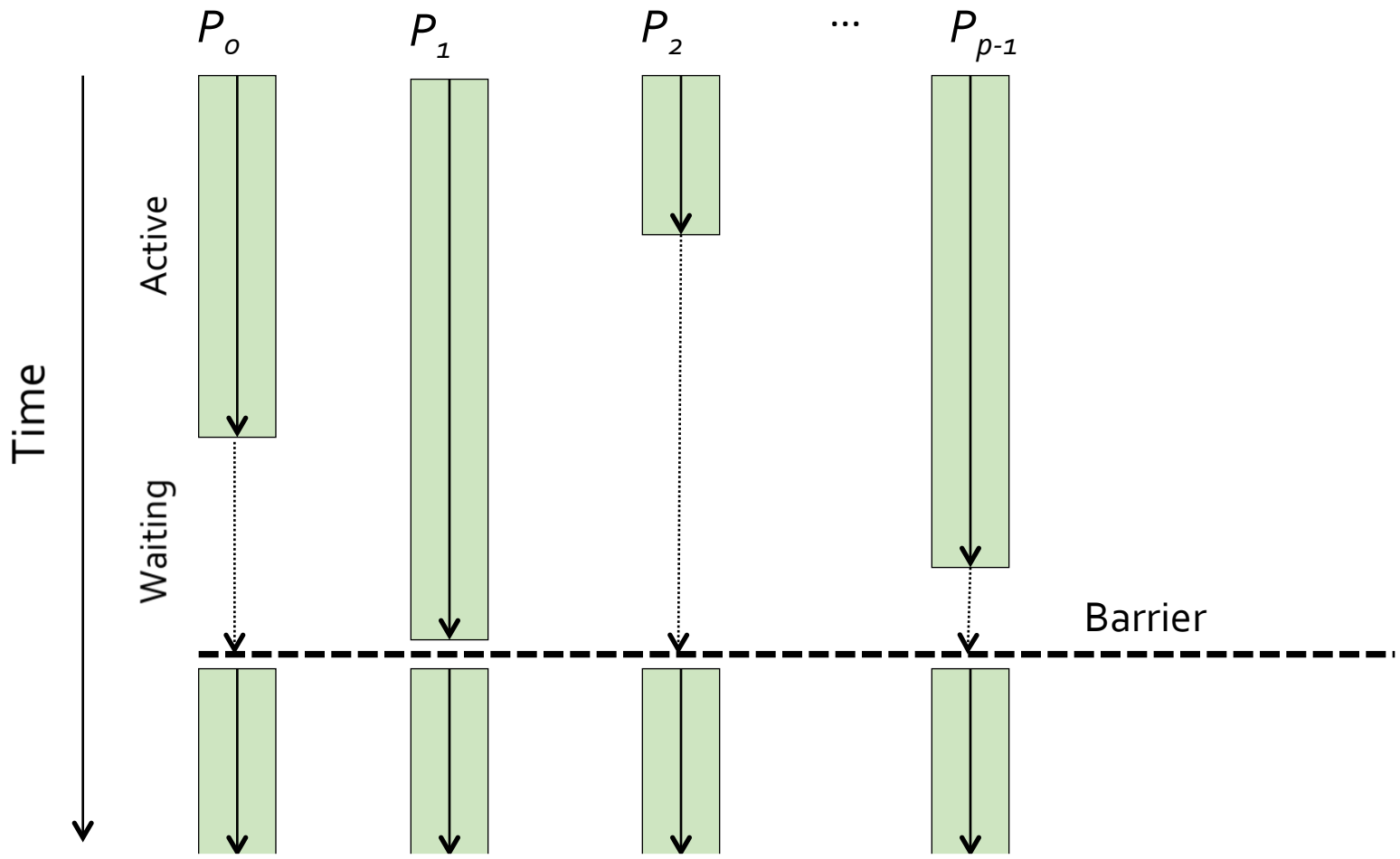
Synchronous computation

In a (fully) synchronous application, all the processes synchronized at regular points.

Barrier

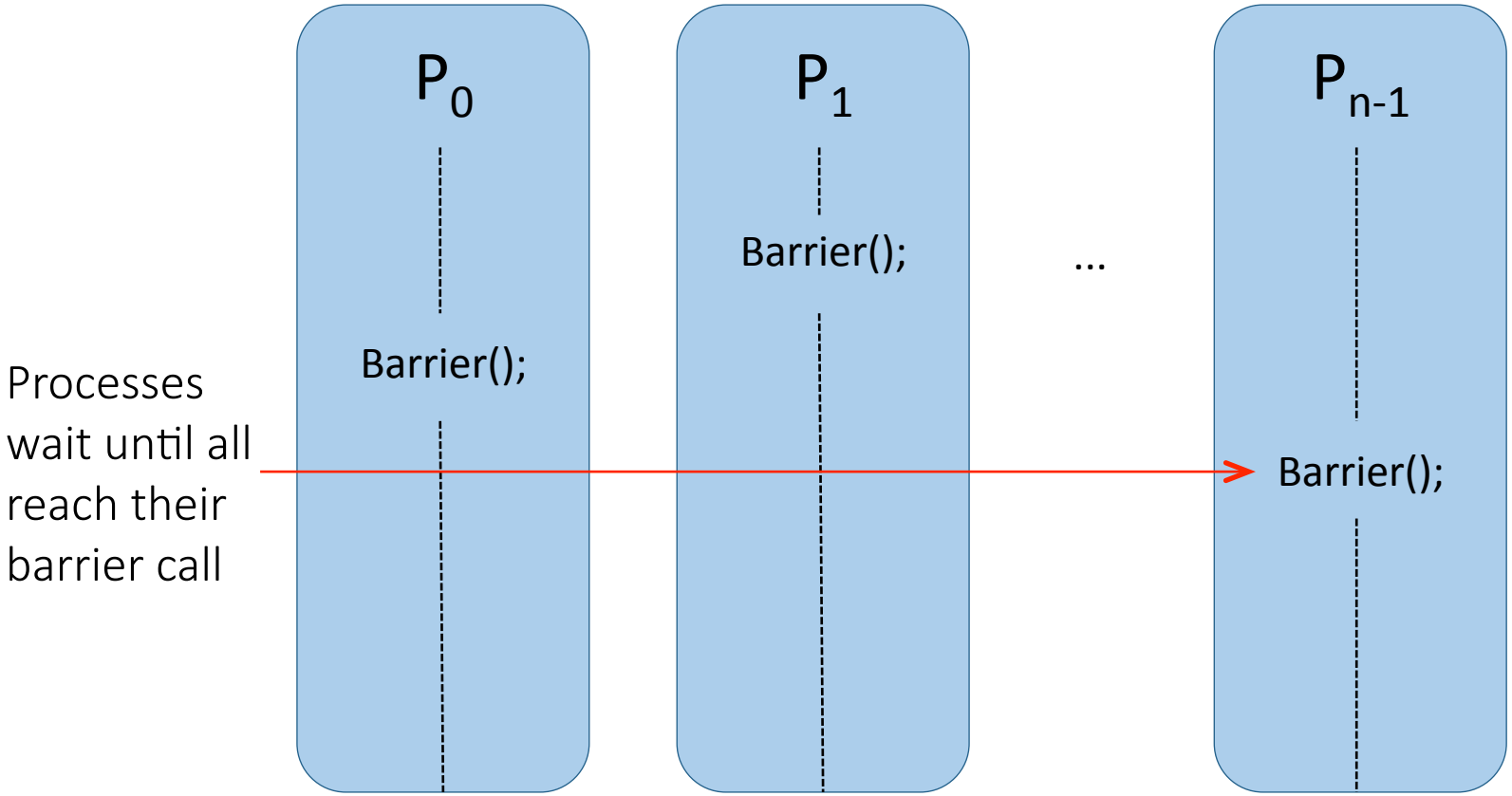
- A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait;
- All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).

