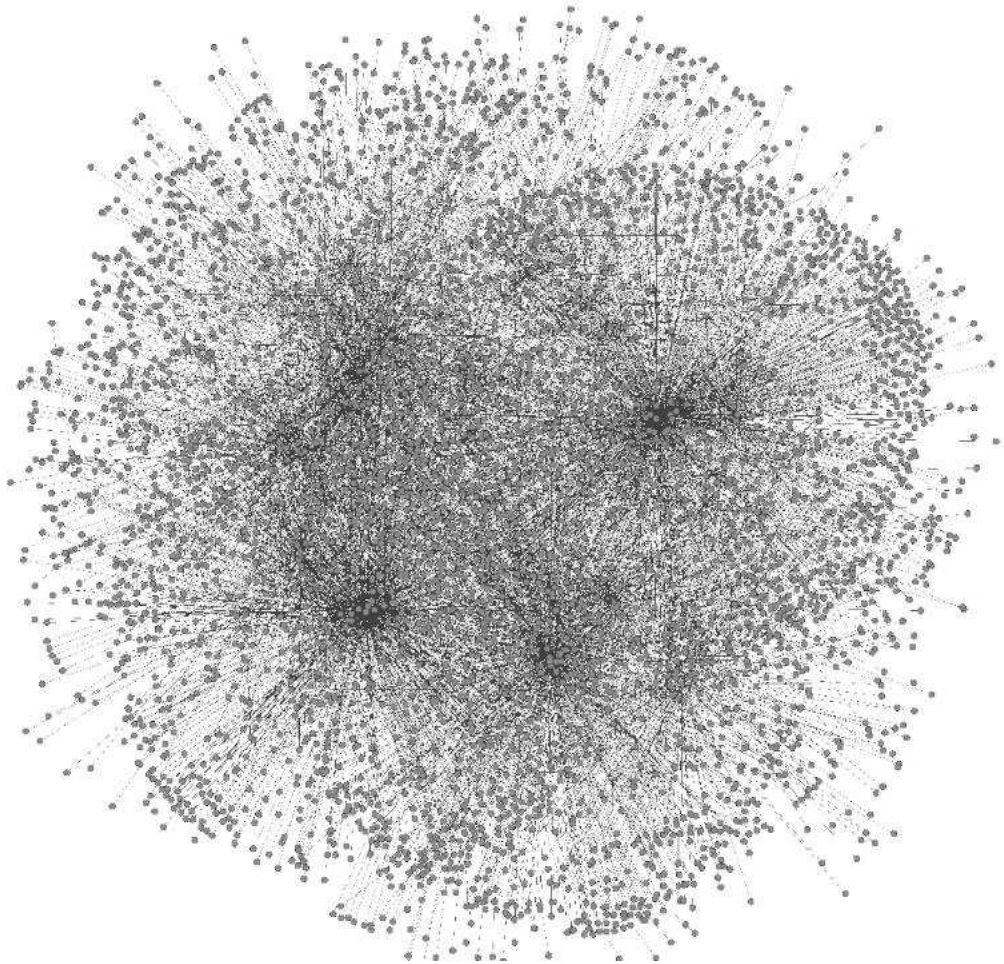
Multicore Computing
Lecture19 – Parallel Graph Processing

남 범 석

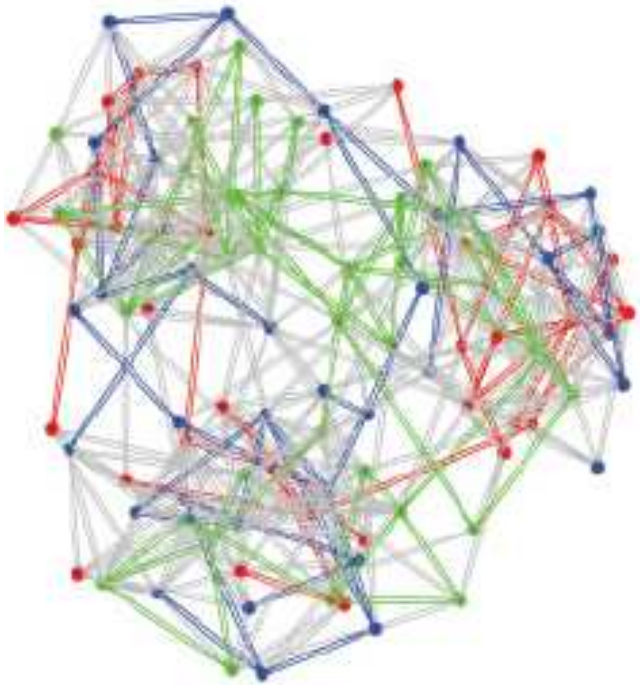
bnam@skku.edu

Parallel Graph Processing

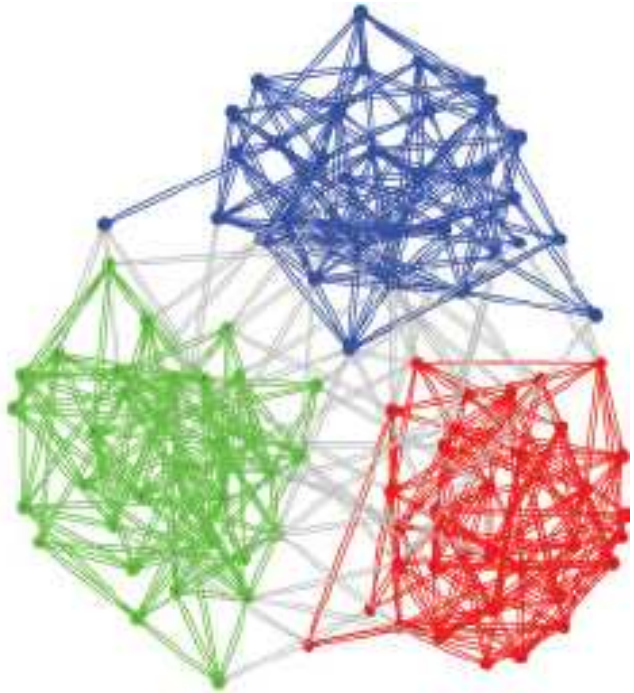
- Large Social Network Graph
  - Too large to store in a single machine



