

COMP4300: Synchronous Computations/Barriers

Chapter 6: Wilkinson and Allen

Alistair Rendell

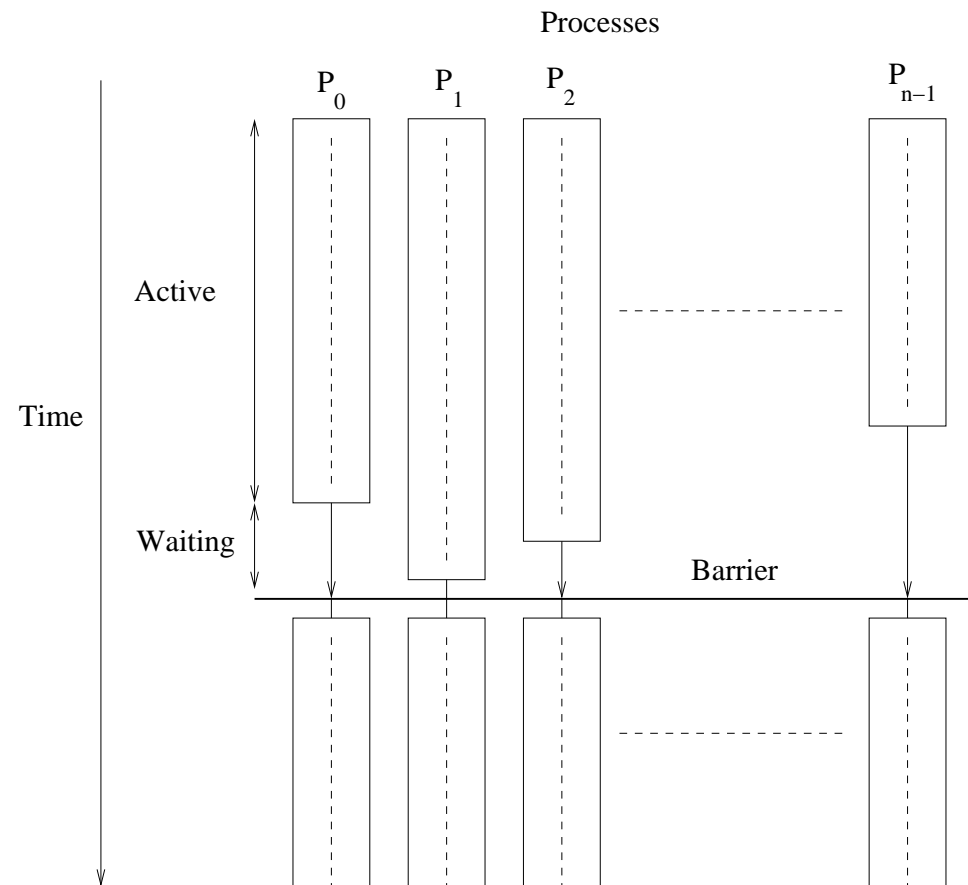
9.1 Barriers

- **Barrier:** a point at which all processes must wait until all other processes have reached that point

