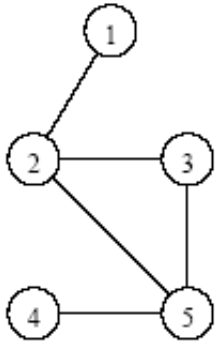
# Parallel Graph Algorithms

# Basic Definitions

- An *undirected graph G* is a pair *(V,E)*, where *V* is a finite set of points called *vertices* and *E* is a finite set of *edges*.
- An edge $e \in E$ is an **unordered** pair *(u,v)*, where *u and* $v \in V$.
- In a directed graph, the edge *e* is an **ordered** pair *(u,v)*. An edge *(u,v)* is *outgoing edge of* vertex *u* and is *incoming edge of* vertex *v*.
- A *path* from a vertex *v* to a vertex *u* is a sequence $<v_0, v_1, v_2, \ldots, v_k>$ of vertices, where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k-1$.
- The length of a path is defined as the number of edges in the path.

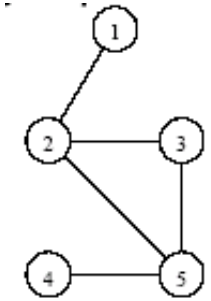
# Representations (Undirected Graphs)

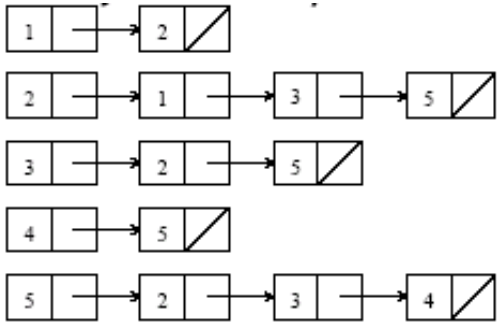
Adjacency matrix representation




$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

Adjacency list representation

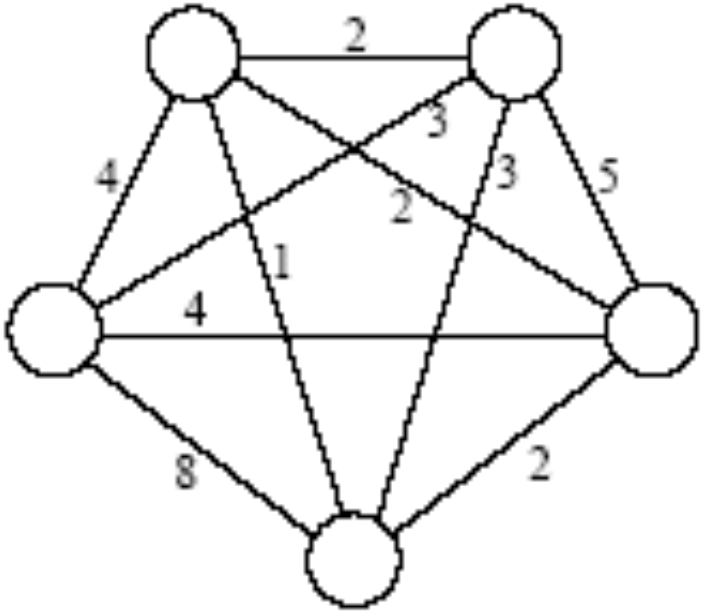


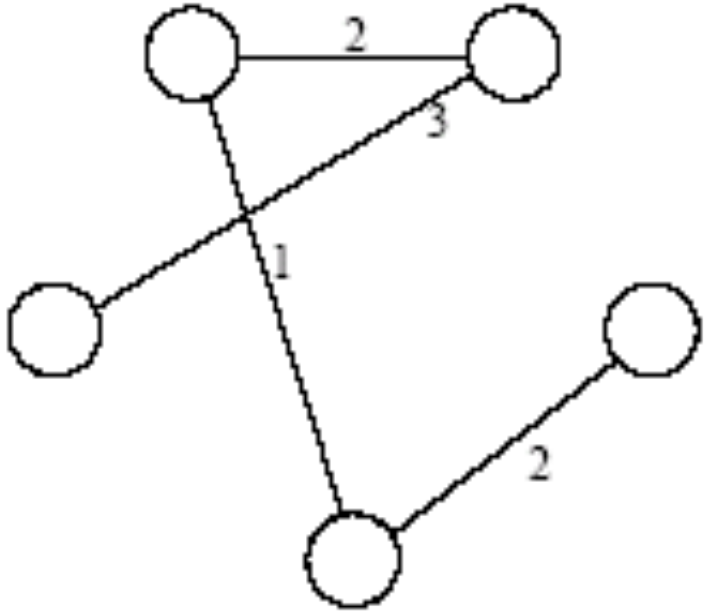



# Problem 1: **Minimum Spanning Tree**

- A *spanning tree* of an **undirected** graph *G* is a sub-graph of *G,* which is a tree containing all the vertices of *G*. So the spanning tree does not contain necessarily all the edges of *G* but a subset.
- In a weighted graph, the weight of a sub-graph is the sum of the weights of the edges in the sub-graph.
- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.

