CSE 6220/CX 4220

Introduction to HPC

Lecture 21: Graph Algorithms

Helen Xu

hxu615@gatech.edu

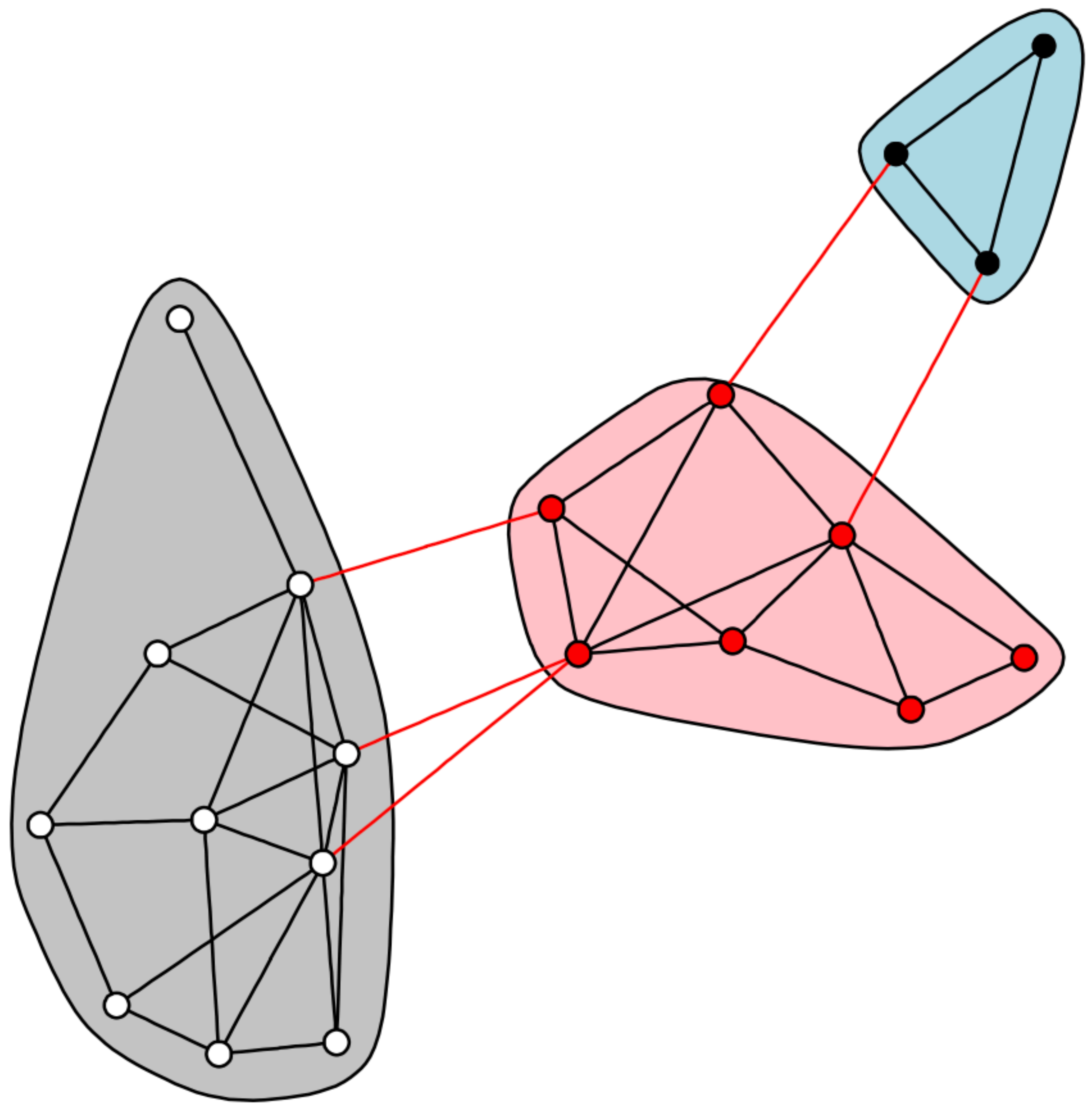
Georgia Tech College of Computing
School of Computational Science and Engineering

Slides from Prof. Srinivas Aluru

Material taken from "Introduction to Parallel Computing",
by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar.

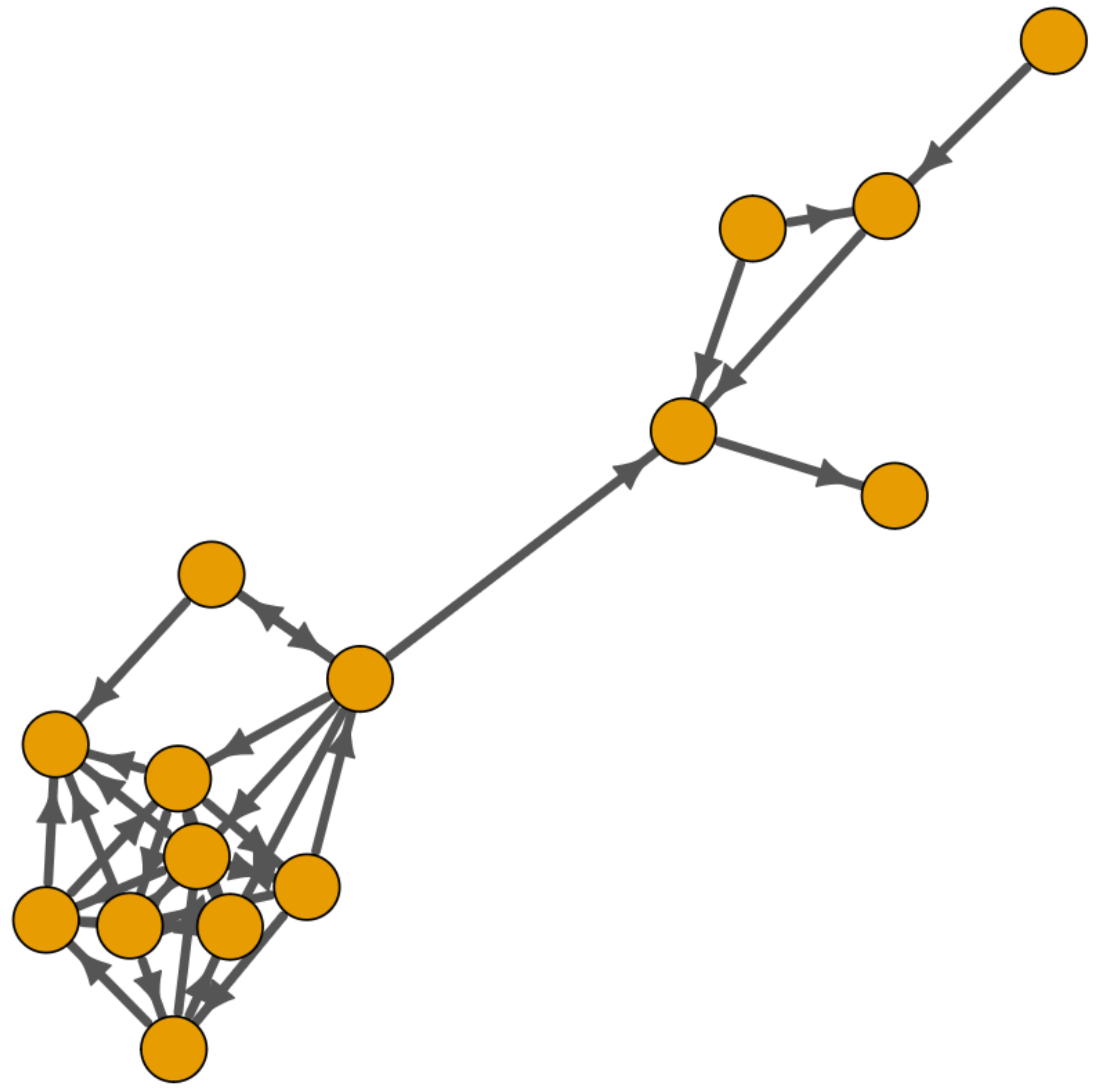
Graphs

- A graph is a set of **vertices** connected by a set of **edges**
- Framing data in this way is intuitive as it emphasizes the relationships in data
- Vertices and edges are frequently augmented with extra data, such as weights



