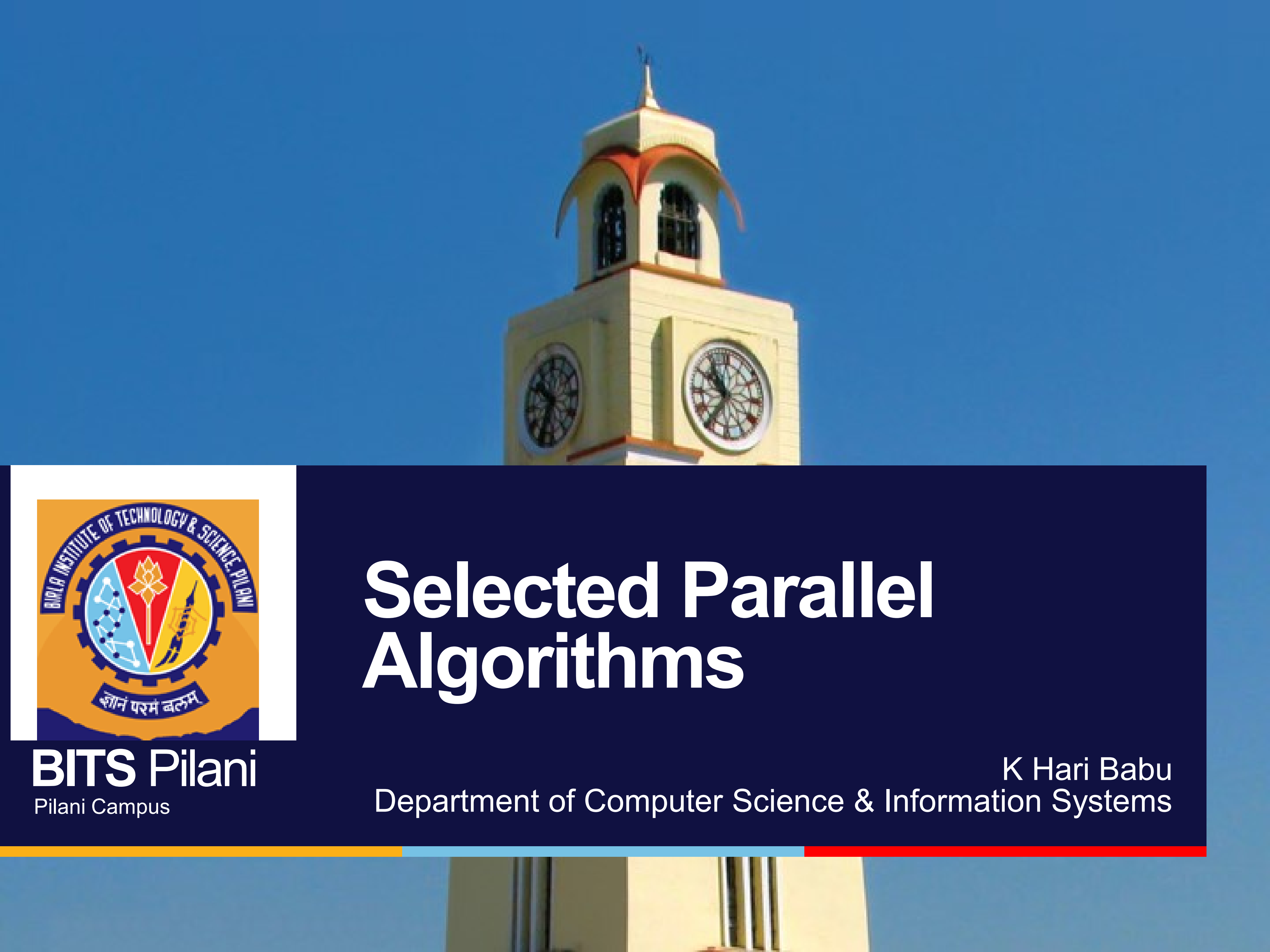


BITS Pilani
Pilani Campus

Selected Parallel Algorithms

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

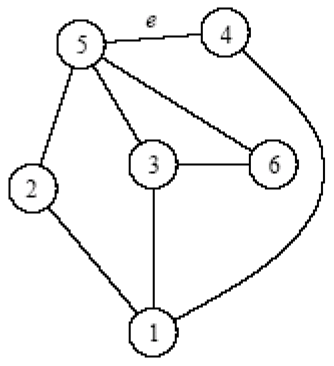
BITS Pilani
Pilani Campus

Graphs-Representation

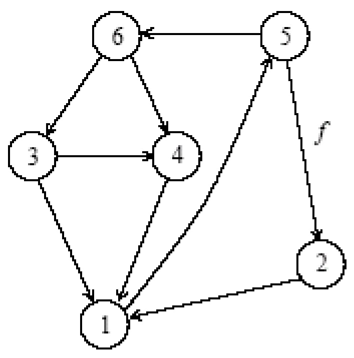
Definitions and Representation

- 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,v* $\in$ *V*.
- In a directed graph, the edge e is an ordered pair *(u,v)*. An edge *(u,v)* is *incident from* vertex *u* and is *incident to* 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.