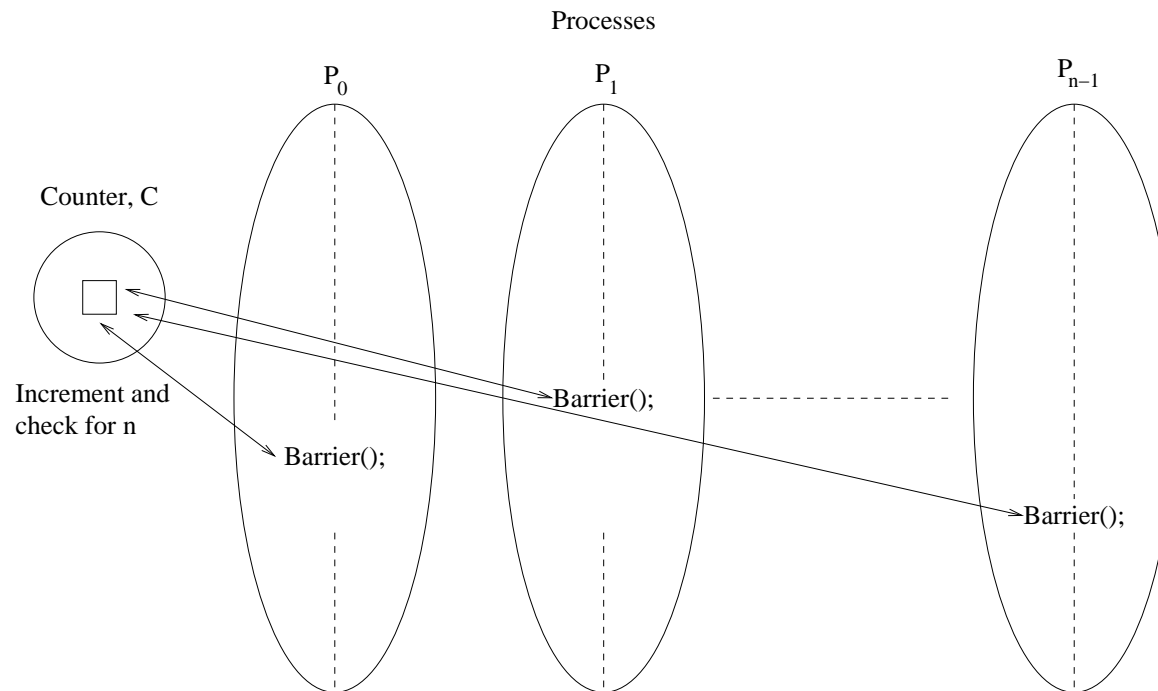
```
MPI_Barrier(MPI_Comm comm)
```

- **Mutual Exclusion:** a barrier that prevents other processes from entering the following region if another process is already in that region
 - Common in shared memory parallel programs
 - Necessary for some MPI-2 operations
- Both are possible sources of overhead

9.2 Barrier



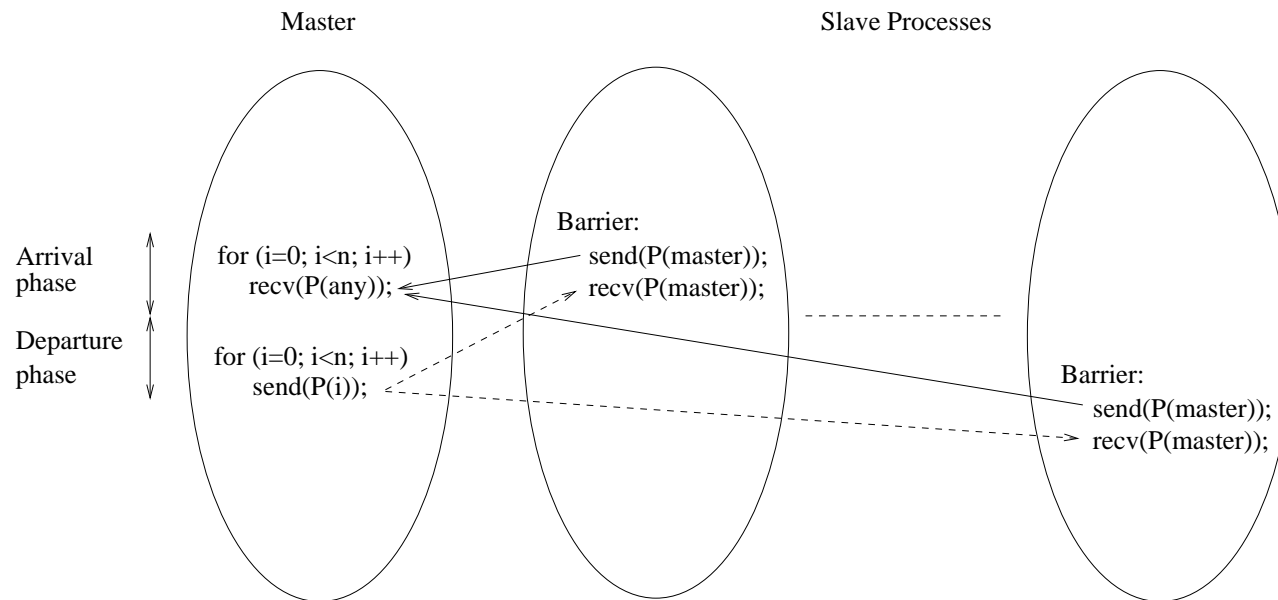
9.3 Counter-based or Linear Barriers



- One process counts the arrival of the other processes
 - When all processes have arrive they are each sent a release message