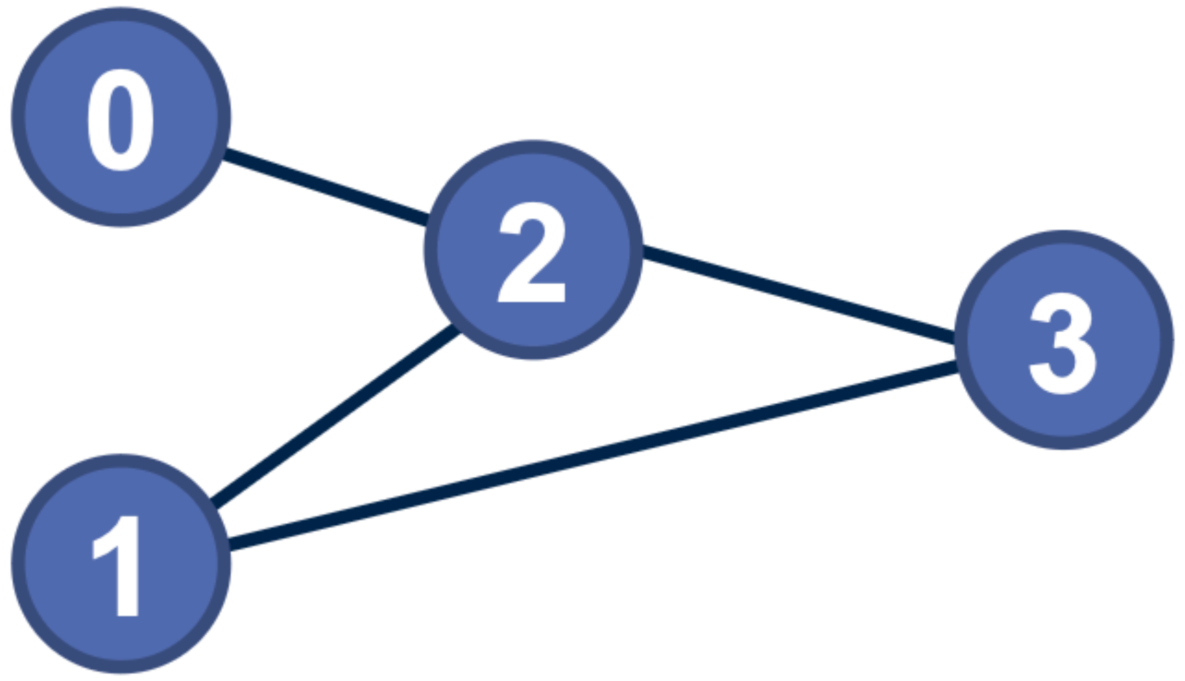
Graph Types

- A graph G is a tuple of sets $G = (V, E)$ such that $E \subseteq V \ X \ V$
- There are many extensions to this basic structure:
  - A directed graph is a graph where edges have directionality, i.e., each edge has an *initial* and *terminal* vertex, also called tail and head, and self loops are possible
  - A multigraph is a graph that allows multiple edges between the same pair of nodes.
  - A hypergraph has edges that can connect more than two vertices



Representing Graphs

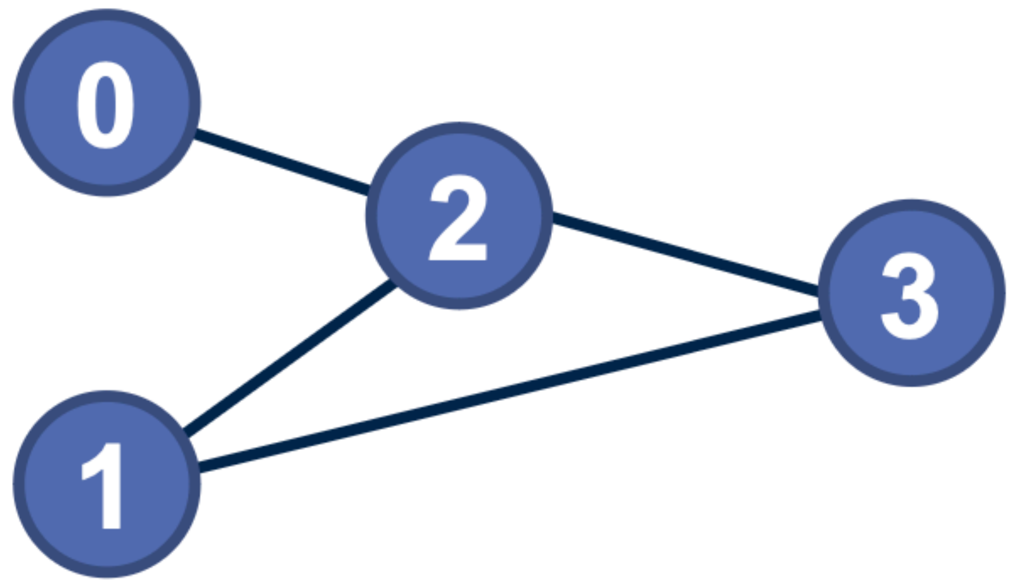
- Given that $E \subseteq [V]^2$, it is natural to represent a graph as a matrix, termed the *adjacency matrix*.



	0	1	2	3
0			1	
1			1	1
2	1	1		1
3		1	1	

- Adjacency matrix of an undirected graph is a symmetric matrix.

Representing Graphs



• Adjacency Matrix	$\Theta(\lvert V\rvert^2)$
• Incidence Matrix	$\Theta(\lvert V\rvert \times \lvert E\rvert)$
• Adjacency List	$\Theta(\lvert V\rvert + \lvert E\rvert)$
• Compressed Sparse Row/Column	$\Theta(\lvert V\rvert + \lvert E\rvert)$

	0	1	2	3
0			1	
1			1	1
2	1	1		1
3		1	1	

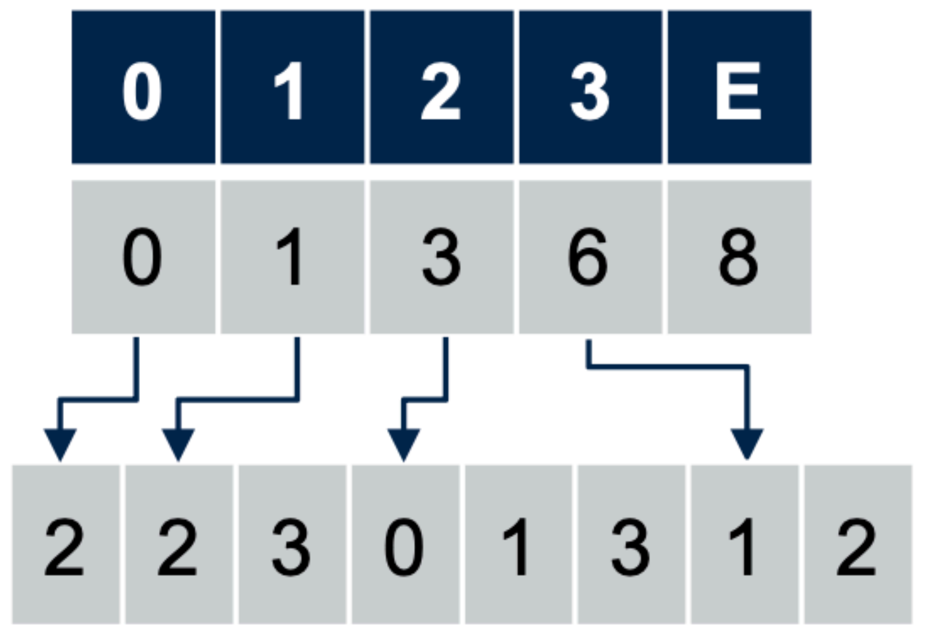
Adjacency Matrix

	e0	e1	e2	e3
0	1			
1		1		1
2	1	1	1	
3			1	1

Incidence Matrix

0	2		
1	2	3	
2	0	1	3
3	1	2	

Adjacency List



0	1	2	3	E
0	1	3	6	8

2	2	3	0	1	3	1	2

CSR

Parallel Algorithms for Dense Graphs

Assume:

- A dense weighted graph
- A large number of vertices

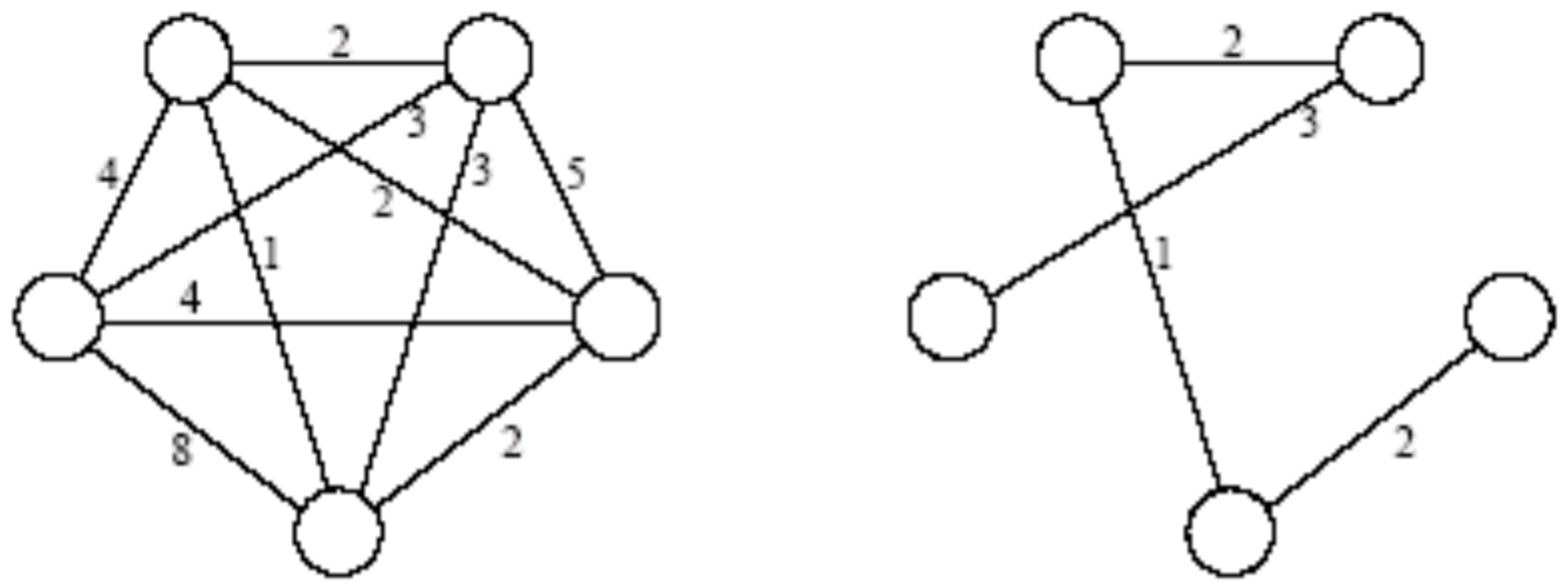
We will first briefly present the sequential version, then a distributed memory parallel version using adjacency matrices