- Graph Partitioning and Clustering



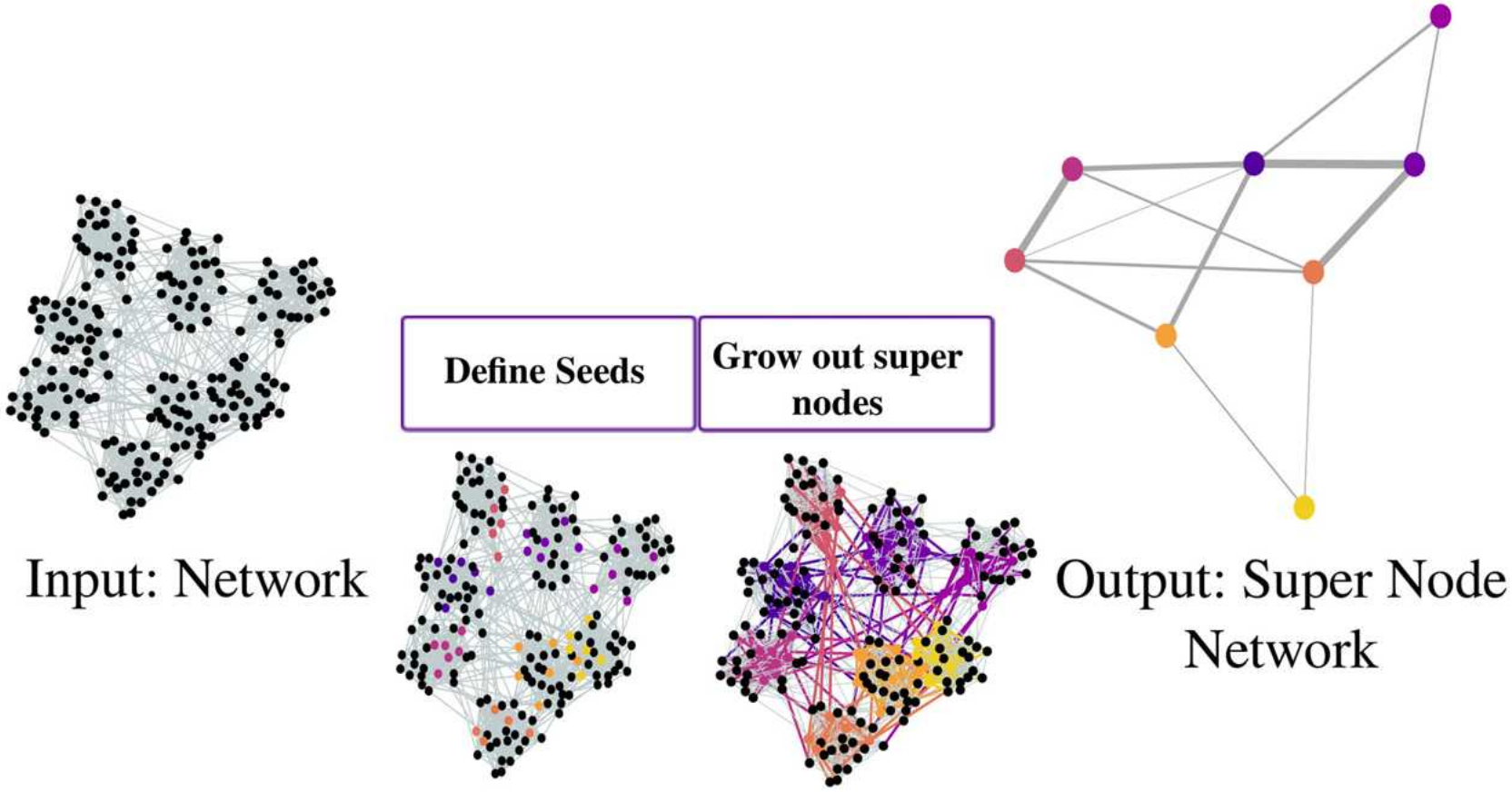
(a) A poor edge-cut partitioning. Vertices are assigned to partitions at random, thus, there are many inter-partition links.



(b) A good edge-cut of the same graph, where vertices that are highly connected are assigned to the same partition.

Parallel Graph Processing

- Lots of interesting on-going research in this area



Back-to-the-Basic

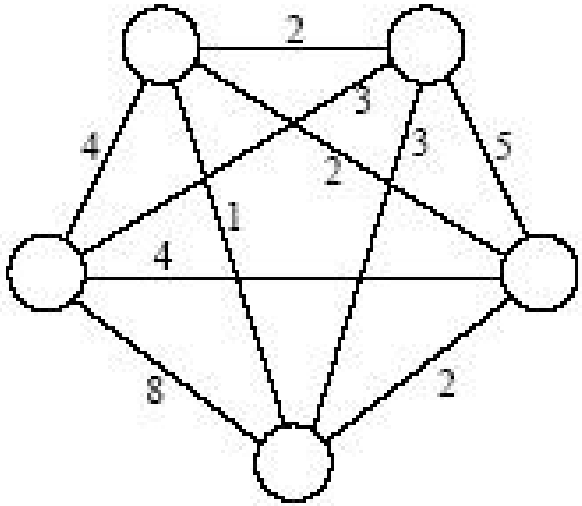
- Minimum Spanning Tree
  - Prim's algorithm
- Single-Source Shortest Path
  - Dijkstra's algorithm
- All-pair Shortest Path
  - Floyd's algorithm

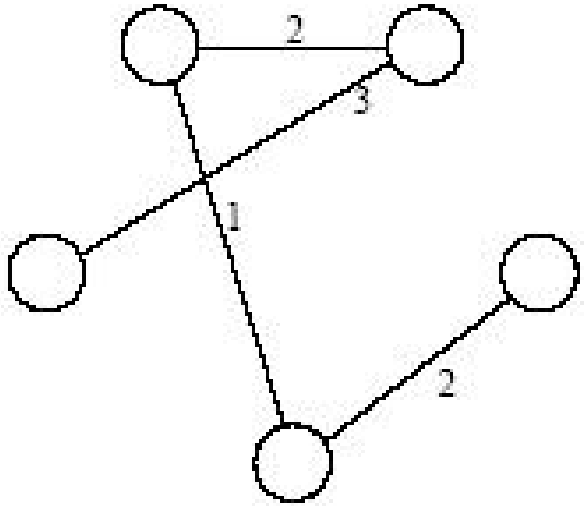
Minimum Spanning Tree

- A spanning tree of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G.
- 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.

