



# Parallel Algorithm Design

CS121 Parallel Computing  
Fall 2024



# Foster's design methodology

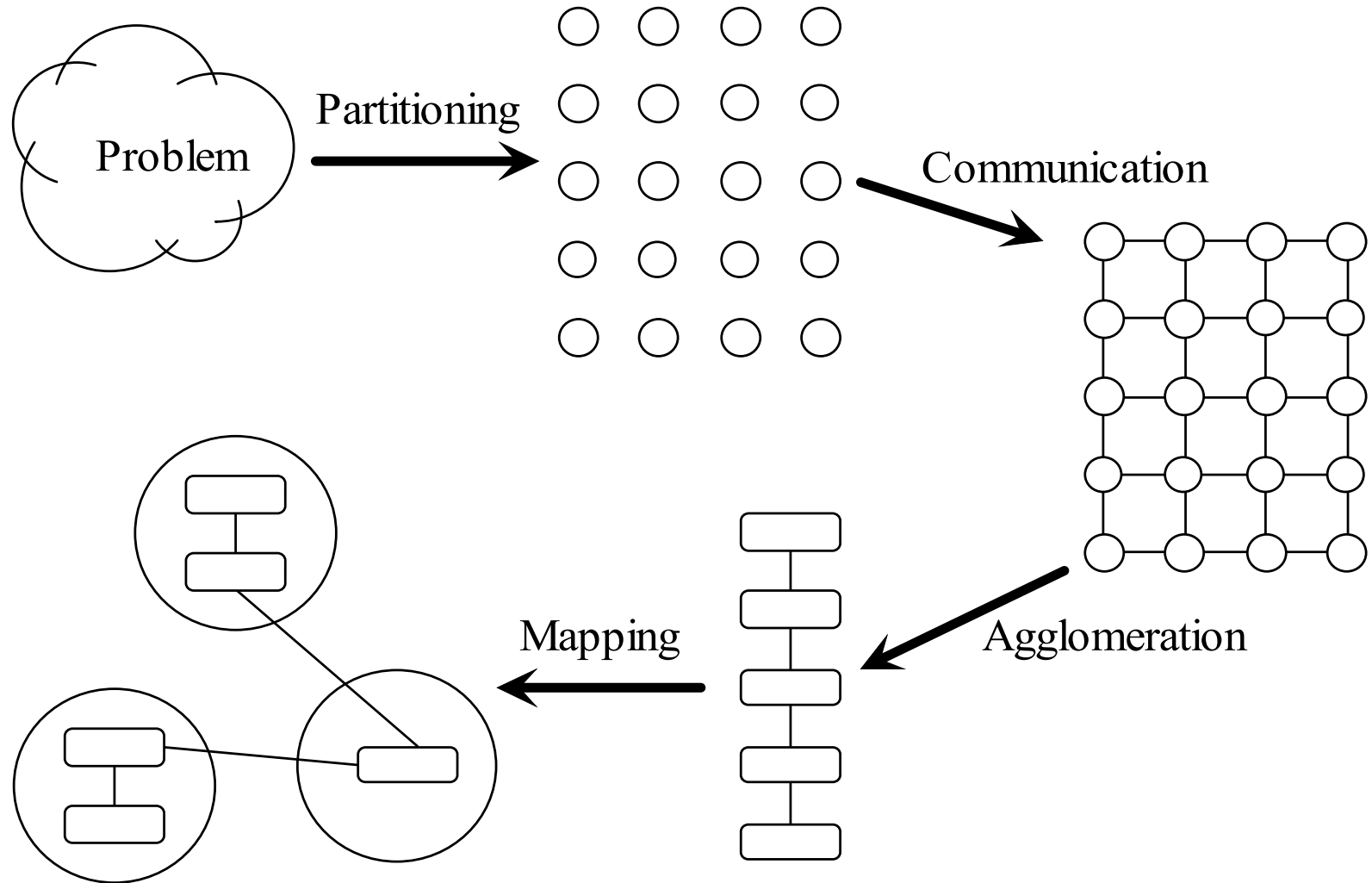
- Encourages development of scalable algorithms by delaying machine-dependent considerations until later steps.
  - I. Foster, Designing and Building Parallel Programs, Addison-Wesley, 1995, available online at <http://www.mcs.anl.gov/~itf/dbpp/>
- Methodology has four steps
  - Partitioning
  - Communication
  - Agglomeration
  - Mapping



# Outline

- Overview of Foster's methodology.
- Data partitioning.
- Example using Floyd-Warshall algorithm.
- Algorithmic partitioning.

# Foster's design methodology





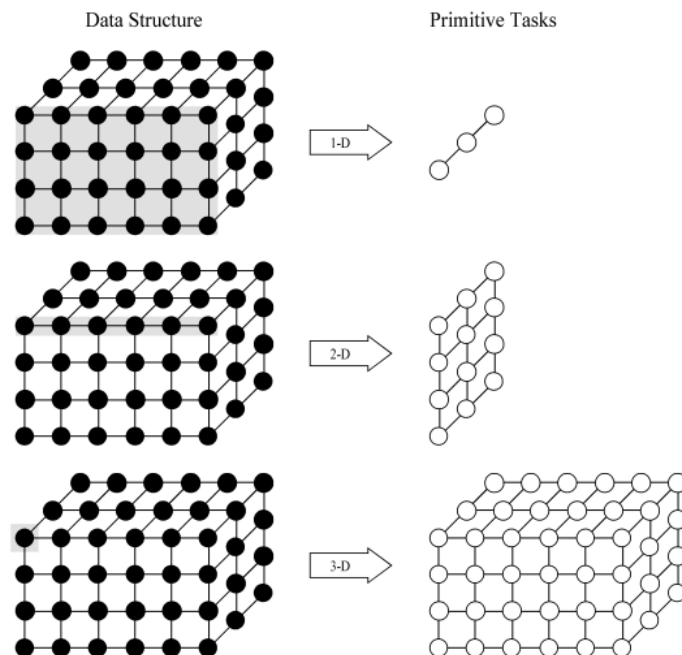
# Partitioning

- The application must be partitioned into a number of tasks that can execute in parallel.
- Data partitioning
  - The application data is divided into parts and each task operates on a different part of the data.
  - Need to associate computations with the data.
  - Also known as domain decomposition.
- Algorithmic partitioning
  - The algorithm is divided into a number of tasks that can execute in parallel.
  - Need to associate data with the tasks.
  - This often yields tasks that can be pipelined.
  - Also known as functional decomposition.