Algorithms

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths

Minimum Spanning Tree

- A ***spanning tree*** of an undirected graph *G* is a subgraph of *G* that is a **tree** containing all the vertices of *G*
  - Recall: a tree is a connected acyclic graph
- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph
- A ***minimum spanning tree*** (MST) for a weighted undirected graph is a spanning tree with minimum weight



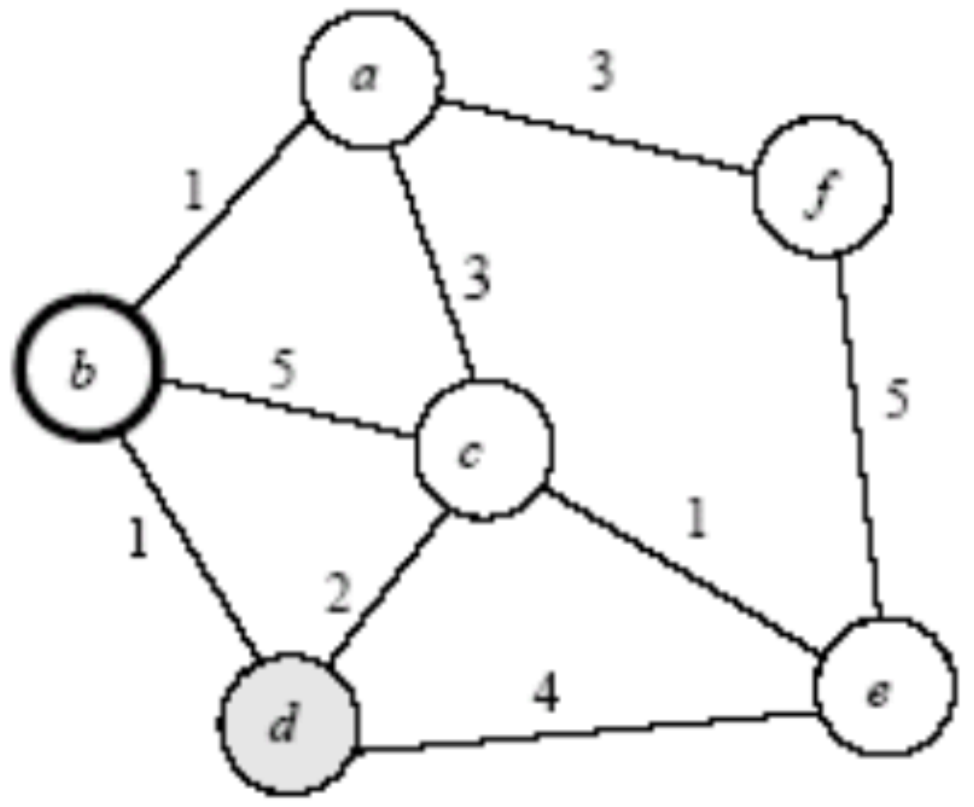
An undirected graph and its minimum spanning tree.

MST: Prim's Algorithm

- Greedy algorithm
- Start by selecting an arbitrary vertex, include it into the MST
- Grow the MST using the **closest connected vertex** to the current MST
- Stop when all vertices are included

MST: Prim’s Algorithm

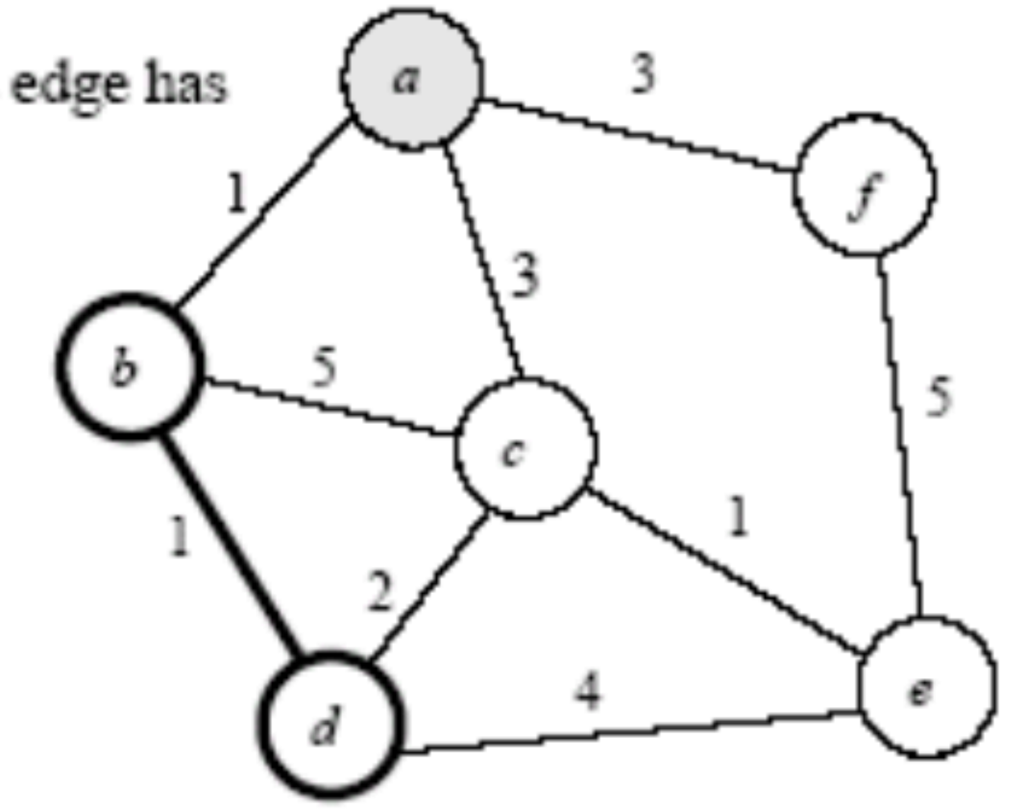
(a) Original graph



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[] & 1 & 0 & 5 & 1 & \infty & \infty \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

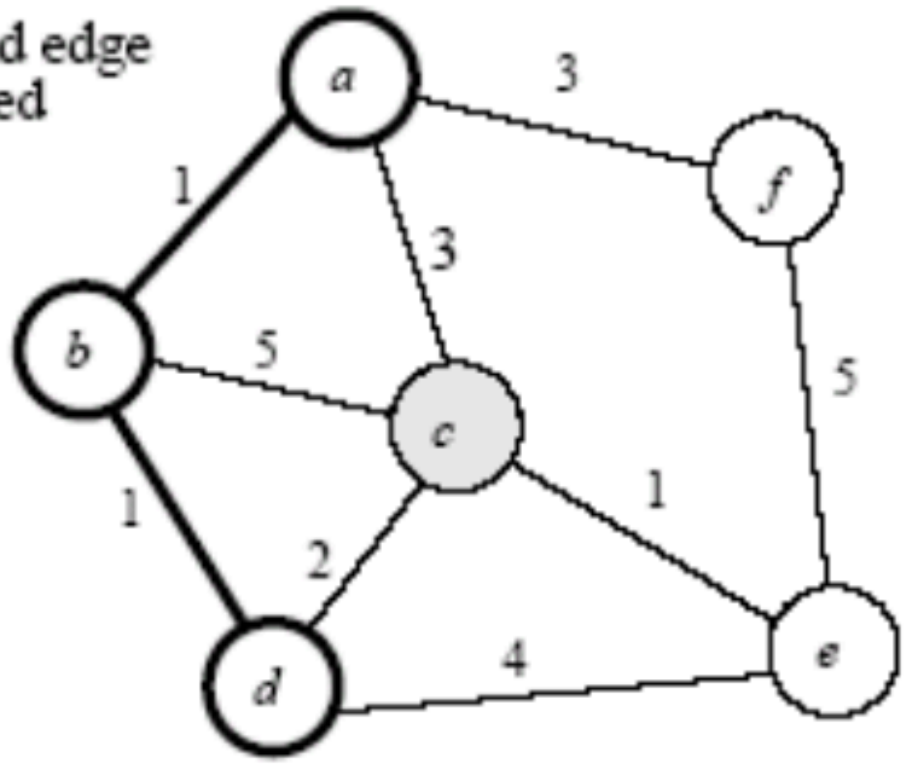
(b) After the first edge has been selected



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[] & 1 & 0 & 2 & 1 & 4 & \infty \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