Processes reaching barrier at different times



Message-passing

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



MPI

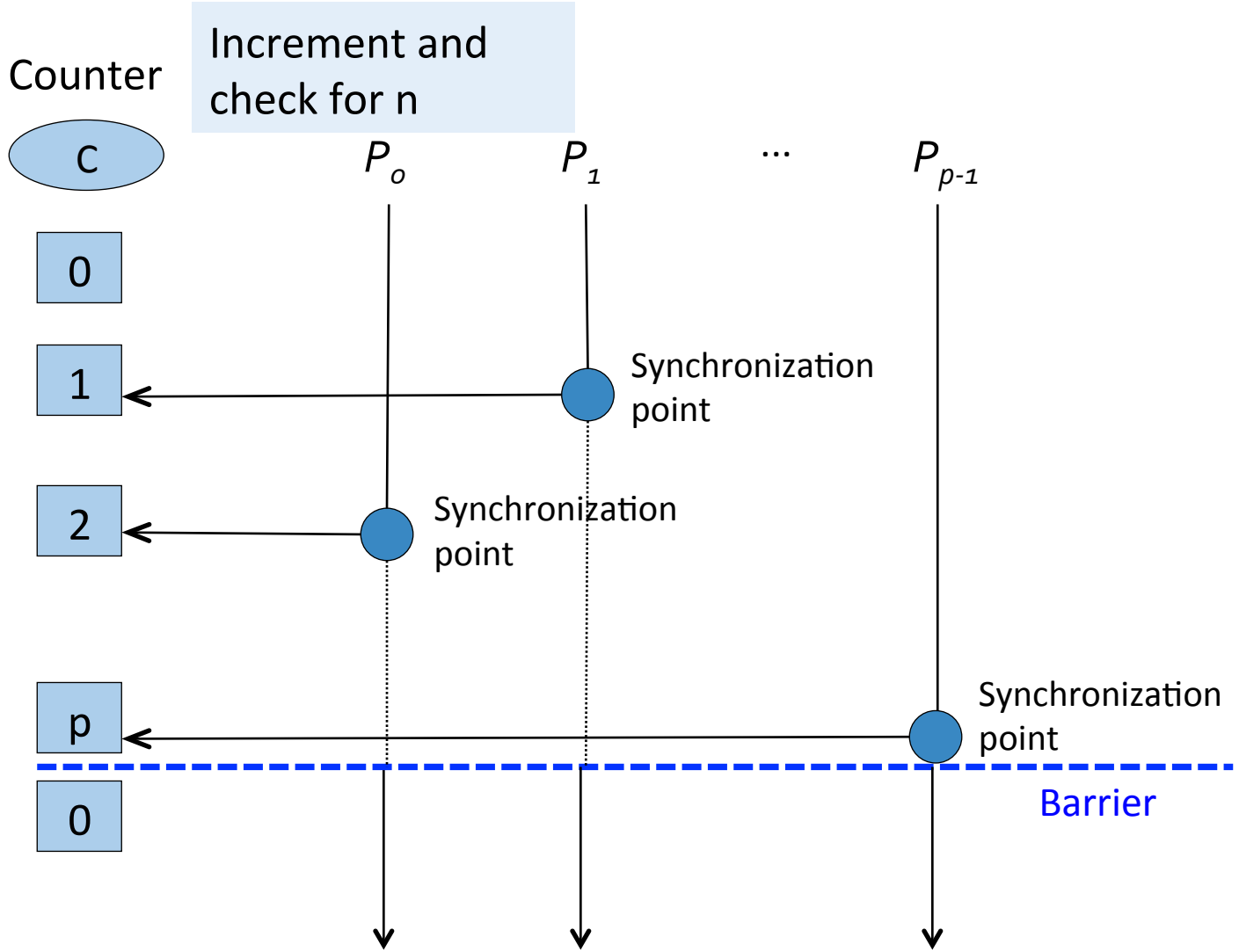
- MPI_Barrier()
- Barrier with a named communicator being the only parameter
- Called by each process in the group, blocking until all members of the group have reached the barrier call and only returning then.

Barrier Implementation

Centralized counter implementation (linear barrier)

- Counter-based barriers often have two phases:
  - A process enters *arrival/trapping phase* and does not leave this phase until all processes have arrived in this phase
  - Then processes move to *departure/ release 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

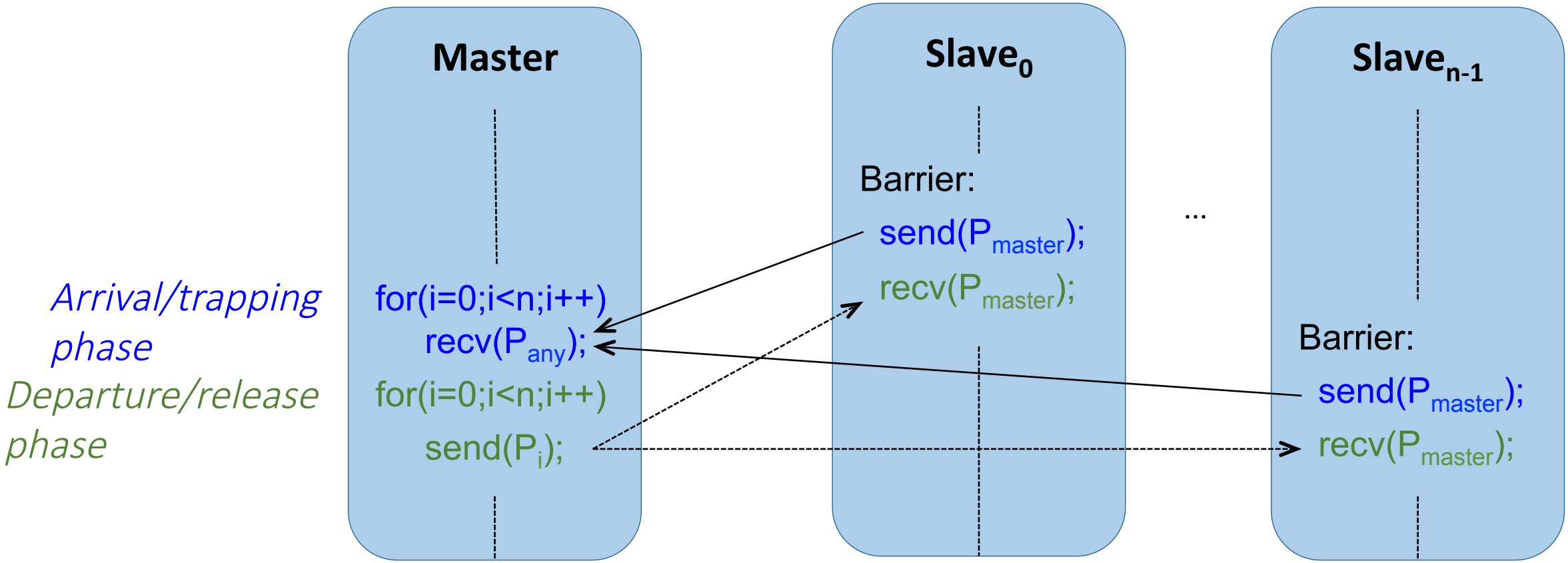
Master

```
     // Arrival/trapping phase
     // Pha nhập/bẫy: đợi đủ p Slave
1.   for (i=0; i<p; i++)
2.      Recv(P_any_slave);
     // Departure/release phase
     // Pha xuất/giải phóng: giải phóng p Slave
3.   for (i=0; i<p; i++)
4.      Send(P_i);
```

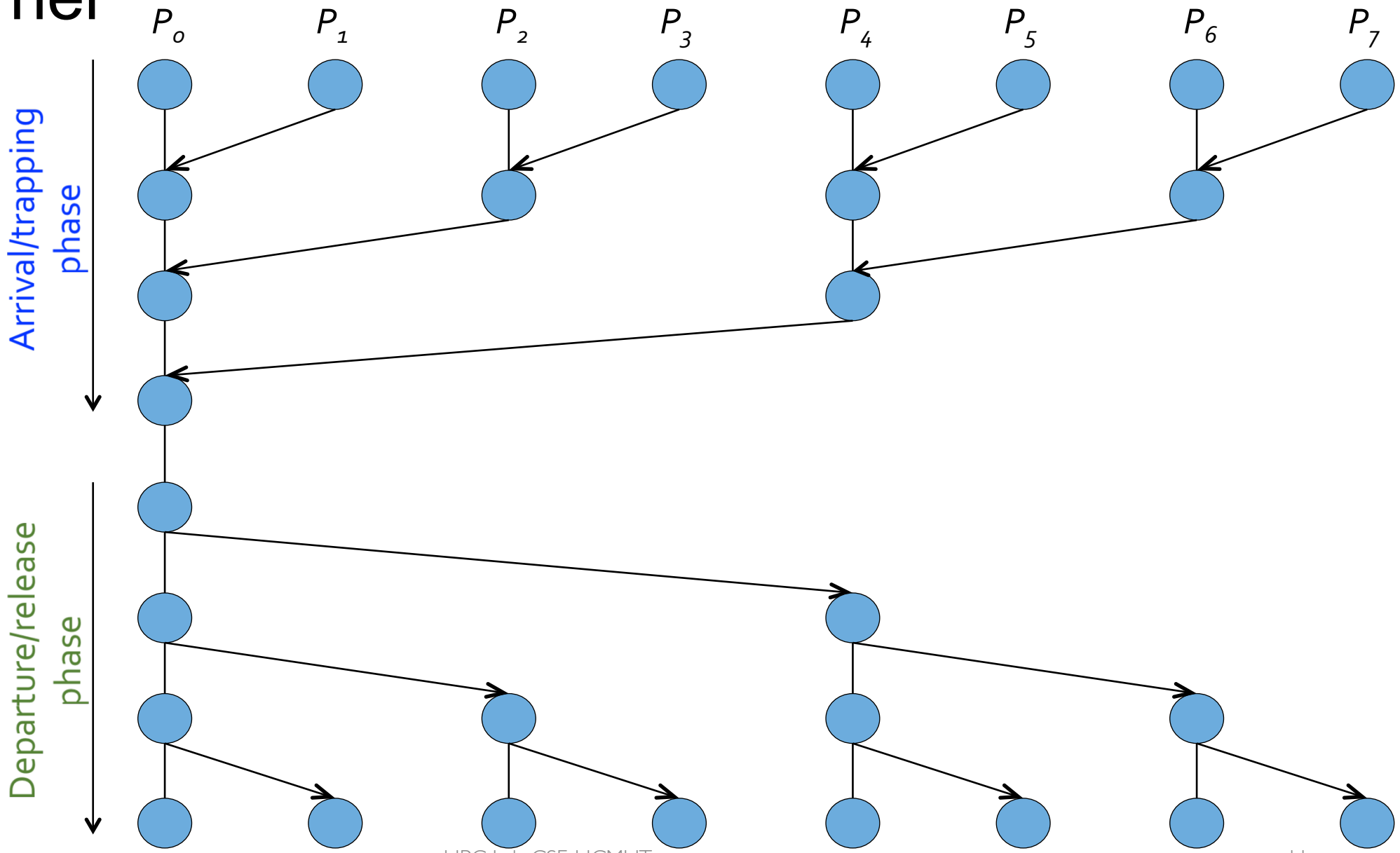
Slave

```
5.   Send(P_master);   // Slave thông báo đã đạt đến rào đồng bộ
6.   Recv(P_master);   // Đợi thông báo từ Master để sang pha xuất
```

