Lesson 6: Synchronous computations

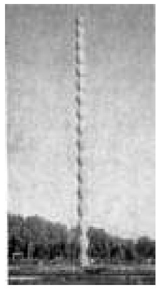
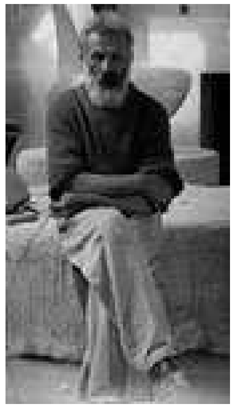
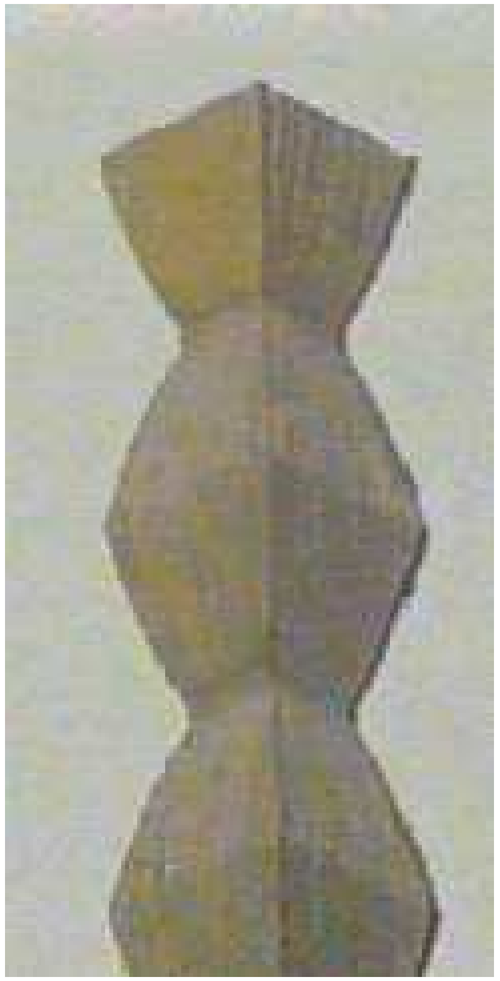
G Stefanescu — University of Bucharest

Parallel & Concurrent Programming
Fall, 2014

Synchronous computations

Synchronous computations:

- *all processes are repeatedly synchronized at regular points*
- a very important class of problems (e.g., 70% of cases in a Caltech set of parallel programming applications)




Brancusi: The endless column

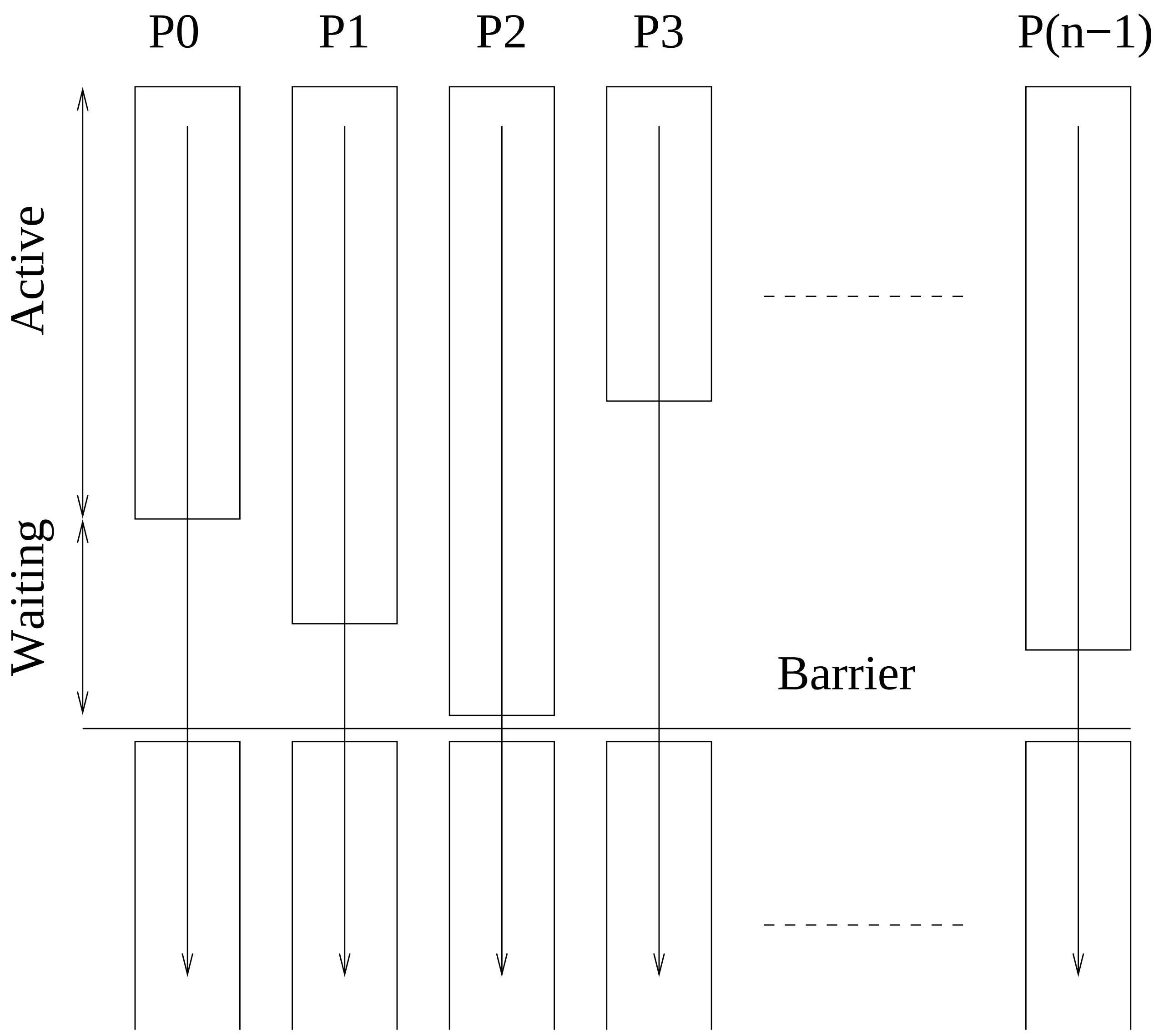
Barrier

Barrier:

- a barrier is a *basic mechanism* for synchronizing processes
- the barrier statement has to be inserted in each process at the point *where it has to wait* for synchronization
- a processes can *continue* from the synchronization point *when all processes have reached the barrier* (or, in a partially synchronous case, a stated number of processes have reached the barrier).