# Partitioning

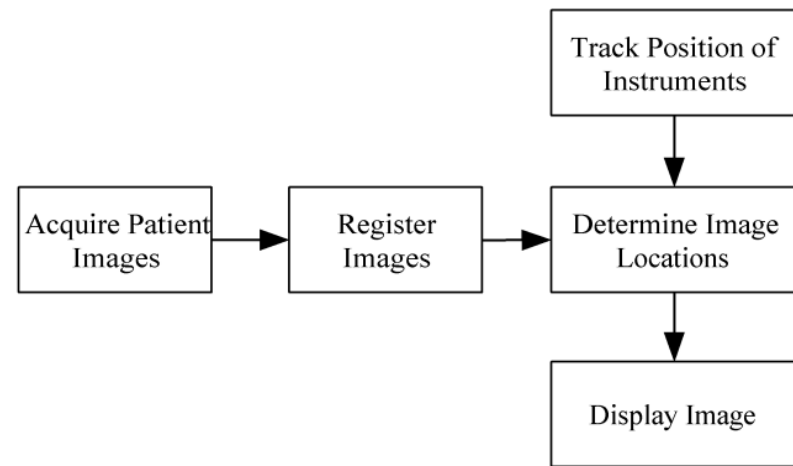
## ■ Data partitioning

- The application data is divided into parts that can be operated on in parallel.



## ■ Algorithmic partitioning

- The algorithm is divided into parts that can be executed in parallel.





# Partitioning checklist

- There should be at least 10x more primitive tasks than processors in the target computer.
- Redundant computations and redundant data storage should be minimized.
- The primitive tasks should be roughly the same size.
- The number of tasks should be an increasing function of problem size.



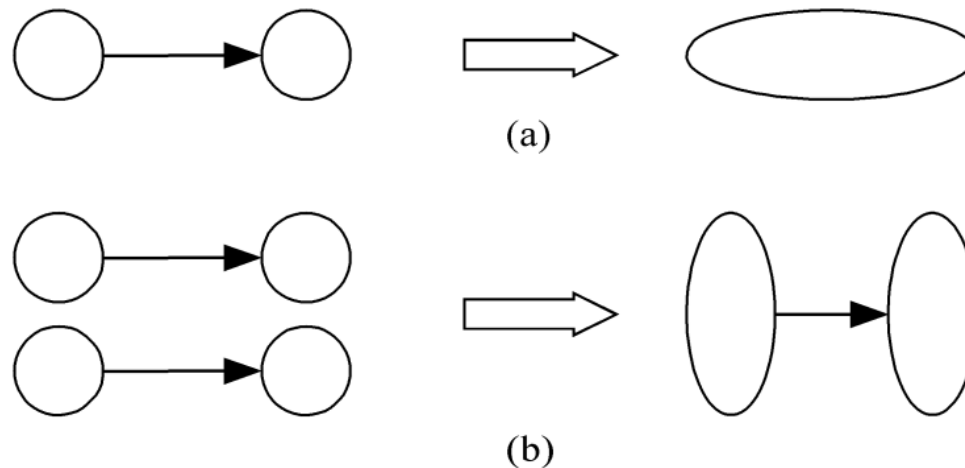
# Communication

- Determine what data is passed between tasks.
- Two kinds of communication
  - Local communication: A task needs values from a small number of other tasks (point-to-point),
  - Global communication: A number of tasks require or contribute data (collective).
- Checklist
  - Communication should be balanced among tasks.
  - Each task should preferably communicate with only a small group of neighbours.
  - Tasks should be able to perform communications concurrently, i.e. no bottlenecks.
  - Task should be able to overlap computations and communication.



# Agglomeration

- Grouping tasks into larger (super)tasks.
- Goal is to improve performance and simplify programming.
  - Eliminate communication between primitive tasks that are agglomerated into one task.
  - Combine groups of sending and receiving tasks so that fewer but larger messages are sent.



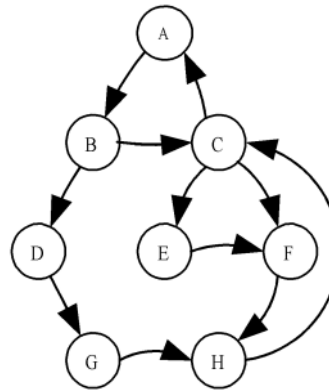
# Mapping

- Assigning (super)tasks to processors

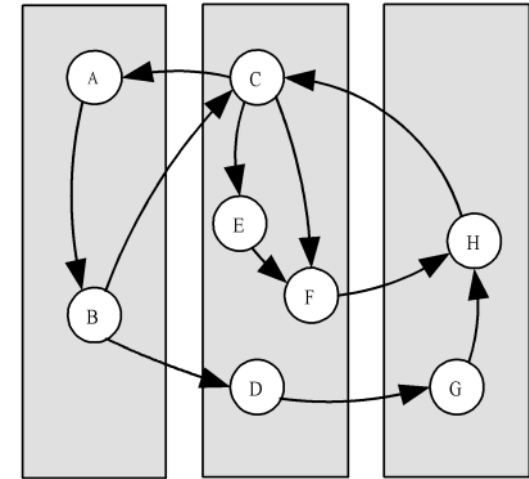
- ☐ With a distributed memory system, mapping can be done by the user.

- Goals

- ☐ Balance the load to maximize processor utilization.
- ☐ Minimize interprocessor communication.
- ☐ Sometimes two goals conflict. Look for best tradeoff.



(a)

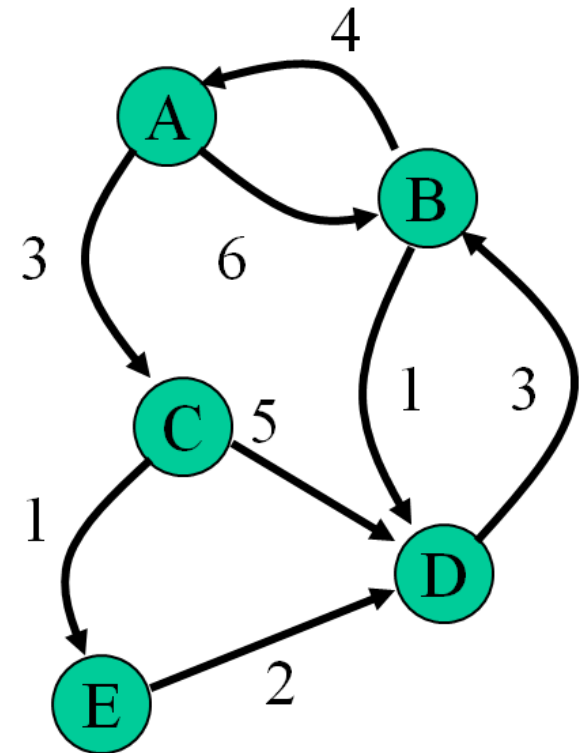


(b)

- If all tasks require the same amount of time, the above mapping would mean the middle processor takes twice as long as the other two.

# All pairs shortest path problem

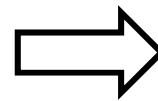
- Given a weighted directed graph of vertices and edges.
  - There may be an edge from vertex  $i$  to  $j$ , but not from  $j$  to  $i$ .
  - Edge weight from vertex  $i$  to  $j$  may be different to from  $j$  to  $i$ .
- Find the shortest path between every pair of vertices.
- Applications
  - Shortest route on a road / communication network.
  - Efficient solution in a state space graph.



# Example

- Represent graph using adjacency matrix.
  - $a[i,j] = 0$  if  $i = j$ .
  - For  $i \neq j$ ,  $a[i,j] = \text{length of edge from } i \text{ to } j$ , or  $\infty$  if no edge.
- Output matrix of shortest path lengths, or  $\infty$  if no path.
- Assume no negative weight cycles for simplicity.