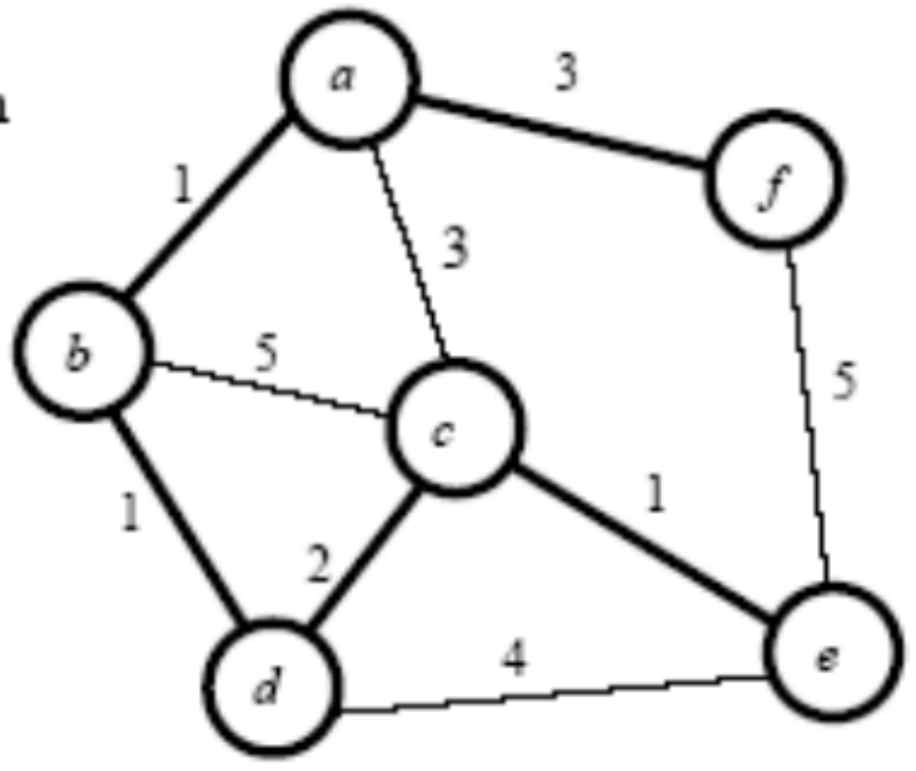
(c) After the second edge has been selected



$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[] & 1 & 0 & 2 & 1 & 4 & 3 \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

(d) Final minimum spanning tree



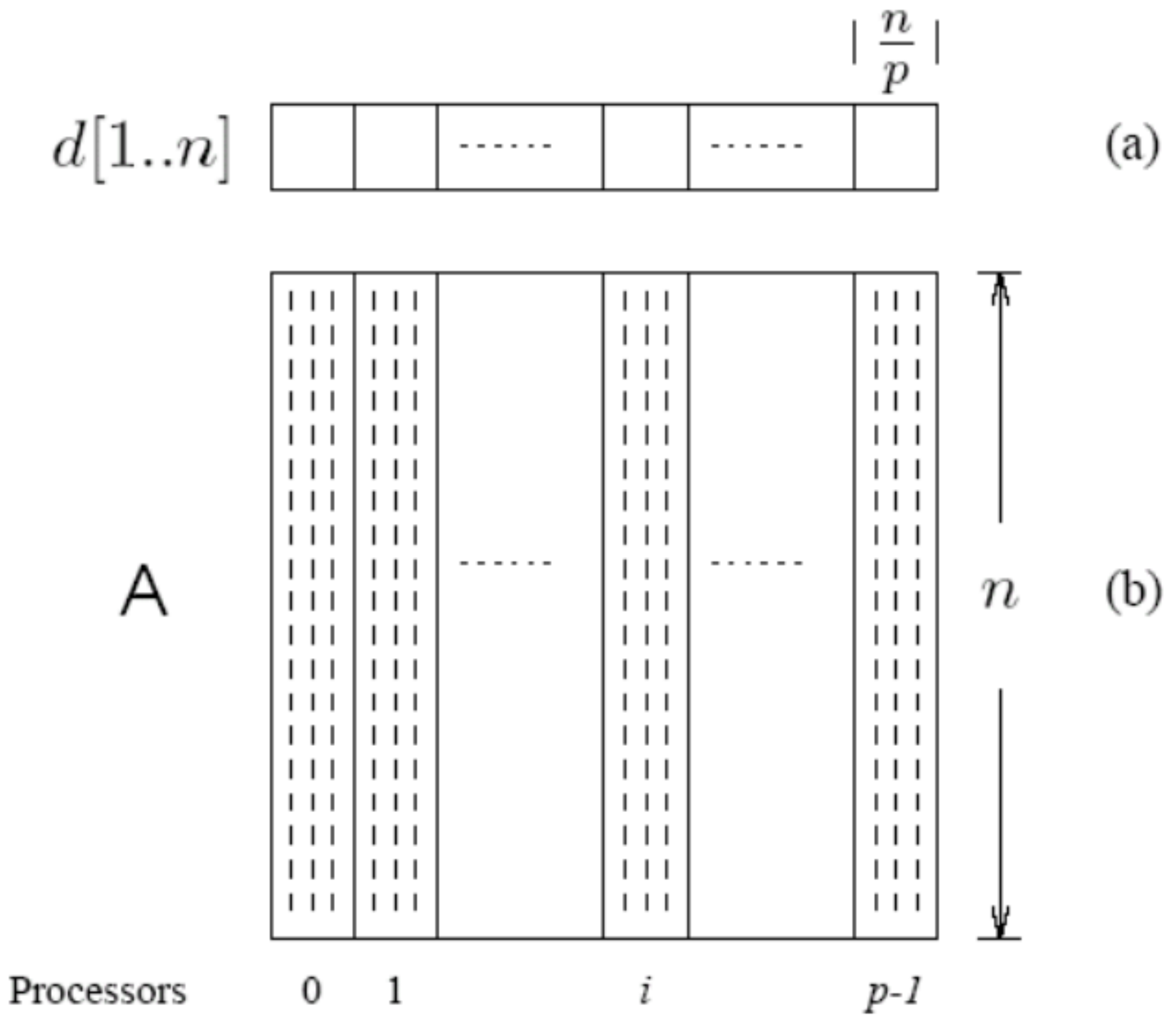
$$\begin{array}{c|cccccc} & a & b & c & d & e & f \\ \hline d[] & 1 & 0 & 2 & 1 & 1 & 3 \end{array}$$

$$\begin{array}{c} a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix}$$

MST: Prim's Algorithm

```
procedure PRIM_MST($V, E, w, r$)
begin
    $V_T := \{r\}$;
    $d[r] := 0$;
    for all $v \in (V - V_T)$ do
        if edge $(r, v)$ exists set $d[v] := w(r, v)$;
        else set $d[v] := \infty$;
    while $V_T \neq V$ do
    begin
        find a vertex $u$ such that $d[u] := \min\{d[v] | v \in (V - V_T)\}$;
        $V_T := V_T \cup \{u\}$;
        for all $v \in (V - V_T)$ do
            $d[v] := \min\{d[v], w(u, v)\}$;
    endwhile
end PRIM_MST
```

Prim's Algorithm: Parallel Formulation



The partitioning of the distance array *d* and the adjacency matrix *A* among *p* processes.

Prim’s Algorithm: Parallel Formulation

- The algorithm works in *n* iterations. In each iteration,
- The smallest non-marked entry in array d is computed using reduction.

  Computation time = $O\left(\frac{n}{p} + \log p\right)$

  Communication time = $O((\tau + \mu) \log p)$
- The selected node is broadcast.

  Communication time = $O((\tau + \mu) \log p)$
- Update the d vector.

  Computation time = $O\left(\frac{n}{p}\right)$
- Total parallel time is given by

  Computation time = $O\left(\frac{n^2}{p} + n \log p\right)$

  Communication time = $O\left((\tau + \mu)\, n \log p\right)$

Efficient up to $O\left(\frac{n}{\log n}\right)$ processors.

Single-Source Shortest Paths

- For a weighted graph $G = (V,E,w)$, the ***single-source shortest paths*** problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V
- Dijkstra's algorithm is similar to Prim's algorithm
- It maintains a **set of nodes** for which the shortest paths are known
- It grows this set based on the node closest to source using one of the nodes in the current shortest path set