..Barrier

Processes reaching the barier at different times:



MPI/PVM barrier routines

In message passing systems, barriers are often provided as library routines

- MPI: `MPI_Barrier()`
  - it is a barrier with only one parameter, namely the communicator group name
  - it has to be called by each process in the group, blocking processes until all members of the group have reached the barrier
- PVM: `pvm_barrier`
  - it is similar
  - PVM has the unusual feature of specifying the number of processes that must reach the barrier in order to release the processes

Implementation

Centralized counter implementations (or *linear barrier*): Such counter-based barriers often have two phases:

- a process enter an arrival phase and does not leave this phase until all processes have arrived in this phase
- then the process move to a departure phase and it is released

Good implementation of a barrier must take into account that a barrier might be used more than once *in a process. So, it 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. The two-phase above handles this scenario.*

..Implementation

Example of implementation:

Master:

```
for (i=0; i<n; i++)
    recv($P_{any}$);
for (i=0; i<n; i++)
    send($P_i$);
```

Slave:

```
send($P_{master}$);
recv($P_{master}$);
```

Tree implementation

Tree implementation - it is *more efficient*; suppose there are eight processes P0,P1,. . . , P7:

—1st stage:

P1 sends message to P0 (when P1 has reached its barrier)

P3 sends message to P2 (when P3 has reached its barrier)

P5 sends message to P4 (when P5 has reached its barrier)

P7 sends message to P6 (when P7 has reached its barrier)

—2nd stage:

P2 sends message to P0 (P1,P2,P3 have reached their barrier)

P6 sends message to P4 (P5,P6,P7 have reached their barrier)

—3rd stage:

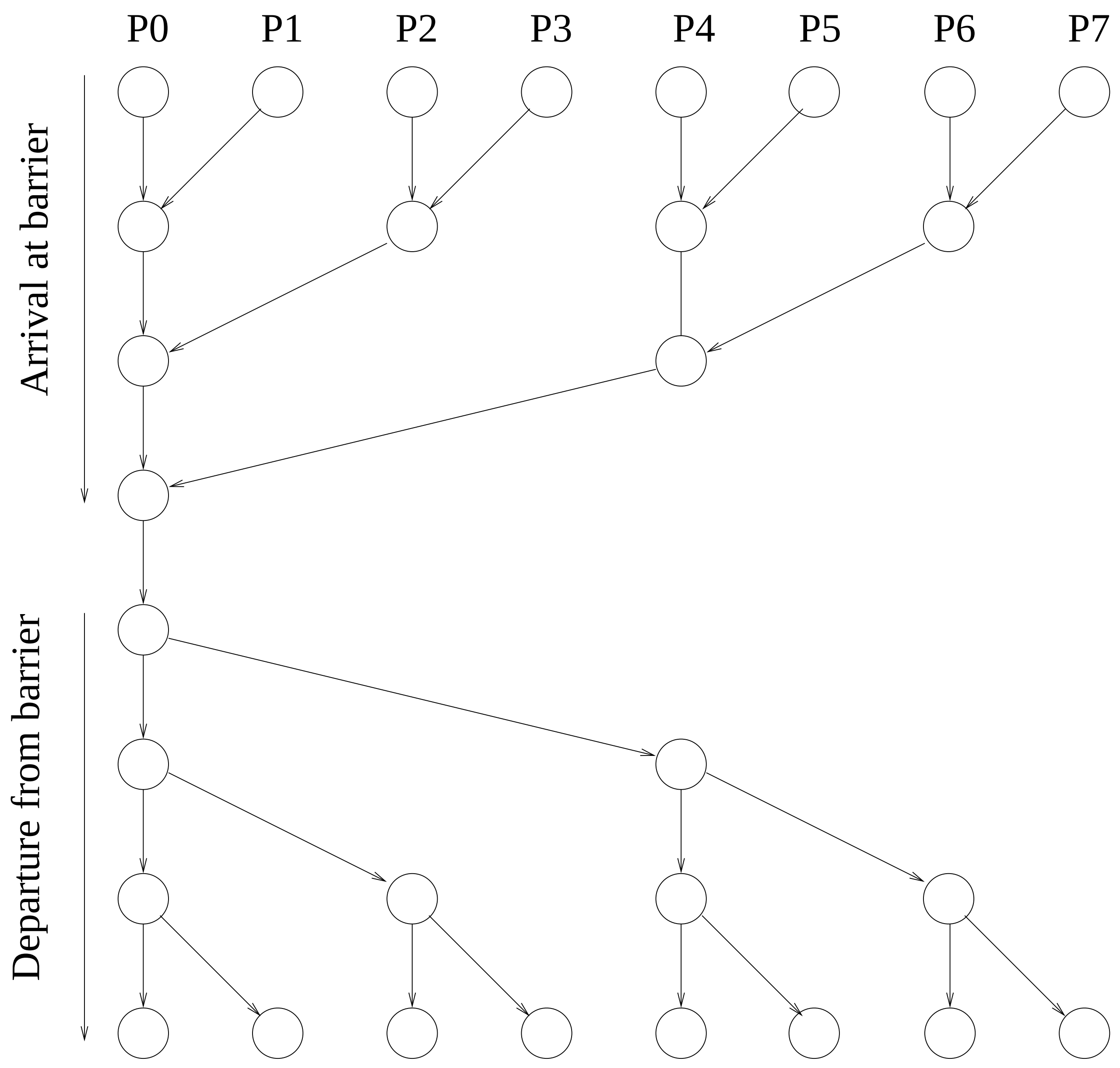
P4 sends message to P0 (P1,P2,P3,P4,P5,P6,P7 have arrived)