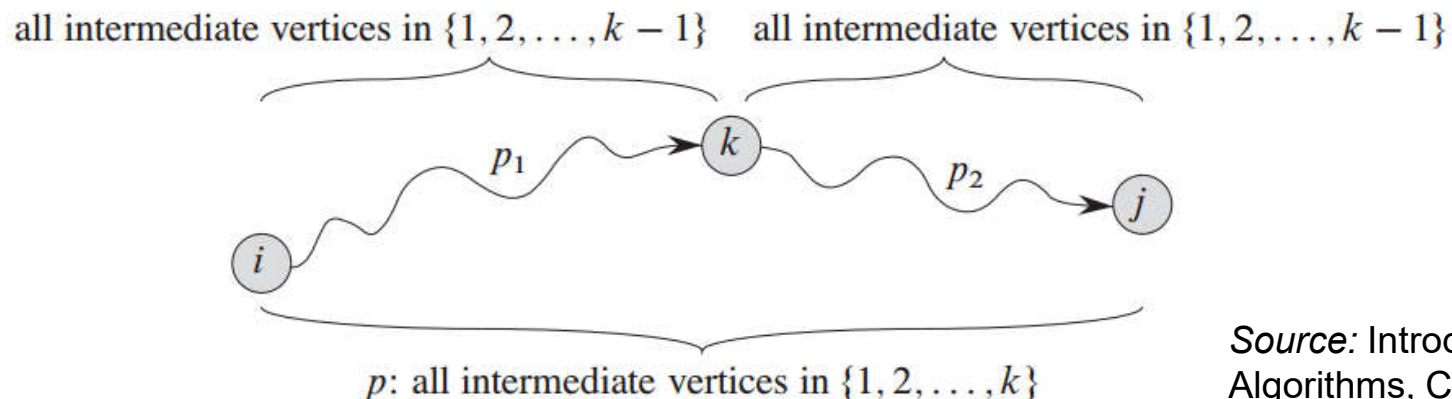
	A	B	C	D	E
A	0	6	3	$\infty$	$\infty$
B	4	0	$\infty$	1	$\infty$
C	$\infty$	$\infty$	0	5	1
D	$\infty$	3	$\infty$	0	$\infty$
E	$\infty$	$\infty$	$\infty$	2	0



	A	B	C	D	E
A	0	6	3	6	4
B	4	0	7	10	8
C	10	6	0	3	1
D	7	3	10	0	11
E	9	5	12	2	0

# Floyd-Warshall algorithm

- Consider the shortest path from  $i$  to  $j$  using only nodes  $1, \dots, k$  as intermediate nodes.
  - Intermediate node is a node on the path besides  $i$  and  $j$ .
  - If the path doesn't exist, indicate its distance by  $\infty$ .
- Assume a path exists. Then it either uses node  $k$  or not.
  - In the first case,  $k$  occurs once in path from  $i$  to  $j$ .
    - The path consists of a subpath from  $i$  to  $k$  using  $1, \dots, k-1$  as intermediate nodes, and a path from  $k$  to  $j$  using  $1, \dots, k-1$  as intermediate nodes.
  - In the second case, the path only uses nodes  $1, \dots, k-1$ .



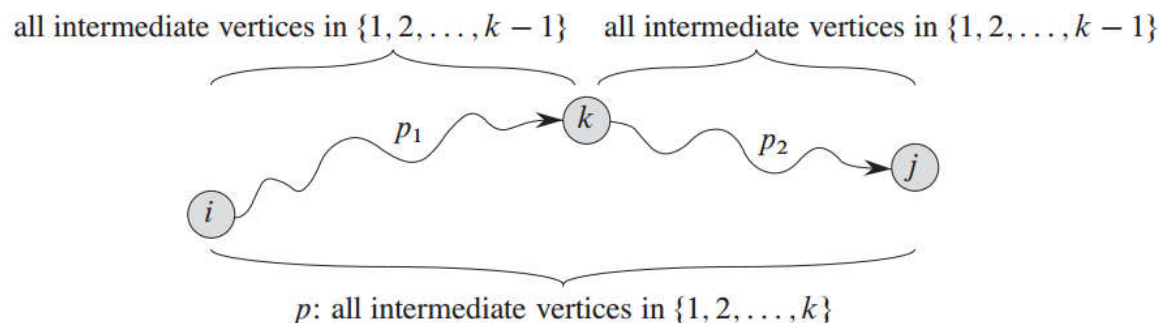
Source: Introduction to Algorithms, Cormen et al.

# Floyd-Warshall algorithm

- Let  $d_{i,j}^k$  = min distance from i to j using 1, ..., k as intermediate nodes.
- For a graph with n nodes, we compute the shortest paths using a dynamic programming formulation that increases k from 0 to n.
  - Represents more and more general paths.
  - Shortest path from i to j equals  $d_{i,j}^n$ .
  - Assume in k'th stage, have already found  $d_{i,j}^{k-1}$  for all i,j. Then

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

- Expresses observation that shortest path from i to j using 1,...,k as intermediate nodes either does not use node k, or uses it once.



# Parallelization: partitioning

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0, \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

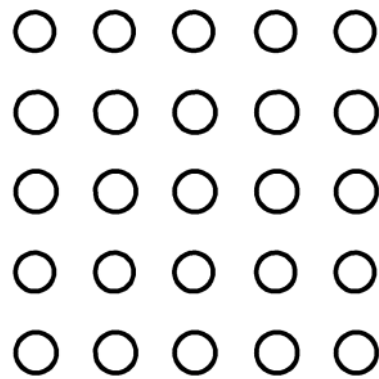
```
for (k = 0; k < n; k++)  
  for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
      a[i, j] = min(a[i, j], a[i, k] + a[k, j]);
```

- To parallelize this algorithm, observe for each  $k$ , all assignments can be done independently.
  - I.e.  $a[i, j]$  and  $a[i', j']$  can be computed in either order, for all  $i, i', j, j'$ .
  - $a[i, j]$  depends on  $a[i, k]$  and  $a[k, j]$ .
  - But  $a[i, k] = \min(a[i, k], a[i, k] + a[k, k])$ , where  $a[k, k] = 0$ .
  - So  $a[i, k]$  doesn't change in iteration  $k$ . Similarly for  $a[k, j]$ .
  - So no matter when  $a[i, j]$  is computed during phase  $k$ , it has same value. Same for  $a[i', j']$ . So  $a[i, j]$  and  $a[i', j']$  are independent.
- Parallel algorithm runs  $n$  phases sequentially (one for each  $k$ ). In each phase, all  $n^2$  assignments done in parallel.