Minimum Spanning Tree

- An undirected graph and its minimum spanning tree.



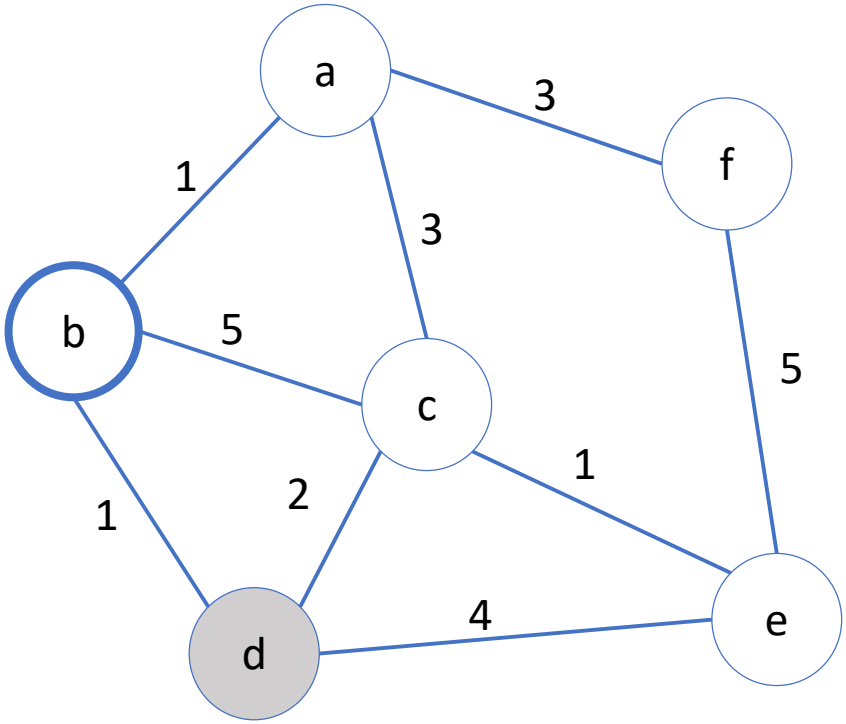


Minimum Spanning Tree: Prim's Algorithm

- Prim's algorithm for finding an MST is a greedy algorithm.
- Start by selecting an arbitrary vertex, include it into the current MST.
- Grow the current MST by inserting into the vertex closest to one of the vertices already in current MST.