Barrier implementation in a message-passing system

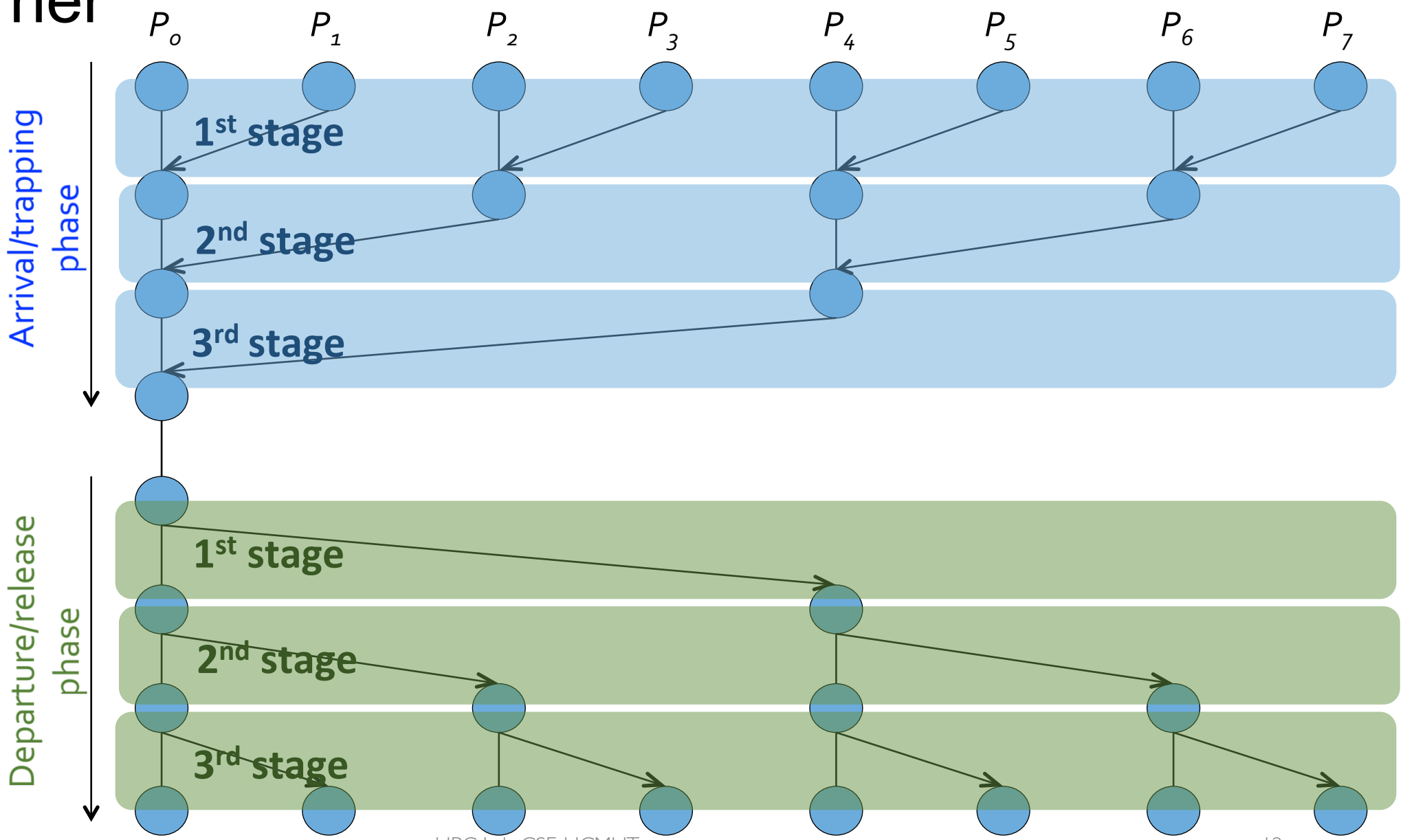


Tree barrier



Tree barrier

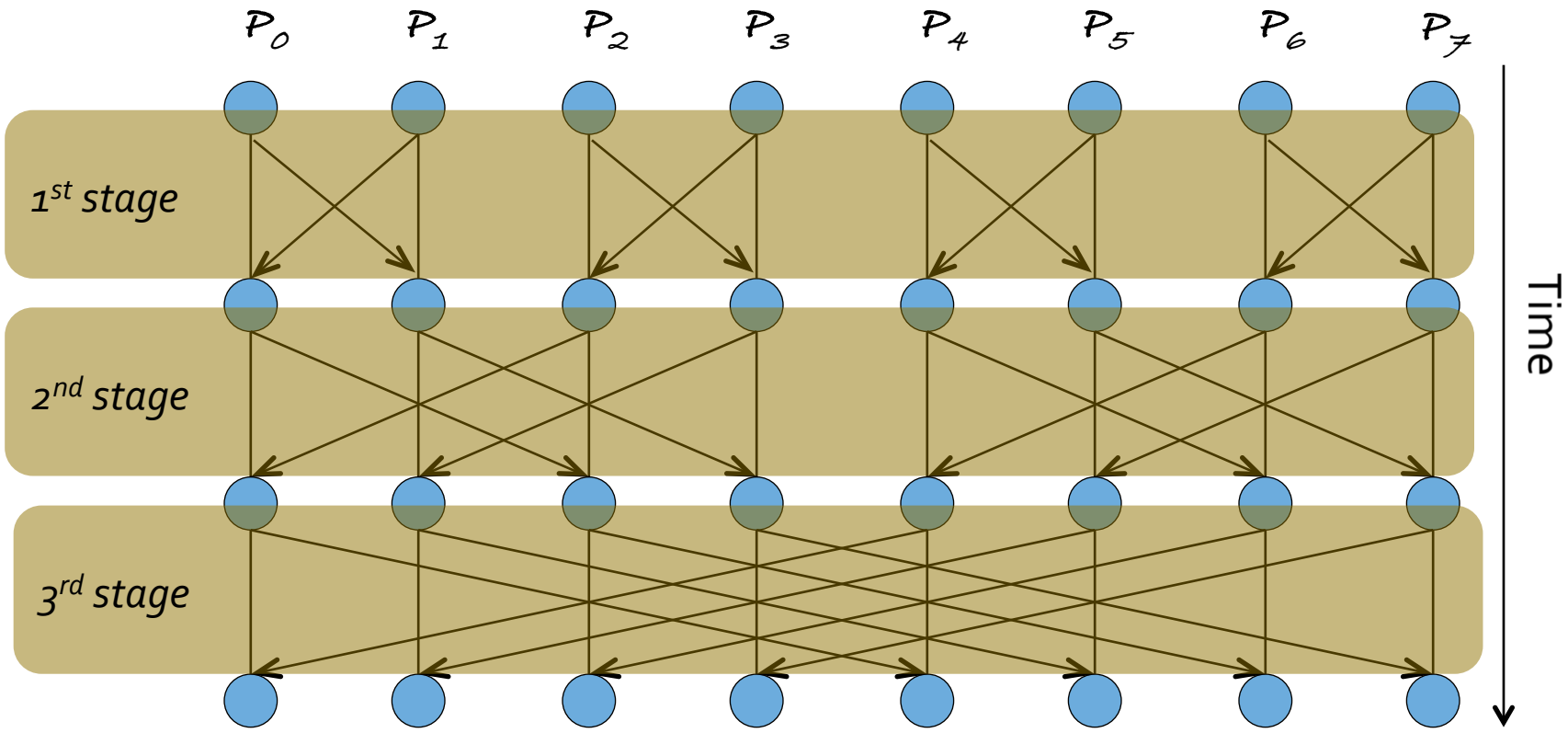
- n processes
- $2.\log n$ stages



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

- n processes
- Log n stages



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:

P_{i-1}		P_i		P_{i+1}
`Recv(`P_i`);`	←	`Send (`P_{i-1}`);`		
		`Send (`P_{i+1}`);`	→	`Recv(`P_i`);`
`Send(`P_i`);`	→	`Recv(`P_{i-1}`);`		
		`Recv(`P_{i+1}`);`	←	`Send(`P_i`);`

✧ 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.

A solution

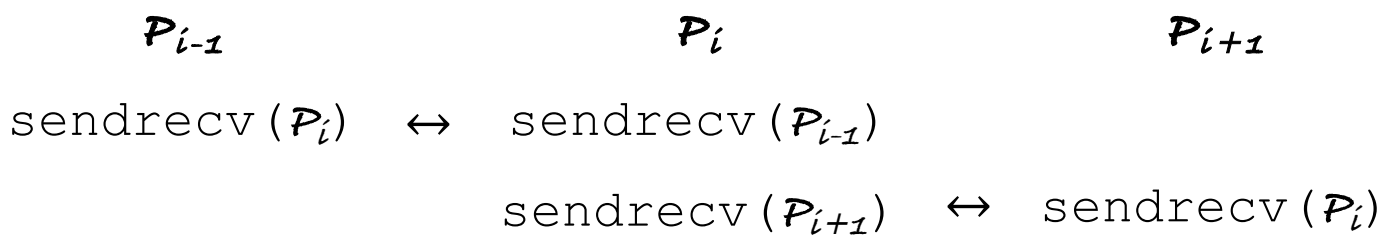
Arrange for one process to receive first and then send and the other process to send first and then receive.

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()



Synchronized Computations

Can be classified as:

- Fully synchronous

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

- Locally synchronous

  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.

