CSC4005 – Distributed and Parallel Computing

Prof. Yeh-Ching Chung

School of Data Science

Chinese University of Hong Kong, Shenzhen

Outline

- Introduction to Parallel Computers
- Message Passing Computing and Programming
- Multithreaded Programming
- CUDA Programming
- OpenMP Programming
- Embarrassingly Parallel Computations
- Partitioning and Divide–and–Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Load Balancing and Termination Detection
- Sorting Algorithms

Synchronous Computations (1)

all process are synchronous at every time poiny

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).

Synchronous Computations (2)

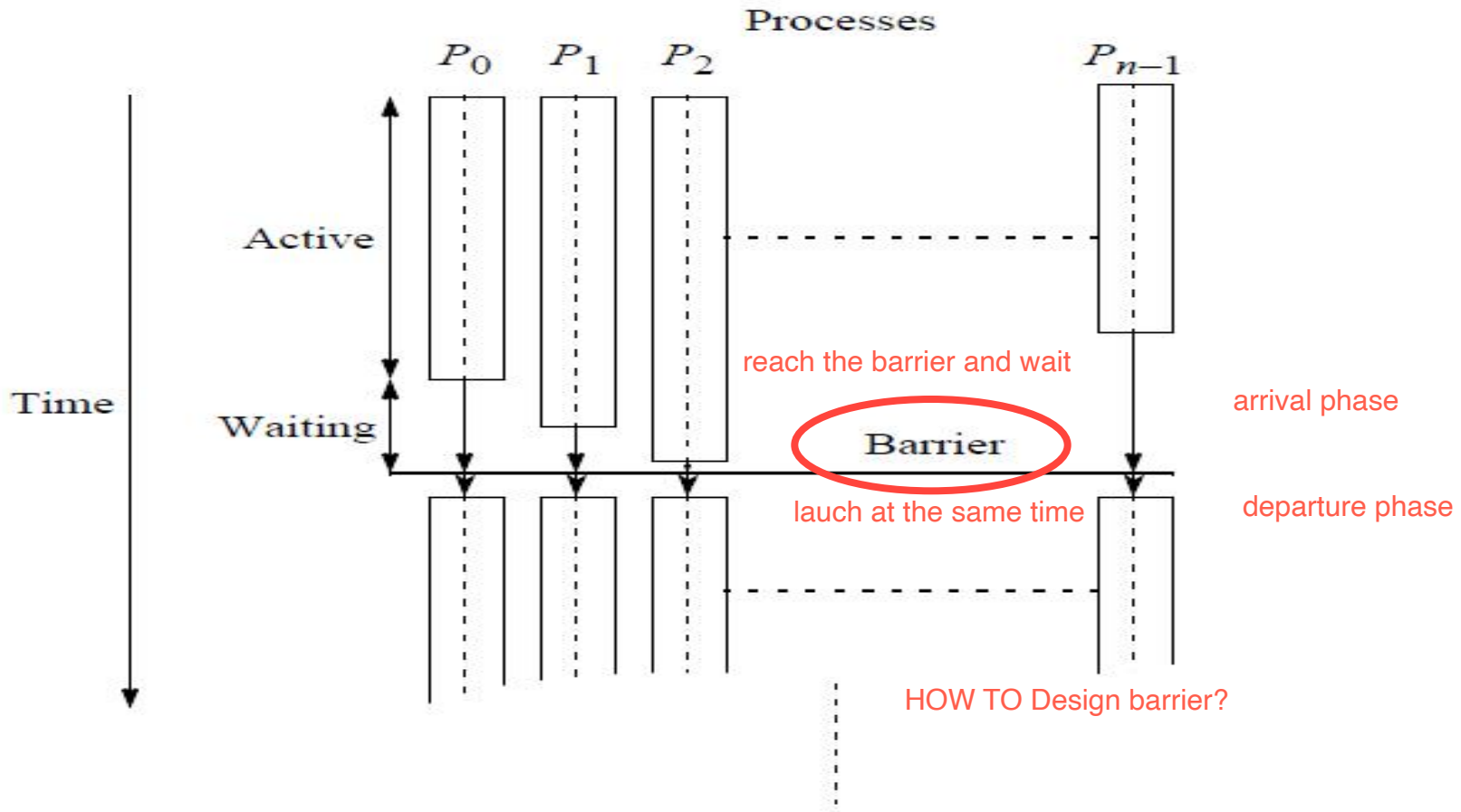


Figure 6.1 Processes reaching the barrier at different times.

Synchronous Computations (3)

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

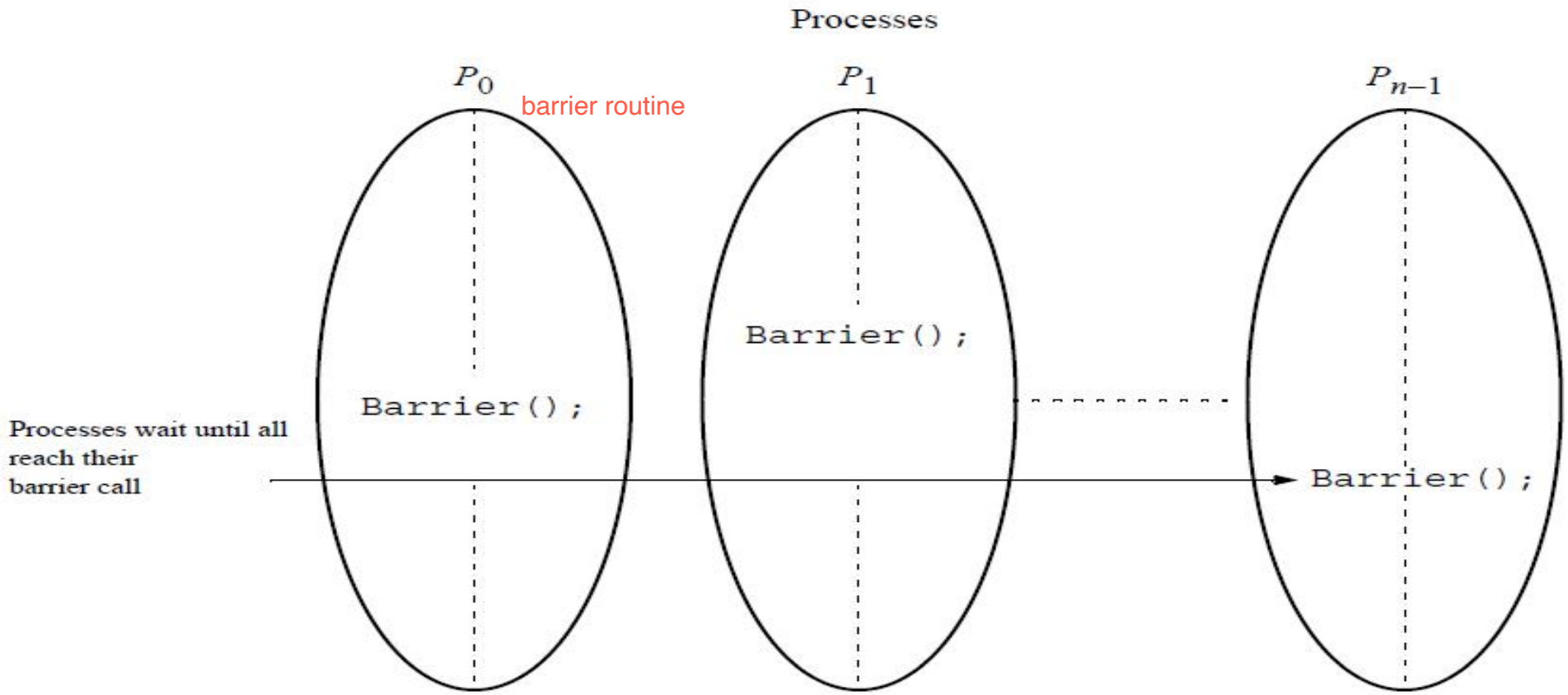


Figure 6.2 Library call barriers.

Synchronous Computations (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.

PVM

`pvm_barrier()` - similar barrier routine used with a named group of processes.

PVM has the unusual feature of specifying the number of processes that must reach the barrier to release the processes.