Minimum Spanning Tree: Prim's Algorithm

(a) Start from b

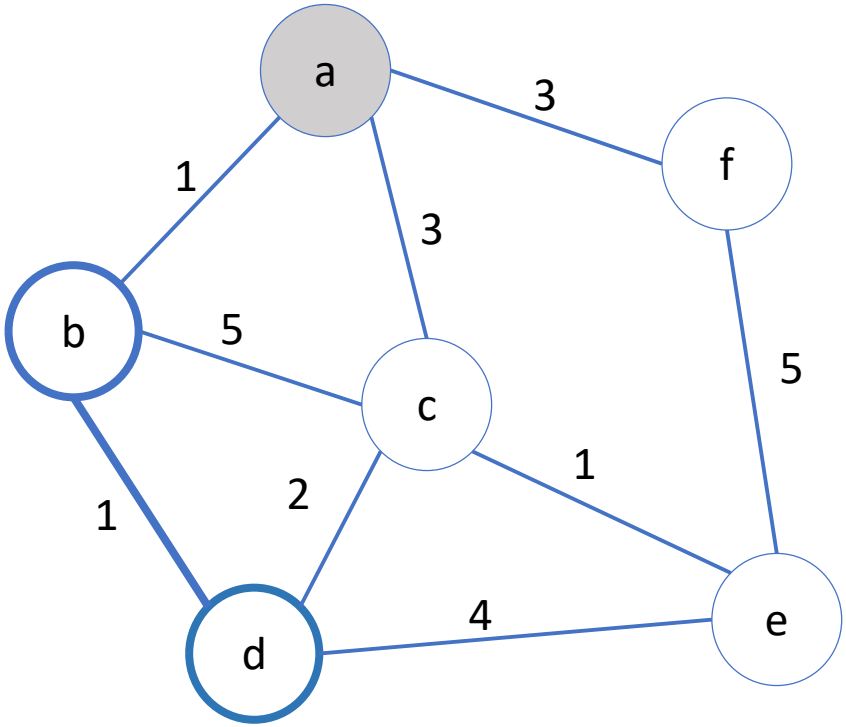


	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

(b) After the first edge is selected

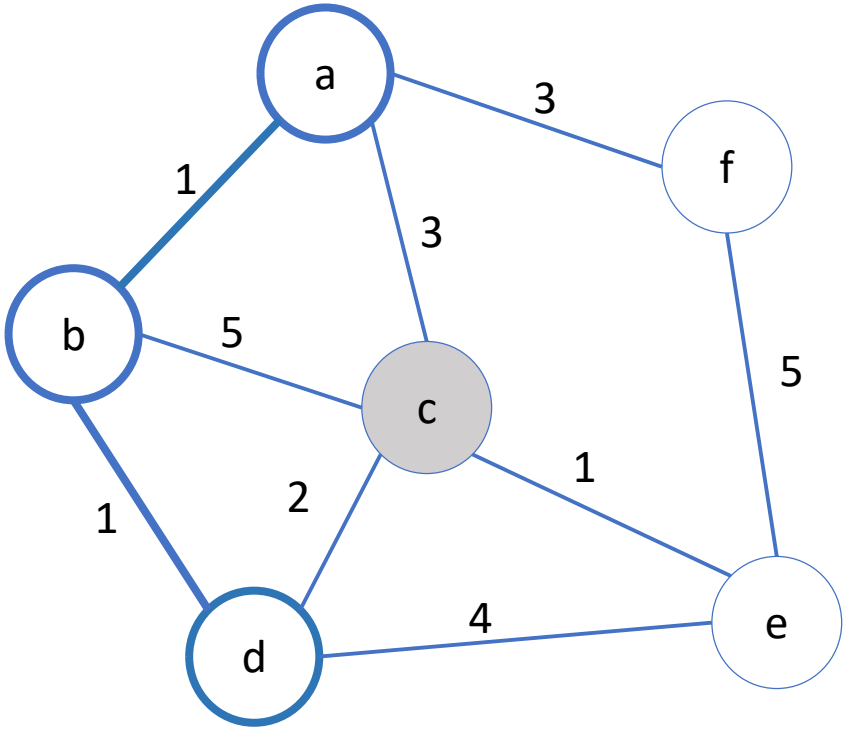


	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

(c) After the second edge is selected

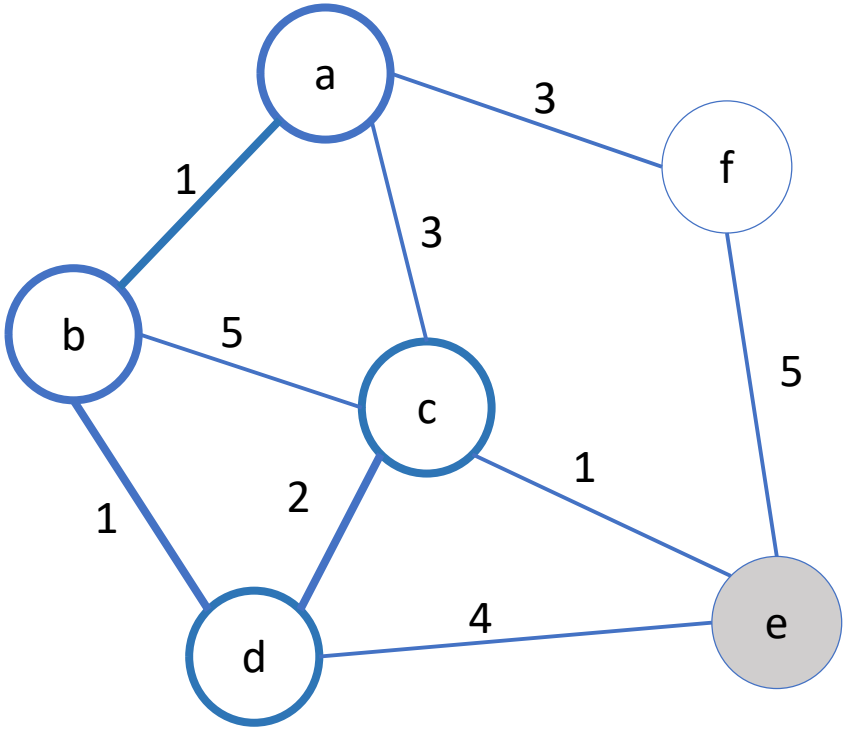


	a	b	c	d	e	f
d[]	1	0	2	1	4	3

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

(d) After the third edge is selected

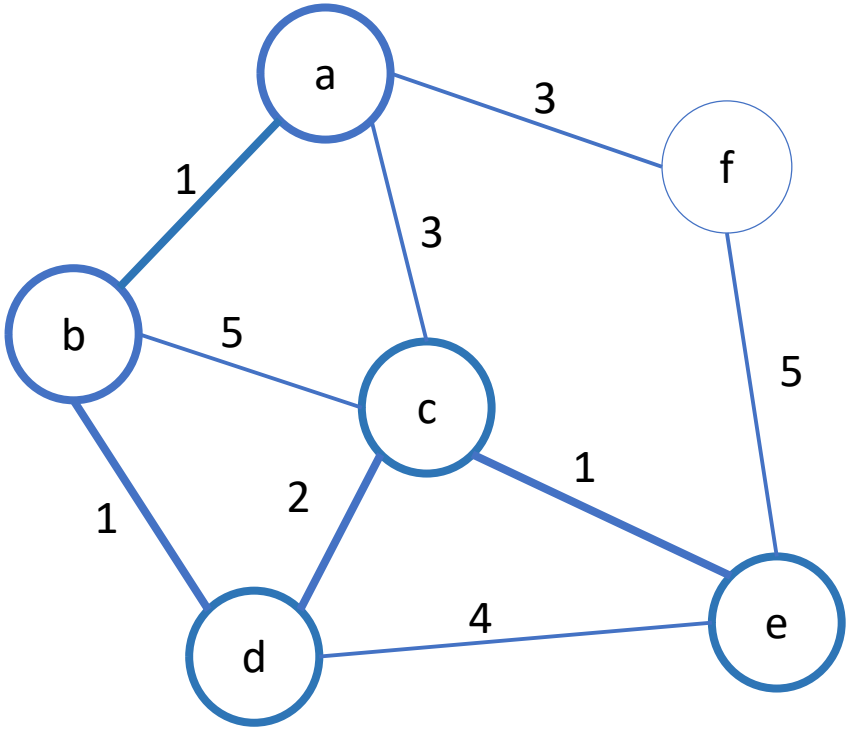


	a	b	c	d	e	f
d[]	1	0	2	1	1	3

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

(f) After the fifth edge is selected

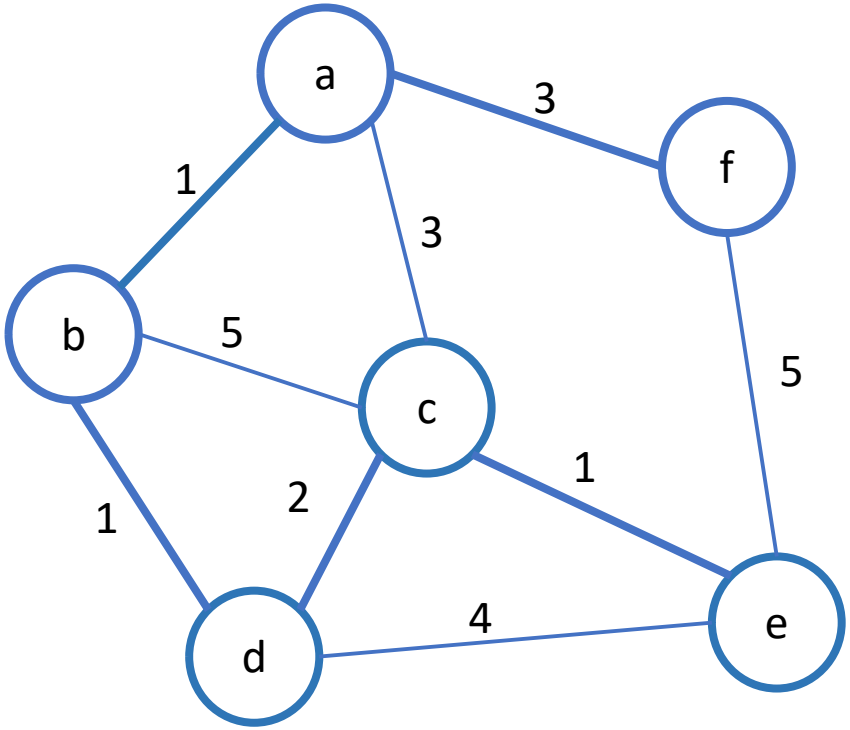


	a	b	c	d	e	f
d[]	1	0	2	1	1	3

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

(g) Final minimum spanning tree



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

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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's Algorithm

- Prim's sequential minimum spanning tree algorithm.

```
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
```

Prim's Algorithm: Parallel Formulation