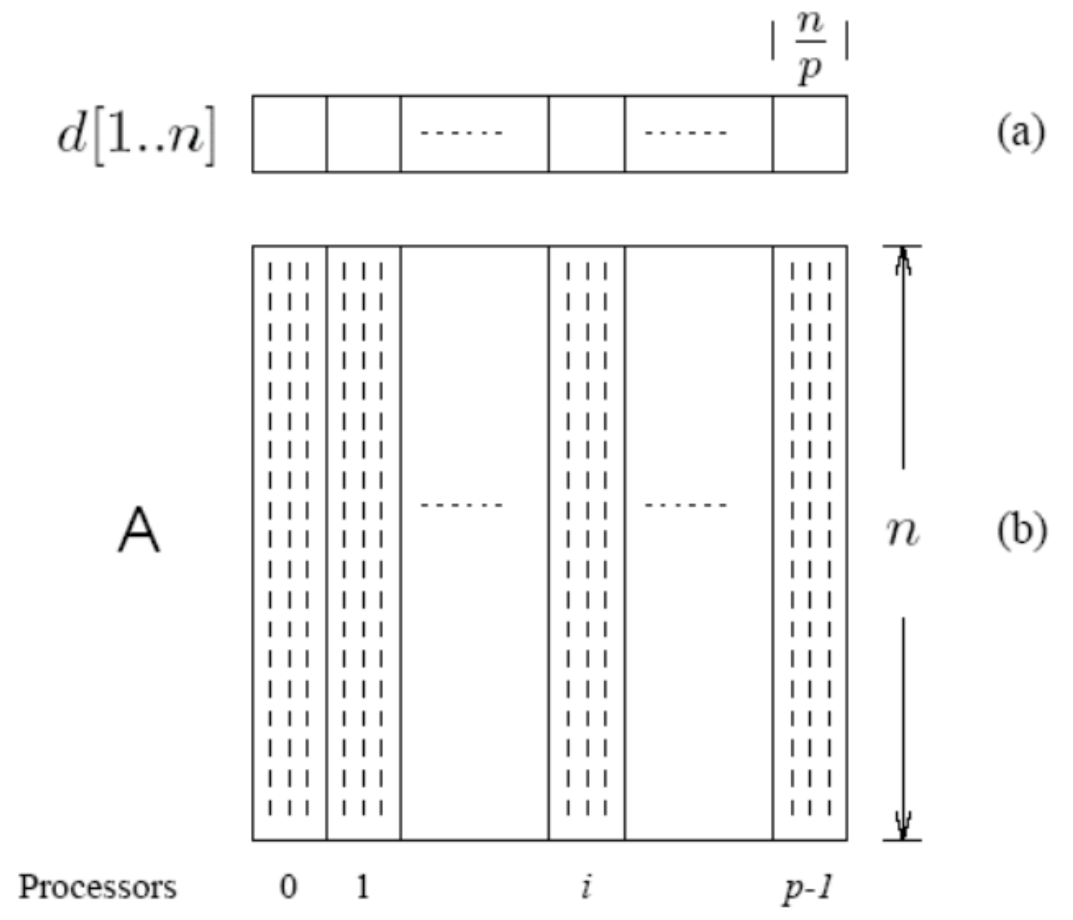
SSSP: Dijkstra’s Algorithm

```
procedure DIJKSTRA_SINGLE_SOURCE_SP($V, E, w, s$)
begin
    $V_T := \{s\}$;
    for all $v \in (V - V_T)$ do
        if $(s, v)$ exists set $l[v] := w(s, v)$;
        else set $l[v] := \infty$;
    while $V_T \neq V$ do
    begin
        find a vertex $u$ such that $l[u] := \min\{l[v] | v \in (V - V_T)\}$;
        $V_T := V_T \cup \{u\}$;
        for all $v \in (V - V_T)$ do
            $l[v] := \min\{l[v], \boxed{l[u] +} w(u, v)\}$;
    endwhile
end DIJKSTRA_SINGLE_SOURCE_SP
```

Dijkstra's sequential single-source shortest paths algorithm.

Dijkstra’s Algorithm: Parallel Formulation

- Very similar to the parallel formulation of Prim's algorithm for minimum spanning trees
- The weighted adjacency matrix is partitioned using the 1D block mapping
- The next node closest to the source is identified using **reduction**.
- The selected node is broadcast to all processors.
- The ℓ vector is updated using local computation.
- The parallel performance of Dijkstra's algorithm is **identical to that of Prim’s algorithm**



The partitioning of the distance array *d* and the adjacency matrix *A* among *p* processes.

Dijkstra on Sparse Graphs

- The two main steps in Dijkstra's algorithm are 1) the find to find the unmarked vertex with the smallest distance d[v], and 2) the relaxation step.
- Step 1) requires O(n) time on a dense vector d, and step 2) happens O(m) times, for total time $O(n^2 + m)$
- The theoretical miniumum uses the Fibonacci heap for step 1) in O(log n) and step 2) in O(1) each, for total time $O(n \log n + m)$.
- In practice, priority queues are usually better due to the constants in Fib heaps, for total runtime $O(n \log n + m \log n) + O(m \log n)$

All-Pairs Shortest Paths

Given a weighted graph *G(V,E,w)*, the ***all-pairs shortest paths*** problem is to find the shortest paths between all pairs of vertices.

A number of algorithms are known for solving this problem

- Matrix multiplication based
- Dijkstra's algorithm
- Floyd's algorithm

All-Pairs Shortest Paths: Matrix-Multiplication Based Algorithm

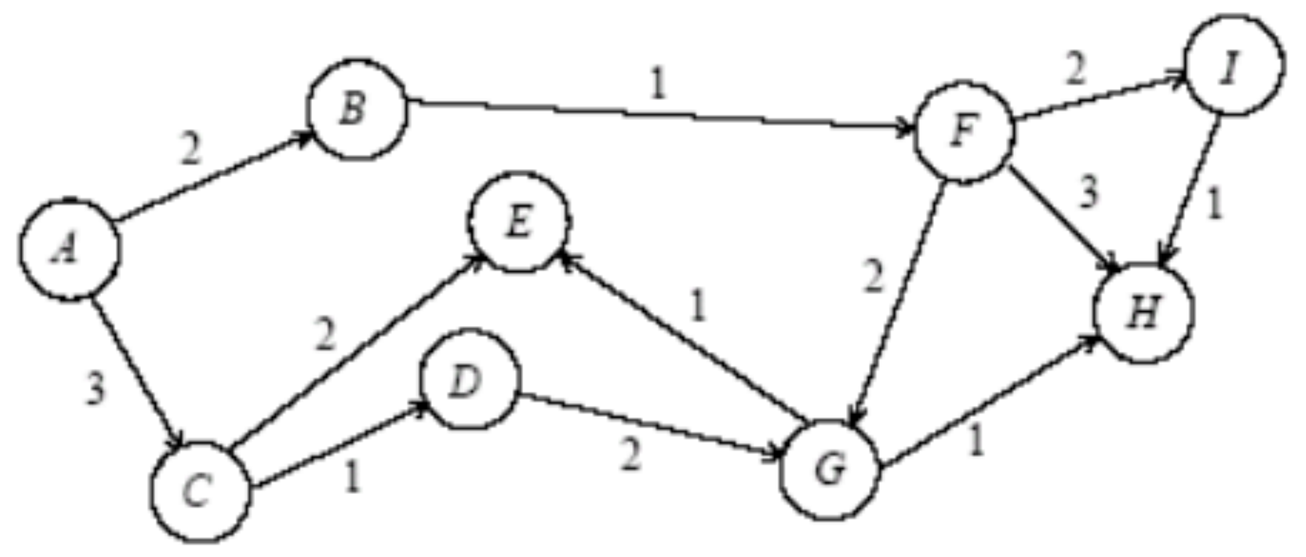
Consider the multiplication of the weighted adjacency matrix with itself

- except, in this case, we **replace the multiplication operation by addition**, and the **addition operation by minimization**.

Notice that the product of weighted adjacency matrix with itself returns a matrix that contains shortest paths of length 2 between any pair of nodes.

It follows from this argument that A^n contains all shortest paths.

Matrix-Multiplication-Based Algorithm



$$A^1 = \begin{pmatrix} 0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^2 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

$$A^4 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix} \quad A^8 = \begin{pmatrix} 0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\ \infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\ \infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\ \infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\ \infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\ \infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0 \end{pmatrix}$$

What if the number of vertices is not a power of 2

- If the number of vertices is not a power of 2, we need to multiply powers of 2 together to add up to the correct power of the matrix A.
- Given n vertices, write n in binary in O(log n) bits. Each 1 in the bit string will be one of the multiplied power-of-2 A matrices.
- For example, suppose n = 5 = 101. $A^4 \cdot A = A^5$

Matrix-Multiplication-Based Algorithm

- A^n is computed by doubling powers - i.e., as A, A^2, A^4, A^8, and so on.
- We need *log n* matrix multiplications, each taking time $O(n^3)$.
- The serial complexity of this algorithm is $O(n^3 \log n)$.
- This algorithm is not optimal, since the best known algorithms have complexity $O(n^3)$.

Matrix-Multiplication-Based Algorithm: Parallel Formulation

- Each of the $\log n$ matrix multiplications can be performed in parallel.
- Using Cannon's algorithm:
  - Computation time = $O\left(\frac{n^3 \log n}{p}\right)$
  - Communication time $O\left(\left(\tau\sqrt{p} + \mu\frac{n^2}{\sqrt{p}}\right)\log n\right)$

Not efficient with respect to the best serial algorithm.

Efficient with respect to the serial algorithm for $p = O(n^2)$ processors.

Dijkstra's Algorithm

- Execute *n* instances of the single-source shortest path problem, one for each of the *n* source vertices.
- Serial run-time complexity is $O(n^3)$.