Synchronous Computations (5)

Implementation

Centralized counter implementation (sometimes called a *linear barrier*):

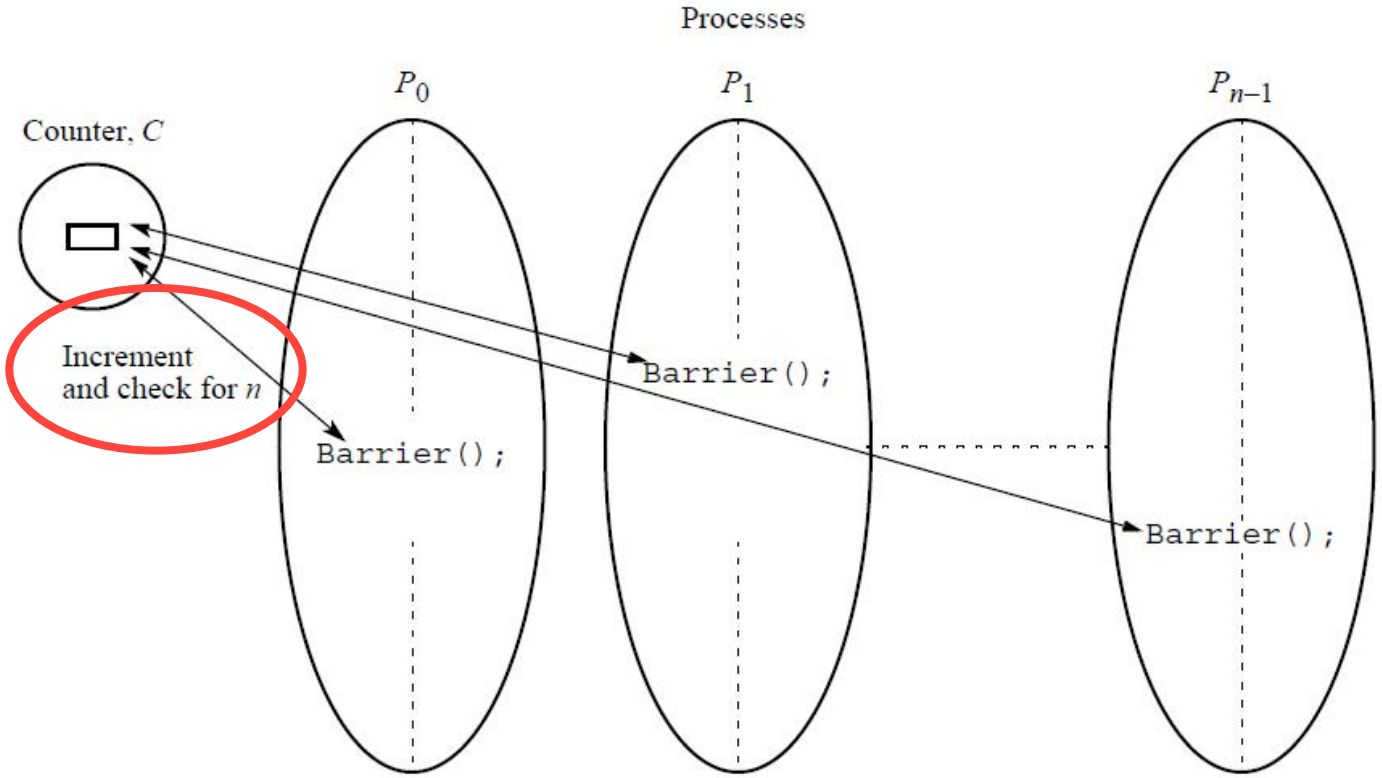


Figure 6.3 Barrier using a centralized counter.

Synchronous Computations (6)

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 of a barrier must take into account that a barrier might be used more than once in a process. 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-above handles this scenario.

Synchronous Computations (7)

Example code:

Master:

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

Slave processes:

```
send(P_master);
recv(P_master);
```

Synchronous Computations (8)

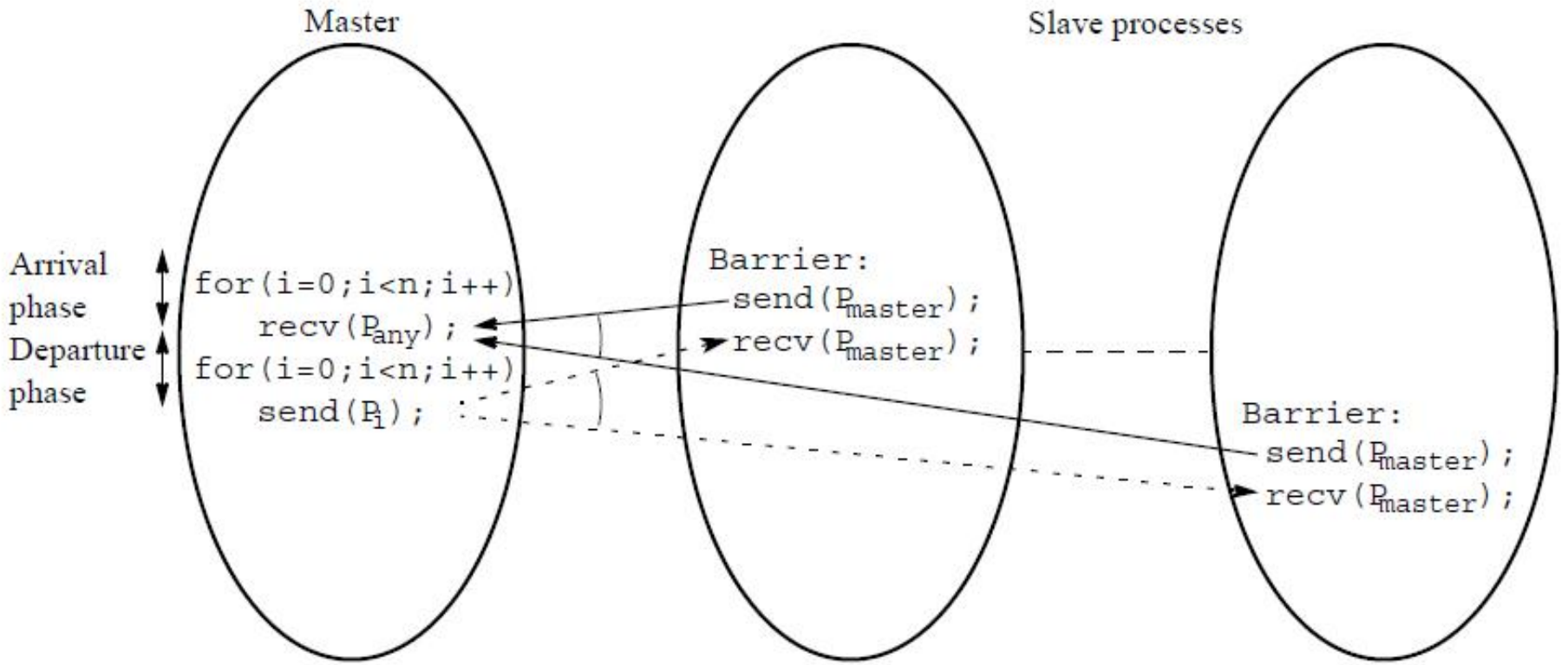


Figure 6.4 Barrier implementation in a message-passing system.

Synchronous Computations (9)

Tree Implementation

More efficient. Suppose there are eight processes, P_0, P_1, P_2, P_3, P_4, P_5, P_6, and 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 and P_3 have reached their barrier)
P_6 sends message to P_4; (P_6 and P_7 have reached their barrier)

Third stage: P_4 sends message to P_0; (P_4, P_5, P_6, and P_7 have reached their barrier)
P_0 terminates arrival phase; (when P_0 reaches barrier and has received message from P_4)

Release with a reverse tree construction.