—Final stage:

P0 terminates the arrival phase (when P0 has reached its barrier and it has received message from P4)

..Tree implementation

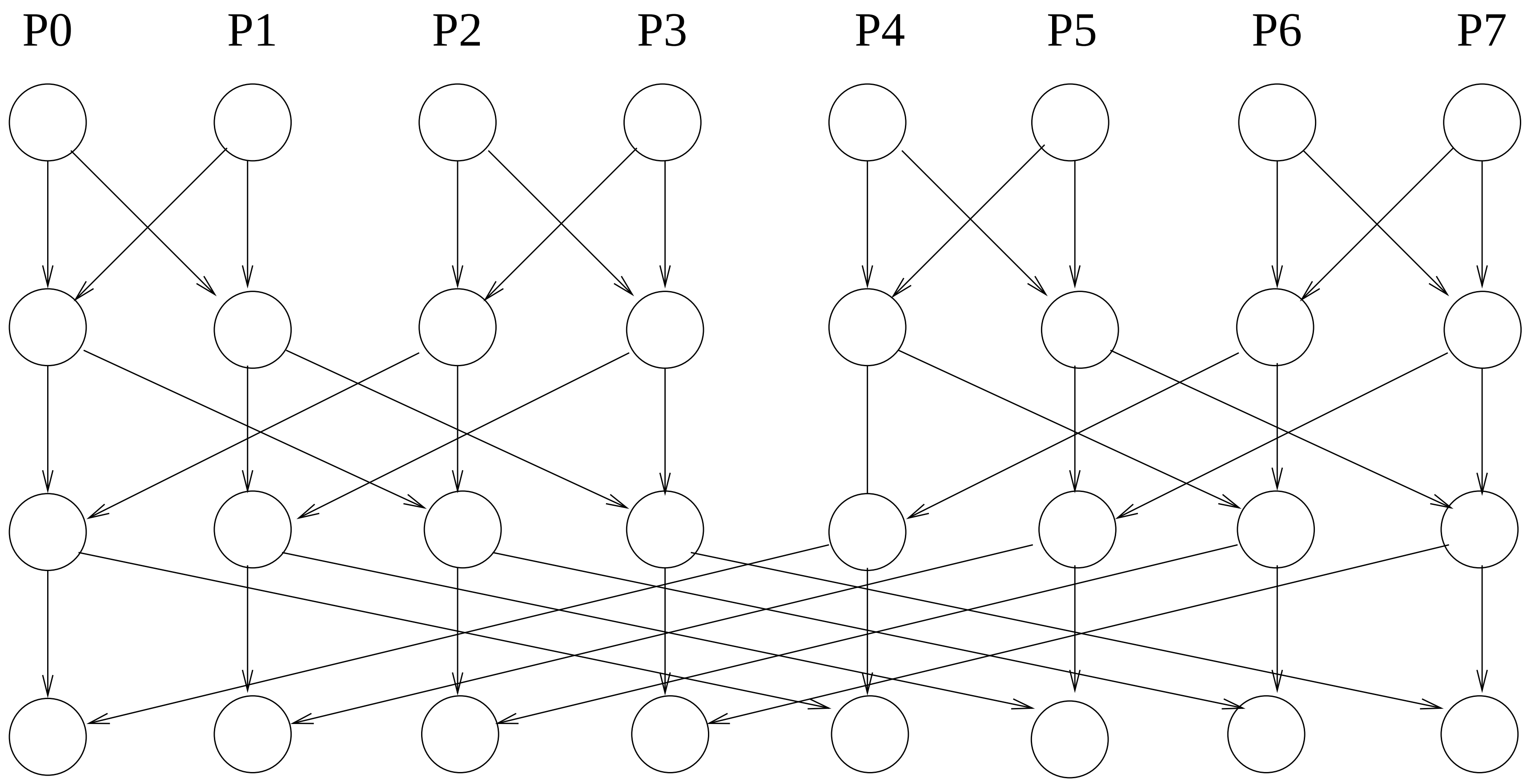
Tree barrier:



Butterfly barrier

Butterfly barrier:

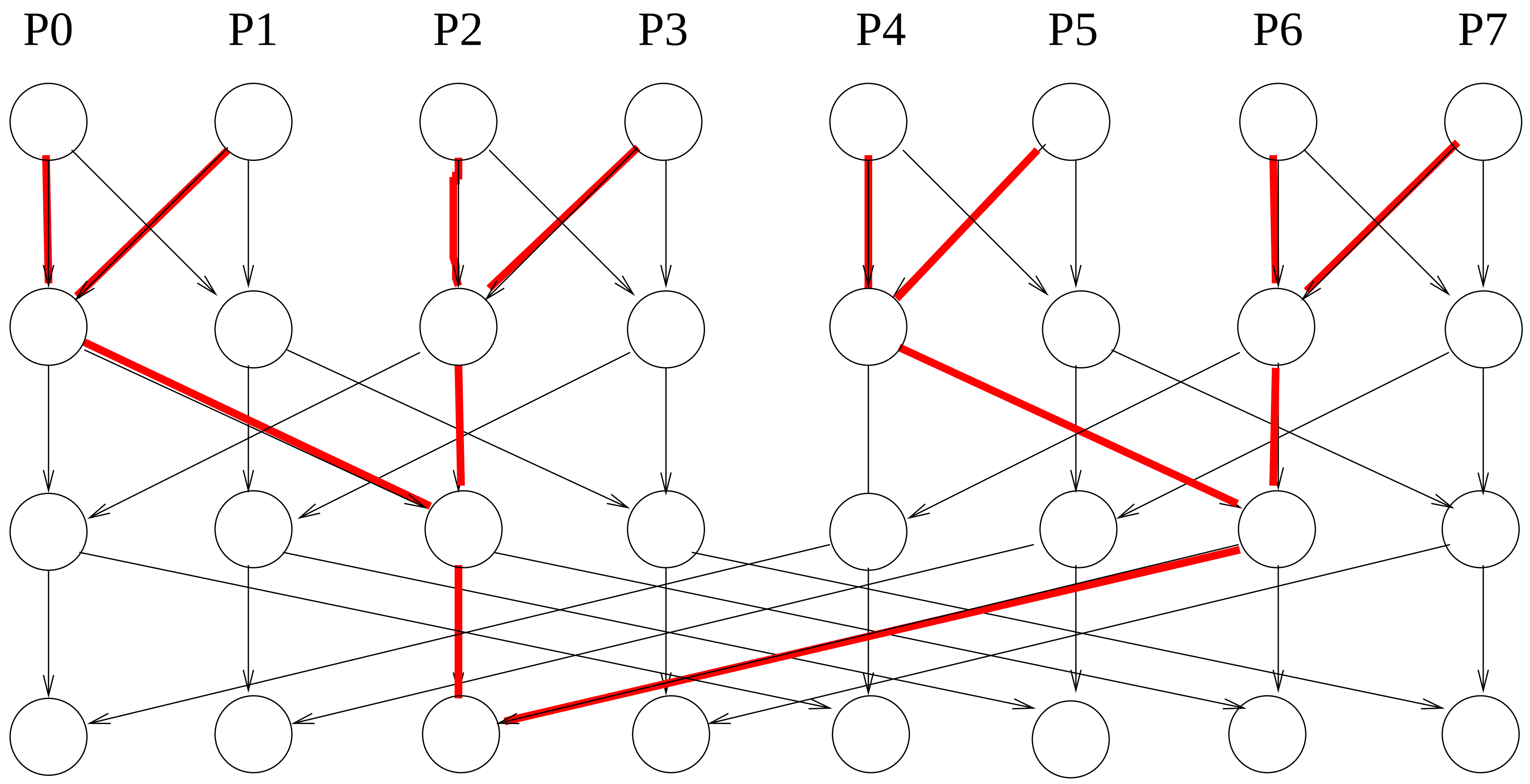
—1st stage: $P0 \longleftrightarrow P1, P2 \longleftrightarrow P3, P4 \longleftrightarrow P5, P6 \longleftrightarrow P7$
—2nd stage: $P0 \longleftrightarrow P2, P1 \longleftrightarrow P3, P4 \longleftrightarrow P6, P5 \longleftrightarrow P7$
—3rd stage: $P0 \longleftrightarrow P4, P1 \longleftrightarrow P5, P2 \longleftrightarrow P6, P3 \longleftrightarrow P7$



..Butterfly barrier

Useful if *data are exchanged* at the barrier.

E.g., observe how P2 gathers data from all processes:



Gahter data from all processes

Partial barriers

Local synchronization: Suppose a process P_i needs to be synchronized and to exchange data with processes P_{i-1} and P_{i+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});
```

Notice that this is not a perfect three-process barrier because, for instance, process P_{i-1} will only synchronized with P_i and continue as soon as P_i allows. Similarly for P_{i+1} and P_i.

Deadlock

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

- Deadlock will occur if *both processes first perform the send, using synchronous routines* (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 *first to receive* and then to send and for the *other* process *first to send* and then to receive.

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

..Deadlock

A different solution is to use *combined deadlock-free blocking* routines, e.g., `sendrecv()` routines.

In MPI there are two versions:

- `MPI_Sendrecv()` and `MPI_Sendrecv_replace()` (the former uses 2 buffers, while the latter uses only one buffer)

Synchronized computations

Data parallel computations:

- *Some operation* is to be performed on *different data* elements *simultaneously* (in parallel)
- This is particularly convenient because:
  - it is *easy to write programs*
  - it easily *scales* to larger problem sizes
  - *many numeric and some non-numeric problems* can be casted into such a data parallel format

Example: 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` may be executed simultaneously by multiple processors using different indices i

Forall

Forall construct: There are special "parallel" constructs in parallel programming languages to specify such a data parallel operation.

E.g.,

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

states that `n` instances of the `body` can be executed simultaneously, one for each value of *i* in the given range.

The term "forall" is unfortunate here, as there are not iterations, each instance performing one execution of the `body`*, only. The previous example may be written as*

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

Prefix sum problem

Prefix sum problem:

- Given a list of numbers $x_0, \ldots, x_{n-1}$, compute *all the partial summations* (i.e., $x_0, x_0 + x_1, x_0 + x_1 + x_2, \ldots$).
- It may use *other associative operations* rather than addition.
- It was *wildly studied*, having practical applications is areas such as: *process allocation, data compaction, sorting, polynomial evaluation, etc.*

..Prefix sum problem

A sequential code may be:

```
for (i=0; i<n; i++){
    sum[i] = 0;
    for (j=0; j <= i; j++)
        sum[i] = sum[i] + x[j]
}
```

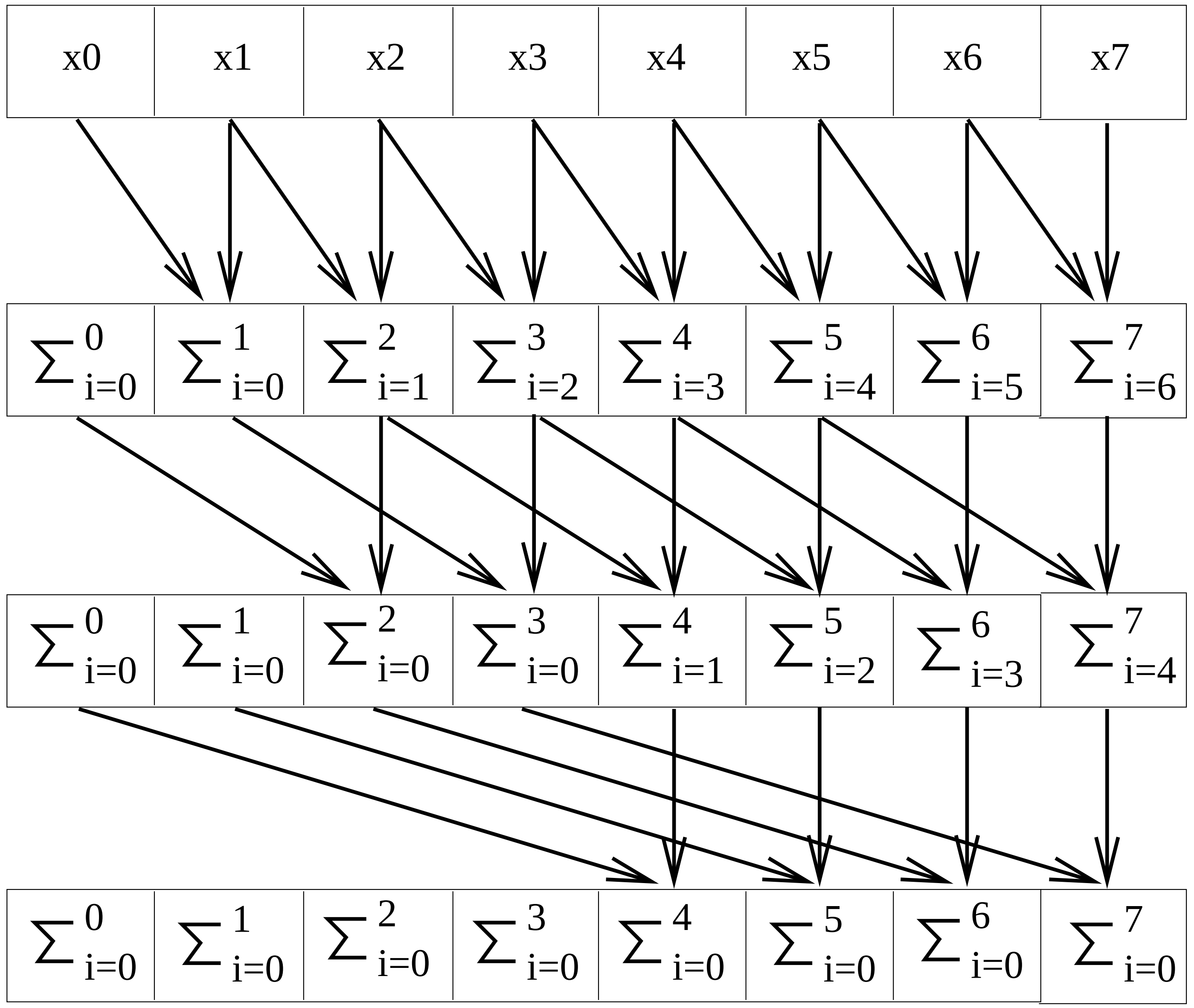
(this is an $O(n^2)$ algorithm)

Parallel code

```
for (j=0; j < log(n); j++)
    forall (i=0; i<n; i++)
        if (i>=$2^j$) x[i] = x[i] + x[i-$2^j$]
```

..Prefix sum problem

A concrete example, namely for 8 numbers:



Synchronous iteration

Synchronous iteration: this term is used to describe a situation where a problem is *solved by iteration* and

- *each iteration step* is composed of *several processes that start together* at the beginning of the iteration step and
- *next iteration step* cannot begin until *all processes have finished* the current iteration step.

Example:

```
for (j=0; j<n; j++){
    i = myrank;
    body(i);
    barrier(mygroup);
}
```

Case studies: 1. Systems of linear equations

Systems of linear equations:

A (general) systems of linear equations looks like follows:

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

where a's and b's are constants and x's are unknown to be found.

..Case studies: Systems

Iterative methods: One way to solve these systems of equations is by *iterations*.

One may rearrange 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 get

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

or

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

Such an equation specifies an unknown in terms of the other unknowns and can be used as an iteration formula (with the hope to get a convergent iteration process)

..Case studies: Systems

Jacobi iteration: in this iterative process (based on the above general schema) *all* values of x's are *updated together*.

It can be proved that Jacobi method will *converge if the diagonal values of a's have an absolute value greater that the sum of the absolute values of the others a's on the row* (or, in other words, if the matrix of a's is diagonally dominant), i.e.,

$$\sum_{j \neq i} |a_{i,j}| < |a_{i,i}| \quad (\forall i)$$

(this is a sufficient, but not a necessary condition)

..Case studies: Systems

Termination (Jacobi iteration): A simple approach is

- to *compare the values computed at one iteration step* to the values obtained at the *previous iteration step* and
- to terminate the computation at the t-th iteration when all these differences are within a *given tolerance*, i.e.,

$$\forall i : |x_i^t - x_i^{t-1}| < \varepsilon \quad (\varepsilon \text{ - error tolerance})$$

(x_k^t denotes the value of x_k at t-th iteration)

Other appropriate termination conditions are:

- $\sqrt{\sum_{i=0}^{n-1} (x_i^t - x_i^{t-1})^2} < \varepsilon$ *(Pacheco) or*
- $\forall i : |\sum_{j=0}^{n-1} a_{i,j} x_i^t - b_i| < \varepsilon$ *(Bertsekes & Tsitsiklis)*

..Case studies: Systems

Sequential code:

```
for (i=0; i<n; i++)
    x[i] = b[i];
for (iter = 0; iter < limit; iter++)
    for (i=0; i<n; i++){
        sum = 0;
        for (j=0; j<n; j++)
            if (i != j)
                sum = sum + a[i][j]*x[j];
        newx[i] = (b[i]-sum)/a[i][i]
    }
    for (i=0; i<n; i++)
        x[i] = newx[i];
}
```

..Case studies: Systems

More efficient sequential code (avoiding the `if` statement):

```
for (i=0; i<n; i++)
    x[i] = b[i];
for (iter = 0; iter < limit; iter++)
    for (i=0; i<n; i++){
        sum = -a[i][i]*x[i];
        for (j=0; j<n; j++)
            sum = sum + a[i][j]*x[j];
        newx[i] = (b[i]-sum)/a[i][i]
    }
    for (i=0; i<n; i++)
        x[i] = newx[i];
}
```

..Case studies: Systems

Parallel code: suppose we have a process P_i for each unknown x_i; the code for process P_i may be:

```
x[i] = b[i];
for (iter = 0; iter < limit; iter++){
    sum = -a[i][i]*x[i];
    for (j=0; j<n; j++)
        sum = sum + a[i][j]*x[j];
    newx[i] = (b[i]-sum)/a[i][i]
    broadcast_receive(&newx[i]);
    global_barrier();
}
```

..Case studies: Systems

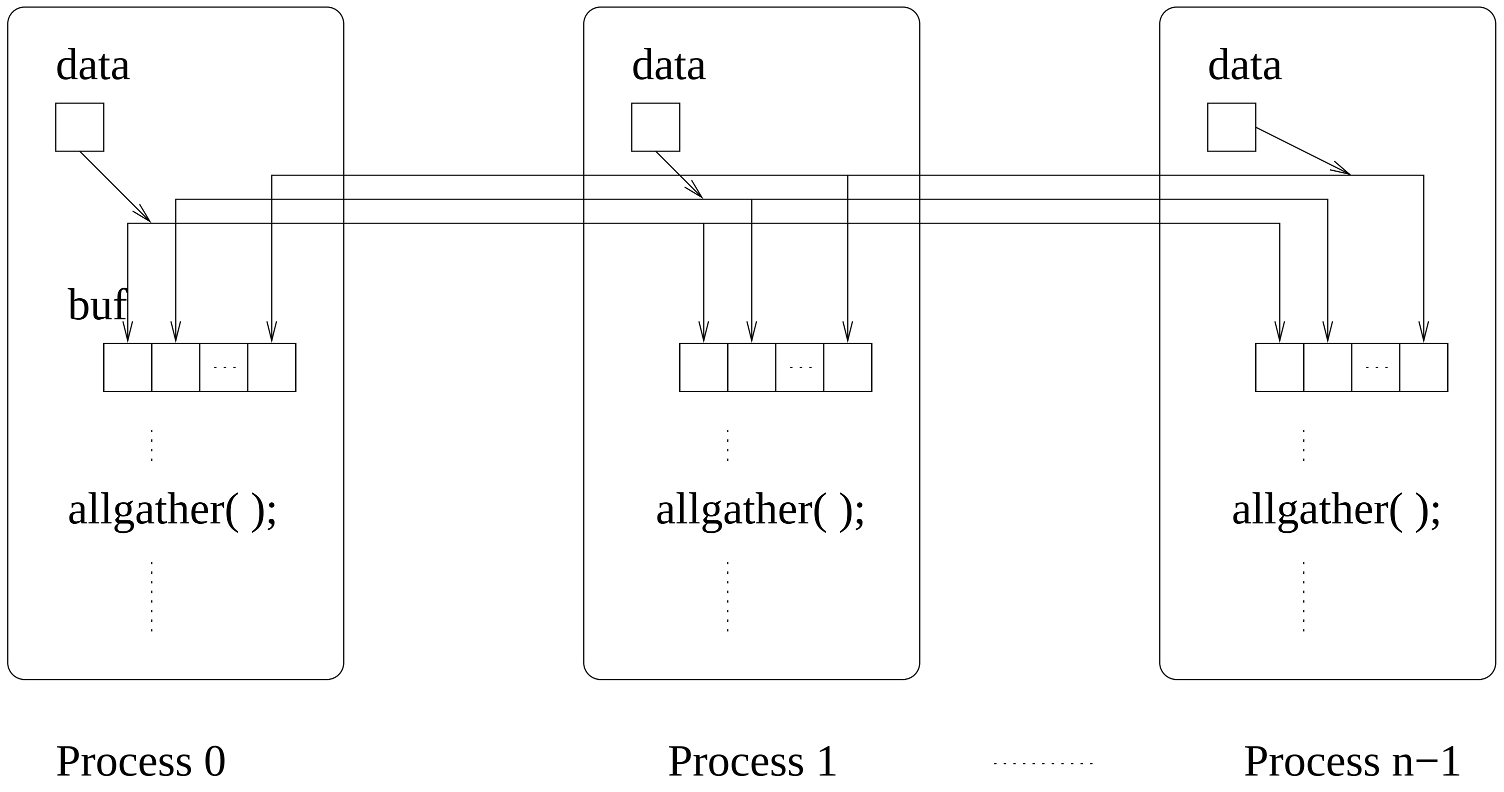
Notice:

- `broadcast_receive()` is used here (1) to *send* the newly computed value of `x[i]` from process P_i *to every* other process and (2) to *collect* data broadcasted *from any* other processes to process P_i

- an alternative simple solution is to use individual `send/recv` routines

..Case studies: Systems

All gather: Broadcast and gather values in *one* composite construction



Notice: `MPI_Allgather` *is also a global barrier, so you do not need to add* `global_barrier()`.

..Case studies: Systems

A version where the termination condition is included:

```
x[i] = b[i];
iter = 0;
do {
    iter++;
    sum = -a[i][i]*x[i];
    for (j=0; j<n; j++)
        sum = sum + a[i][j]*x[j];
    newx[i] = (b[i]-sum)/a[i][i]
    broadcast_receive(&newx[i]);
} while (tolerance() && (iter < limit));
```

..Case studies: Systems

Partitioning:

- Usually the number of *processors* is *smaller* that the number of *unknowns*, hence each process can be responsible for computing a group of unknowns
- one may use
  - *block* partitions: allocate consecutive unknowns to a process
  - *cyclic* partition: P_0 handles $x_0, x_p, \ldots, x_{((n/p)-1)p}$, etc. (this is worse here as many such complex indices have to be computed)

..Case studies: Systems

Analysis: Suppose there are n equations and p processors; hence a processor handle n/p unknowns. Suppose there are τ iterations. Then:

- Computation time: $t_{comp} = n/p(2n+4)\tau$
- Communication time (broadcast):
  $t_{comm} = p(t_{startup} + (n/p)t_{data})\tau = (pt_{startup} + nt_{data})\tau$
- Overall: $t_p = (n/p(2n+4) + pt_{startup} + nt_{data})\tau$

When $t_{startup} = 10000$ and $t_{data} = 50$, for various p and τ this has a minimum *value around $p \approx 16$, $\tau \approx 0.4 \times 10^6$.*

Case studies: 2. Heat distribution problem

Heat distribution problem: Suppose *a square metal sheet has known temperatures along each of its edges*. The goal is to *find temperature distribution of the metal square* (at equilibrium).

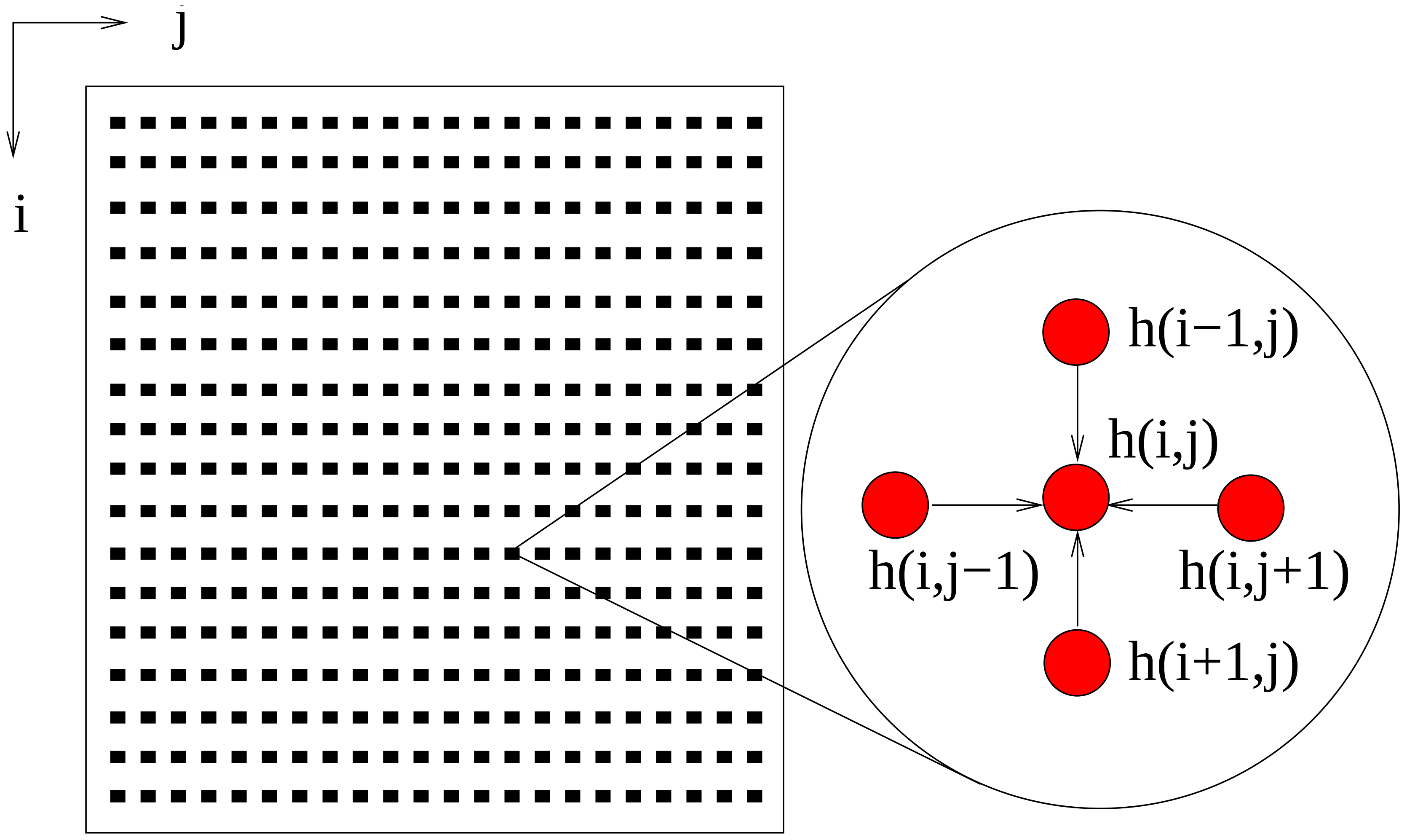
Solution:

- divide the area into a *fine mesh* of points $h_{i,j}$
- the evolution in time is obtained by using the following relation

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

- as in a usual iterative method, we *repeat* for a fixed number of iterations or until the difference between the values at two consecutive iteration steps is less that a very small prescribed amount in each point.

..Case studies: Heat distribution



..Case studies: Heat distribution

Connection with systems of linear equations: For convenience, the points are numbered from 1 to k^2. Each point will have an associated equation

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

leading to a system of linear equations

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

This is the *finite difference* system associated to *Laplace equation*.

Back: We come back to our double indices. The temperature on the boundary is known, i.e., we know $h_{0,r}, h_{n,r}, h_{r,0}, h_{r,n}$ for any $0 \leq r \leq n$. A sequential code (including a termination condition) is:

..Case studies: Heat distribution