Definitions and Representation



(a)



(b)

a) An undirected graph and (b) a directed graph.

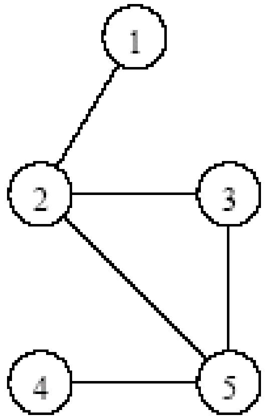
Definitions and Representation

- An undirected graph is *connected* if every pair of vertices is connected by a path.
- A *forest* is an acyclic graph, and a *tree* is a connected acyclic graph.
- A graph that has weights associated with each edge is called a *weighted graph*.

Definitions and Representation

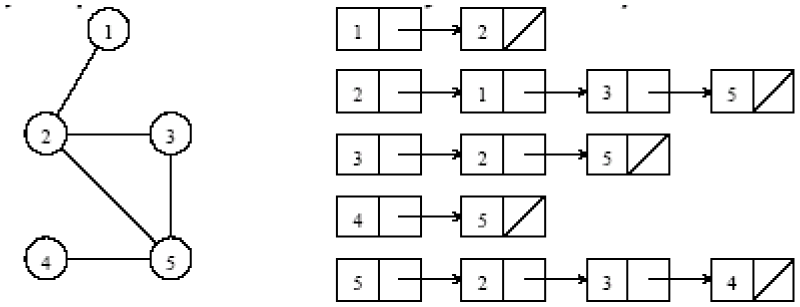
- Graphs can be represented by their adjacency matrix or an edge (or vertex) list.
- Adjacency matrices have a value $a_{i,j} = 1$ if nodes *i* and *j* share an edge; 0 otherwise. In case of a weighted graph, $a_{i,j} = w_{i,j}$, the weight of the edge.
- The *adjacency list* representation of a graph $G = (V,E)$ consists of an array $Adj[1..|V|]$ of lists. Each list $Adj[v]$ is a list of all vertices adjacent to *v*.
- For a grapn with *n* nodes, adjacency matrices take $\Theta(n^2)$ space and adjacency list takes $\Theta(|E|)$ space.

Definitions and Representation



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

An undirected graph and its adjacency matrix representation.



An undirected graph and its adjacency list representation.

innovate achieve lead

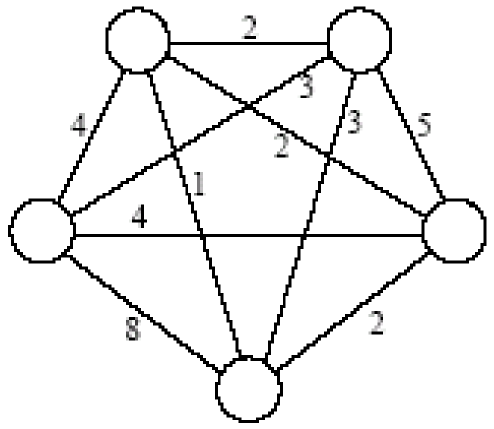
BITS Pilani
Pilani Campus

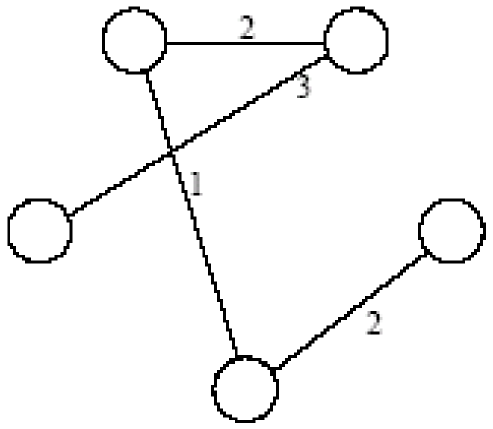
Minimum Spanning Tree – Prim's Algorithms

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





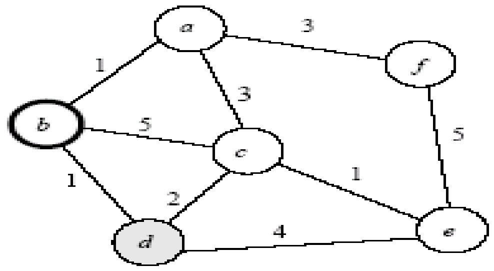
An undirected graph and its minimum spanning tree.