Dijkstra's Algorithm: Parallel Formulation

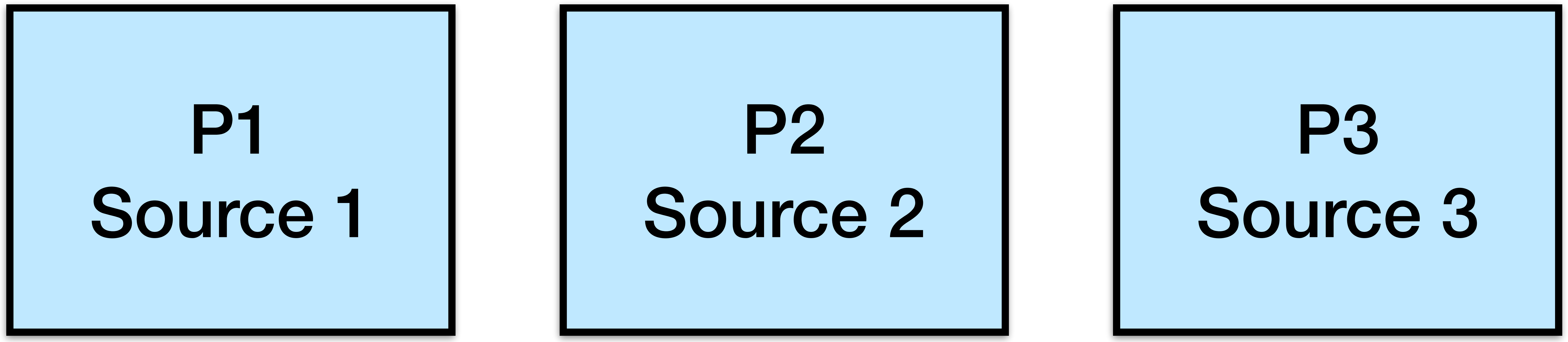
- Two parallelization strategies:
  - execute each of the *n* shortest path problems on different processors (source partitioned).
  - use the parallel formulation of the shortest path problem to increase number of processors that can be used (source parallel).

Use up to n processors

Each one can use up to n processors

Dijkstra's Algorithm: Source-Partitioned Formulation

- Use n processors. Processor P_i finds the shortest paths from vertex v_i to all other vertices by executing Dijkstra's sequential single-source shortest paths algorithm.
- It requires no inter-process communication (provided that the adjacency matrix is replicated on all processes).
- The parallel run time of this formulation is: $O(n^2)$.
- Perfectly efficient up to n processors (no communication involved whatsoever).



Dijkstra’s Algorithm: Source Parallel Formulation

- In this case, each of the shortest path problems is further executed in parallel. We can therefore use up to n^2 processors.
- Given p processors ($p > n$), each single source shortest path problem is executed by p/n processors.
- Using previous results, this takes:

$$\text{Computation time} = O\left(\frac{n^3}{p} + n \log p\right)$$

$$\text{Communication time} = O\left((\tau + \mu)\, n \log p\right)$$

Efficient up to $O\left(\frac{n^2}{\log n}\right)$ processors.

END OF FIRST LECTURE HERE

Floyd's Algorithm

- For any pair of vertices v_i, $v_j \in V$, consider all paths from v_i to v_j whose intermediate vertices belong to the set $\{v_1, v_2, \ldots, v_k\}$. Let $p_{i,j}^{(k)}$ (of weight $d_{i,j}^{(k)}$ be the minimum-weight path among them.
- If vertex v_k is not in the shortest path from v_i to v_j, then $p_{i,j}^{(k)}$ is the same as $p_{i,j}^{(k-1)}$.
- If v_k is in $p_{i,j}^{(k)}$, then we can break $p_{i,j}^{(k)}$ into two paths - one from v_i to v_k and one from v_k to v_j . Each of these paths uses vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$.

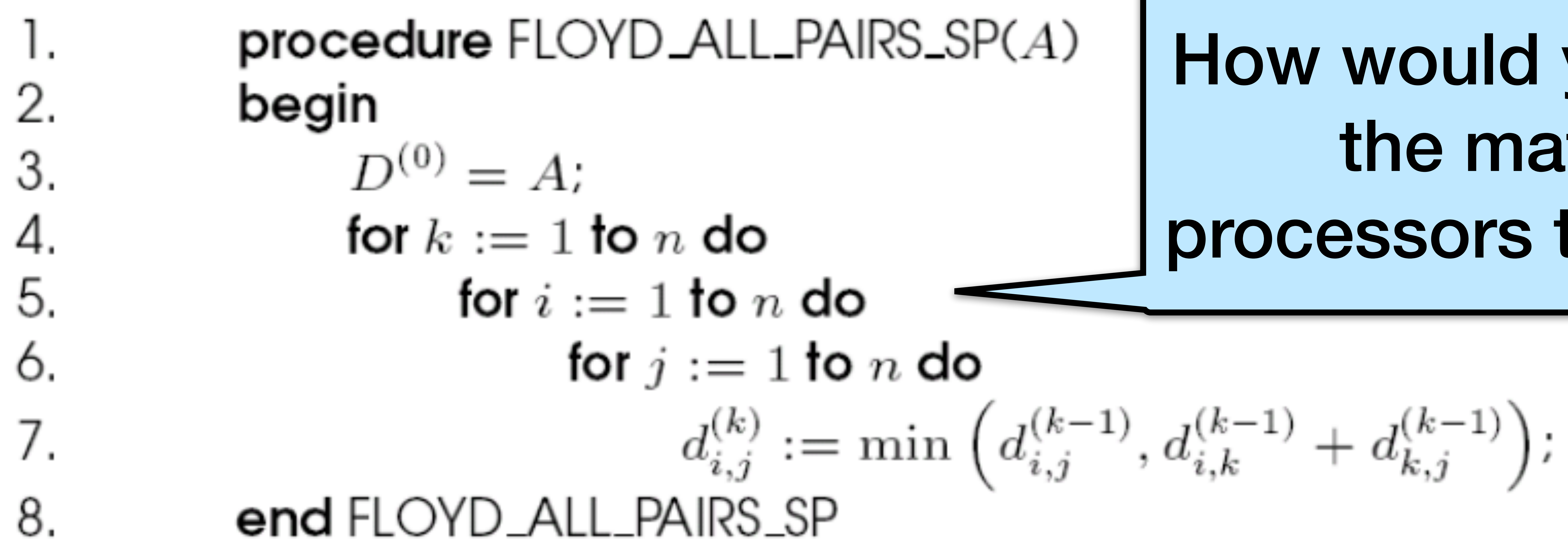
Floyd’s Algorithm

From our observations, the following recurrence relation follows:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\left\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

This equation must be computed for each pair of nodes and for *k = 1, n*. The serial complexity is *O(n^3)*.

Floyd’s Algorithm



```
1. procedure FLOYD_ALL_PAIRS_SP($A$)
2. begin
3.     $D^{(0)} = A$;
4.     for $k := 1$ to $n$ do
5.         for $i := 1$ to $n$ do
6.             for $j := 1$ to $n$ do
7.                 $d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$;
8. end FLOYD_ALL_PAIRS_SP
```

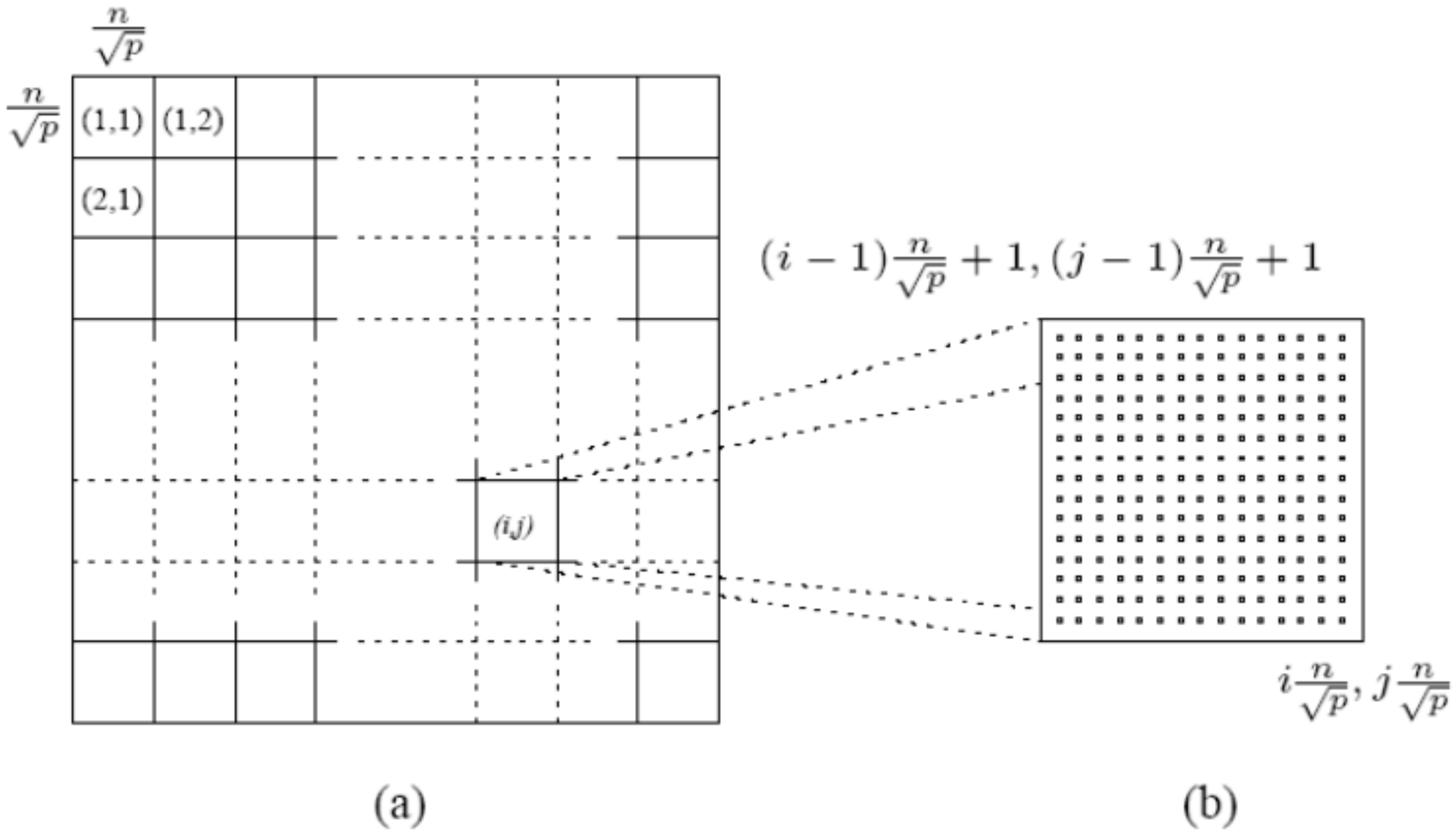
How would you divide up the matrix on p processors to parallelize?