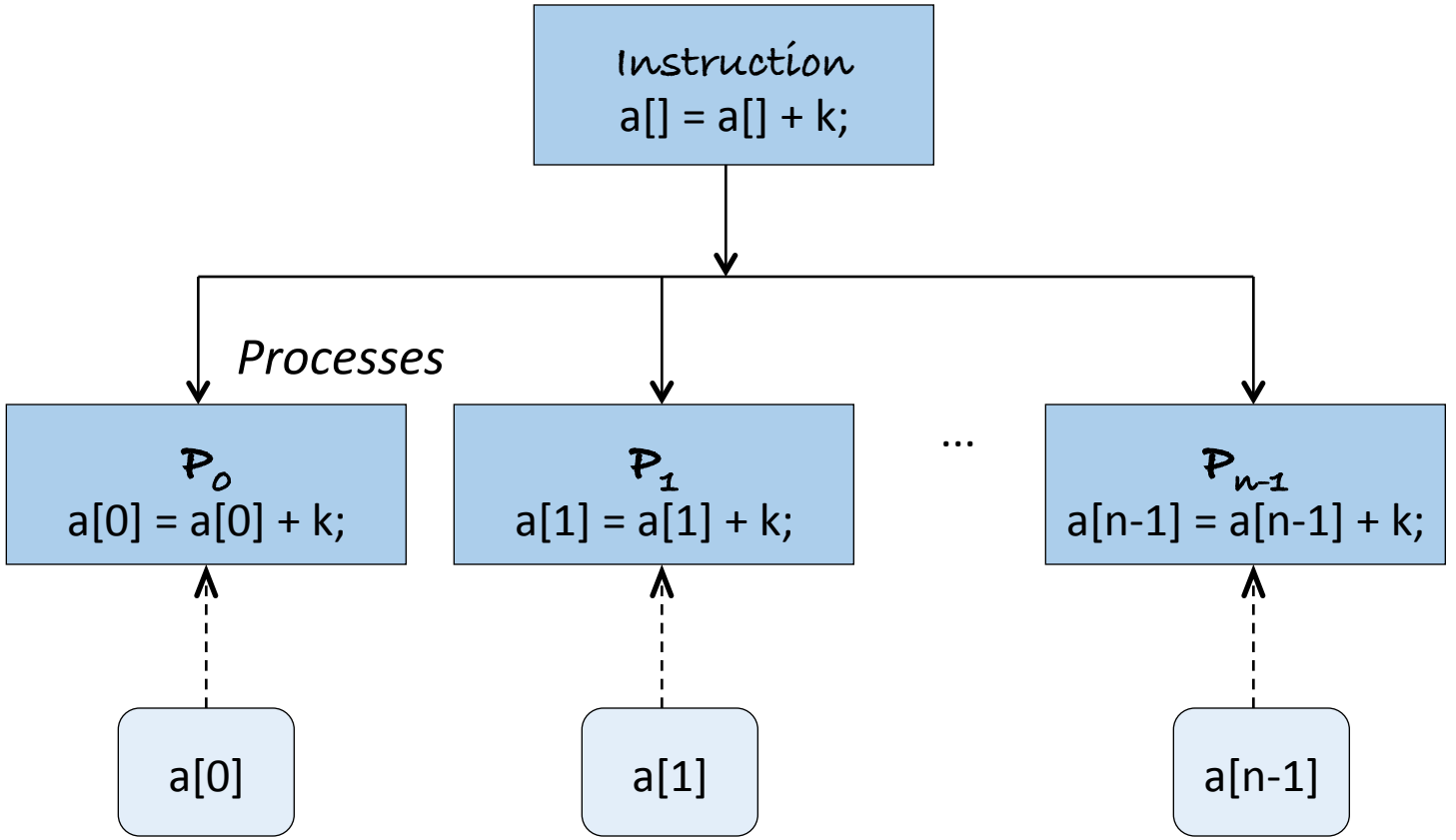
- Particularly convenient because:
  - Ease of programming (essentially only one program)
  - Can scale easily to larger problem sizes
  - Many numeric and some non-numeric problems can be cast in a data parallel form.

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 $(0 < i \leq n)$.



Forall construct

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

```
forall (i=0; i<n; i++)
   S;
```

states that n instances of the statements of the body (S) 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.

Example

- 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.

- SPMD: to add **k** to the elements of an array:

```
i = Get_Rank();    // Pi có Rank=i với 0 ≤ i ≤ n-1
a[i] = a[i] + k;   // Thực thi S trong vòng lặp thứ i
Barrier(group_p); // Đồng bộ rào cản cho tất cả n tiến trình
```

Prefix sum problem

0:	x_0
1:	$x_0 \oplus x_1$
2:	$x_0 \oplus x_1 \oplus x_2$
	…
n-1:	$x_0 \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1}$

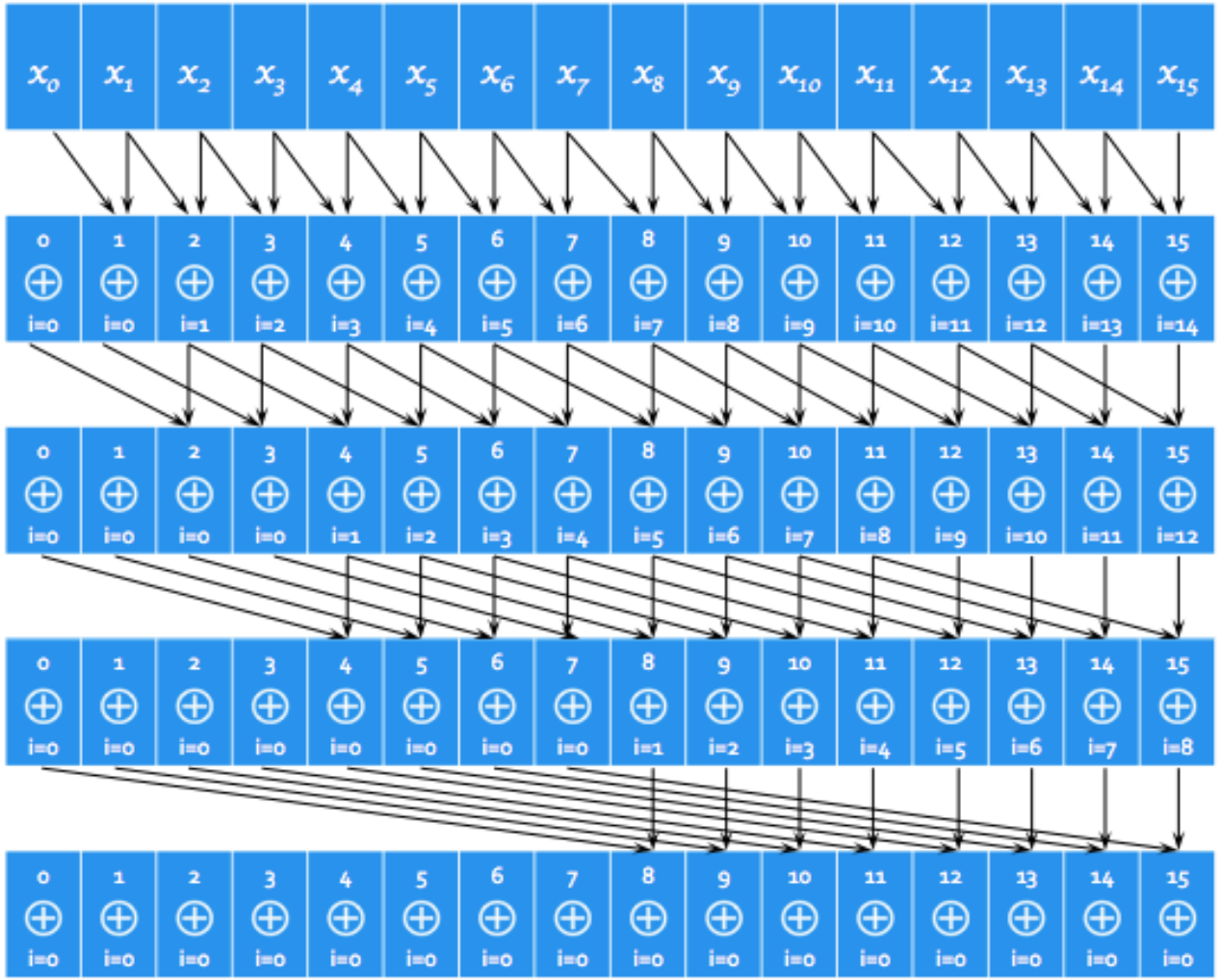
x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	2	3	4	5	6	7	8

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	6	10	15	21	28	36

```
1.  Sequential_Prefix_sums (n, x[]) {
2.       for (int i = 0; i < n; i++)
3.            x[i] = x[i-1] ⊕ x[i];
4.       return x[];
5.  }
```

O(n)