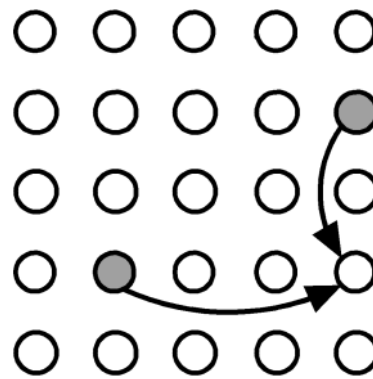
# Parallelization: communication

## ■ Communication in each phase $a[i,j] = \min(a[i,j], a[i,k] + a[k,j])$

The primitive tasks



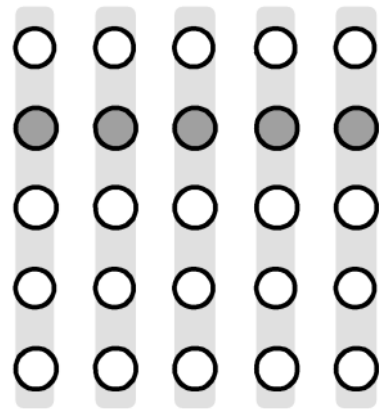
(a)



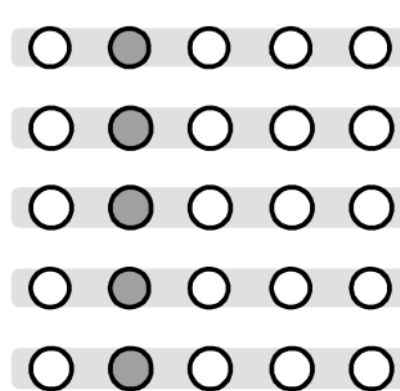
(b)

- Updating  $a[3,4]$  when  $k=1$ .
- In general, every entry in row  $i$  needs  $a[i,k]$ , and every entry in column  $j$  needs  $a[k,j]$ .

So in phase  $k$ ,  $a[k,j]$ , for each  $j$ , broadcasts its value to the  $j$ 'th column.



(c)



(d)

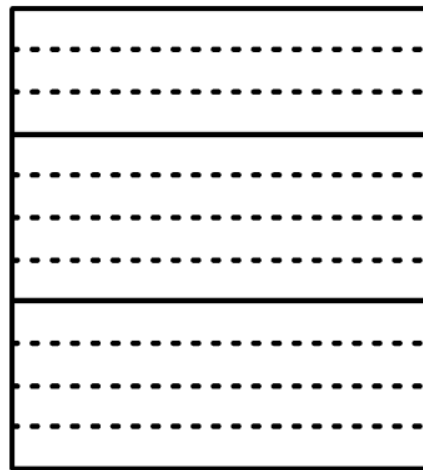
Also,  $a[i,k]$ , for each  $i$ , broadcasts its value to the  $i$ 'th row.



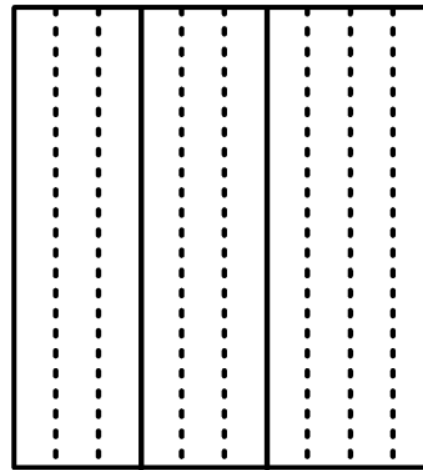
# Agglomeration and mapping

- Number of tasks is static, always  $n^2$ .
- Computation time per task is constant.
- Agglomerate tasks to minimize communication.
- Create one task per MPI process.
- Can choose either (a) row striping, or (b) column striping.
  - Row striping more convenient in C.

Each process has a strip of rows, avoid communication within same strip (row).



(a)



(b)

Each process has a strip of columns, avoid communication within same strip (column).



# MPI pseudocode

- Assume  $n \times n$  matrix on  $p$  processors ( $n$  exactly divisible by  $p$ ), where  $\text{part}[i,j]$  holds strip of rows of  $i$ 'th processor.
- We're ignoring code to distribute matrix and collect results.

```
s = n/p;                                /* rows in strip */
for (k = 0; k < n; k++) {
    root = k/s;                           /* owner of row */
    if (myrank == root) {
        offset = k - myrank*s;            /* offset of row */
        for (j = 0; j < n; j++)           /* copy row */
            temp[j] = part[offset][j];
    }
    bcast(temp, n, root, P_group);         /* broadcast row */
    for (i = 0; i < s; i++)               /* update strip */
        for (j = 0; j < n; j++)
            part[i][j] =
                min(part[i][j], part[i,k] + temp[j]); }
```

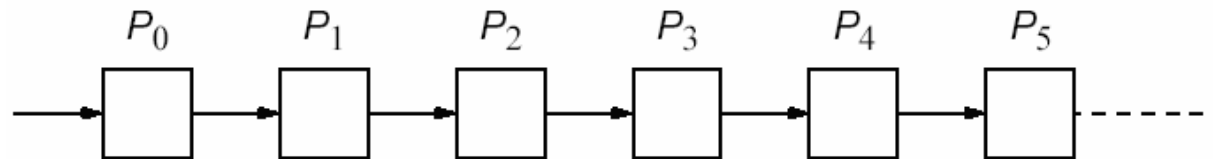


# Analysis of algorithm

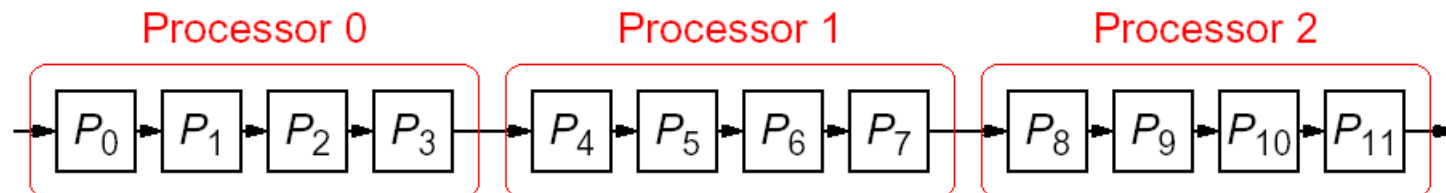
- Sequential running time  $t_s = O(n^3)$ .
- Parallel execution
  - In each iteration of  $k$ , there is a communication phase and a computation phase.
  - Communication phase
    - Broadcast row to other processors
    - $t_{\text{comm}} = (t_{\text{startup}} + 4 n t_{\text{data}}) \log p$ , assuming each element 4 bytes.
  - Computation phase  $t_{\text{comp}} = cn^2 / p$ .
  - Total time
    - $t_p = n (t_{\text{startup}} + 4 n t_{\text{data}} \log p + cn^2 / p) = O(n^2 \log p + n^3 / p)$ .
- Speedup
  - $t_s / t_p = O(n^3) / O(n^2 \log p + n^3/p) = O(p)$ .

# Algorithmic partitioning

- Divide an algorithm into several parts, each taking turns processing the data.
- One example is pipelined computations.
  - The problem is divided into a sequence of steps.
  - The steps must be executed one after another.
  - Each step is executed by a separate task.



- If number of stages in pipeline is greater than number of processors, assign group of steps to each processor.