Synchronous Computations (10)

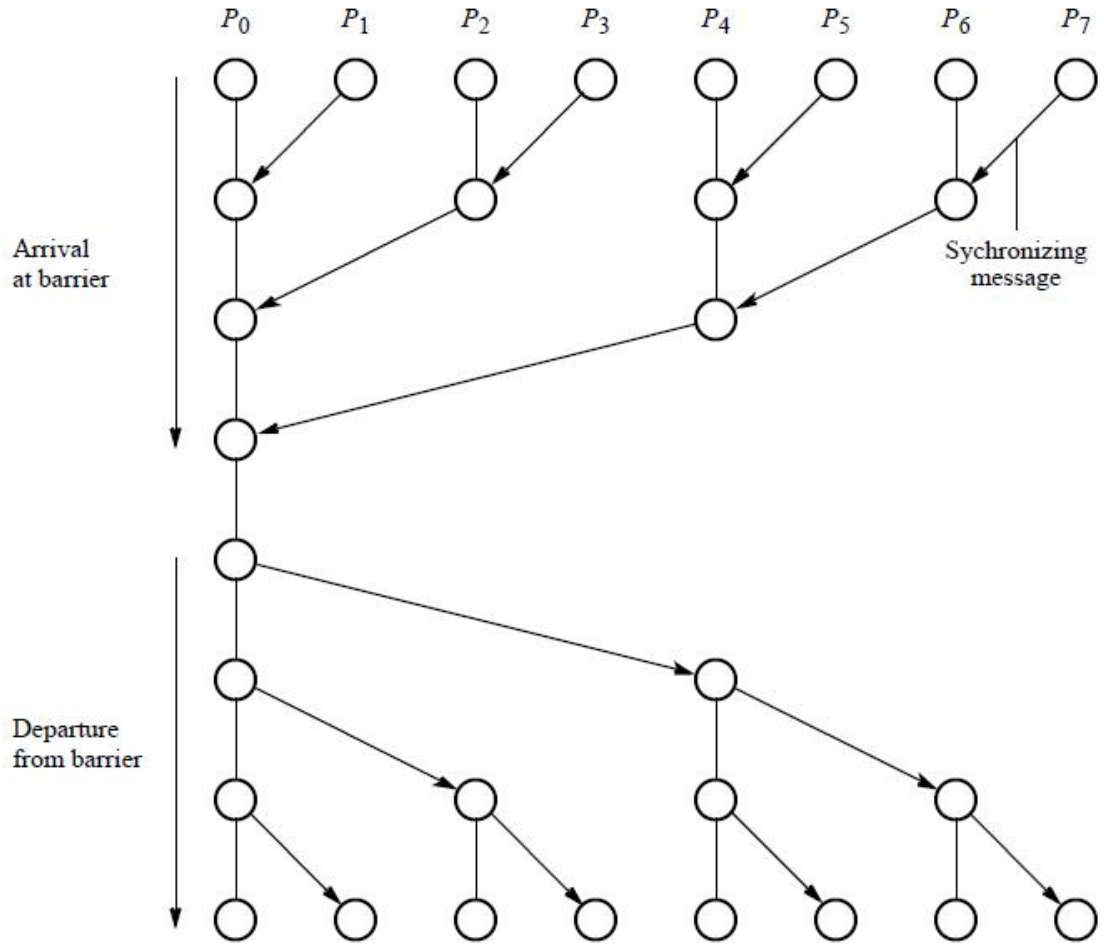


Figure 6.5 Tree barrier.

Synchronous Computations (11)

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

Nlog(N)

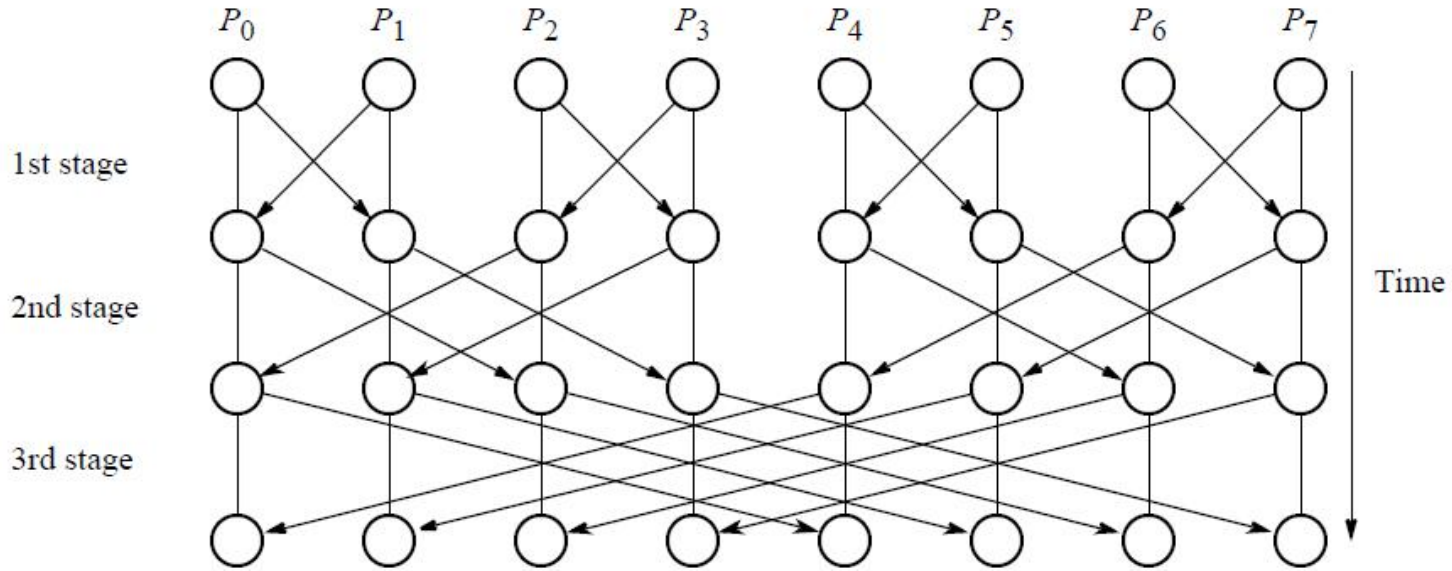


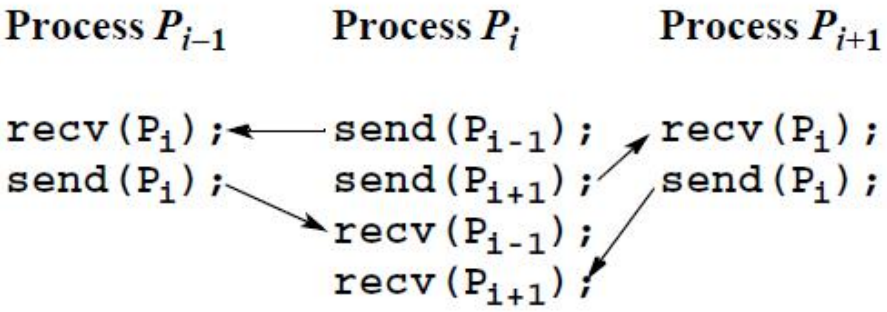
Figure 6.6 Butterfly construction.

Synchronous Computations (12)

Local Synchronization

Example

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



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

Synchronous Computations (13)

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.

Synchronous Computations (14)

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.

Synchronous Computations (15)

Combined deadlock-free blocking `sendrecv()` routines

MPI provides routine `MPI_Sendrecv()` and `MPI_Sendrecv_replace()`.

blocking send

Example

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

Synchronous Computations (16)

function

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.

Synchronous Computations (17)

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` ($0 < i \leq n$).

Synchronous Computations (18)