# In a Picture



# Relationship with **Traveling Salesman Problem (TSP)**

- Normally for TSP complete graphs are used (**there is always a route in between two cities no matter how long it takes**)
- An incomplete graph for MST can be completed by adding edges with a very large weight (note that this will not have any effect on the solution)
- A solution of the TSP yields a cycle with minimal weight. By deleting any edge this would result in a spanning tree
- So a solution of TSP cannot have less weight than the weight of the MST
- So the weight of MST is a lower bound on the weight of TSP

# Sequential Algorithms for MST

- Borůvka's algorithm (1926), Kruskal's algorithm (1956) and Prim's algorithm (1957)
- (Historical note) Borůvska's algorithm was used in 1926 to construct an efficient electricity network in Moravia (Czech Republic)*
- Kruskal's and Prim's algorithm are both based on the selecting a single lightest weight edge in each step of the algorithm

*The algorithm was rediscovered by Choquet in 1938;[4] again by Florek, Łukasiewicz, Perkal, Steinhaus, and Zubrzycki[5] in 1951; and again by **Sollin [6] in 1965**. Because Sollin was the only computer scientist in this list living in an English speaking country, this algorithm is frequently called Sollin's algorithm.

# Light-Edge Property

Given a weighted undirected graph G = (V, E), then for any cut set S (S $\subset$ E), the minimal weighted edge in S has to be an edge of the MST

A cut sets S cuts the graph into two sets U and V\U such that any path from a node x in U to a node y in V\U contains an edge from S

Proof: Assume we have a cut set S which contains an edge $e=(x,y)$ with minimal weight, which is not part of the MST. Then there is a path P in MST, which connects *x* and *y* and which does not contain *e*. So, because *x* and *y* are on opposite sides of *e*, next to *e* there must be an edge *e'* in S with *e'* on the path P. Now add *e* to the MST = MST', then *e* and *e'* are part of a cycle in MST'. Delete *e'* from MST', and we obtain another MST with a lesser weight ( $w(e) < w(e')$ ). Contradiction.

# Kruskal's Algorithm

As decribed by Kruskal in 1956:

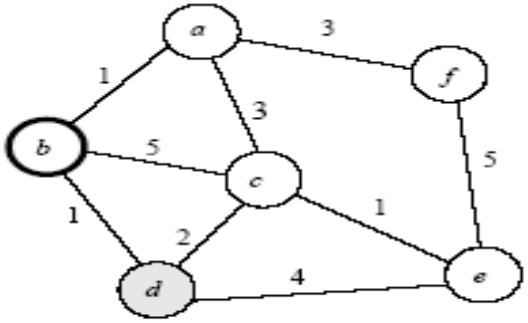
> "Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen"

# Prim's Algorithm

PRIM_MST(V, E, w, r): Given V, E, and w weight function, build MST starting from vertex r

```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that d[u] := min{d[v]|v ∈ (V − V_T)};
11.           V_T := V_T ∪ {u};
12.           for all v ∈ (V − V_T) do
13.               d[v] := min{d[v], w(u, v)};
14.       endwhile
15.   end PRIM_MST
```

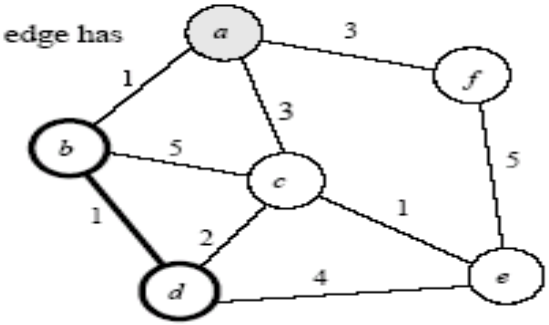
(a) Original graph




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 5 | 1 | ∞ | ∞ |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 3 | ∞ | ∞ | ∞ | 5 | 0 |