# Example: sum of sequence

- Power series for  $\sin \theta = \theta - \theta^3/3! + \theta^5/5! - \theta^7/7! + \dots$

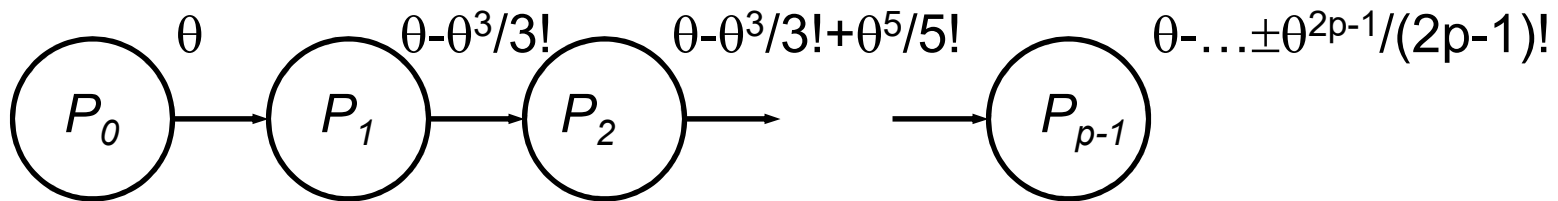
```
r = theta;  
sum = theta;  
for (i=1; i < p; i++) {  
    r = - r*theta*theta/(2*i*(2*i+1));  
    sum = sum + r;  
}
```

- The loop could be unfolded to give.

```
sum = theta;  
sum = sum - theta3/3!;  
sum = sum + theta5/5!;  
sum = sum - theta7/7!;  
...
```

# Pipeline for unfolded loop

- Use as many stages (tasks) as required by the precision of the result.
- Can pass  $\theta$ , partial product and partial sum to next processor in pipeline.
  - More details in later slide.

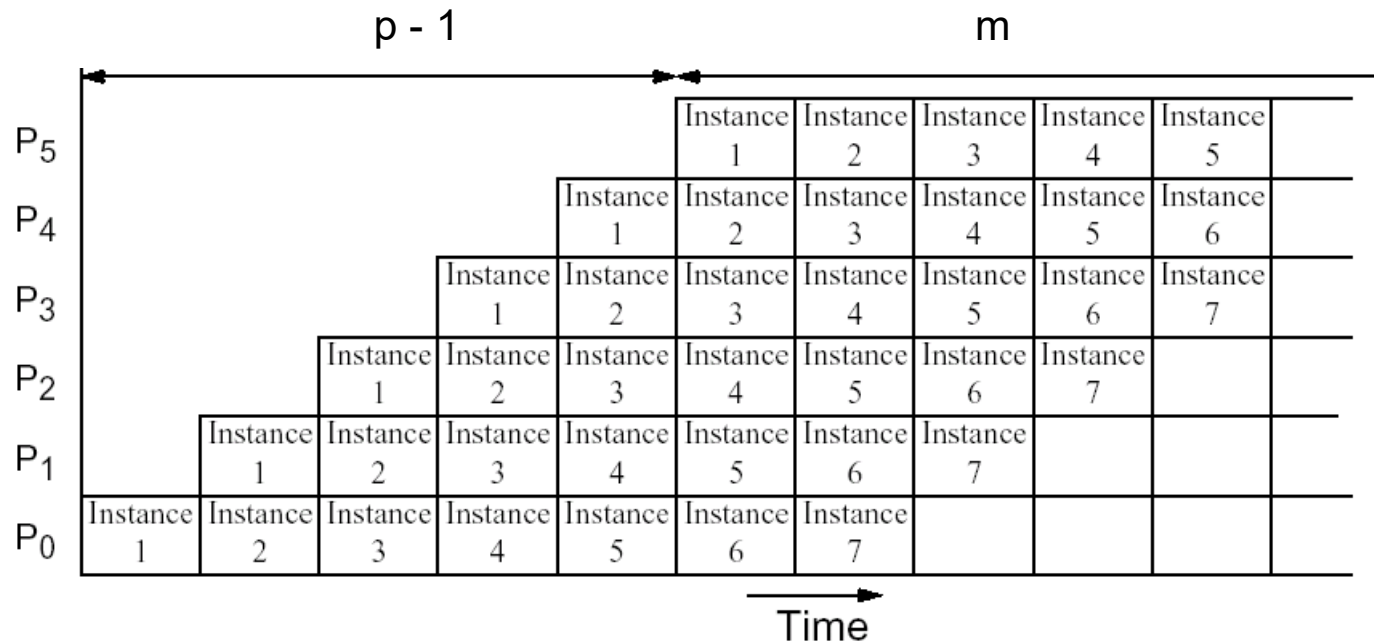




# Types of pipelines

- Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations.
  1. If more than one instance of the problem is to be executed.
  2. If a series of data items must be processed, each requiring multiple operations.
  3. If information to start the next process can be passed forward before the process has completed all its internal operations.