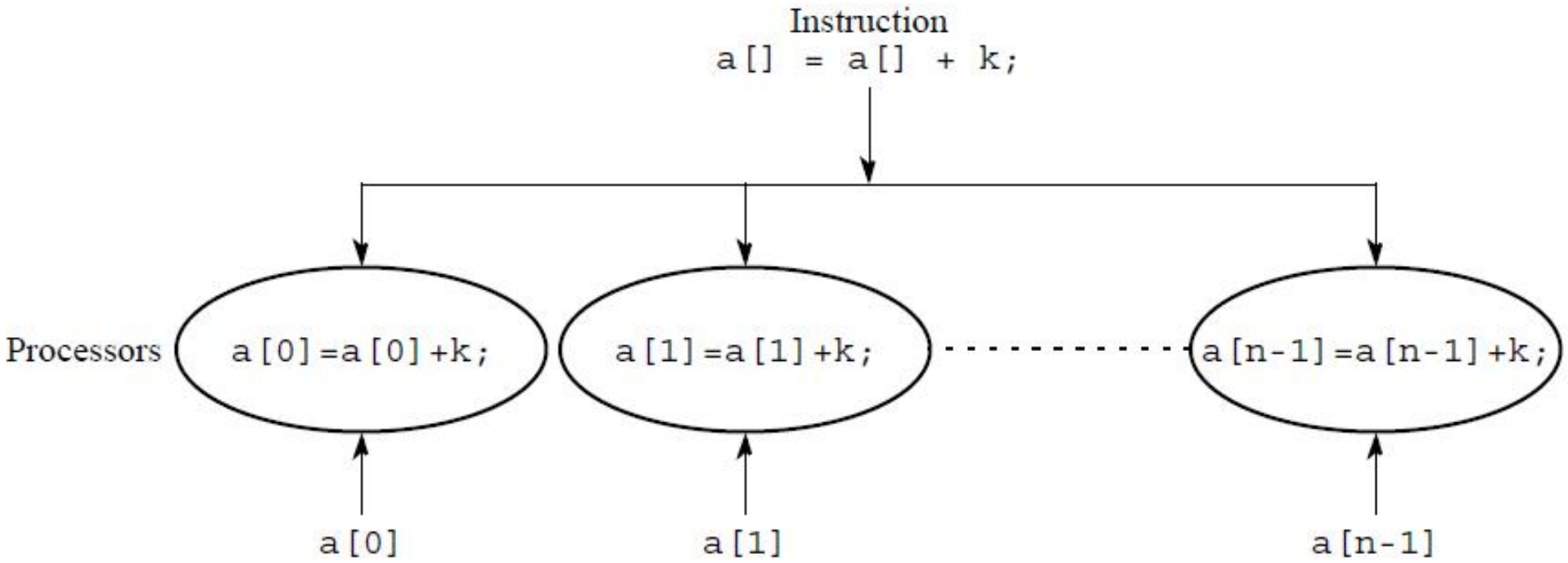


Figure 6.7 Data parallel computation.

Forall Construction (1)

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.

Forall Construction (2)

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 $n - 1$.

Prefix Sum Problem (1)

Given a list of numbers, $x_0, \ldots, x_{n-1}$, 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.

The sequential code for the prefix problem could 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.

Prefix Sum Problem (2)

Data parallel method of adding all partial sums of 16 numbers

Sequential code

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

Parallel code

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

Prefix Sum Problem (3)

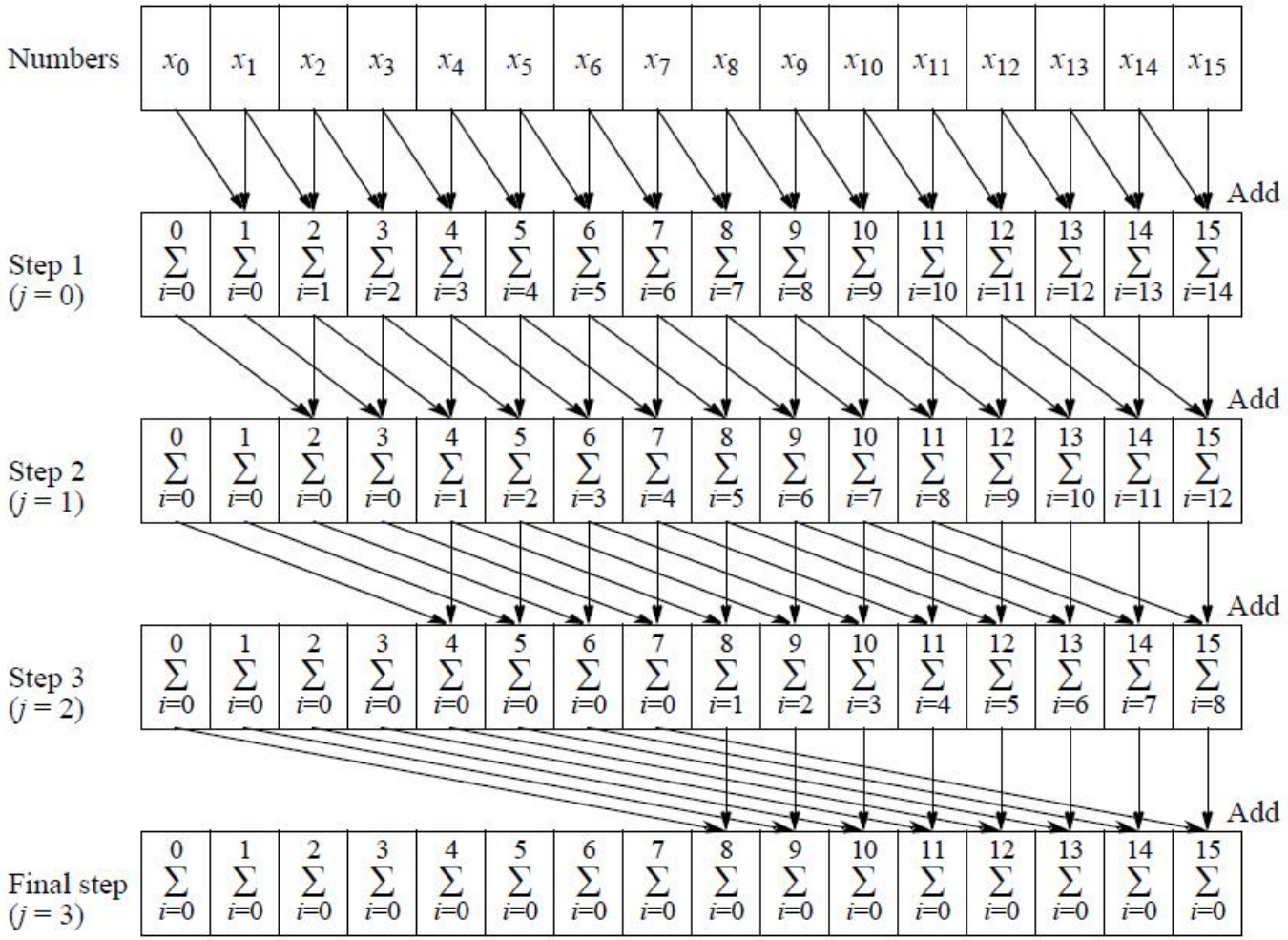


Figure 6.8 Data parallel prefix sum operation.

log（n) steps

Synchronous Iteration (Synchronous Parallelism)

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

The `forall` construct could be used to specify the parallel bodies of the synchronous iteration:

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

or:

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

Solving a System of Linear Equations by Iteration (1)

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

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

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

Solving a System of Linear Equations by Iteration (2)

One way to solve these equations for the unknowns is by iteration. By rearranging the ith equation:

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

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.

Solving a System of Linear Equations by Iteration (3)

Jacobi Iteration

Iterative method described called a *Jacobi iteration* – all values of x are updated together.

It 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.

Solving a System of Linear Equations by Iteration (4)

Termination

A simple, common approach is to compare values computed in each iteration to the values obtained from the previous iteration, and then to terminate the computation in the tth iteration when all values are within a 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.

Solving a System of Linear Equations by Iteration (5)