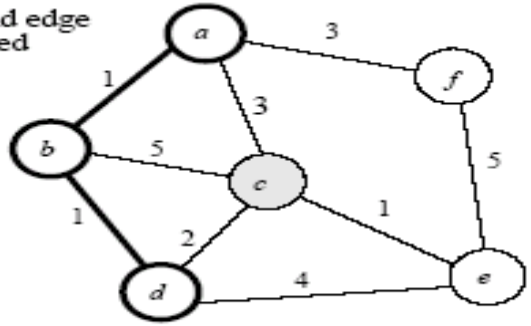
(b) After the first edge has been selected




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 4 | ∞ |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 3 | ∞ | ∞ | ∞ | 5 | 0 |

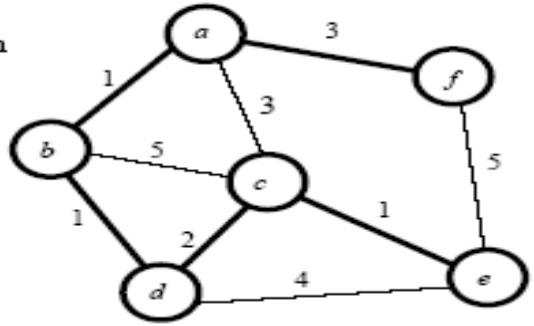
(c) After the second edge has been selected




| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 4 | 3 |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 3 | ∞ | ∞ | ∞ | 5 | 0 |