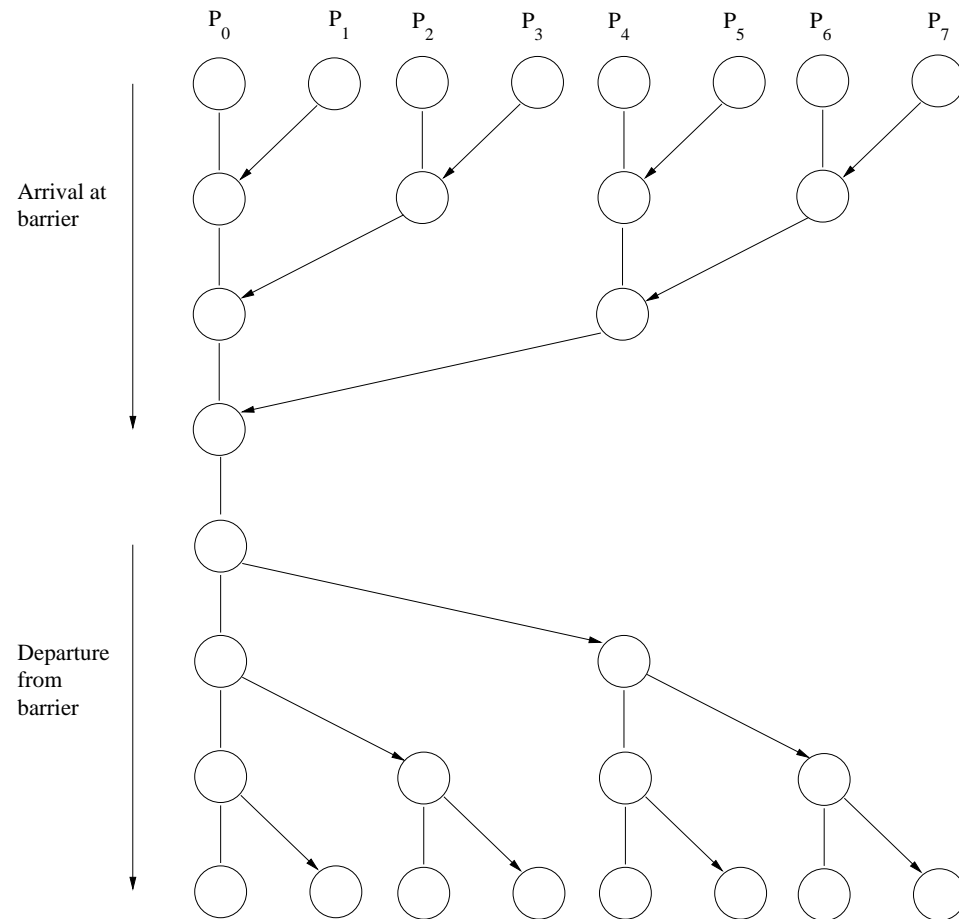
9.4 Implementation

- *Arrival phase*: process sends message to central counter
- *Departure phase*: process receives message from central counter
- Implementations must handle possible time delays
 - e.g. two barriers in quick succession