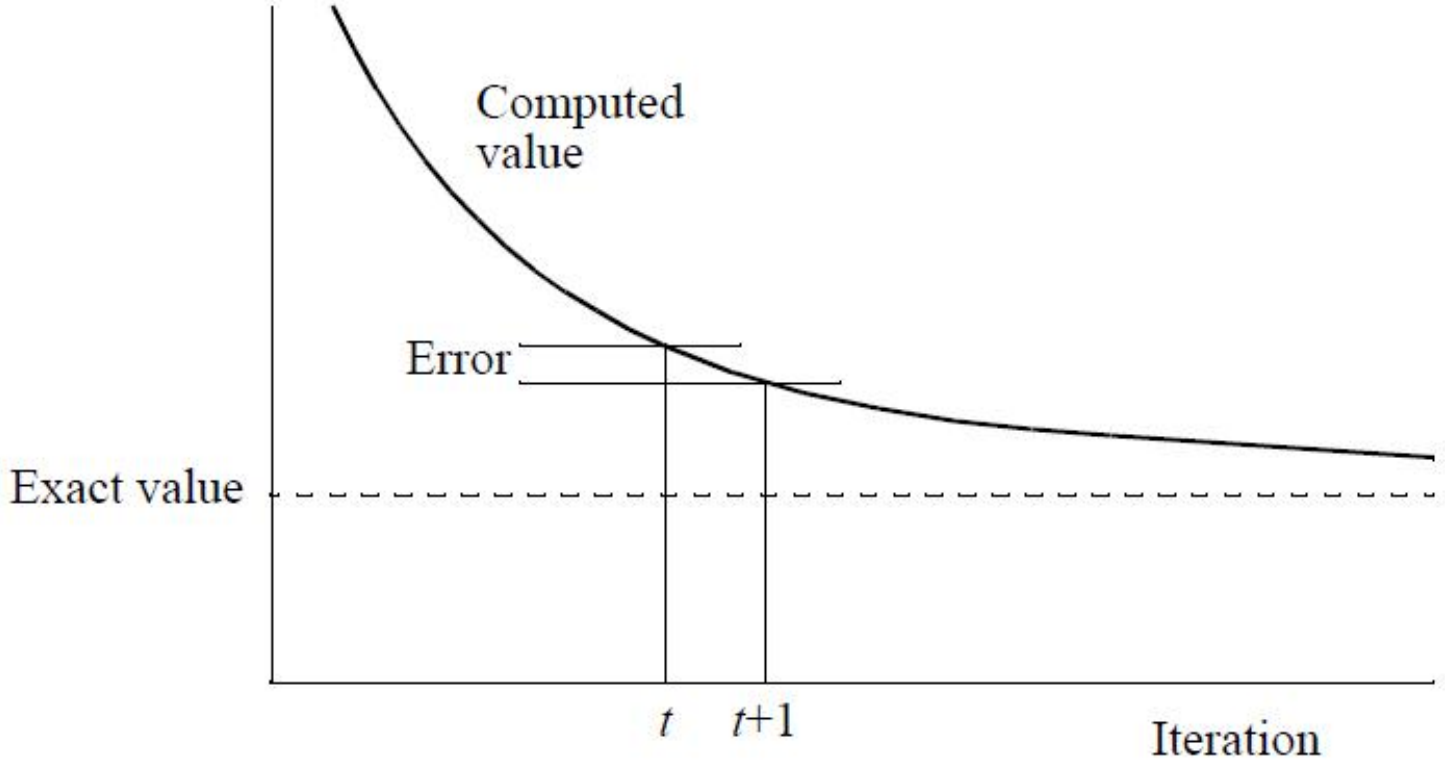


Figure 6.8 Convergence rate.

Solving a System of Linear Equations by Iteration (6)

Sequential Code

Given arrays `a[][]` and `b[]` holding constants in the equations, `x[]` holding unknowns, and fixed number of iterations:

```
for (i = 0; i < n; i++)
    x[i] = b[i];                        /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 0; i < n; i++) {    /* for each unknown */
      sum = 0;
      for (j = 0; j < n; j++)  /* summation of a[][]x[] */
        if (i != j) sum = sum + a[i][j] * x[j];
      new_x[i] = (b[i] - sum) / a[i][i];/*compute unknown*/
    }
    for (i = 0; i < n; i++)
      x[i] = new_x[i];                  /* update values */
}
```

Solving a System of Linear Equations by Iteration (7)

Slight more efficient sequential code:

```
for (i = 0; i < n; i++)
    x[i] = b[i];                          /*initialize unknowns*/
for (iteration = 0; iteration < limit; iteration++) {
    for (i = 0; i < n; i++) {             /* for each unknown */
      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*/
    }
    for (i = 0; i < n; i++) x[i] = new_x[i];/*update */
}
```

Solving a System of Linear Equations by Iteration (8)

Parallel Code

Process P_i could be of the form

```
x[i] = b[i];                          /*initialize unknown*/
for (iteration = 0; iteration < limit; iteration++) {
    sum = -a[i][i] * x[i];
    for (j = 0; j < n; j++)           /* compute summation */
      sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];/* compute unknown */
    broadcast_receive(&new_x[i]);     /* broadcast value */
    global_barrier();                 /* wait for all processes */
}
```

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

An alternative simple solution is to return to basic `send()`s and `recv()`s.

Solving a System of Linear Equations by Iteration (9)

Allgather

Broadcast and gather values in one composite construction.

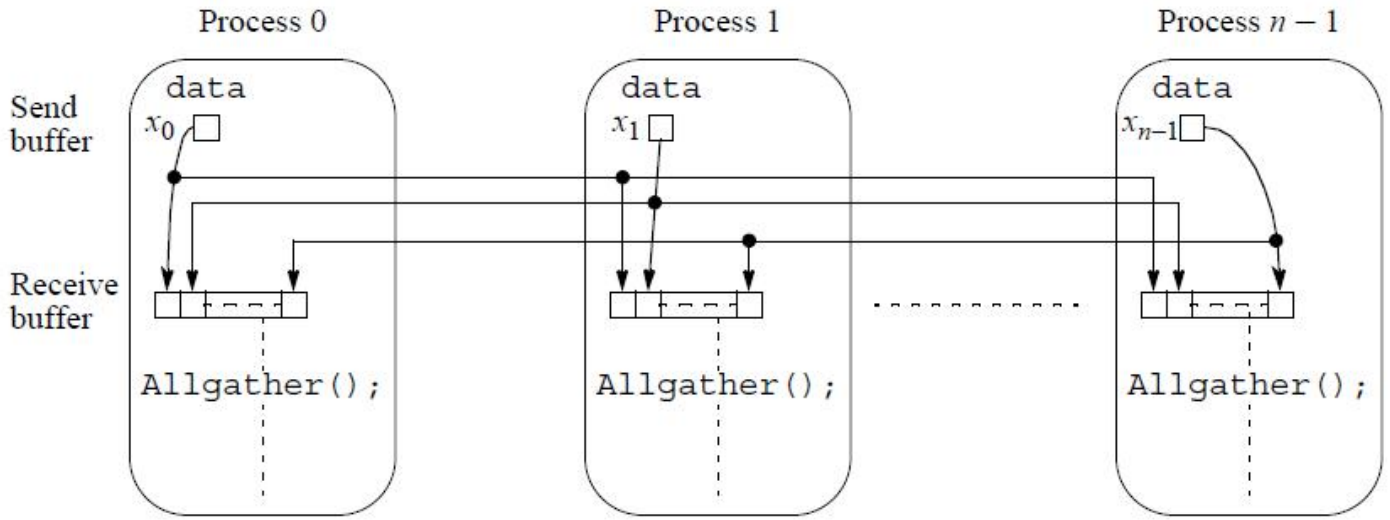


Figure 6.9 Allgather operation.

Solving a System of Linear Equations by Iteration (10)

To iterate until the approximations are sufficiently close:

```
x[i] = b[i];                          /*initialize unknown*/
iteration = 0;
do {
    iteration++;
    sum = -a[i][i] * x[i];
    for (j = 1; j < n; j++)   /* compute summation */
      sum = sum + a[i][j] * x[j];
    new_x[i] = (b[i] - sum) / a[i][i];/* compute unknown */
    broadcast_receive(&new_x[i]);/*broadcast value & wait */
} while (tolerance() && (iteration < limit));
```

where `tolerance()` returns **FALSE** if ready to terminate; otherwise it returns **TRUE**.

Solving a System of Linear Equations by Iteration (11)

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. In the problem at hand, each process can be responsible for computing a group of unknowns.

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 $x_0, x_p, x_{2p}, \ldots, x_{((n/p)-1)p}$, processor P_1 is allocated $x_1, x_{p+1}, x_{2p+1}, \ldots, x_{((n/p)-1)p+1}$, 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).

Solving a System of Linear Equations by Iteration (12)

Analysis

Suppose there are n equations and p processors.

A processor operates upon n/p unknowns.

Suppose there are τ iterations.

One iteration has a computational phase and a broadcast communication phase.

Computation.

$$t_{\text{comp}} = n/p(2n + 4)\tau$$

Communication.

$$t_{\text{comm}} = p(t_{\text{startup}} + (n/p)t_{\text{data}})\tau = (pt_{\text{startup}} + nt_{\text{data}})\tau$$

Overall.

$$t_p = (n/p(2n + 4) + pt_{\text{startup}} + nt_{\text{data}})\tau$$

The resulting total execution time has a minimum value.