(d) Final minimum spanning tree

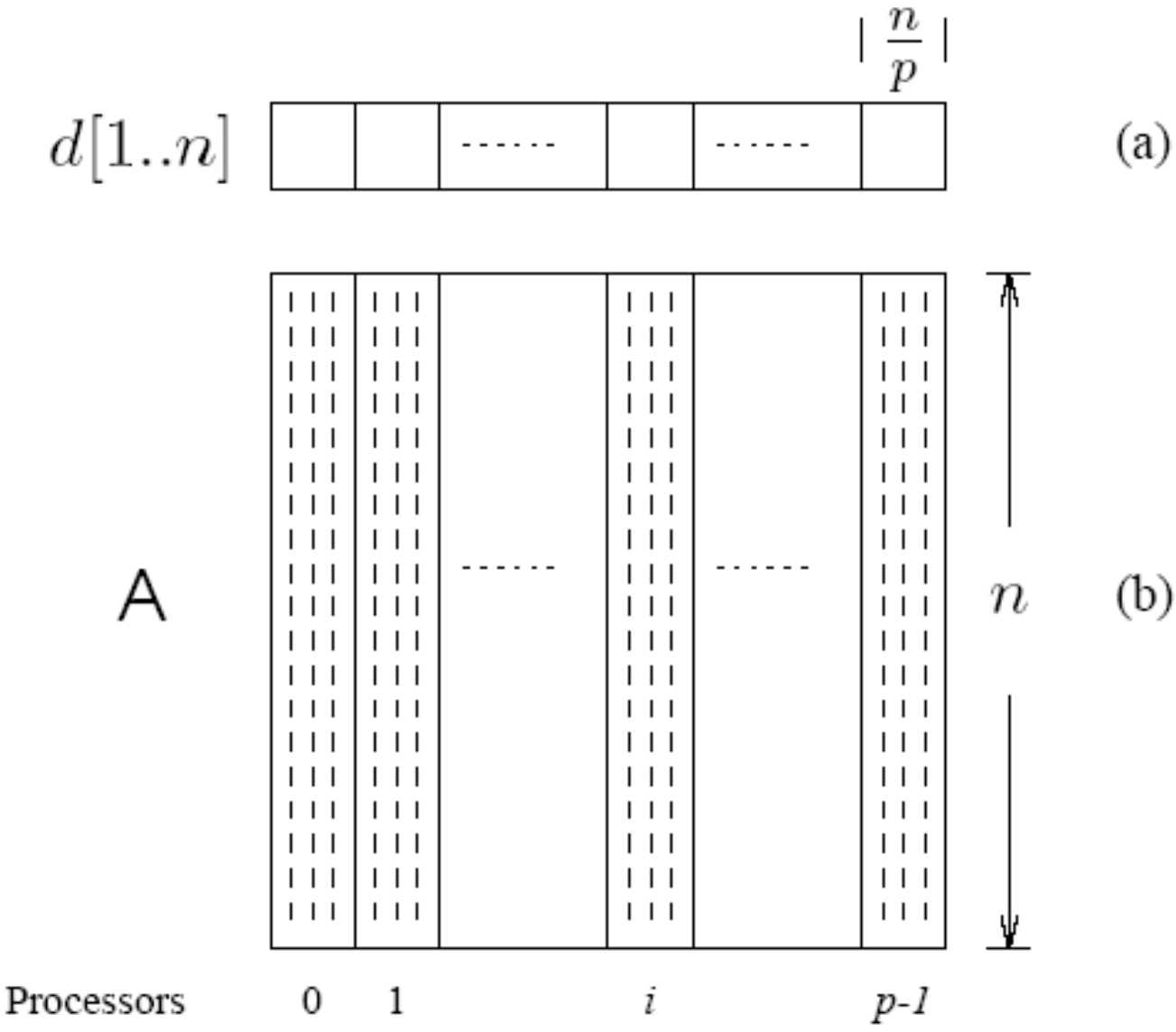



| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| d[] | 1 | 0 | 2 | 1 | 1 | 3 |

| | a | b | c | d | e | f |
|---|---|---|---|---|---|---|
| a | 0 | 1 | 3 | ∞ | ∞ | 3 |
| b | 1 | 0 | 5 | 1 | ∞ | ∞ |
| c | 3 | 5 | 0 | 2 | 1 | ∞ |
| d | ∞ | 1 | 2 | 0 | 4 | ∞ |
| e | ∞ | ∞ | 1 | 4 | 0 | 5 |
| f | 3 | ∞ | ∞ | ∞ | 5 | 0 |

# Prim's Algorithm: Parallel Formulation

- The algorithm works in *n* outer iterations - it is hard to execute these iterations concurrently.
- The inner loop is relatively easy to parallelize. Let *p* be the number of processes, and let *n* be the number of vertices.
- The adjacency matrix is partitioned in a 1-D block fashion (column slices), with distance vector *d* partitioned accordingly. See next slide.
- In each step, each processor selects the locally closest node, followed by a global reduction to select globally closest node.
- This node is inserted into MST, and the choice is broadcasted to all processors.
- Each processor updates its part of the *d* vector locally.


$\frac{n}{p}$
$d[1..n]$
(a)
A
$n$
(b)
Processors
0
1
$i$
$p$-$1$

# Computational Aspects

- The cost to select the minimum entry is $O(n/p + \log p)$.
- The cost of a broadcast is $O(\log p)$.
- The cost of local update of the $d$ vector is $O(n/p)$.
- The parallel time per iteration is $O(n/p + \log p)$.
- The total parallel time ($n$ iterations) is given by $O(n^2/p + n \log p)$.

# Borůvka's Algorithm (1926)

While there are edges remaining:

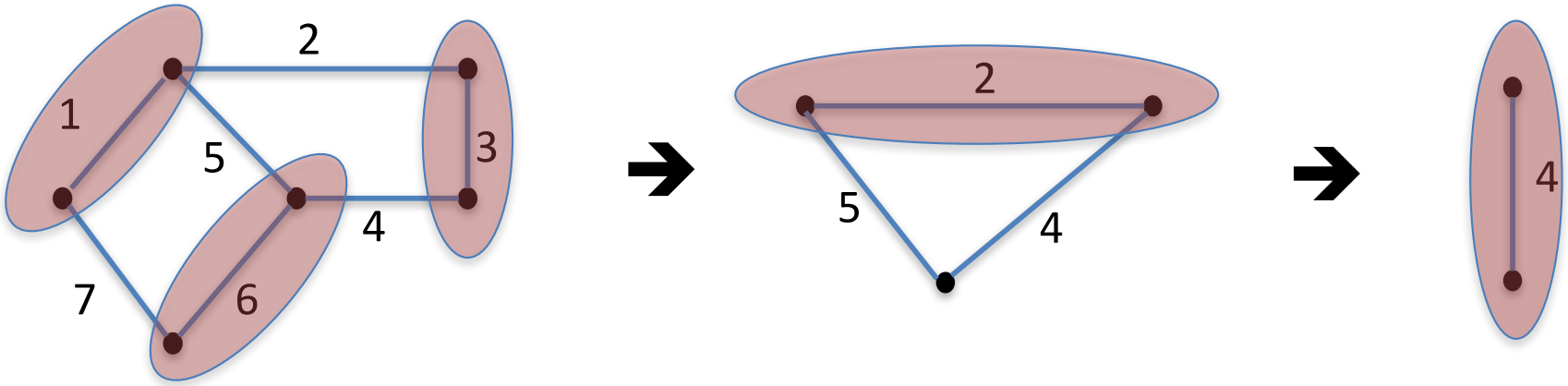
(1) select the minimum weight edge out of each vertex and contract each connected component defined by these edges into a vertex;