- d[] change after every loop
- Greedy algorithm → Outer iteration is hard to parallelize
- The inner loop is relatively easy to parallelize.
  - The adjacency matrix is partitioned in a 1-D block fashion, with distance vector d partitioned accordingly.
  - In each step, a 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.

Prim's Algorithm: Parallel Formulation

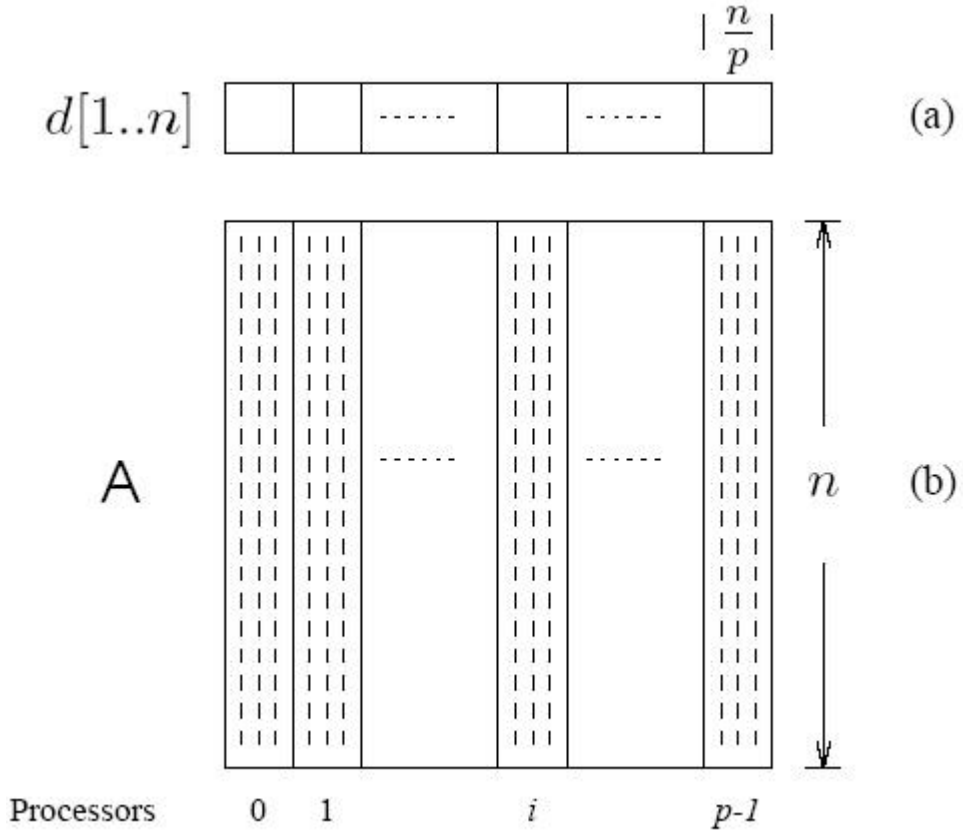
1-D block partitioning: V_i per P_i.

For each iteration:

- P_i computes a local min $d_i[u]$.
- All-to-one reduction to P_0 to compute the global min.
- One-to-all broadcast of u.
- Local updates of d[v].

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 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 $2(n/p + \log p)$.
- The total parallel time is $2(n^2/p + n \log p)$
- In Big O notation, parallel cost is $O(n^2/p + n \log p)$.
  - Constant coefficients are allowed to be dropped in Big O.
  - But it is often misleading in distributed and parallel algorithms since communication cost is not cheap

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.
  - 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.