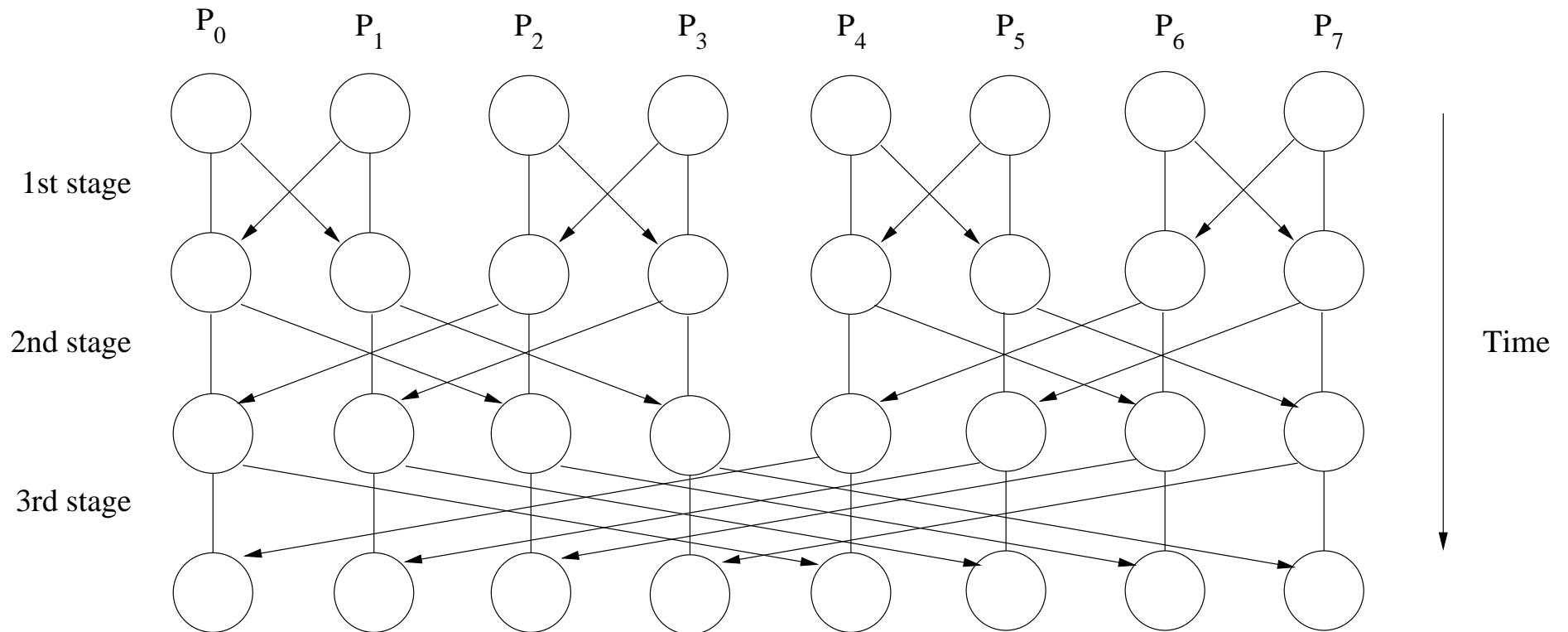
- Cost $O(n)$

9.5 Tree Based Barriers



- NOTE - broadcast does not ensure synchronization
- Cost $2 \log n$ or $O(\log n)$

9.6 Butterfly Barrier (Butterfly/Omega Network)



- Cost $2 \log n$ or $O(\log n)$

9.7 Degrees of Synchronization

- From fully - loosely synchronous
 - The more synchronous your computation the more potential overhead
- SIMD: synchronized at the instruction level
 - Provides ease of programming (one program)
 - Well suited for data decomposition
 - Applicable to many numerical problems
 - forall statement introduced to specify data parallel operations

```
forall (i=0; i<n; i++){  
    data parallel work  
}
```

9.8 Synchronous Example: Jacobi Iterations

- Jacobi iteration solves a system of linear equations iteratively

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

⋮

$$\begin{array}{l} a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 + \cdots \\ a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 + \cdots \\ a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 + \cdots \end{array} \qquad \begin{array}{l} a_{2,n-1}x_{n-1} = b_2 \\ a_{1,n-1}x_{n-1} = b_1 \\ a_{0,n-1}x_{n-1} = b_0 \end{array}$$

where there are n equations and n unknowns $(x_0, x_1, x_2, \cdots, x_{n-1})$

9.9 Jacobi Iterations

- Consider equation i as:

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

re-cast as

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

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

- Strategy: guess x then iterate and hope it converges!
 - Converges if diagonally dominant