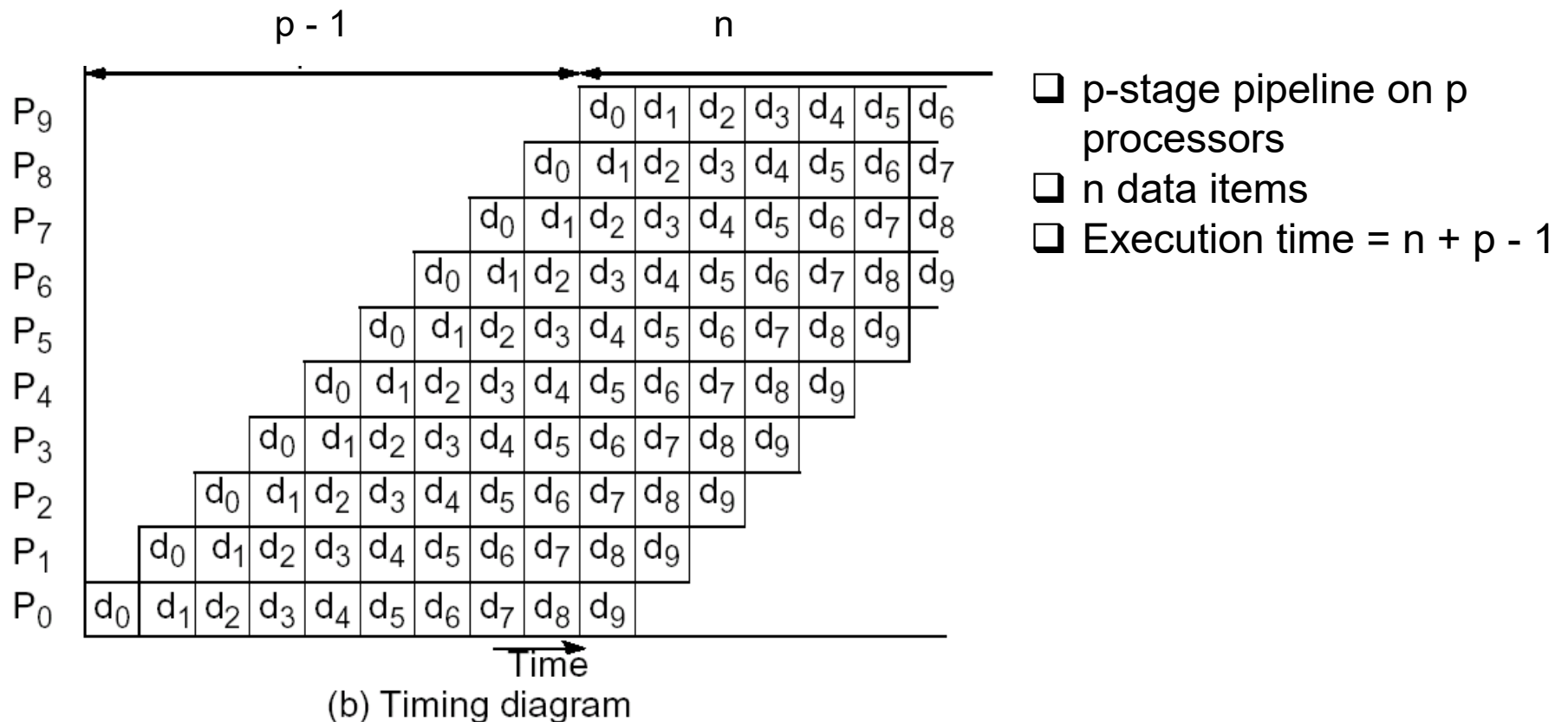
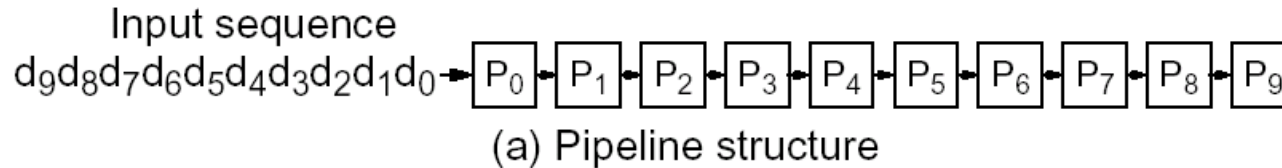
# Type 1: Space-time diagram



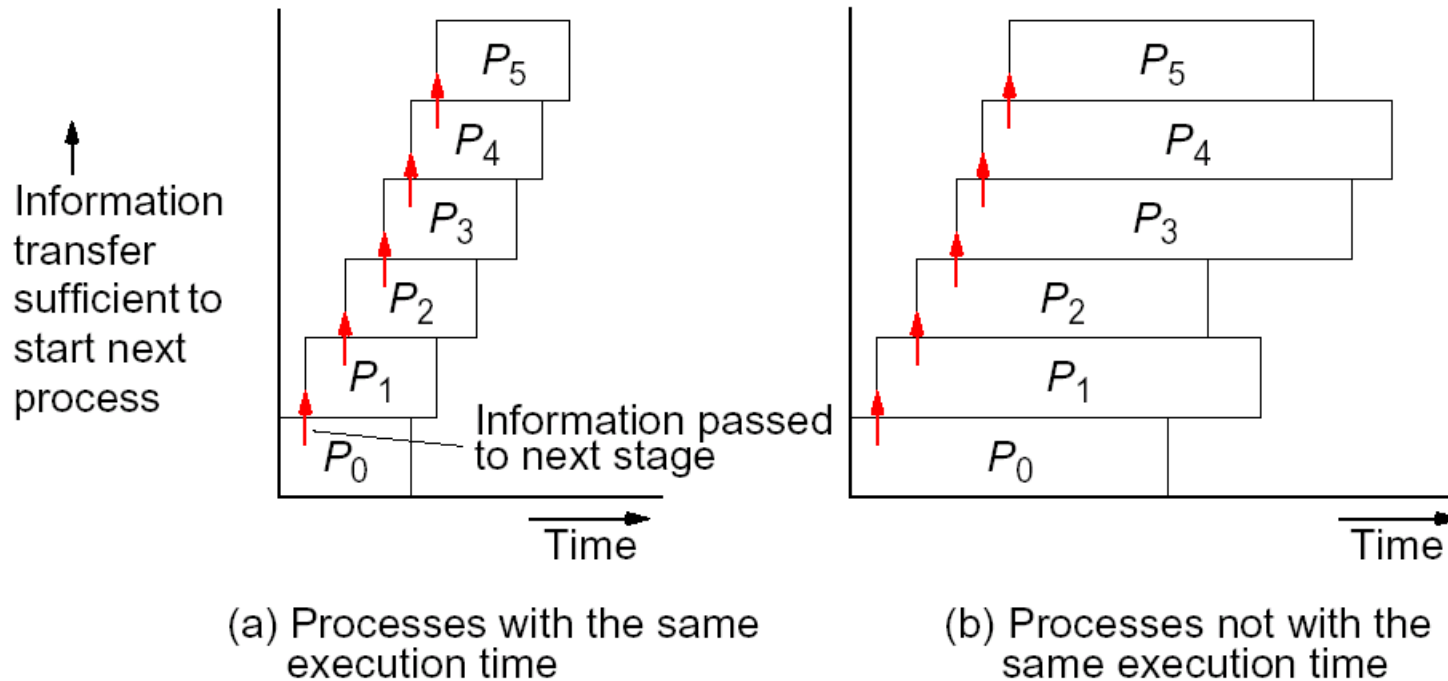
- ☐  $p$ -stage pipeline on  $p$  processors.
- ☐  $m$  instances of the problem.
- ☐ Execution time =  $m + p - 1$ .



# Type 2: Space-time diagram



# Type 3: Space-time diagram



Pipeline processing where information passes to next stage before end of process.



# Analysis of types 1 and 2

## ■ Assumptions and model

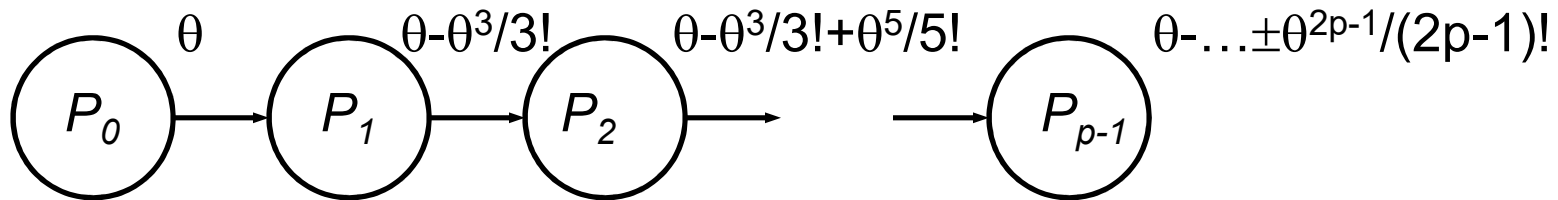
- $p$  processors performing  $p$  stages (tasks).
- Each task consists of a number of cycles, i.e. basic units of computation.
- All cycles of all tasks take about same amount of time, say  $t_{\text{cycle}}$ .