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 it the vertex closest to one of the vertices already in current MST.

Minimum Spanning Tree: Prim's Algorithm

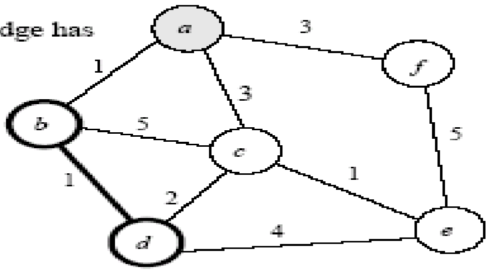
(a) Original graph



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

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

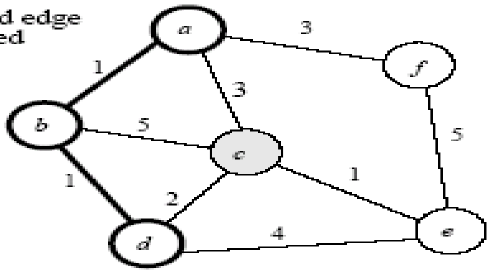
(b) After the first edge has been selected



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

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

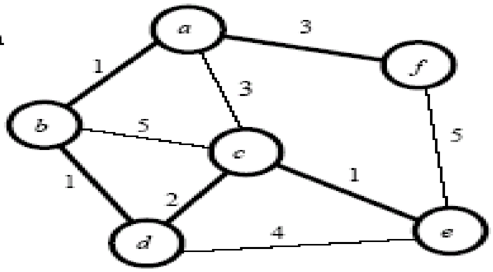
(c) After the second edge has been selected



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

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

(d) Final minimum spanning tree



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

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

Prim's minimum spanning tree algorithm.