Floyd's all-pairs shortest paths algorithm. This algorithm computes the all-pairs shortest paths of the graph *G = (V,E)* with adjacency matrix *A*.

Floyd’s Algorithm: Parallel Formulation Using 2D Block Mapping

- Matrix $D^{(k)}$ is divided into p blocks (submatrices) of size $(n / \sqrt{p})\ x\ (n / \sqrt{p})$.
- Each processor updates its part of the matrix during each iteration.
- To compute $d_{l,r}^{k}$ processor $P_{i,j}$ must get $d_{l,k}^{(k-1)}$ and $d_{k,r}^{(k-1)}$.
- In general, during the k^{th} iteration, each of the $\sqrt{p}$ processors containing part of the k^{th} row sends it to the $\sqrt{p}$ - 1 processors in the same column.
- Similarly, each of the $\sqrt{p}$ processors containing part of the k^{th} column sends it to the $\sqrt{p}$ - 1 processors in the same row.

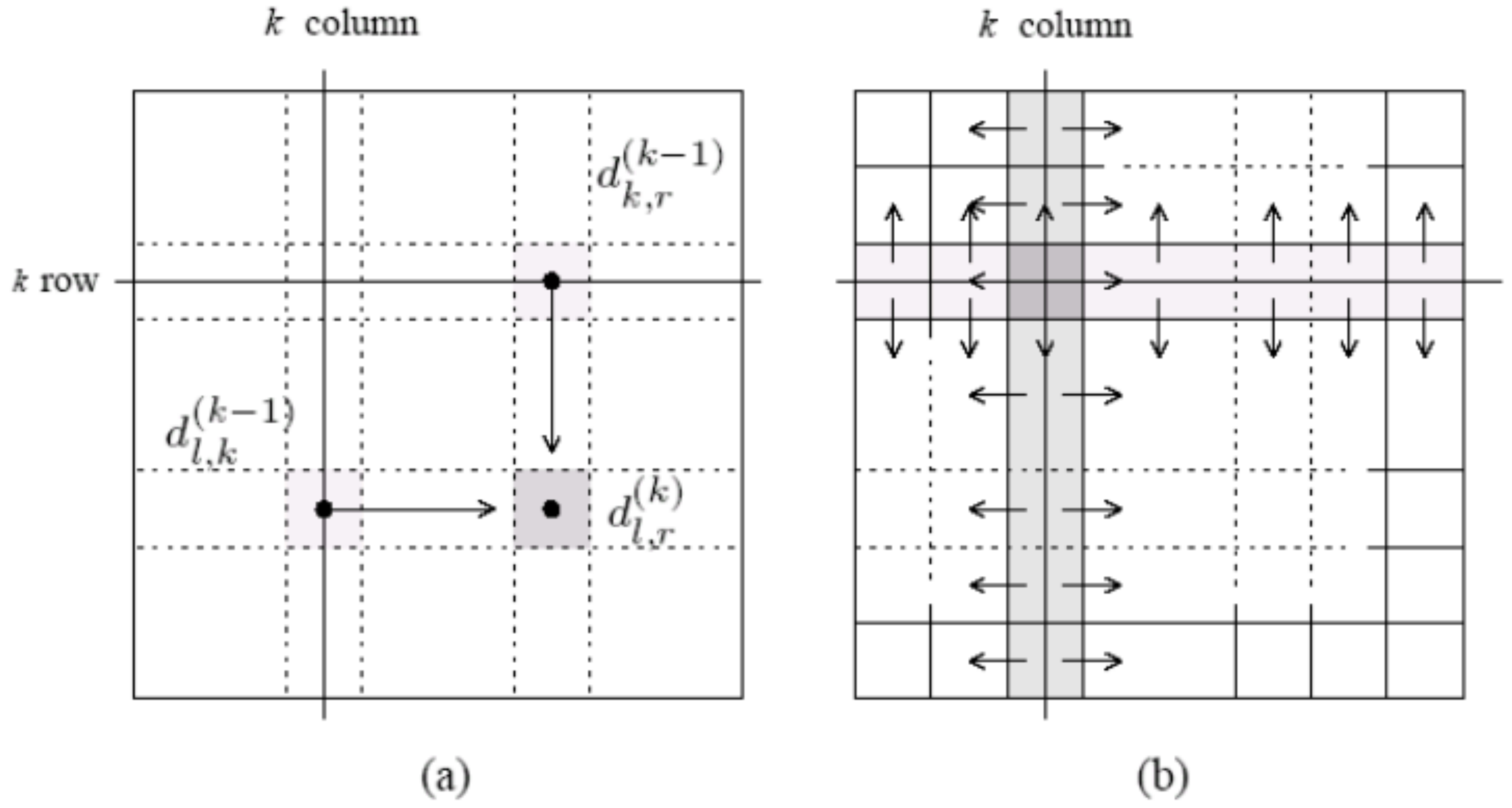
Floyd's Algorithm: Parallel Formulation Using 2D Block Mapping



$\frac{n}{\sqrt{p}}$
$\frac{n}{\sqrt{p}}$
(1,1)
(1,2)
(2,1)
(i,j)
$(i-1)\frac{n}{\sqrt{p}}+1, (j-1)\frac{n}{\sqrt{p}}+1$
$i\frac{n}{\sqrt{p}}, j\frac{n}{\sqrt{p}}$
(a)
(b)

(a) Matrix $D^{(k)}$ distributed by 2-D block mapping into $\sqrt{p}$ x $\sqrt{p}$ subblocks, and (b) the subblock of $D^{(k)}$ assigned to process $P_{i,j}$.

Floyd's Algorithm: Parallel Formulation Using 2D Block Mapping



(a) Communication patterns used in the 2-D block mapping. When computing $d_{l,r}^{(k)}$, information must be sent to the highlighted processor from two other processors along the same row and column. (b) The row and column of $\sqrt{p}$ processors that contain the k^{th} row and column send them along processor columns and rows.

Floyd's Algorithm: Parallel Formulation Using 2D Block Mapping

```
1.   procedure FLOYD_2DBLOCK($D^{(0)}$)
2.   begin
3.       for $k := 1$ to $n$ do
4.       begin
5.           each process $P_{i,j}$ that has a segment of the $k^{th}$ row of $D^{(k-1)}$;
                 broadcasts it to the $P_{*,j}$ processes;
6.           each process $P_{i,j}$ that has a segment of the $k^{th}$ column of $D^{(k-1)}$;
                 broadcasts it to the $P_{i,*}$ processes;
7.           each process waits to receive the needed segments;
8.           each process $P_{i,j}$ computes its part of the $D^{(k)}$ matrix;
9.       end
10.  end FLOYD_2DBLOCK
```

Floyd's parallel formulation using the 2-D block mapping. $P_{*,j}$ denotes all the processors in the j^{th} column, and $P_{i,*}$ denotes all the processors in the i^{th} row. The matrix $D^{(0)}$ is the adjacency matrix.

Floyd's Algorithm: Parallel Formulation Using 2D Block Mapping

- During each iteration of the algorithm, the processors containing segments of the *k*th row and *k*th column perform a one-to-all broadcast along their respective columns/rows. The message size of this broadcast is $n/\sqrt{p}$ elements.

- Once the communication is received, computing the local submatrix for the current iteration takes $O(n^2/p)$ time.