Single-Source Shortest Paths: Dijkstra's Algorithm

- Dijkstra's sequential single-source shortest paths 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
```

Dijkstra's Algorithm: Parallel Formulation

- Very similar to the parallel formulation of Prim's MST algorithm.
  - The weighted adjacency matrix is partitioned along columns
  - The node closest to the source in local node is globally reduced.
  - The node is broadcast to all processors and the l-vector updated.

- The parallel performance of Dijkstra's algorithm is identical to that of Prim's algorithm.

All-Pairs Shortest Paths

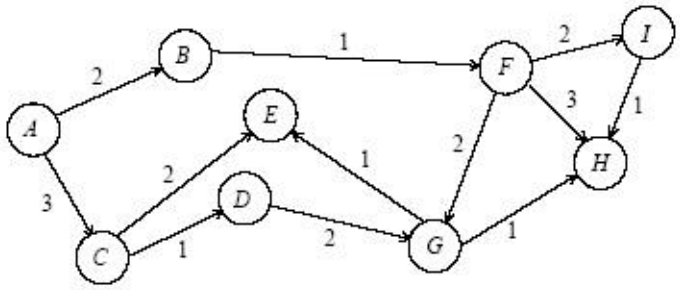
- Given a weighted graph G(V,E,w), the all-pairs shortest paths problem is to find the shortest paths between all pairs of vertices vi, vj $\in$ V.
- A number of algorithms are known for solving this problem.

All-Pairs Shortest Paths: Matrix-Multiplication

- Multiplication of weighted adjacency matrix with itself → a matrix for shortest paths of length 2
  - 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.

- A^n contains all shortest paths.

All-Pairs Shortest Paths: Matrix-Multiplication



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

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

Matrix-Multiplication Based Algorithm: Parallel Formulation

- 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 procedure is $O(n^3 \log n)$.
- We can parallelize matrix multiplications and each of the log n matrix multiplications.

Dijkstra's Algorithm: Parallel Formulation

- Run single-source shortest path problem n times
  - For each source node

- Two parallelization strategies
  - Source Partitioned:
    - Execute each shortest path algorithm on a different processor
  - Source Parallel:
    - Parallelize each shortest path algorithm to further increase concurrency

Dijkstra's Algorithm: Source Partitioned Formulation

- Each processor P_i runs the shortest paths from vertex v_i
- Adjacency matrix is replicated at all processes.
  - Thus, no inter-process communication
- The parallel run time is: $\Theta(n^2)$ where p=n.
  - n/p x $O(n^2)$
    - Complexity of Dijkstra's original algorithm = $O(n^2)$
- Fixed parallelism
  - It can use no more than n processors.

Dijkstra's Algorithm: Source Parallel Formulation

- Execute each of the shortest path problems in parallel
- We can use up to n^2 processors.
- Given p processors (p > n), each single source shortest path problem is executed by p/n processors.
- This takes time:
  - Parallel computation: n/n x $O(n^2 / p/n)$
  - Communication cost:
    - In each step, p processors need to find the global min distance → parallel reduction: O(log p), i.e., O(n log p)

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta(n \log p).}^{\text{communication}}$$

All-Pairs Shortest Paths: Floyd's Algorithm

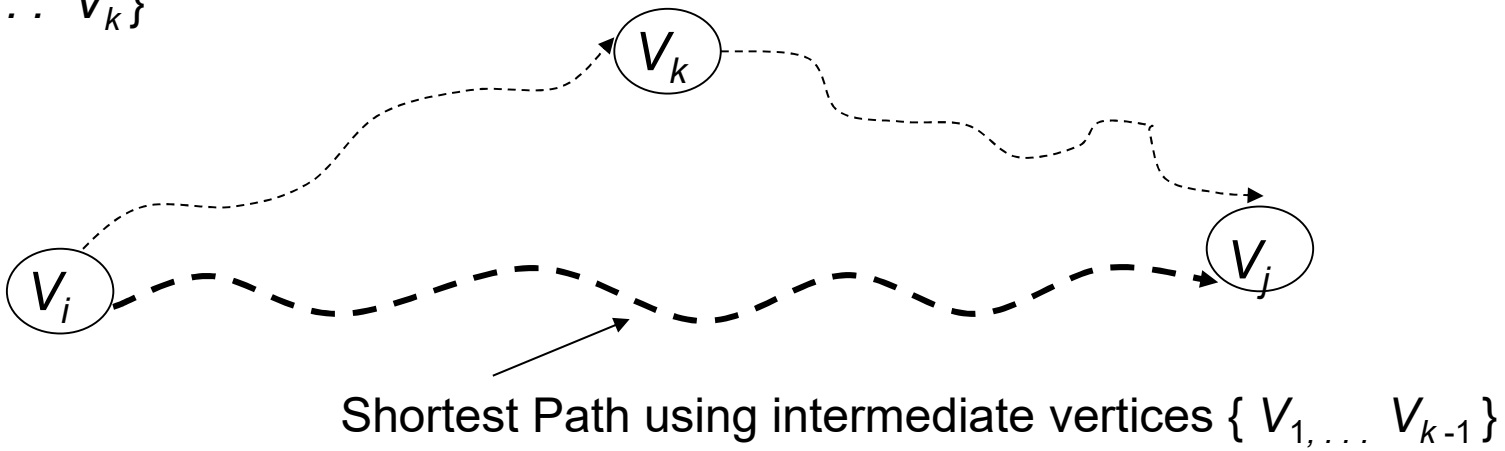
- Let D(k)[i,j]=weight of a shortest path from v_i to v_j using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - D(0)=W
  - D(n)=D which is the goal matrix
- How do we compute D(k) from D(k-1) ?

The Recursive Definition:

- Case 1: A shortest path from v_i to v_j does not use v_k.
  Then D(k)[i,j]= D(k-1)[i,j].

- Case 2: A shortest path from v_i to v_j does use v_k.
  Then D(k)[i,j]= D(k-1)[i,k]+ D(k-1)[k,j].

Shortest path using intermediate vertices
$\{V_1, \ldots V_k\}$



The Recursive Definition:

- Since

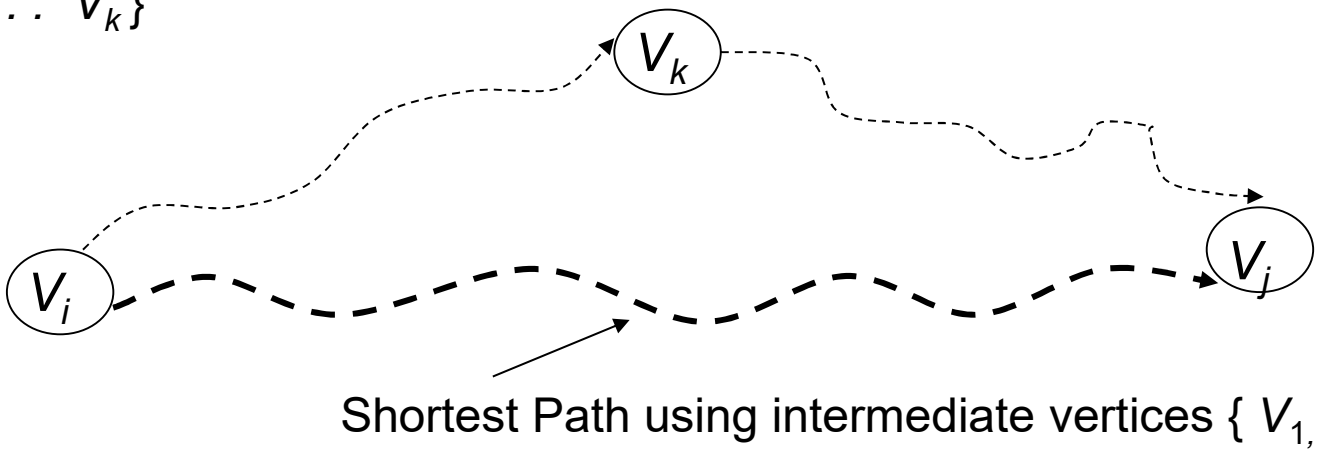
  D(k)[i,j]= D(k-1)[i,j] or

  D(k)[i,j]= D(k-1)[i,k]+ D(k-1)[k,j].

  We conclude:

  D(k)[i,j]= min{ D(k-1)[i,j], D(k-1)[i,k]+ D(k-1)[k,j] }.

Shortest path using intermediate vertices $\{V_1, \ldots V_k\}$



The pointer array P

- Used to enable finding a shortest path
- Initially the array contains 0

- Each time that a shorter path from i to j is found → the k that provided the minimum is saved in P[i, j]
- P[i, j] = highest index node on the path from i to j

- To print the intermediate nodes on the shortest path, a recursive procedure that prints the shortest paths from i and k, and from k to j can be used

Floyd's Algorithm Using n+1 D matrices

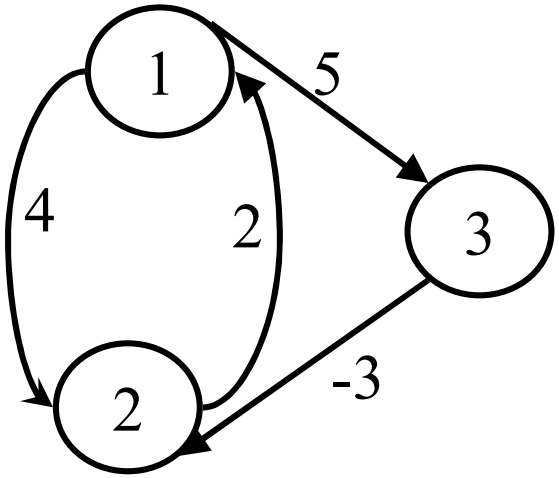
// Computes shortest distance between all pairs of

// nodes, and saves P to enable finding shortest paths

// $O(n^3)$