## ■ Parallel execution time $t_p = t_{\text{cycle}} * n_c$ .

- Time for one cycle  $t_{\text{cycle}} = t_{\text{comp}} + t_{\text{comm}}$ .
- Number of cycles  $n_c$  given by space-time diagram.
  - For type 1,  $n_c = m + p - 1$  for  $m$  instances.
  - For type 2,  $n_c = n + p - 1$  for  $n$  data items.

# Sum of sequence (type 1)

- Power series  $\sin \theta = \theta - \theta^3/3! + \theta^5/5! - \theta^7/7! + \dots$



```
if (myrank == 0)                                /* first */
    send(theta, theta, theta, P1);
else                                            /* not first */
    recv(&theta, &r, &sum, Pi-1);
    r = - r*theta*theta/(2*i*(2*i+1));
    sum = sum + r;
    if (myrank < p-1 )                        /* not last */
        send(theta, r, sum, Pi+1);
}
/* final result in Pp-1 */
```

# Analysis

## ■ Parallel execution

- Use most complex cycle to upper bound computation time.

```
recv(&theta, &r, &sum, Pi-1);  
r = - r*theta*theta/(2*i*(2*i+1));  
sum = sum + r;  
send(theta, r, sum, Pi+1);
```

- Computation time  $t_{\text{comp}} = O(1)$ .
- Communication time  $t_{\text{comm}} = 2(t_{\text{startup}} + 3t_{\text{data}})$ .
- Time for one cycle  $t_{\text{cycle}} = t_{\text{comp}} + t_{\text{comm}}$ .

## ■ Single instance of problem

- Number of cycles  $n_c = p$  for single instance.
- Time  $t_p = t_{\text{cycle}} * n_c = p(2(t_{\text{startup}} + 3t_{\text{data}}) + O(1))$ .

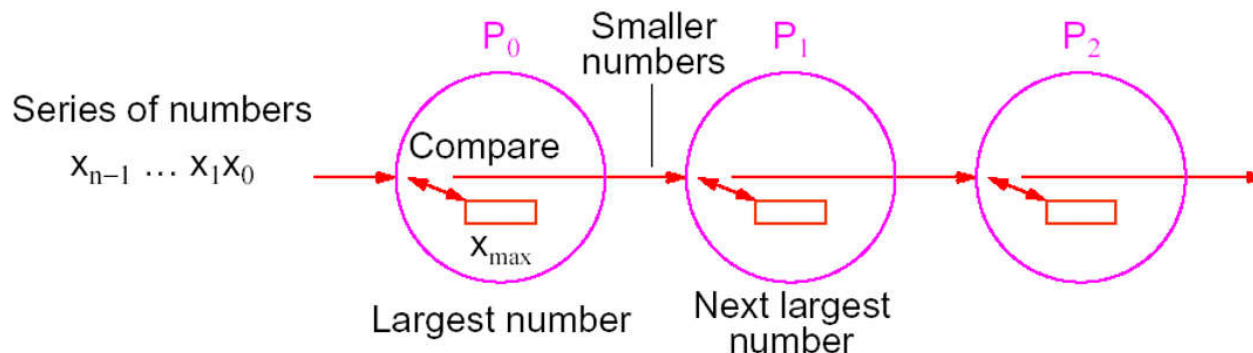


# Analysis

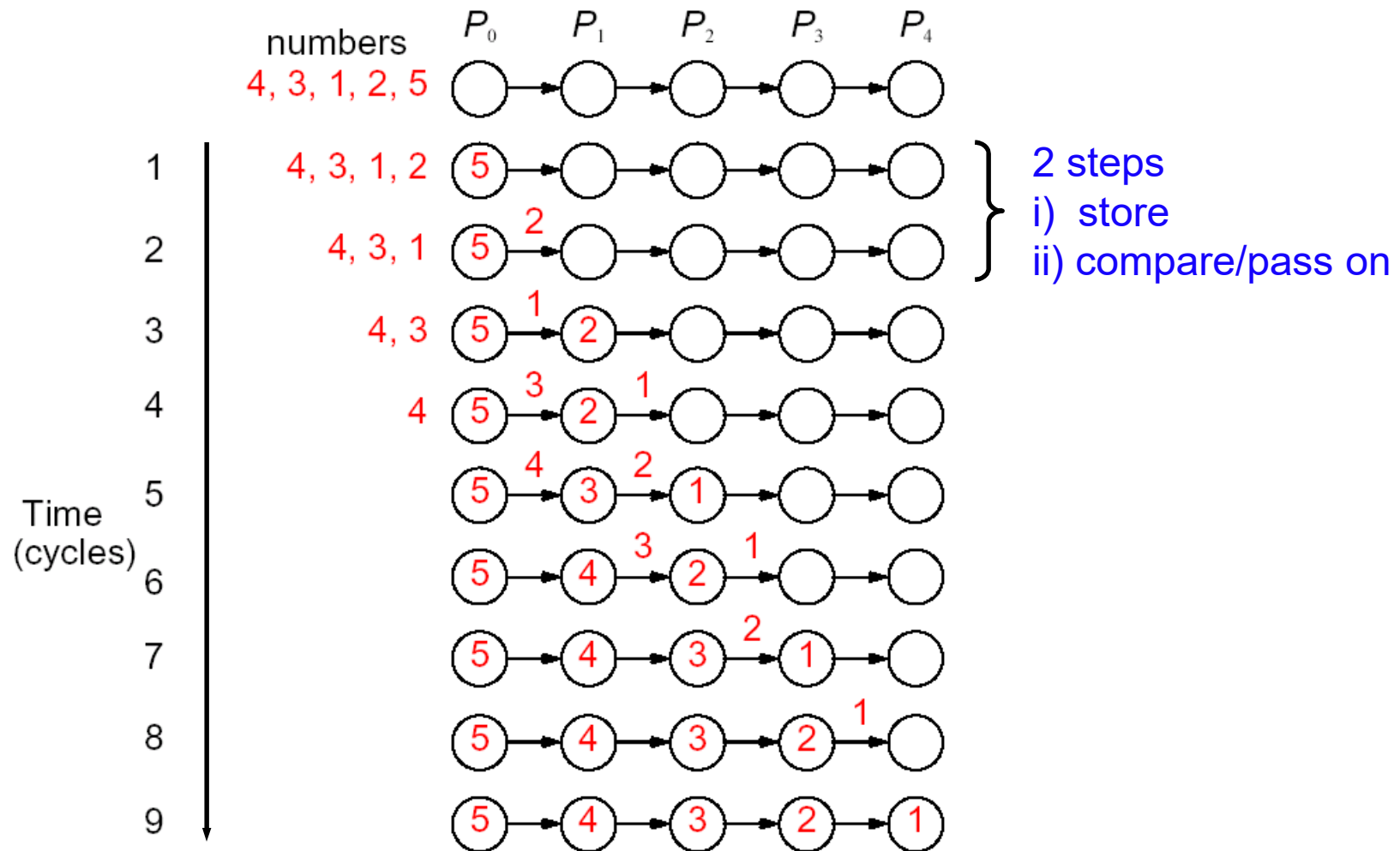
- Pipeline is efficient only if we have more than one problem to solve, i.e. more than one value of  $\theta$ .
  - Assume we compute sin for  $m$  values of  $\theta$ .
- Parallel execution
  - Number of cycles  $n_c = m + p - 1$  for  $m$  instances.
  - Time  $t_p = t_{\text{cycle}} * n_c = (2(t_{\text{startup}} + 3t_{\text{data}}) + O(1)) * (m + p - 1)$ .
  - Average execution time per instance is  $t_a = t_p / m = O((m + p) / m)$ .
  - For large  $m$  (i.e. continuous supply of values of  $\theta$ ) average execution time per instance is
    - $t_a = 2(t_{\text{startup}} + 3t_{\text{data}}) + O(1) = O(1)$ .
    - A result is produced each constant number of pipeline cycles.