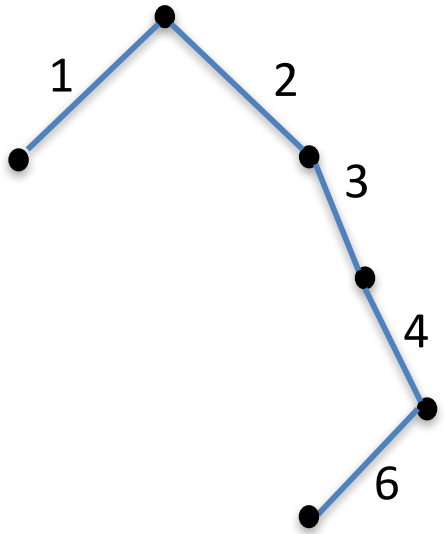
(2) remove self edges, and when there are redundant edges keep the minimum weight edge; and

(3) add all selected edges to the MST.

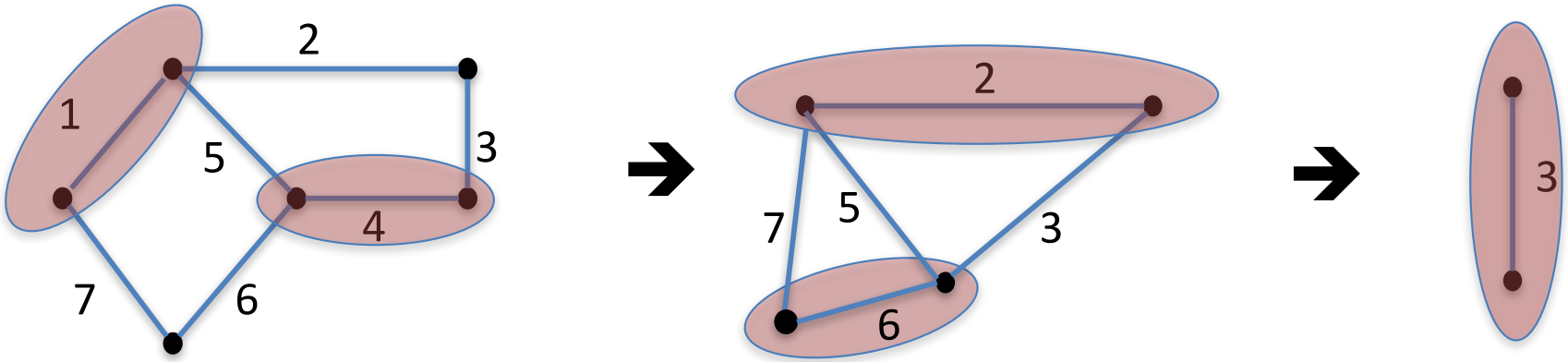
Note that this formulation is inherently parallel while computers were not invented at that time, or maybe **because** computers were not invented yet

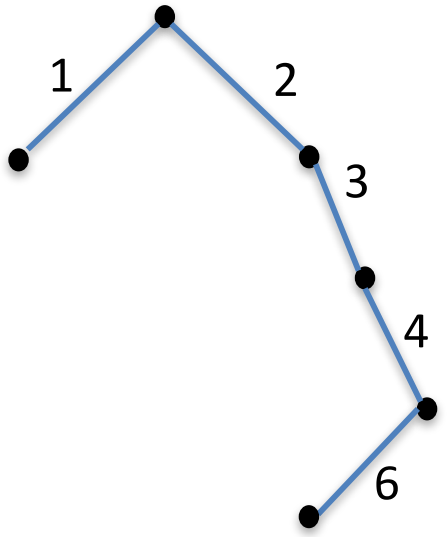
# Example



# Example (other execution order)




**The Same!!!!**

# Notes to Borůvka's Algorithm

- At each step the contractions of nodes *u* and *v* with (*u*,*v*) a minimal edge can be executed in parallel with the contraction of nodes *x* and *w* with (*x*,*w*) a minimal edge, if $v \neq x$ and $u \neq w$. (Note, $u \neq x$ and $v \neq w$ automatically holds)
- So at each step at least ½ |V| vertices are eliminated ➔ at most log (n) steps are required
- However, also the amount of available parallelism is reduced by an half after each step ➔ uneven load balance