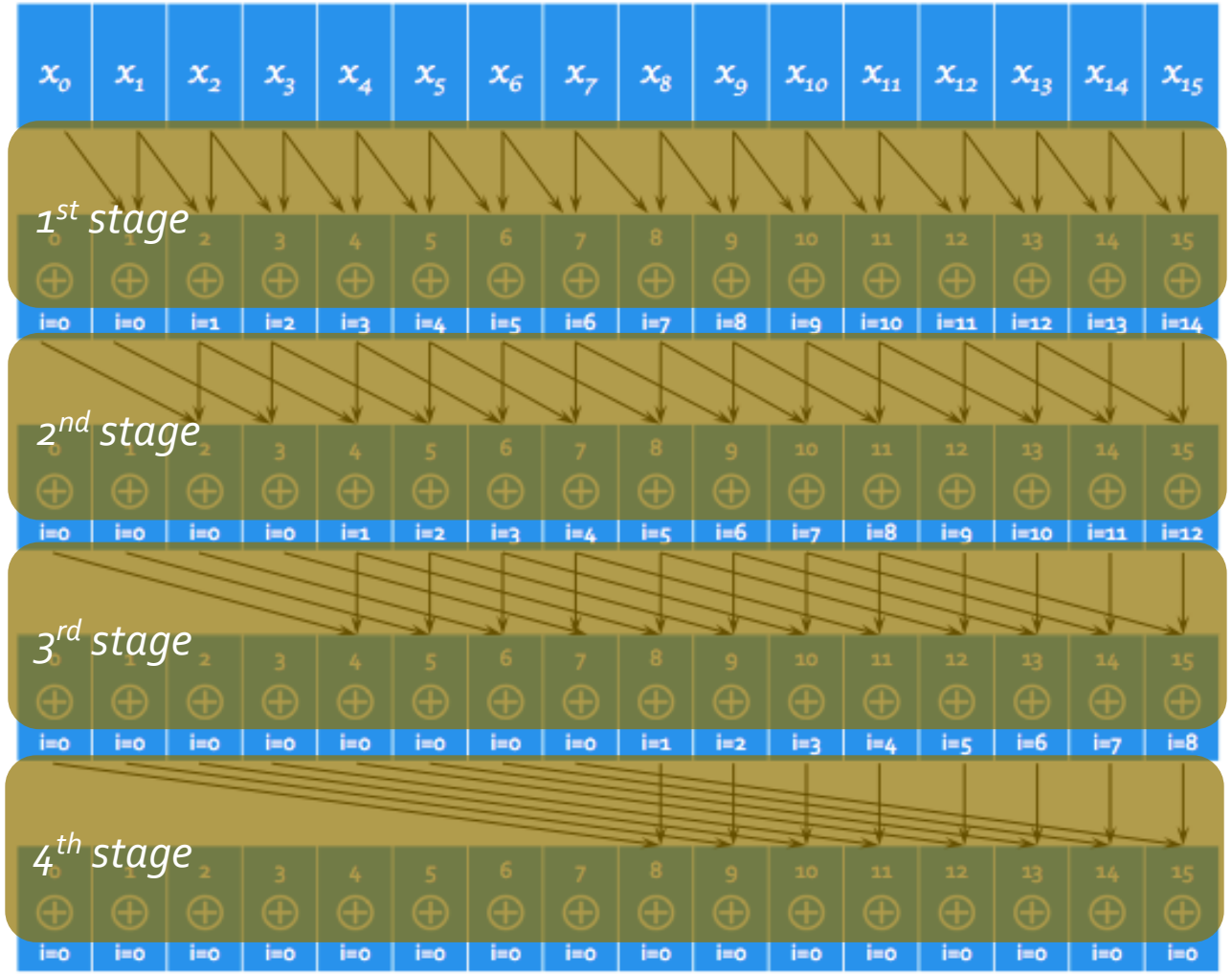
Data parallel example - prefix sum problem




```
1. Parallel_Prefix_sums (n, x[]) {
2.       for (int i = 0; i < ⌈log n⌉; i++)
3.           forall (int j = 0; j < n; j++)
4.               if (j ≥ 2^i)
5.                   x[j] = x[j] + x[j-2^i];
6.       return x[];
7.   }
```

O(logn)

Data parallel example - prefix sum problem




```
1. Parallel_Prefix_sums (n, x[]) {
2.       for (int i = 0; i < ⌈log n⌉; i++)
3.           forall (int j = 0; j < n; j++)
4.               if (j ≥ 2^i)
5.                   x[j] = x[j] + x[j-2^i];
6.       return x[];
7.   }
```

O(logn)

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**:

```
for (j=0; j<n; j++)        // Lặp n bước
  forall (i=0; i<p; i++) // p tiến trình thực hiện
      S(i);                // Công việc của mỗi tiến trình P_i
```

- SIMD:

```
for (j=0; j<n; j++) {  // Lặp n bước
  i = Get_Rank();    // P_i có rank=i với 0 ≤ i ≤ n-1
  S(i);              // Công việc của mỗi tiến trình P_i
  Barrier(group_p); // Đồng bộ rào cản cho tất cả p tiến trình
}
```

Solving a general system of linear equations by Iteration
(fully synchronous computation)

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

$$a_{0,0}x_o + a_{0,1}x_1 + a_{0,2}x_2 + \ldots + a_{0,n-1}x_{n-1} = b_0 \quad (\text{Equation}_0)$$

$$a_{1,0}x_o + a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n-1}x_{n-1} = b_1 \quad (\text{Equation}_1)$$

$$a_{2,0}x_o + a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n-1}x_{n-1} = b_2 \quad (\text{Equation}_2)$$

....

$$a_{n-1,0}x_o + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1} \quad (\text{Equation}_{n-1})$$

where the unknowns are x_o, x_1, x_2, ..., x_{n-1}.

Solution

By rearranging the i^{th} equation:

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

or

$$x_i = \frac{1}{a_{i,i}}(b_i - \sum_{j=0, j\neq i}^{n-1} a_{i,j}x_j) \ \ (0 \leq i < n)\,.$$

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

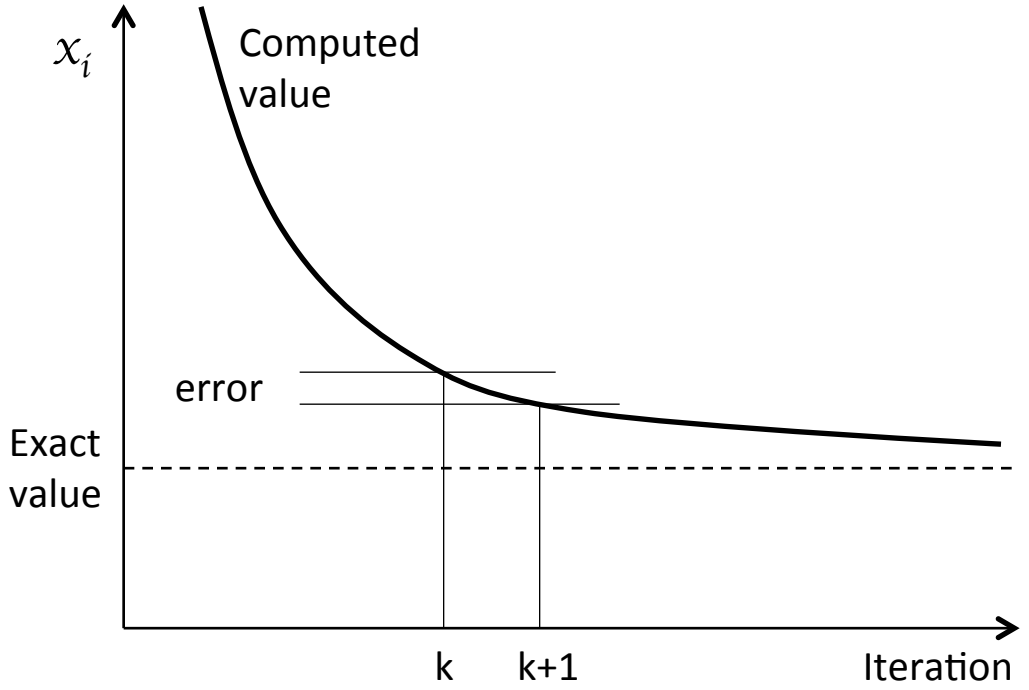
- Can be proven that the Jacobi method will converge if the diagonal values of a have an absolute value greater than the sum of the absolute values of the other a's on the row (the array of a's is diagonally dominant) i.e. if

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

- 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



$$\left|x_i^k - x_i^{k-1}\right| < \varepsilon \ (\forall i: 0 \leq i < n)$$

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

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

Algorithm (1)

- Solving a general system of linear equations by Jacobi Iteration
- OpenMP
- MPI???

```
1. Sync_comp_linear_equation (x[n]) {
2.   for (i=0; i<n; i++)
3.     x[i] = b[i];  // Nghiệm khởi đầu
     // Lặp đến Max_loop
4.   for (k=0; k<Max_loop; k++) {
5.     forall (i=0; i<n; i++) { // Lướt qua từng phương trình
6.       sum = -a[i][i] * x[i];
7.       for (j=0; j<n; j++)
8.         sum = sum + a[i][j] * x[j];
9.       new_x[i] = (b[i] - sum) / a[i][i]; // Tính x_i^k
10.    }
11.    acc  = Tolerance(x, new_x);// Điều kiện kết thúc
12.    forall (i=0; i<n; i++)
13.      x[i] = new_x[i]; // Cập nhật giá trị x_i^k ở bước lặp k
14.    if acc return 0;   // Kết thúc khi thoả điều kiện
15.  }
16.  return 0; // Kết thúc khi đạt Max_loop bước
17.}
```