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

- Terminate when

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

9.10 Sequential Jacobi Code

- Ignoring convergence testing

```
for (i = 0; i < n; i++) x[i] = b[i];

for (iter = 0; iter < max_iter; 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]
        }
        new_x[i] = (b[i] - sum)/ a[i][i];
    }
    for (i = 0; i < n; i++) x[i] = new_x[i];
}
```

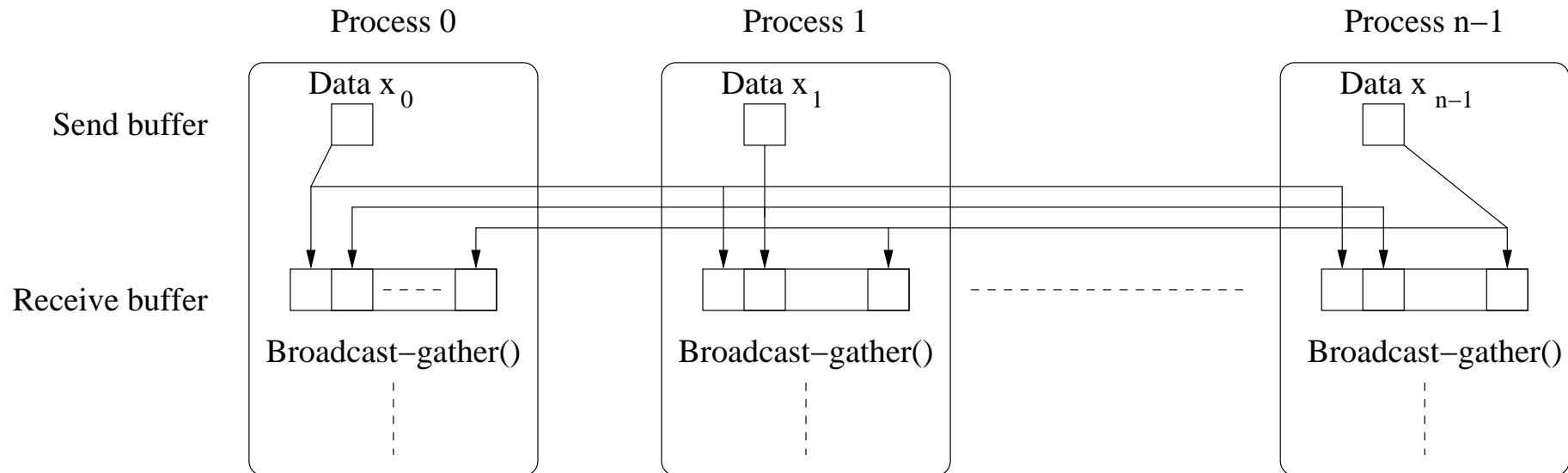
9.11 Parallel Jacobi Code

- Ignoring convergence testing and assuming parallelisation over n processes

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

- `broadcast_gather()` sends the local `new_x[i]` to all processes and collects their new values

9.12 Broadcast_gather



9.13 Partitioning

- Normally number of processes much less than number of data items
 - *Block partitioning*: allocate groups of consecutive unknowns to processes
 - *Cyclic partitioning*: allocate in a round robin fashion
- Analysis: τ iterations, n/p unknowns per process