# Input Data Partitioning

- Recall separator sets (nested dissection) for undirected graphs, based on levellization (BFS).
- The set of nodes V is divided into P disjoint subsets and separator sets:

$$V = V_1 \cup S_2 \cup V_2 \cup S_3 \ldots S_P \cup V_P$$

P = number of processors and $|V_i|$ about equal for all i

- Distribute the edges E such that each processor i has

$E_i = \{ (u,v) \mid u \in V_i \text{ and } v \in V_i \}$, and

$Left\_E_i = \{ (u,v) \mid u \in S_i \text{ and } v \in V_i \}$, and

$Right\_E_i = \{ (u,v) \mid u \in V_i \text{ and } v \in S_{i+1} \}$

➔ First phase every processor computes in parallel an MST for each $E_i$

➔ Second these partial MST's are knitted together by "synchronizing" the choice of minimum weight edge of $Right\_E_i$ with $Left\_E_{i+1}$

# Problem 2: **Single-Source Shortest Paths**

- For a weighted graph $G = (V,E,w,s)$, the *single-source shortest paths* problem is to find the shortest paths from a vertex $s \in V$ to all other vertices in $V$ ($w$ is the weigth function of the edges).
- 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.

# Dijkstra's Algorithm

```
1.   procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.   begin
3.       V_T := {s};
4.       for all v ∈ (V − V_T) do
5.           if (s, v) exists set l[v] := w(s, v);
6.           else set l[v] := ∞;
7.       while V_T ≠ V do
8.       begin
9.           find a vertex u such that l[u] := min{l[v]|v ∈ (V − V_T)};
10.          V_T := V_T ∪ {u};
11.          for all v ∈ (V − V_T) do
12.              l[v] := min{l[v], l[u] + w(u, v)};
13.      endwhile
14.  end DIJKSTRA_SINGLE_SOURCE_SP
```

# Similarities!!!!!!!!

## Prim's Algorithm for MST

**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**

## Dijkstra's Algorithm for Single Source Shortest Path

**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**
$l[v] := \min\{l[v], l[u] + w(u, v)\}$;
**endwhile**

# 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 1-D block mapping (column slicing).
- Each process selects, locally, the node closest to the source, followed by a global reduction to select next node.
- The node is broadcast to all processors and the *l*-vector updated.

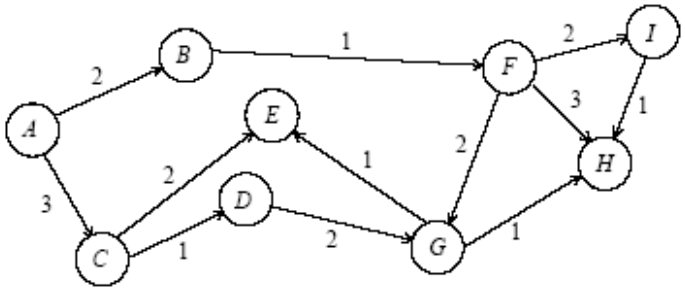
# Problem 3: **All-Pairs Shortest Paths**

- 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 $v_i$, $v_j \in V$.
- A number of algorithms are known for solving this problem: Matrix-Multiplication Based algorithm, Dijkstra's algorithm, Floyd's algorithm.

# Matrix-Multiplication Based Algorithm

- Consider the multiplication of the weighted adjacency matrix with itself - except, in this case, we replace the multiplication operation in matrix multiplication 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.

# In a Picture




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

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

# Computational Aspects

- For (semi) complete graphs and sequential execution:
  - $A^n$ is computed by doubling powers - i.e., as $A$, $A^2$, $A^4$, $A^8$, and so on.
  - We need *log n* (dense) matrix multiplications, each taking time $O(n^3)$.
  - The serial complexity of this procedure is $O(n^3 \log n)$.
- For (semi) complete graphs and parallel execution:
  - Each of the *log n* matrix multiplications can be performed in parallel.
  - We can use $n^3$ processors to compute each matrix-matrix product in time *log n*.
  - The entire process takes $O(\log^2 n)$ time.

Note that for incomplete graphs (leading to sparse matrices) this complexity does not change very much, because sparse x sparse matrix multiply very easily lead to full matrices.

# Dijkstra's Algorithm for All-Pairs Shortest Paths

## Sequential Execution:

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

## Parallel Execution:

- execute each of the *n* shortest path problems on a different processor (source partitioned), or
- use a parallel formulation of the shortest path problem to increase concurrency (source parallel)

# Source Partitioned Formulation

- Use $n$ processors, each 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 interprocess communication (provided that the adjacency matrix is replicated at all processes).
- The parallel run time of this formulation is: $O(n^2)$. $O(n^2)$ is the same time complexity as Prim's algorithm.
- While the algorithm is cost optimal, it can only use $n$ processors.