- Total parallel run-time:

  Computation time = $O\left(\frac{n^3}{p}\right)$

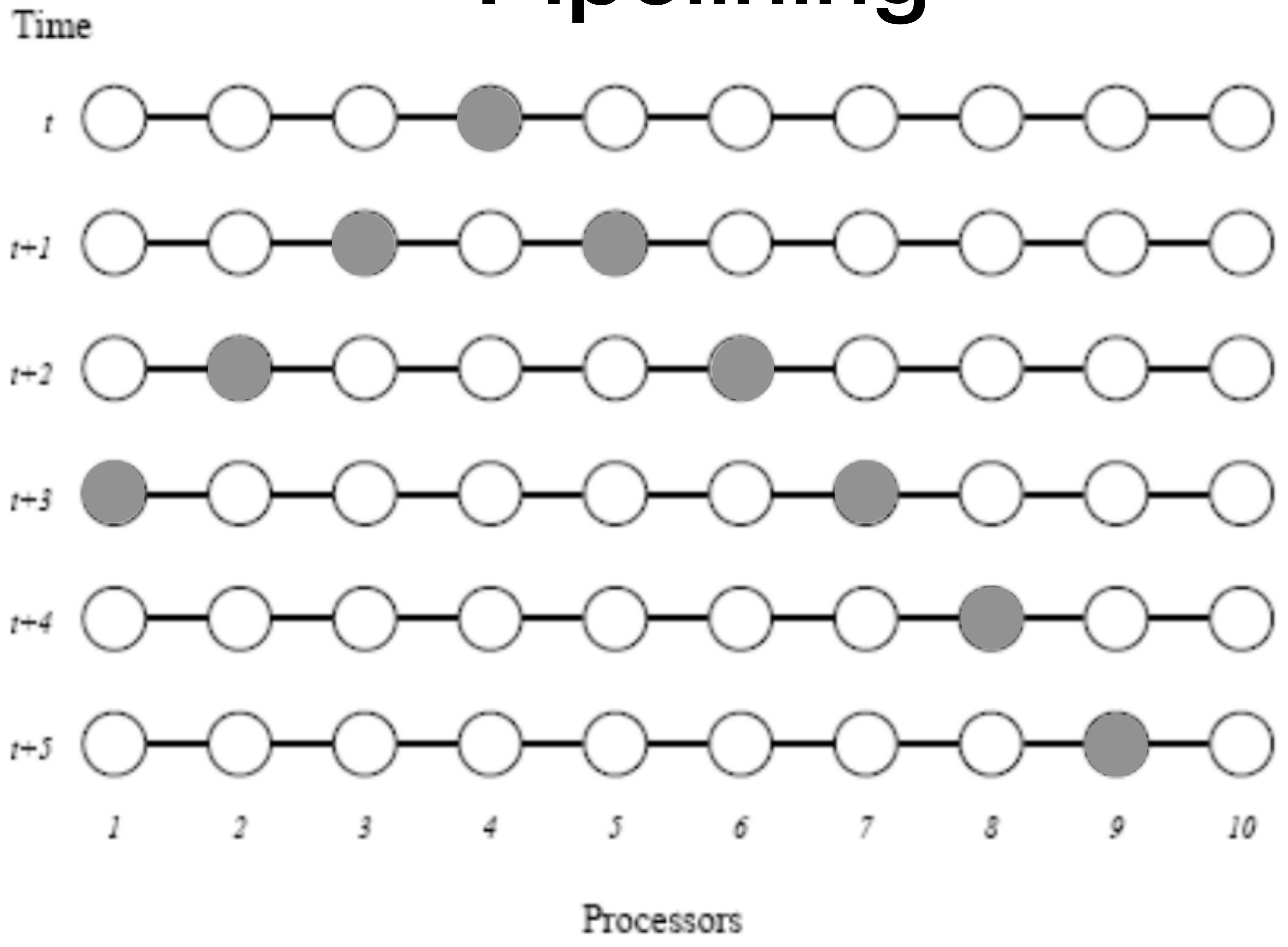
  Communication time $O\left(\left(\tau + \mu \frac{n}{\sqrt{p}}\right) n \log p\right)$

Efficient up to $O\left(\frac{n^2}{log^2 n}\right)$ processors.

Floyd's Algorithm: Speeding Thing Up By Pipelining

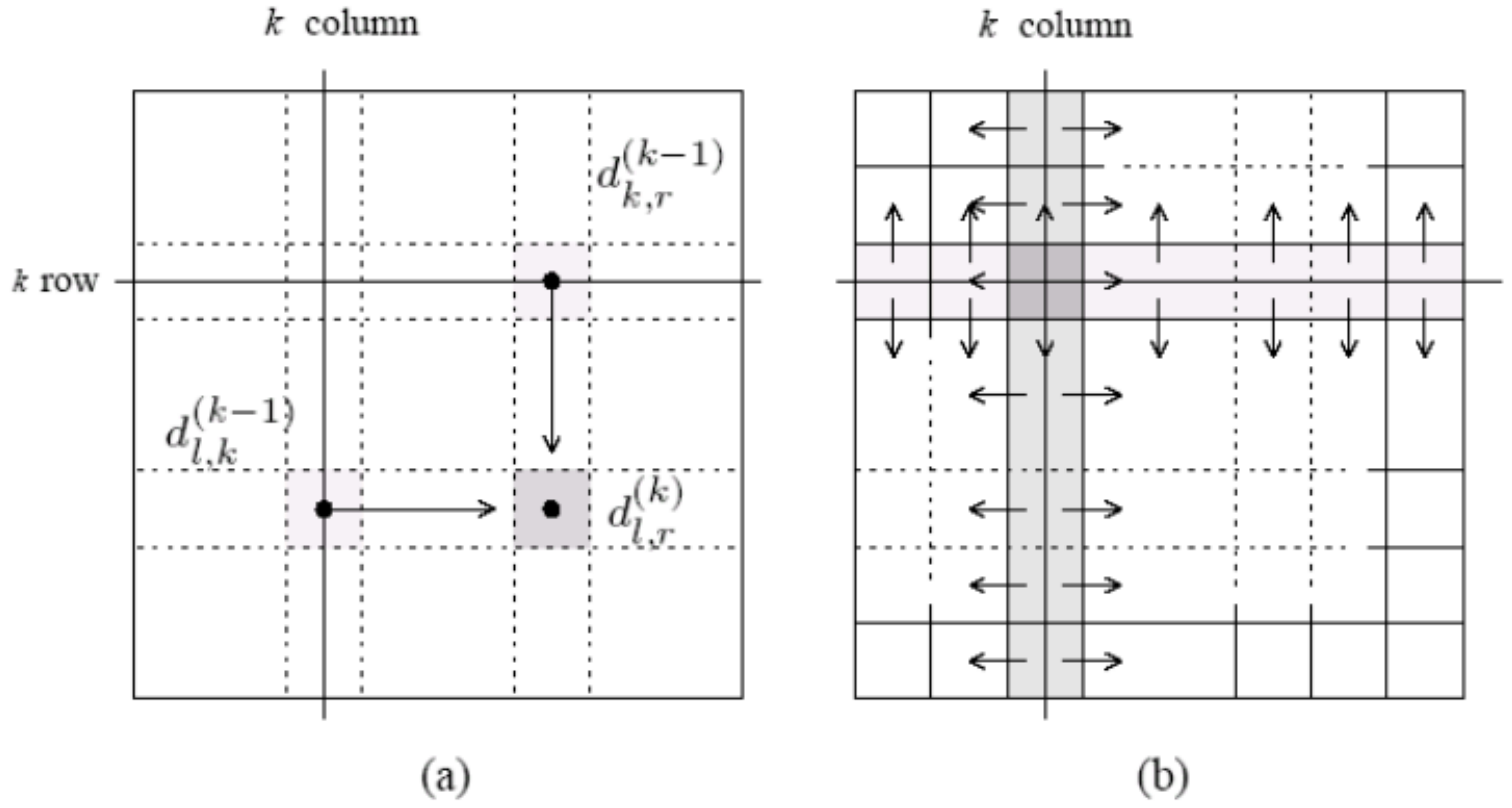
- The broadcast step in parallel Floyd's algorithm can be removed without affecting the correctness of the algorithm.
- A process starts working on the k^{th} iteration as soon as it has computed the $(k\text{-}1)^{th}$ iteration and has the relevant parts of the $D^{(k-1)}$ matrix.

Floyd's Algorithm: Speeding Thing Up By Pipelining



Communication protocol followed in the pipelined 2-D block mapping formulation of Floyd's algorithm. Assume that process 4 at time *t* has just computed a segment of the k^{th} column of the $D^{(k-1)}$ matrix. It sends the segment to processes 3 and 5. These processes receive the segment at time *t + 1* (where the time unit is the time it takes for a matrix segment to travel over the communication link between adjacent processes). Similarly, processes farther away from process 4 receive the segment later. Process 1 (at the boundary) does not forward the segment after receiving it.

Floyd's Algorithm: Parallel Formulation Using 2D Block Mapping



(a) Communication patterns used in the 2-D block mapping. When computing $d_{l,r}^{(k)}$, information must be sent to the highlighted processor from two other processors along the same row and column. (b) The row and column of $\sqrt{p}$ processors that contain the k^{th} row and column send them along processor columns and rows.

Floyd’s Algorithm: Speeding Thing Up By Pipelining

For $p = n^2$:

- Use the virtual mesh topology and use the mesh links for communicating elements.
- Send the received element to the neighbor and then compute. By the time the processor needs to broadcast, it would have received and passed along all the elements from previous processors in the same row/column.
- Communications for two iterations will not use the same links at the same time.
- Total number of time steps = $n + (n - 1)$ (iterations + pipelining)
- Run-time is $O(n)$ (unit computation and communication repeated $O(n)$ times).

Floyd’s Algorithm: Speeding Thing Up By Pipelining

For $p < n^2$:

Computation time = $O\left(\frac{n^3}{p} + n\right)$

Communication time $O\left(\left(\tau\, n + \mu \frac{n^2}{\sqrt{p}}\right)\right)$

Efficient for up to $O(n^2)$ processors.