Solving a System of Linear Equations by Iteration (13)

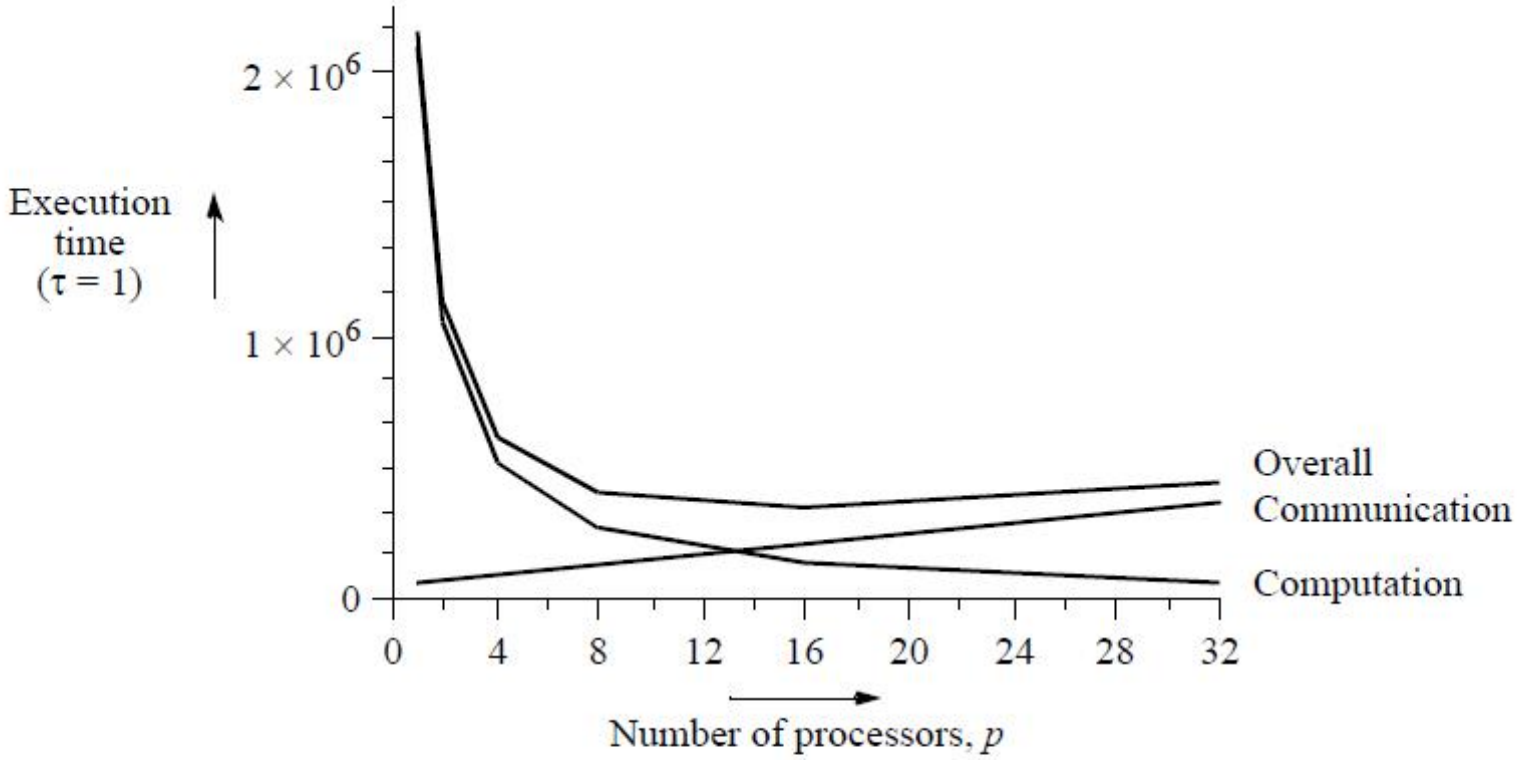


Figure 6.10 Effects of computation and communication in Jacobi iteration.

Heat Distribution Problem (1)

A square metal sheet has known temperatures along each of its edges. Find the temperature distribution.

Dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points.

Convenient to describe the edges by points adjacent to the interior points. The interior points of $h_{i,j}$ are where $0 < i < n$, $0 < j < n$ [$(n-1) \times (n-1)$ interior 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 of a point is less than some very small prescribed amount.

Heat Distribution Problem (2)

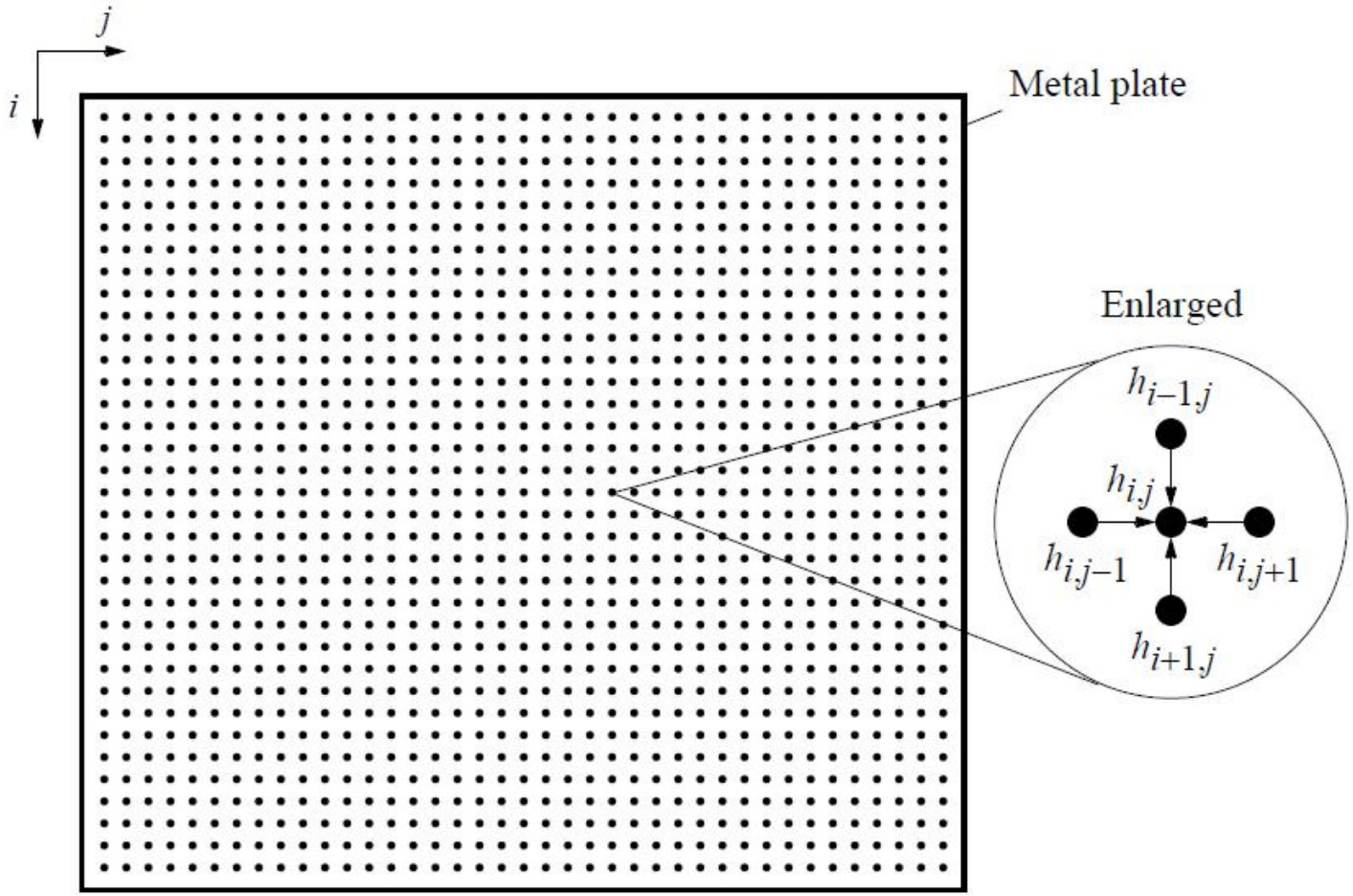


Figure 6.11 Heat distribution problem.