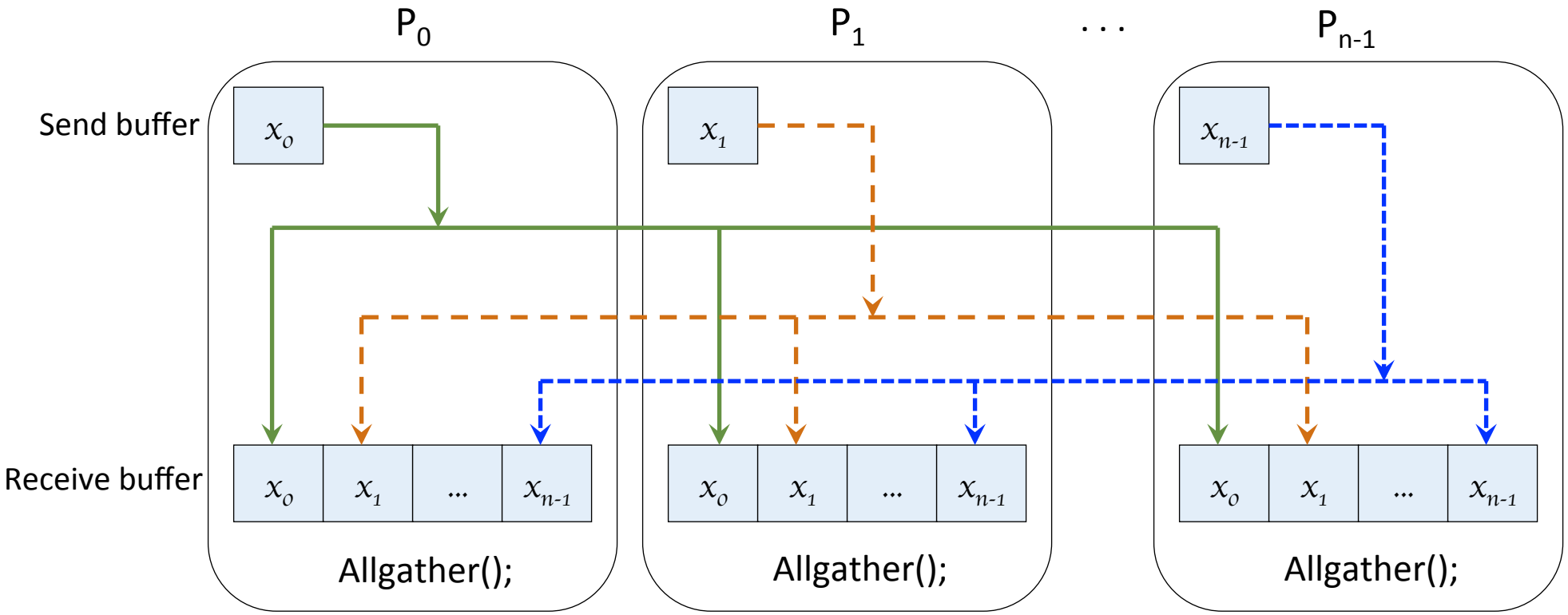
Algorithm (2)

- MPI
- Allgather

```
1. DS_Sync_comp_linear_equation () { // Tiến trình Pi
2.     x[i] = b[i];  // Nghiệm khởi đầu
     // Lặp đến Max_loop
3.   for (k=0; k<Max_loop; k++) {
4.     sum = -a[i][i] * x[i];
5.     for (j=0; j<n; j++)
6.       sum = sum + a[i][j] * x[j];
7.     new_x[i] = (b[i] - sum) / a[i][i]; // Tính x_i^k
       // Gởi new_x[i] và nhận new_x[j] từ tất cả Pj (j≠i)
8.     Allgather(new_x[i], new_x);
       // Đồng bộ rào cản cho tất cả n tiến trình
9.     Barrier(group_p);
10.    acc  = Tolerance(x, new_x);// Điều kiện kết thúc
11.    for (j=0; j<n; j++)
12.      x[j] = new_x[j]; // Cập nhật giá trị x_i^k ở bước lặp k
13.    if acc return 0;   // Kết thúc khi thoả điều kiện
14.  }
15.  return 0; // Kết thúc khi đạt Max_loop bước
16.}
```

Allgather

Broadcast and gather values in one composite construction.



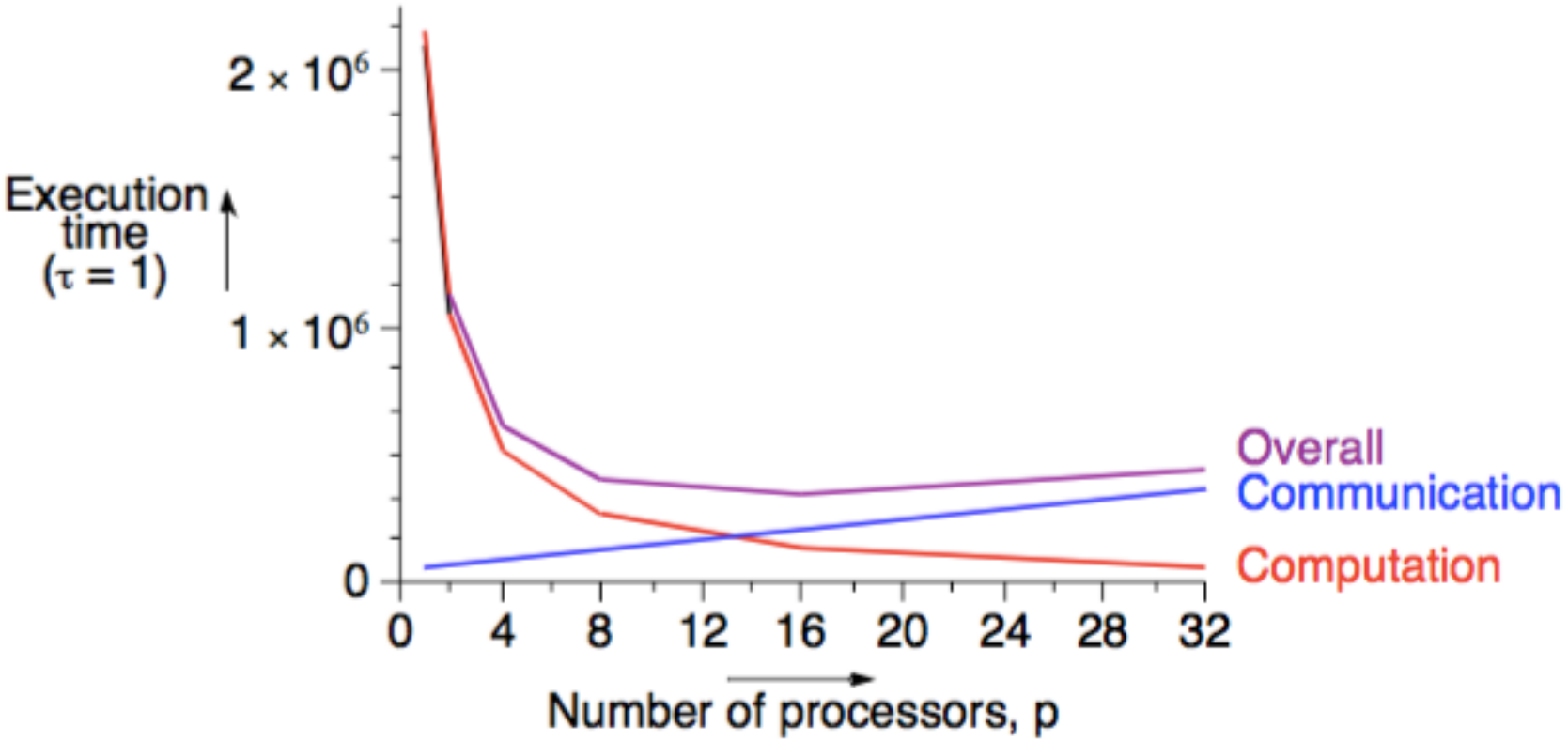
Partitioning

- 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

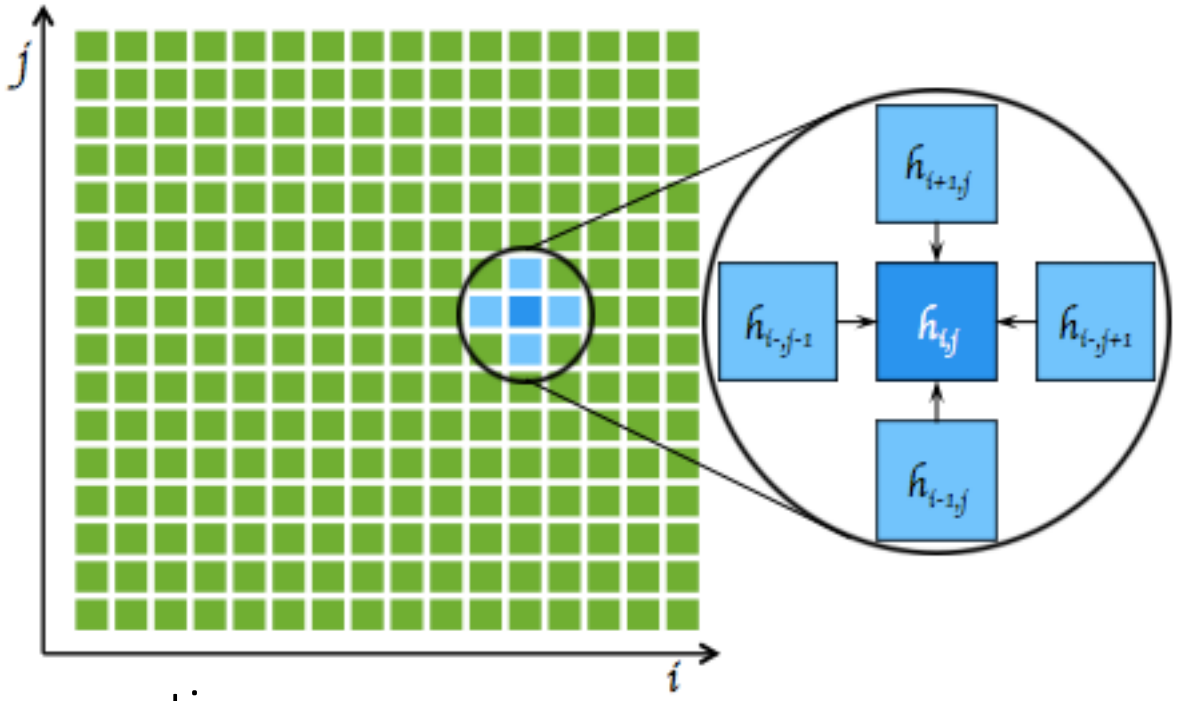
- Cyclic allocation no particular advantage here (Indeed, may be disadvantageous because the indices of unknowns have to be computed in a more complex way).