```
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++)
      for (j=1; j<n; j++)
         h[i][j] = g[i][j];
   cont = FALSE;
   for (i=1; i<n; i++)
      for (j=1; j<n; j++)
         if (!converged(i,j){
            continue = TRUE;
            break;
         }
      }
} while (continue == TRUE)
```

..Case studies: Heat distribution

Parallel code: (version with a fixed number of iterations and a process $P_{i,j}$ for each point:)

```
for (iter=0; iter<limit; iter++)
     h[i][j] = 0.25*(w+e+n+s);
     send(&h[i][j], $P_{i-1,j}$); /* nonblocking sends */
     send(&h[i][j], $P_{i+1,j}$);
     send(&h[i][j], $P_{i,j-1}$);
     send(&h[i][j], $P_{i,j+1}$);
     recv(&w, $P_{i-1,j}$); /* blocking receivers */
     recv(&e, $P_{i+1,j}$);
     recv(&n, $P_{i,j-1}$);
     recv(&s, $P_{i,j+1}$);
}
```

Notice: It is important to use nonblocking sends, *otherwise the processes deadlock. On the other hand, some blocking routines (as our* blocking receives*) are also needed - here, a local synchronization technique is used.*

..Case studies: Heat distribution

Some care is needed for *processes operating at the edges*:

- one may allocate processes for the edges which simply send the data; e.g., for $P_{0,j}$

```
    for (iter=0; iter<limit; iter++)
        send(&h[0][j],$P_{1,j}$)
```

- other possibility will be to remove `send/recv` statements form the code of processes acting near the border, e.g.,

..Case studies: Heat distribution

```
 ⋮
if (i != 1) send(&h[i][j],$P_{i-1,j}$);
if (i != n-1) send(&h[i][j], $P_{i+1,j}$);
if (j != 1) send(&h[i][j], $P_{i,j-1}$);
if (j != n-1) send(&h[i][j], $P_{i,j+1}$);
if (i != 1) recv(&w, $P_{i-1,j}$);
if (i != n-1) recv(&e, $P_{i+1,j}$);
if (j != 1) recv(&n, $P_{i,j-1}$);
if (j != n-1) recv(&s, $P_{i,j+1}$);
⋮
```

..Case studies: Heat distribution

Partitioning: As the number of points is usually large, one has to allocate more points to a process. Natural partitions are into *square blocks* or *strips*.

- Block partition: Communication time is $t_{commsq} = 8(t_{startup} + \sqrt{(n/p)}t_{data})$
- Strip partition: Communication time is $t_{commcol} = 4(t_{startup} + \sqrt{n}t_{data})$.

..Case studies: Heat distribution

Communication time: Strips are better for large startup time; for small startup time blocks are better. Indeed:

$$\text{strips better than blocks}$$
$$\text{iff}$$
$$t_{commcol} < t_{commsq}$$
$$\text{iff}$$
$$4(t_{startup} + \sqrt{n}t_{data}) < 8(t_{startup} + \sqrt{(n/p)}t_{data})$$
$$\text{iff}$$
$$t_{startup} > \sqrt{n}(1 - \frac{2}{\sqrt{p}})t_{data}$$

..Case studies: Heat distribution

Implementation details:

- a good technique is to use an additional row of points (*ghost points*) at each edge of the area of a process that hold the values from the adjacent edge of a neighbor process
- code with ghost points (for row strips; $m = \sqrt{n}$ and $m1 = m/p$)

```
for (i=1; i<m1; i++)
     for (j=1; j<m; 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<m1; i++)
     for (j=1; j<m; j++)
          h[i][j] = g[i][j];
send(&g[1][1],&m, $P_{i-1}$);
send(&g[1][m],&m, $P_{i+1}$);
recv(&h[1][0],&m, $P_{i-1}$);
recv(&h[1][m+1],&m, $P_{i+1}$);
```

..Case studies: Heat distribution

Unsafe send/receive:

- If *all* processes *first send, then receive* the amount of buffering may be large. If there is *no storage available*, then locally blocking `send()` could become as a synchronous `send()` and *deadlock may occur*.

- A *solution* to this problem is to alternate `send/recv` routines. E.g., in the case of row strips partitions one may use:

..Case studies: Heat distribution

```
if(myid % 2) == 0){
     send(&g[1][1],&m,$P_{i-1}$);
     recv(&h[1][0],&m,$P_{i-1}$);
     send(&g[1][m],&m,$P_{i+1}$);
     recv(&h[1][m+1],&m,$P_{i+1}$);
} else {
     recv(&h[1][0],&m,$P_{i-1}$);
     send(&g[1][1],&m,$P_{i-1}$);
     recv(&h[1][m+1],&m,$P_{i+1}$);
     send(&g[1][m],&m,$P_{i+1}$);
}
```

..Case studies: Heat distribution

Alternatives for safe communication:

- *combine send and receive* in the same routine `MPI_Sendrecv()` (which is guaranteed not to deadlock)
- use `MPI_Bsend()` where the user *provides explicit buffering* storage
- use nonblocking routines like `MPI_Isend()` and `MPI_Irecv()` - the *routines return imediately* and one has to use separate routines to test if the communication was successful

Notice: MPI routines used for successful communication testing include:
`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, `MPI_Testany()`.

Cellular automata

Cellular automata represent an interesting "synchronous" formal model where:

- the problem space is divided into *cells*
- each cell can be in one of a finite number of *states*
- cells are affected by *their neighbors* according to certain *rules*
- all cells are affected simultaneously, leading to an *evolution* from one "generation" to another

..Cellular automata

Conway Game of life: Rules

- every organism with *two or three* neighbors *survives* for the next generation
- every organism with *four or more* neighbors *dies* from over-population
- every organism with *one neighbor or none dies* from isolation
- an empty cell adjacent to *exactly three* occupied neighbors will *give birth* to an organism

Many applications of celullar automata in: computer science, physics, biology, etc.