- Computation - decreases with p

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

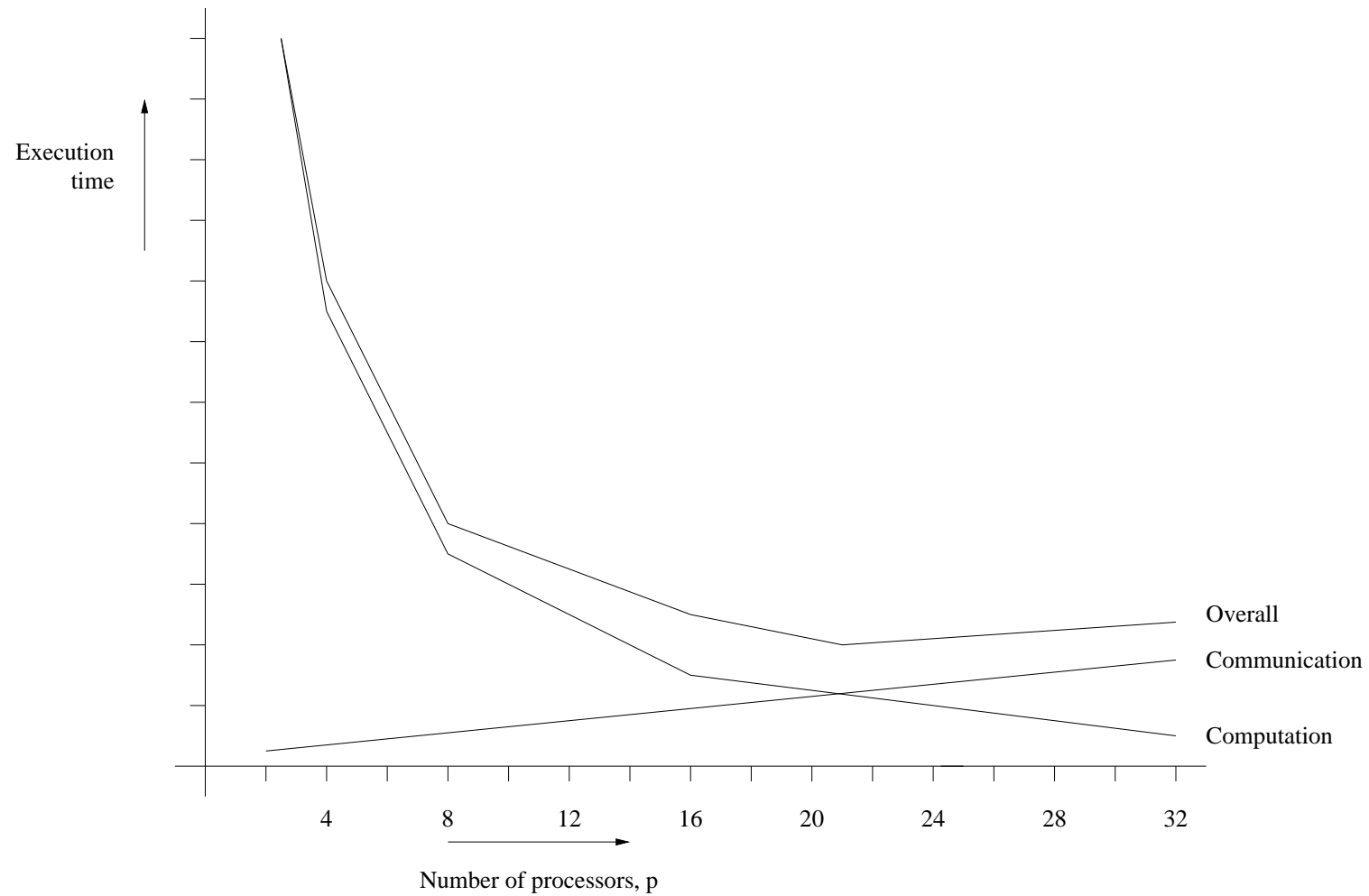
- Communication - increases with p

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

- Total - has an overall minimum

$$t_{tot} = ((2n + 4)n/p + pt_{stup} + nt_{data})\tau$$

9.14 Parallel Jacobi Iteration Time



9.15 Locally Synchronous Example: Heat Distribution Problem

Consider a metal sheet with known temperature along the sides but unknown temperatures in the middle - find temperature in the middle

- Finite difference approximation to the Laplace equation

$$\frac{\partial^2 T(x, y)}{\partial x^2} + \frac{\partial^2 T(x, y)}{\partial y^2} = 0$$