Effects of computation and communication in Jacobi iteration



Heat distribution problem
(Locally synchronous computation)

- 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.

Sequential algorithms

```
1. Seq_heat_distribution_ver1 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           g[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Cập nhật giá trị nhiệt mới tại bước k và h[i][j]
7.       for (i=1; i<n; i++)
8.         for (j=1; j<n; j++)
9.           h[i][j] = g[i][j];
   // Kiểm tra điều kiện kết thúc
10.      continue = false;
11.      for (i=1; i<n; i++)
12.        for (j=1; j<n; j++)
13.          if !converged(i, j) {
14.            continue = true;
15.            break;
16.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
17.   } while ((continue == true) && (k < (Max_loop-1)))
18. }
```

```
1. Seq_heat_distribution_ver2 () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       for (i=1; i<n; i++)
5.         for (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                              h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

Parallel algorithm

```
  // Lặp đến Max_loop
1.  for (k=0; k<Max_loop; k++) {
2.     h = 0.25 * (l + r + d + u);
  // Send() ở chế độ
  // không bị chặn (non-blocking)
3.     Send(&h, P_{i-1,j});
4.     Send(&h, P_{i+1,j});
5.     Send(&h, P_{i,j-1});
6.     Send(&h, P_{i,j+1});
  // Recv() ở chế độ hay đồng bộ
  // (synchronous) bị chặn (blocking)
7.     Recv(&l, P_{i-1,j});
8.     Recv(&r, P_{i+1,j});
9.     Recv(&d, P_{i,j-1});
10.    Recv(&u, P_{i,j+1});
11. }
```

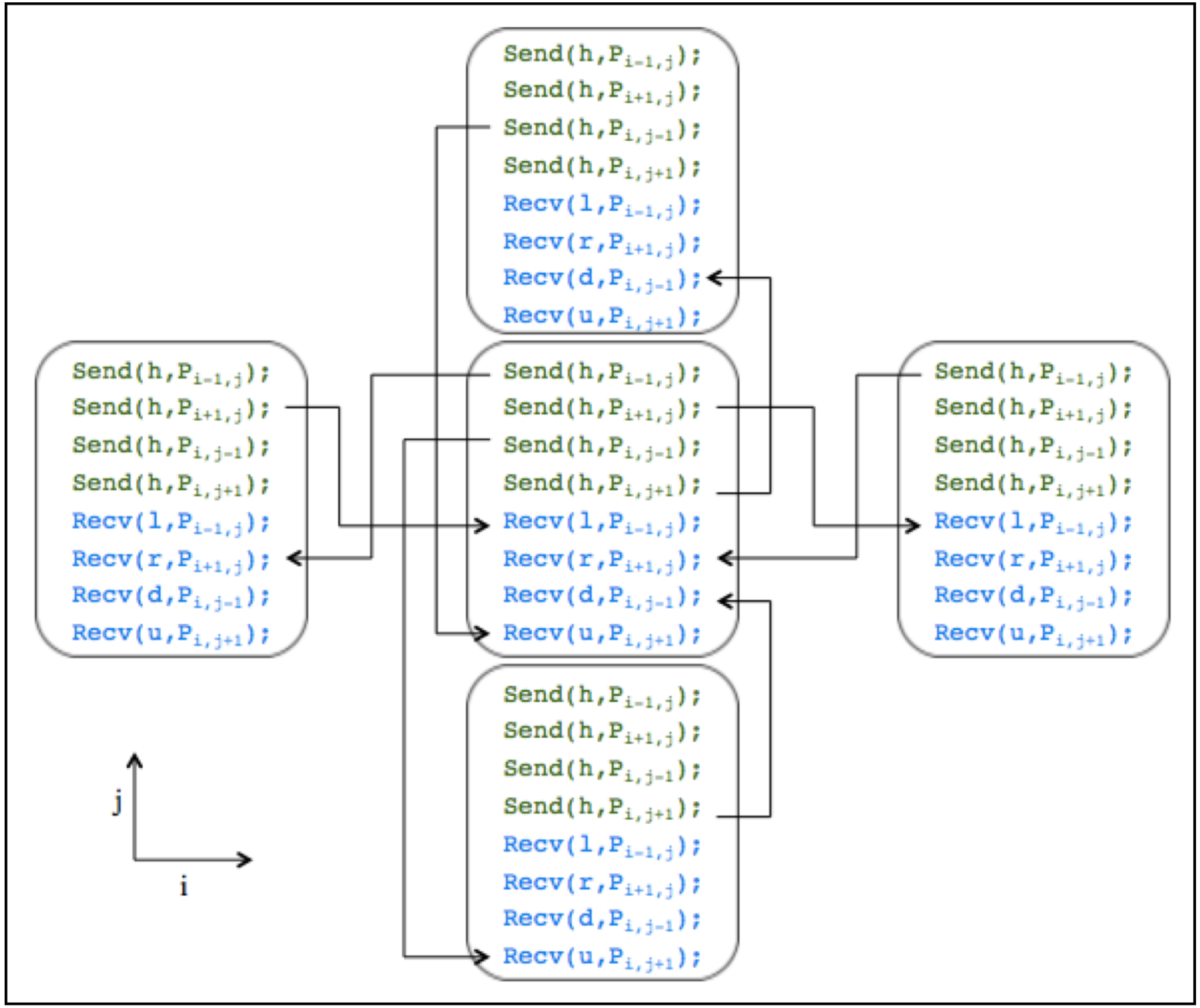
Local barrier