# 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.

# 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 $\{v_1, v_2, \ldots, v_{k-1}\}$.

# As a consequence:

From these 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 $O(n^3)$.

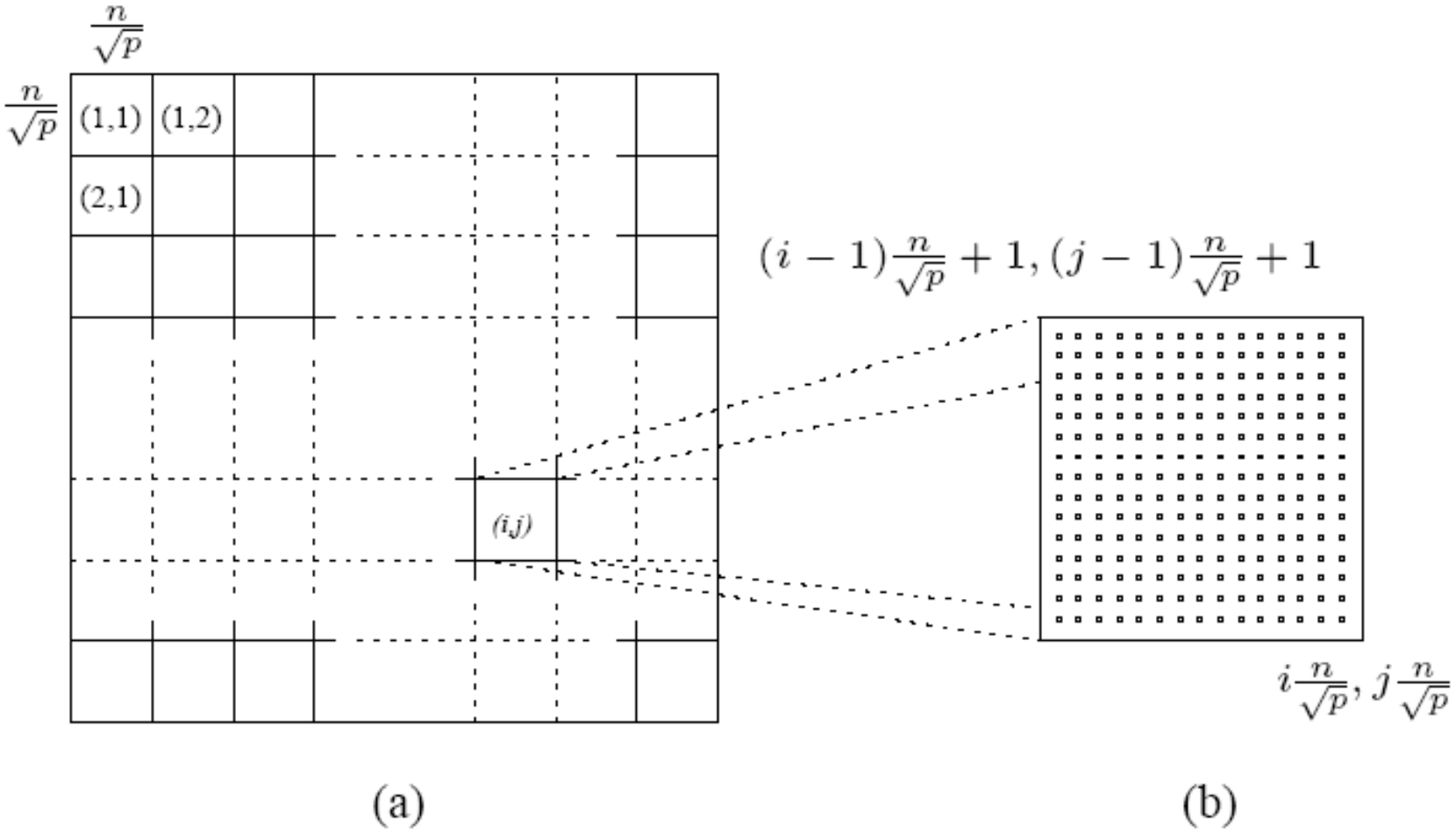
# In (pseudo) code

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

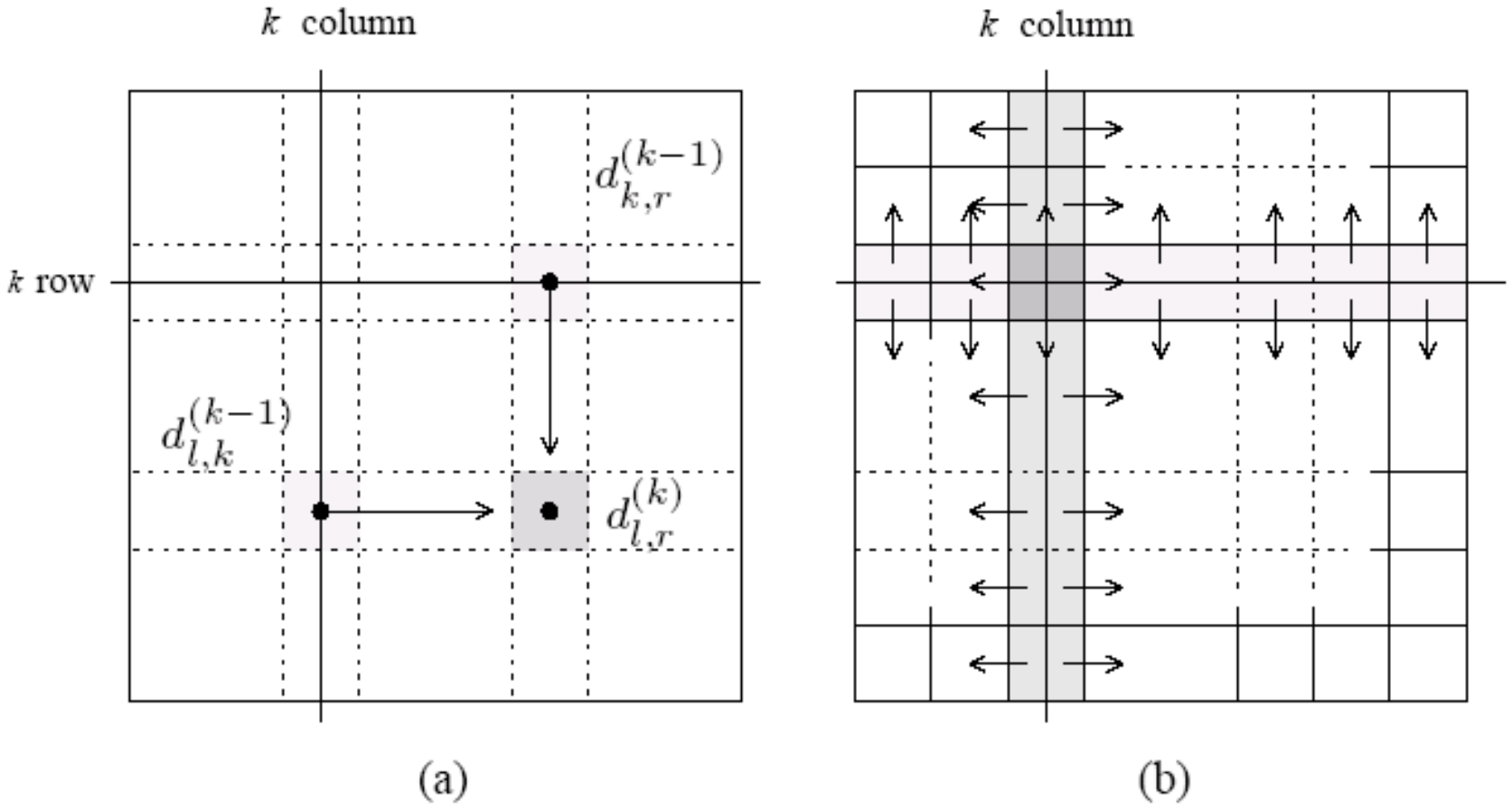
# Floyd's Algorithm: Parallel Execution

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

# In a Picture


$\frac{n}{\sqrt{p}}$
$\frac{n}{\sqrt{p}}$
(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)

# In a Picture: continued



# In (pseudo) code

```
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
```

# Computational Aspects

- During each iteration of the algorithm, the $k^{th}$ row and $k^{th}$ column of processors perform a one-to-all broadcast along their rows/columns.
- The size of this broadcast is 2 times $n/\sqrt{p}$ elements, taking time $O((n \log p)/\sqrt{p})$.
- The synchronization step takes time $O(\log p)$, *so neglicible*.
- The computation time is $O(n^2/p)$.
- The total parallel run time ($n$ step) of the 2-D block mapping formulation of Floyd's algorithm is giving a total of $O(n^3/p) + O(n^2 \log p/\sqrt{p})$