```
1. D0  ← W    // initialize D array to W [ ]
2. P ← 0      // initialize P array to [0]
3. for k ← 1 to n
4.         do for i ← 1 to n
5.                 do for j ← 1 to n
6.                         if (D(k-1)[ i, j ] > D(k-1)[ i, k ] + D(k-1)[ k, j ] )
7.                                 then  D(k)[ i, j ] ← D(k-1)[ i, k ] + D(k-1)[ k, j ]
8.                                         P[ i, j ] ← k;
9.                                 else D(k)[ i, j ] ← D(k-1)[ i, j ]
```

Floyd's All-to-All Shortest Paths Example



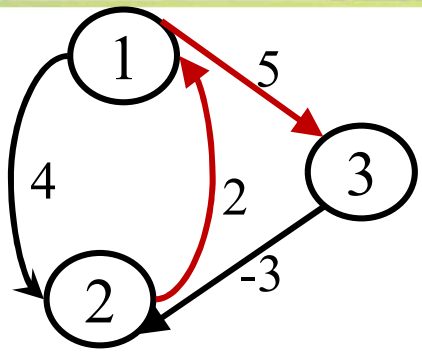
$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$P =$

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

Floyd's All-to-All Shortest Paths Example



$D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

k = 1
Vertex 1 can be intermediate node

$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

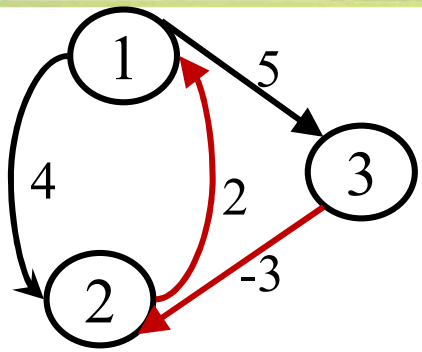
D1[2,3] = min(D0[2,3], D0[2,1]+D0[1,3])
= min (∞, 7)
= 7

D1[3,2] = min(D0[3,2], D0[3,1]+D0[1,2])
= min (-3,∞)
= -3

$P =$

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

Floyd's All-to-All Shortest Paths Example



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

k = 2
Vertices 1, 2 can be intermediate

$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

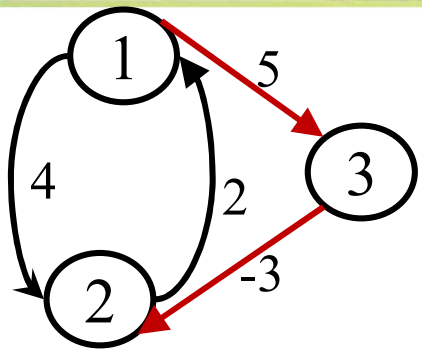
D2[1,3] = min(D1[1,3], D1[1,2]+D1[2,3])
= min (5, 4+7)
= 5

D2[3,1] = min(D1[3,1], D1[3,2]+D1[2,1])
= min (∞, -3+2)
= -1

$P =$

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

Floyd's All-to-All Shortest Paths Example



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

k = 3
Vertices 1, 2, 3 can be intermediate

$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

D3[1,2] = min(D2[1,2], D2[1,3]+D2[3,2])
= min (4, 5+(-3))
= 2

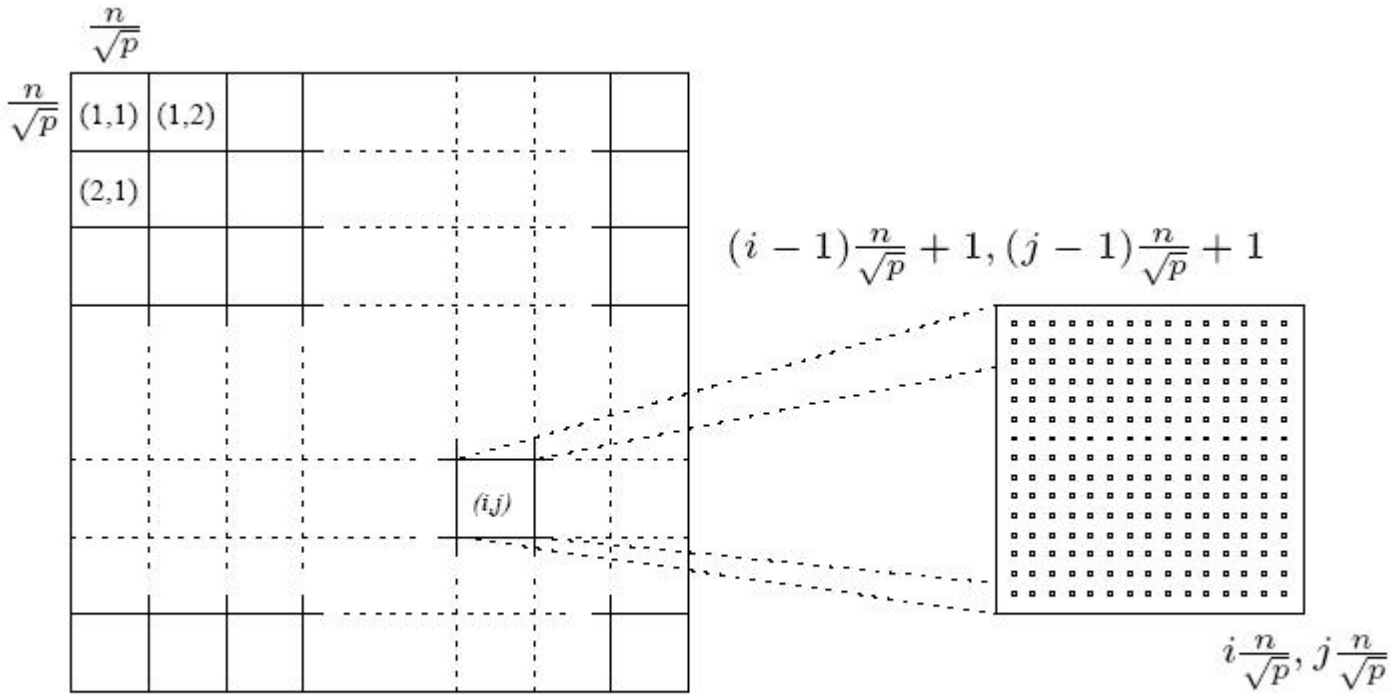