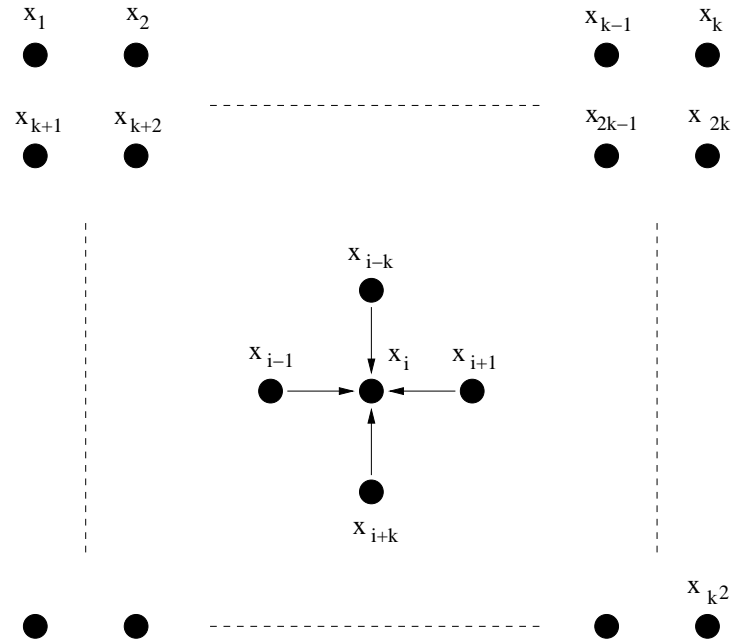
$$\frac{T(x + \delta x, y) - 2T(x, y) + T(x - \delta x, y)}{\delta x^2} + \frac{T(x, y + \delta y) - 2T(x, y) + T(x, y - \delta y)}{\delta y^2} = 0$$

- Assuming an even grid of $n \times n$ points (i.e. $\delta x = \delta y$) denoted as $h_{i,j}$
- Temperature at any point is an average of surrounding points

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

- Problem is very similar to “Game of Life”, ie what happens in a cell depends upon its neighbours

9.16 Array Ordering



- We will solve iteratively

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

- But may also be written as a system of linear equations

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

9.17 Heat Equation: Sequential Code

- Fixed number of iterations and square mesh
- Beware of what happens at edges!

```
for (iter = 0; iter < max_iter; iter++){
  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];
}
```

9.18 Heat Equation: Parallel Code

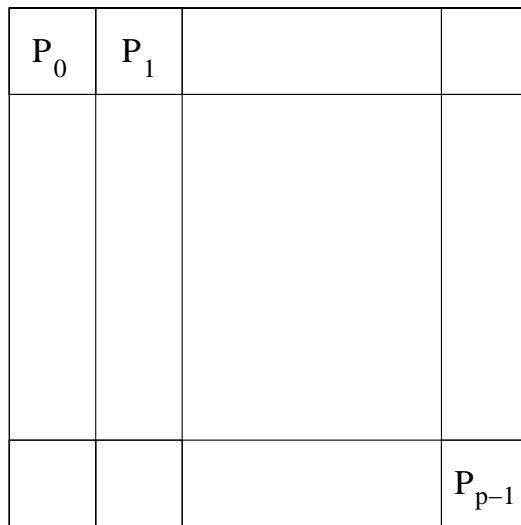
- One point per process
- Non blocking sends

```
for (iter = 0; iter < max_iter; iter++){  
    g = 0.25*(w + x + y + z);  
    send(&g, P(i-1,j));  
    send(&g, P(i+1,j));  
    send(&g, P(i,j-1));  
    send(&g, P(i,j+1));  
    recv(&g, P(i-1,j));  
    recv(&g, P(i+1,j));  
    recv(&g, P(i,j-1));  
    recv(&g, P(i,j+1));  
}
```

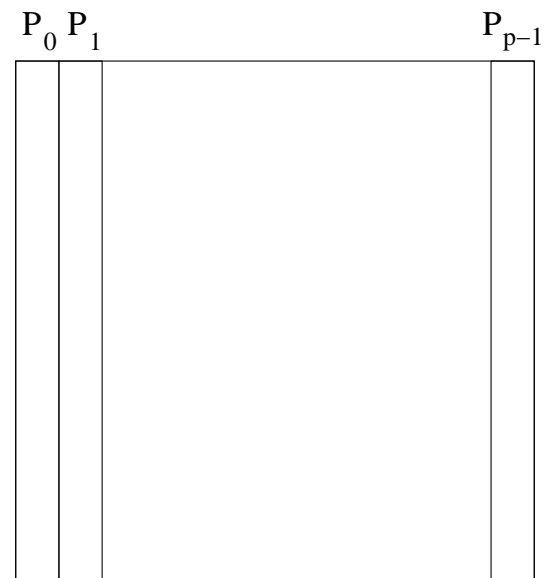
- Sends and receives provide a local barrier
 - Each process synchronizes with 4 others surrounding processes