Heat Distribution Problem (3)

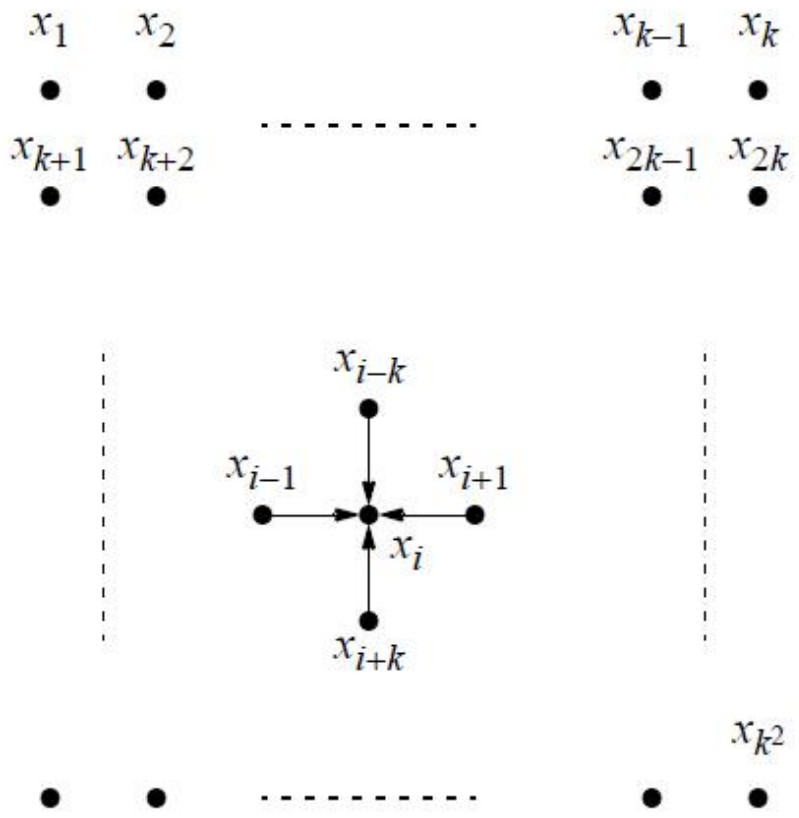


Figure 6.12 Natural ordering of heat distribution problem.

Heat Distribution Problem (4)

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];
}
```

Heat Distribution Problem (5)

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);
```

Heat Distribution Problem (6)

Parallel Code

Version with a fixed number of iterations, process $P_{i,j}$ (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 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. Each process synchronized with its four neighbors by the `recv()`s.

Heat Distribution Problem (7)

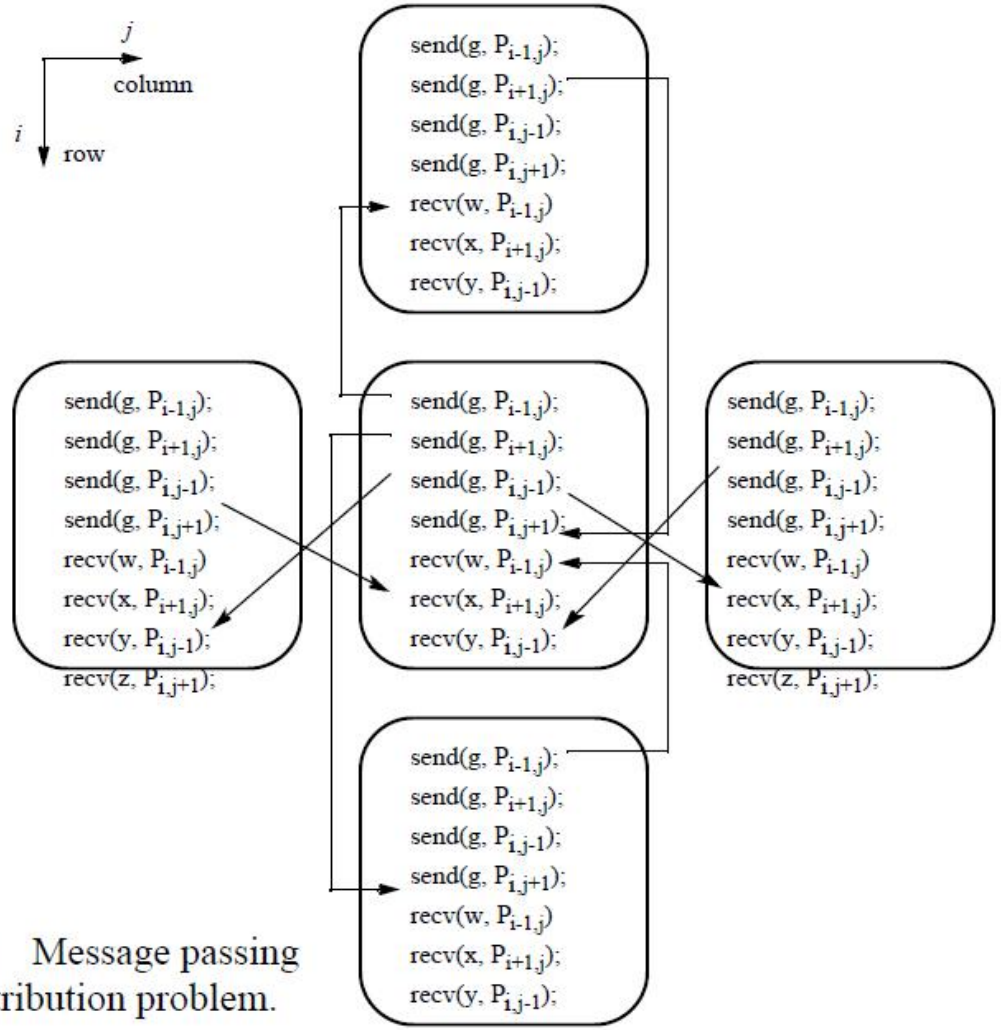


Figure 6.13 Message passing for heat distribution problem.

Heat Distribution Problem (8)

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);
```

Heat Distribution Problem (9)

To handle the processes operating at the edges:

```
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);
```

Heat Distribution Problem (10)

Partitioning

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

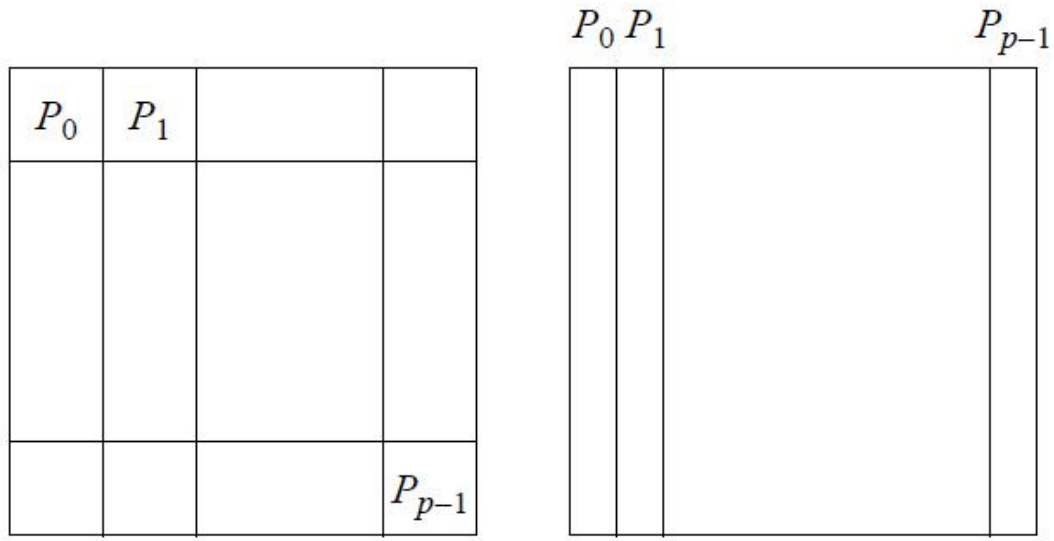


Figure 6.14 Partitioning heat distribution problem.

Heat Distribution Problem (11)

Block partition:

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

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

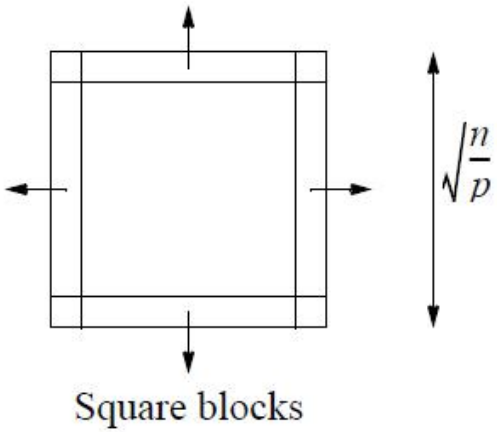


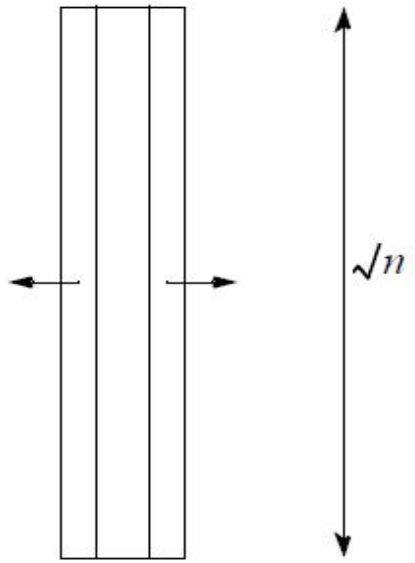
Figure 6.15 Communication consequences of partitioning.

Heat Distribution Problem (12)

Strip partition

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

$$t_{commcol} = 4(t_{startup} + \sqrt{n}t_{data})$$



Strips

Heat Distribution Problem (13)

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

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

or

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

$(p \geq 9)$.

Heat Distribution Problem (14)

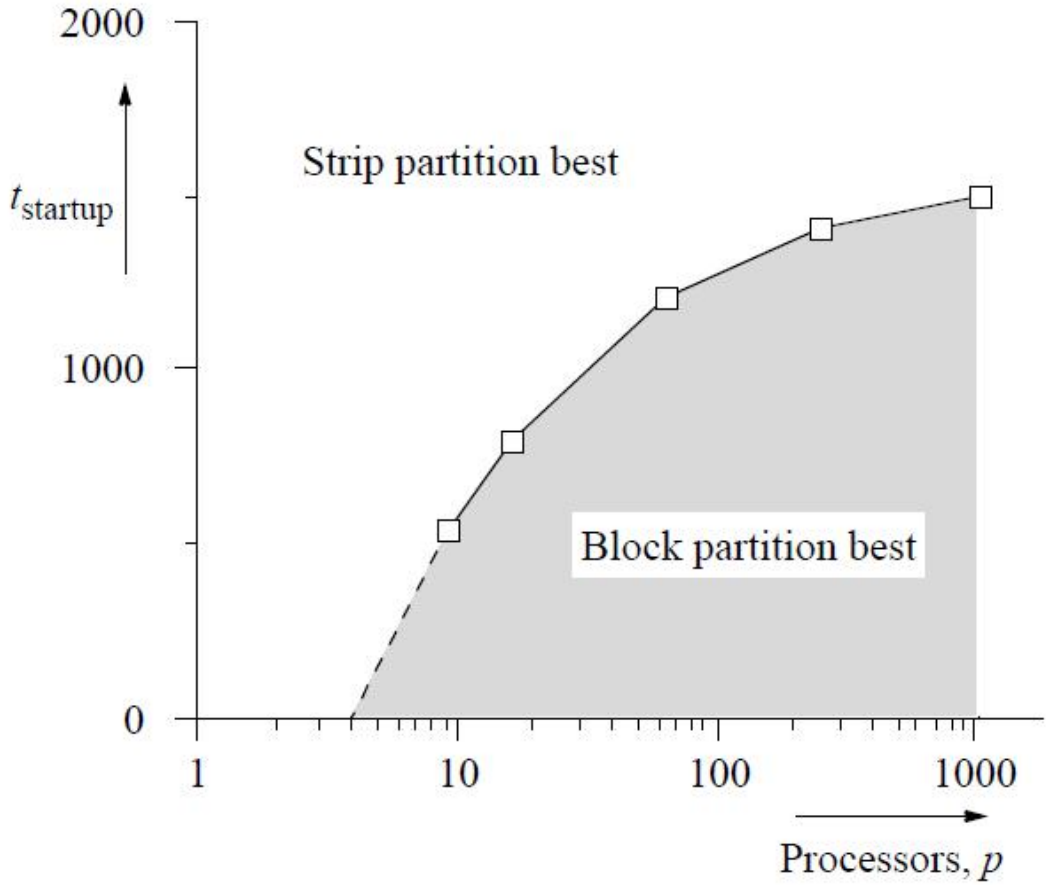


Figure 6.16 Startup times for block and strip partitions.

Heat Distribution Problem (15)

Ghost Points

An additional row of points at each edgethat hold the values from the adjacent edge.

Each array of points is increased to accommodate the ghost rows.

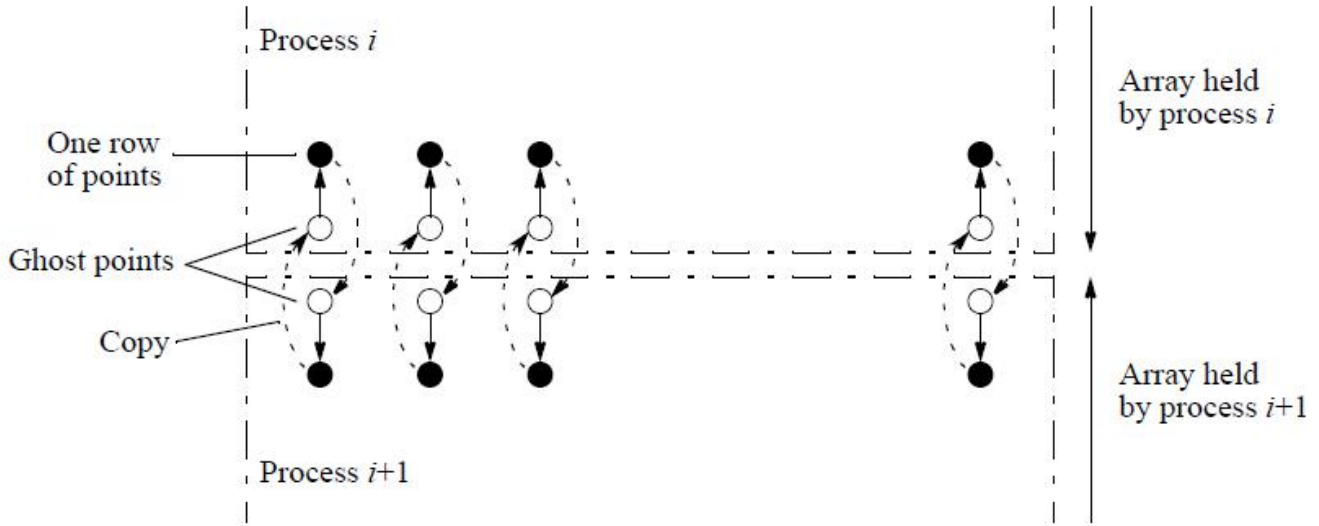


Figure 6.17 Configurating array into contiguous rows for each process, with ghost points.

Heat Distribution Problem (16)

Safety and Deadlock

When all processes send their messages first and then receive all of their messages, as in all the code so far, is described as "unsafe" in the MPI literature because it relies upon buffering in the `send()`s. The amount of buffering is not specified in MPI.

If a send routine has insufficient storage available when it is called, the implementation should be such to delay the routine from returning until storage becomes available or until the message can be sent without buffering.

Hence, the 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.

Heat Distribution Problem (17)

Making the code safe

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

```
if ((myid % 2) == 0) {          /* even processes */
    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 {                        /* odd numbered processes */
    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});
}
```

Then even synchronous `send()`s would not cause deadlock. In fact, a good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

Heat Distribution Problem (18)

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()` — here the routine returns immediately, and a separate routine is used to establish whether the message has been received (`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`, `MPI_Testall()`, or `MPI_Testany()`)

A pseudocode segment using the third method is

```
isend(&g[1][1], &m, P_{i-1});
isend(&g[1,m], &m, P_{i+1});
irecv(&h[1][0], &m, P_{i-1});
irecv(&h[1][m+1], &m, P_{i+1});
waitall(4);
```

Wait routine becomes a barrier, waiting for all message-passing routines to complete.

Cellular Automata (1)

In this approach, the problem space is first 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, and all cells are affected simultaneously in a "generation."

The rules are reapplied in subsequent generations so that cells evolve, or change state, from generation to generation.

The most famous cellular automata is the "Game of Life" devised by John Horton Conway, a Cambridge mathematician, and published by Gardner (Gardner, 1967).

Cellular Automata (2)

The Game of Life

Board game; Board consists of a (theoretically infinite) two-dimensional array of cells. Each cell can hold one "organism" and has eight neighboring cells, including those diagonally adjacent. Initially, some of the cells are occupied in a pattern.

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."

Cellular Automata (3)

"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

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.

Cellular Automata (4)

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.

Similar examples: "foxes and rabbits" -The behavior of the rabbits is to move around happily whereas the behavior of the foxes is to eat any rabbits they come across.

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.