# Insertion sort (type 2)

- Use a pipeline to sort  $n$  numbers in decreasing order using  $p = n$  processors.
- Each processor stores the largest number received so far and passes on all smaller numbers.
- If a new number is larger than the currently stored number, the stored number is passed on and replaced by the new number.
- When all numbers have been passed through the pipeline,  $P_0$  has the largest number,  $P_1$  has next largest number, etc.



# Insertion sort







# Insertion sort

- Initialization and basic cycle for  $P_i$ .

- $x$  is current max value at  $P_i$ .

```
recv(&number,  $P_{i-1}$ );  
if (number >  $x$ ) {  
    send( $x$ ,  $P_{i+1}$ );  
     $x$  = number;  
}  
else  
    send(number,  $P_{i+1}$ );
```

- Process  $P_i$  needs to pass on  $(n - i - 1)$  numbers, as there are  $(n - i - 1)$  processors to its right.

- It executes the basic cycle  $(n - i - 1)$  times.



# Analysis

- Sequential execution

- $t_s = (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 = O(n^2)$ .

- Parallel execution

- Each basic cycle consists of one recv, one send and a compare/exchange operation.
  - Computation time  $t_{\text{comp}} = 3$ .
  - Communication time  $t_{\text{comm}} = 2(t_{\text{startup}} + t_{\text{data}})$ .
  - Time for one cycle  $t_{\text{cycle}} = t_{\text{comp}} + t_{\text{comm}}$
  - Number of cycles  $n_c = n + p - 1 = 2n - 1$ .
  - Time  $t_p = t_{\text{cycle}} * n_c = (2(t_{\text{startup}} + t_{\text{data}}) + 3) * (2n - 1) = O(n)$ .



# Solving linear equations (type 3)

- Solving linear equations in upper (or lower) triangular form.
  - Used in Gaussian elimination.
- Process  $P_i$  behaves as follows.
  - Receives partial solution from  $P_{i-1}$ .
  - Passes on partial solution to  $P_{i+1}$ .
  - Computes next part of solution and sends to  $P_{i+1}$ .
- This is a type 3 pipeline computation.
  - Each process has a different computation time.



# Solving linear equations

## Upper-triangular form

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

where the  $a$ 's and  $b$ 's are constants and the  $x$ 's are unknowns to be found.



# Back substitution

First, the unknown  $x_0$  is found from the last equation; i.e.,

$$x_0 = \frac{b_0}{a_{0,0}}$$

Value obtained for  $x_0$  substituted into next equation to obtain  $x_1$ ; i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

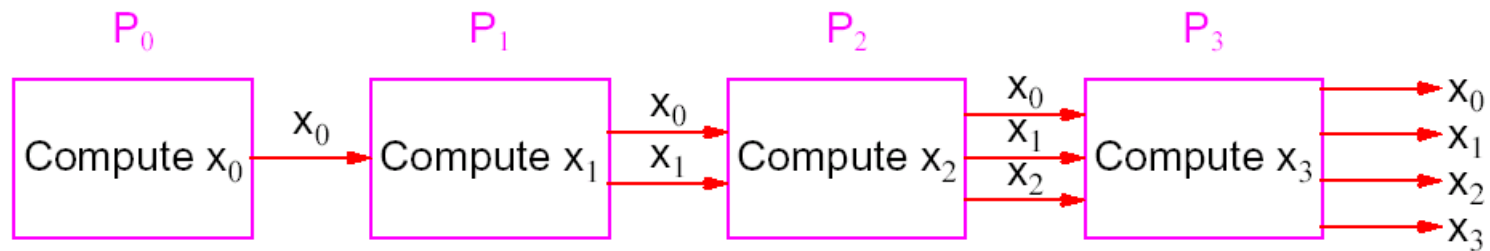
Values obtained for  $x_1$  and  $x_0$  substituted into next equation to obtain  $x_2$ :

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

# Pipeline solution

First pipeline stage computes  $x_0$  and passes  $x_0$  onto the second stage, which computes  $x_1$  from  $x_0$  and passes both  $x_0$  and  $x_1$  onto the next stage, which computes  $x_2$  from  $x_0$  and  $x_1$ , and so on.



$p = n$   
n equations  
n unknowns

Type 3 pipeline computation

# Parallel code

## ■ Process $P_i$ ( $0 < i < n$ )

```
sum = 0;
for (j = 0; j < i; j++) {
    recv(&x[j], Pi-1);
    send(x[j], Pi+1);
    sum = sum + a[i][j] * x[j];
}
x[i] = (b[i] - sum) / a[i][i];
send(x[i], Pi+1);
```

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

## ■ Process $P_0$

```
x[0] = b[0] / a[0][0];
send(x[0], Pi+1);
```

## ■ $p = n$ stages, with final result on $P_n$ .



# Analysis

## ■ Sequential execution

- Iteration  $i$  of loop performs  $i$  multiplications and additions, one subtraction and one division.
- $t_s = O(1+2+\dots+n-1) = O(n^2)$ .

## ■ Parallel execution

- Process  $P_i$  performs  $i$  multiplications and additions, one subtraction and one division.
- Process  $P_i$  has one recv and one send before process  $P_{i+1}$  can start, so time to pass data is  $2(t_{\text{startup}} + t_{\text{data}})$ .
- Process  $P_{n-1}$  starts at time  $(n-1) * 2(t_{\text{startup}} + t_{\text{data}})$ .



# Analysis

## ■ Parallel execution

- Process  $P_{n-1}$  performs  $n-1$  multiplications and additions, one division and one subtraction.
- Computation time  $t_{\text{comp}} = 2(n-1) + 2$ .
- Process  $P_{n-1}$  has  $(n-1)$  recv and  $n$  sends.
  - Communication time  $t_{\text{comm}} = (2n-1)(t_{\text{startup}} + t_{\text{data}})$ .
- Time  $t_p = (n-1) * 2(t_{\text{startup}} + t_{\text{data}}) + (2n-1) (t_{\text{startup}} + t_{\text{data}}) + 2(n-1) + 2 = O(n)$ .
- Speedup  $= t_s / t_p = O(n)$  with  $n$  processors.