```
1. Parrallel_heat_distribution () {
2.   do {
3.     for (k=0; k<Max_loop; k++) { // Lặp đến Max_loop
   // Tính toán giá trị nhiệt mới tại bước k, không tính ở biên
4.       forall (i=1; i<n; i++)
5.         forall (j=1; j<n; j++)
6.           h[i][j] = 0.25 * (h[i-1][j] + h[i+1][j] +
                               h[i][j-1] + h[i][j+1]);
   // Kiểm tra điều kiện kết thúc
7.       continue = false;
8.       for (i=1; i<n; i++)
9.         for (j=1; j<n; j++)
10.          if !converged(i, j) {
11.            continue = true;
12.            break;
13.          }
   // Dừng khi đạt điều kiện kết thúc hoặc lặp đủ Max_loop bước
14.   } while ((continue == true) && (k < (Max_loop-1)))
15. }
```

- Message-passing

  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 (1)



Message passing for heat distribution problem (2)

- Master/Slave

$$h_{i,j} \leftrightarrow h$$

$$h_{i-1,j} \leftrightarrow l$$

$$h_{i+1,j} \leftrightarrow r$$

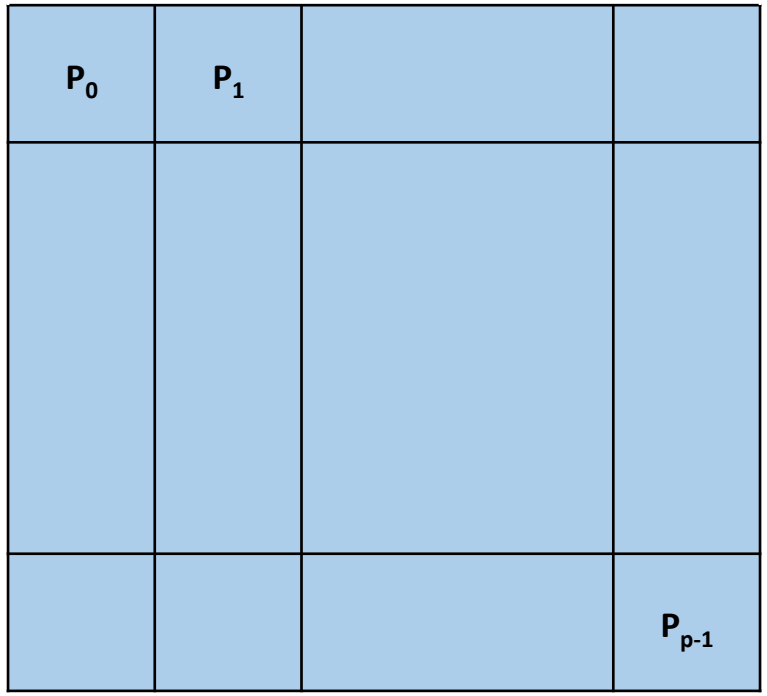
$$h_{i,j-1} \leftrightarrow d$$

$$h_{i,j+1} \leftrightarrow u$$

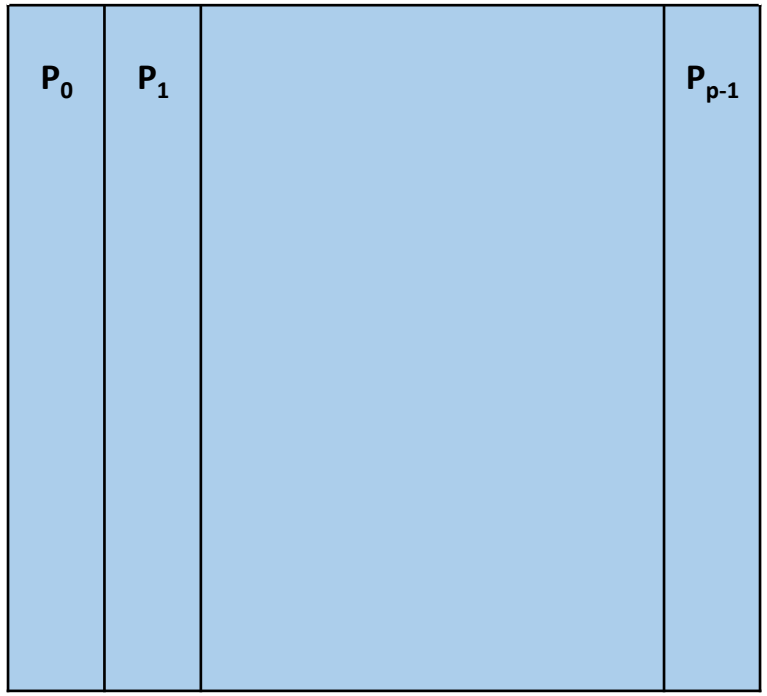
```
1.  k=0; // Bước lặp thứ k
2.  do {
3.    k++;
4.    h = 0.25 * (l + r + d + u);
  // Send() ở chế độ không bị chặn (non-blocking)
5.    Send(&h, P_{i-1,j});
6.    Send(&h, P_{i+1,j});
7.    Send(&h, P_{i,j-1});
8.    Send(&h, P_{i,j+1});
  // Recv() ở chế độ hay đồng bộ(synchronous) bị chặn (blocking)
9.    Recv(&l, P_{i-1,j});
10.   Recv(&r, P_{i+1,j});
11.   Recv(&d, P_{i,j-1});
12.   Recv(&u, P_{i,j+1});
13. } while (!converged(i, j) && (k < Max_loop));
14. Send(&h, &i, &j, &k, P_master);
```

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

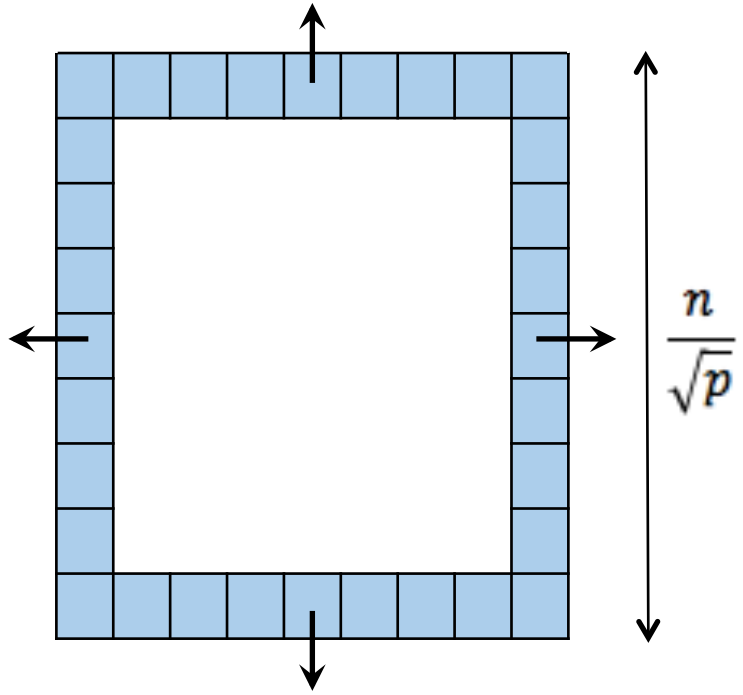


Strips (columns)

Block partition

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

$$T_{commB} = 8(t_{startup} + \frac{n}{\sqrt{p}} t_{data})$$



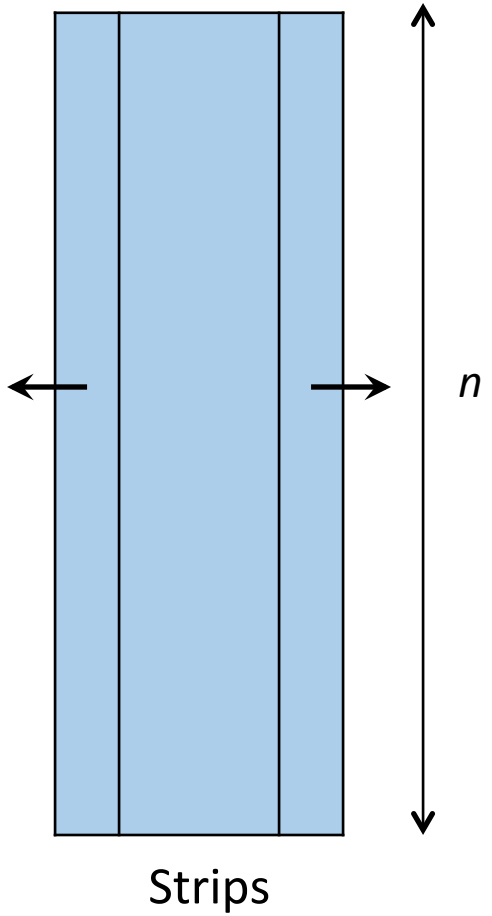
Square blocks

Communication consequences of partitioning

Strip partition

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

$$T_{commC} = 4(t_{startup} + nt_{data})$$

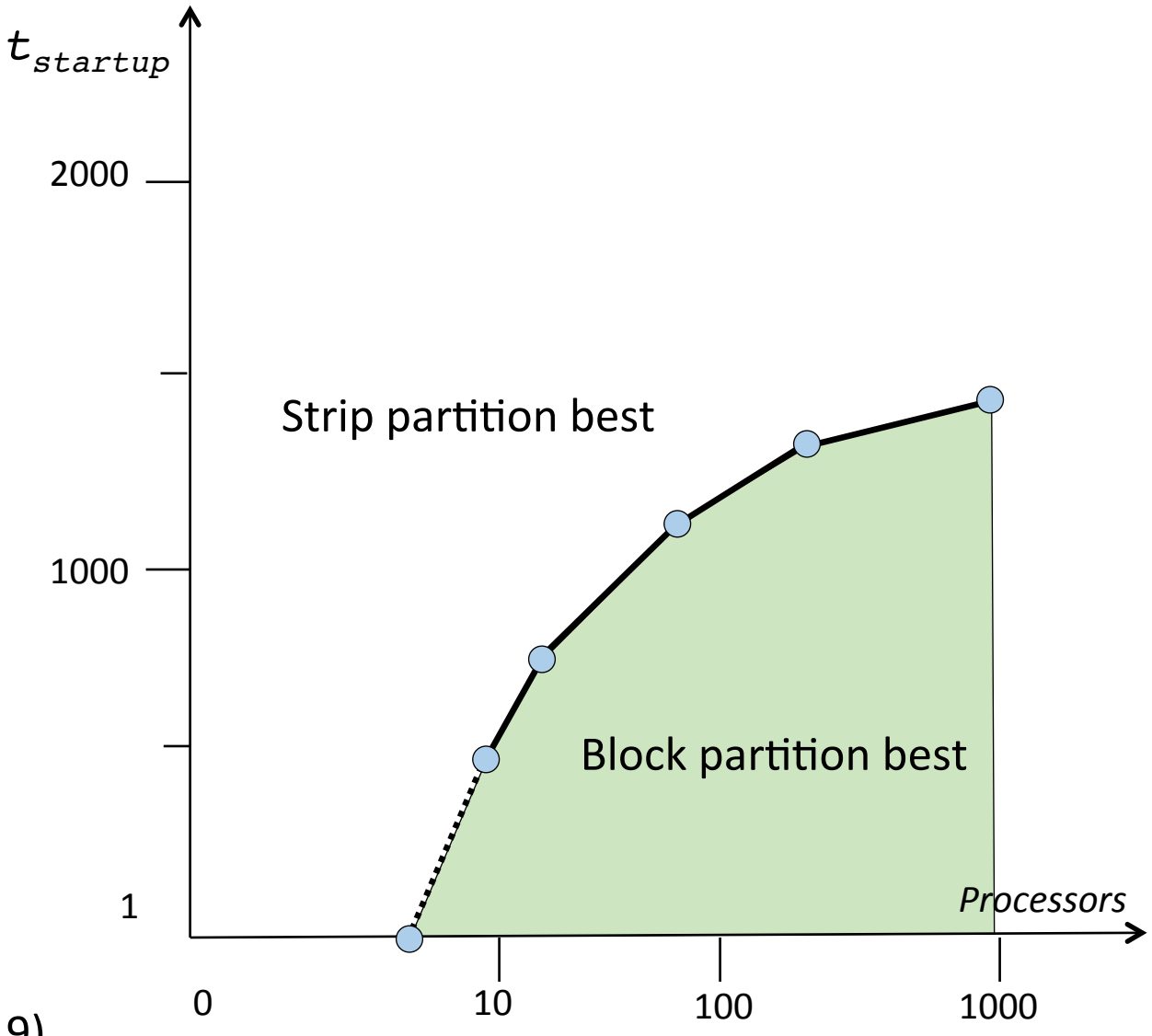


Optimum

- 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

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

$$\Leftrightarrow t_{startup} > n(1 - \frac{1}{\sqrt{p}}) t_{data} \quad \text{(p >= 9)}$$



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 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:

P_{i-1}		P_i		P_{i+1}
`Recv(`P_i`);`	←	`Send(`P_{i-1}`);`		
`Send(`P_i`);`	→	`Recv(`P_{i-1}`);`		
		`Recv(`P_{i+1}`);`	←	`Send(`P_i`);`
		`Send(`P_{i+1}`);`	→	`Recv(`P_i`);`

- 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().

Cellular Automata
(Fully synchronous problems)

- 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
- Also good assignment for graphical output.

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.”

Sharks and Fishes
(Simple fun examples of Cellular Automata)

- An ocean could be modeled as a three-dimensional array of cells
- Each cell can hold one fish or one shark (but not both)
- Fishes 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.

SHARK

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 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.

Serious applications for Cellular Automata

- 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.