9.19 Heat Equation: Partitioning

- Normally more than one point per process
- Option of either block or strip partitioning



Block Partitioning



Strip Partitioning

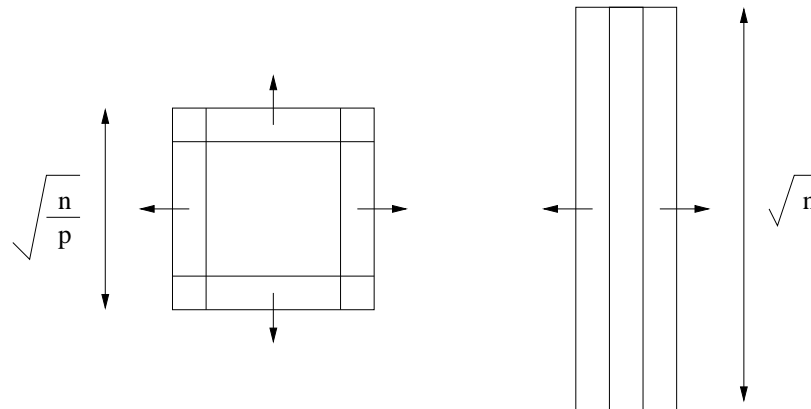
9.20 Block/Strip Communication Comparison

- Block: four edges exchanged (n data points, p processes)

$$t_{comm} = 8\left(t_{stup} + \sqrt{\frac{n}{p}}t_{data}\right)$$

- Strip: two edges exchanged

$$t_{comm} = 4\left(t_{stup} + \sqrt{np}t_{data}\right)$$



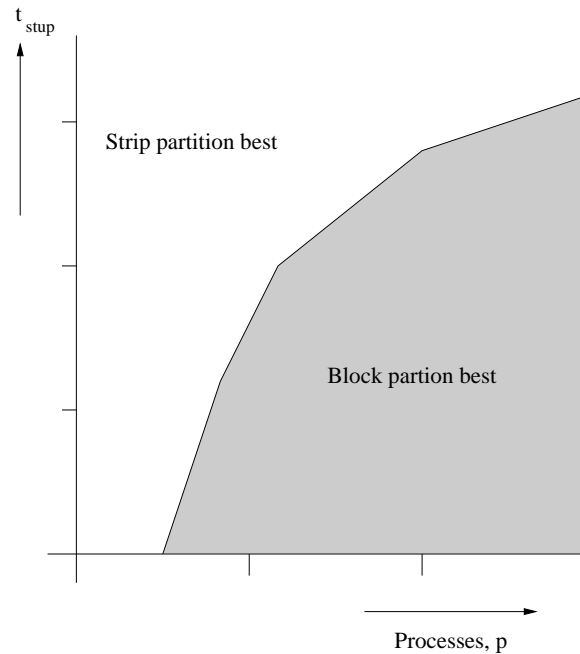
Block Communications

Strip Communications

9.21 Block/Strip Optimum

- Block communication is larger than strip if

$$\begin{aligned}8(t_{stup} + \sqrt{\frac{n}{p}}t_{data}) &> 4(t_{stup} + \sqrt{nt_{data}}) \\ \implies t_{stup} &> \sqrt{n}\left(1 - \frac{2}{\sqrt{p}}\right)t_{data}\end{aligned}$$



9.22 Safety and Deadlock

- With all processes sending and then receiving data the code is unsafe - it relies on buffering in the send routine
 - Potential for deadlock (as in lab 1)!
- Alternative #1: reorder sends and receives (eg strip partitioning)

```
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));
}
```

9.23 Alt# 2: Asynchronous Communication using Ghostpoints

- Assign extra receive buffers for edges where data is exchanged
- Use asynchronous calls (MPI_Isend)