D3[2,1] = min(D2[2,1], D2[2,3]+D2[3,1])
= min (2, 7+ (-1))
= 2

$P =$

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

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

- Matrix D(k) is divided into p blocks of size (n / √p) x (n / √p).
- Each processor updates its part of the matrix per iteration.

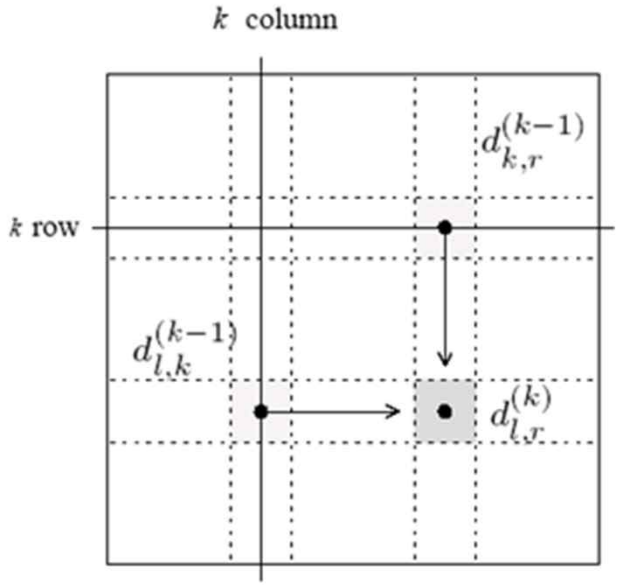


(a) Matrix D(k)

(b) Subblock of D(k) assigned to process $P_{i,j}$

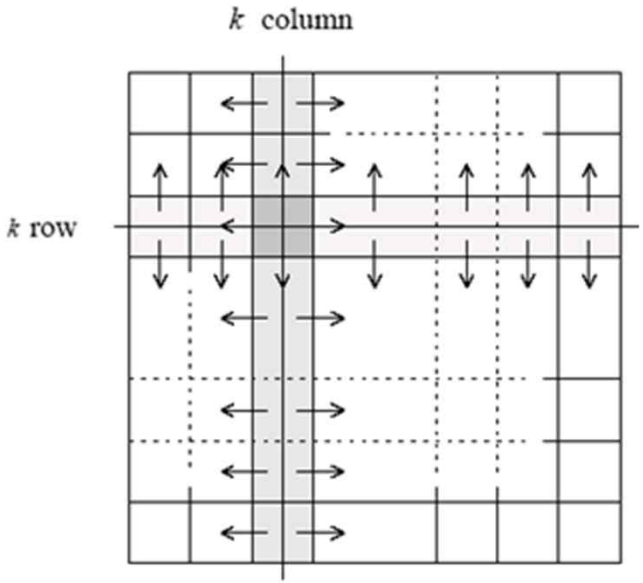
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping

- To compute d(k)[l,r], $P_{i,j}$ must get d(k-1)[l,k] and d(k-1)[k,r].
  - d(k-1)[l,k] is on a process along the same row
  - d(k-1)[k,r] is on a process along the same column



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

- In general, during the k^{th} iteration, each process containing the k^{th} row sends it to the other processes in the same column.
- Communication patterns used in the 2-D block mapping.
  - Similarly, each process containing the k^{th} column sends it to the other processes in the same row.



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

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

```
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 Algorithm: Parallel Formulation Using 2-D Block Mapping

- 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 $n/\sqrt{p}$ elements, taking time $\Theta((n \log p)/\sqrt{p})$.
- The synchronization step takes time $\Theta(\log p)$.
- The computation time is $\Theta(n^2/p)$.
- The parallel run time of the 2-D block mapping formulation of Floyd's algorithm is

$$T_P = \overbrace{\Theta\left(\frac{n^3}{p}\right)}^{\text{computation}} + \overbrace{\Theta\left(\frac{n^2}{\sqrt{p}}\log p\right)}^{\text{communication}}.$$

Floyd's Algorithm: Speeding Things Up by Pipelining

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