Minimum Spanning Tree: 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
```

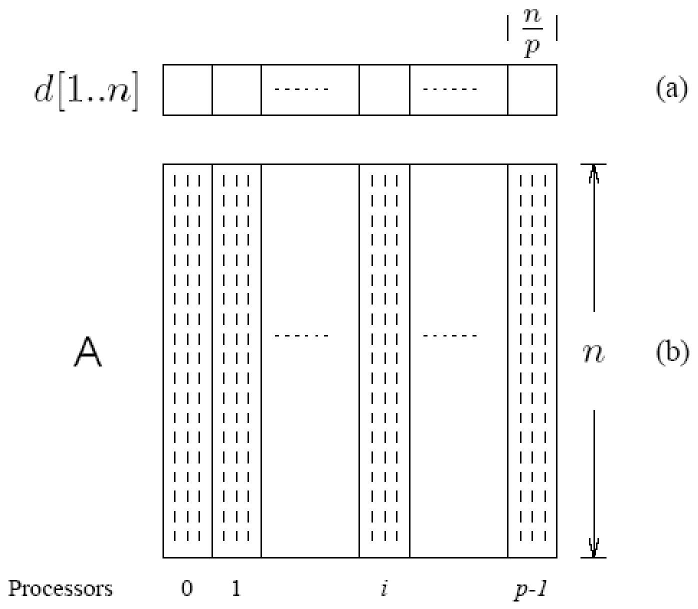
Prim's sequential minimum spanning tree algorithm.

Prim's Algorithm: Parallel Formulation

- Prim's algorithm is iterative.
  - Each iteration adds a new vertex to the minimum spanning tree. Since the value of d[v] for a vertex v may change every time a new vertex u is added in V_T , it is hard to select more than one vertex to include in the minimum spanning tree.
  - Thus, it is not easy to perform different iterations of the while loop in parallel. However, each iteration can be performed in parallel as follows.
- Let p be the number of processes, and let n be the number of vertices in the graph. The set V is partitioned into p subsets using the 1-D block mapping.

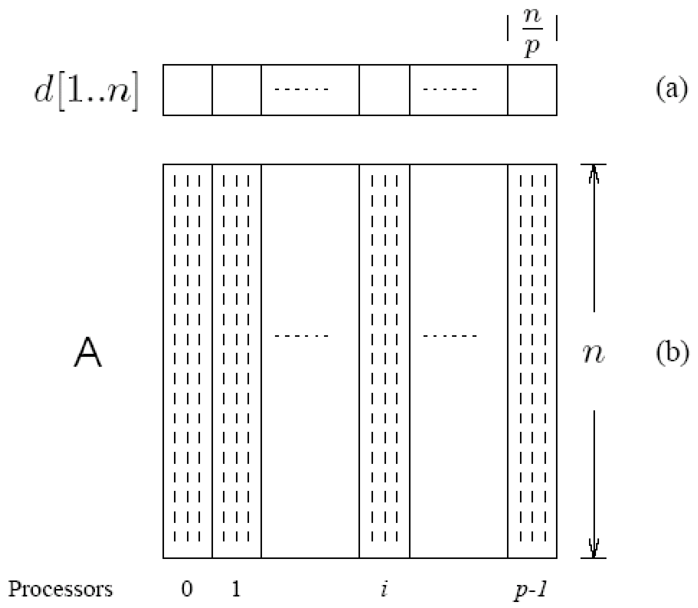
Prim's Algorithm: Parallel Formulation

- Each subset has n/p consecutive vertices, and the work associated with each subset is assigned to a different process.
- Let V_i be the subset of vertices assigned to process P_i for i = 0, 1, ..., p - 1. Each process P_i stores the part of the array d that corresponds to V_i i.e. process P_i stores d [v] such that $v \in Vi$
- Each process P_i computes $d_i[u] = \min\{d_i[v] | v \in (V-V_T) \cap V_i\}$ during each iteration of the while loop
- The global minimum is then obtained over all $d_i[u]$ by using the all-to-one reduction operation and is stored in process P_0.



Prim's Algorithm: Parallel Formulation

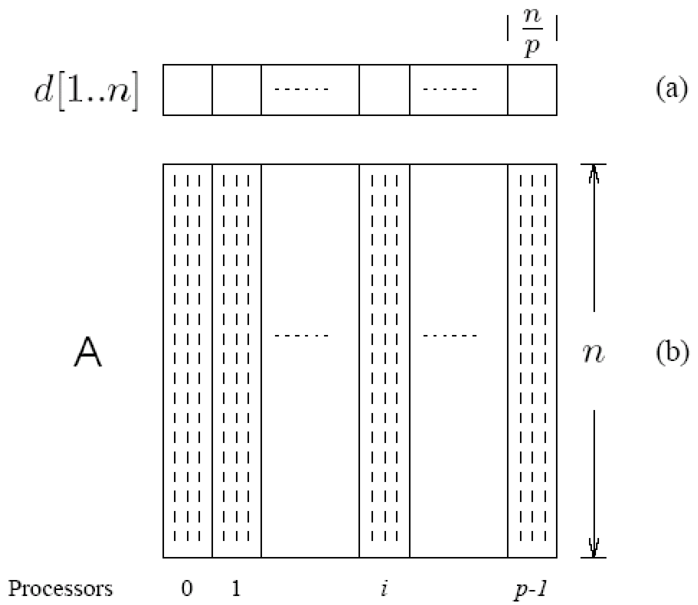
- Process P_0 now holds the new vertex u, which will be inserted into V_T. Process P_0 broadcasts u to all processes by using one-to-all broadcast
- The process P_i responsible for vertex u marks u as belonging to set V_T. Finally, each process updates the values of d[v] for its local vertices
- When a new vertex u is inserted into V_T, the values of d[v] for vi ∈ $(V-V_T)$ must be updated
  - The process responsible for v must know the weight of the edge (u, v). Hence, each process P_i needs to store the columns of the weighted adjacency matrix corresponding to set V_i of vertices assigned to it. This corresponds to 1-D block mapping of the matrix



Prim's Algorithm: Parallel Formulation

- The space to store the required part of the adjacency matrix at each process is Q(n^2/p)
- The computation performed by a process to minimize and update the values of d[v] during each iteration is Q(n/p)
- One to all broadcast, all-to-one reduction takes Q(log p)
- The parallel run time of this formulation is given by

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



innovate achieve lead

BITS Pilani
Pilani Campus

Single-Source Shortest Paths - Dijkstra's Algorithm

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 solves the single-source shortest-paths problem on both directed and undirected graphs with non-negative weights
- Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known
    - The main difference is that, for each vertex u, Dijkstra's algorithm stores l[u], the minimum cost to reach vertex u from vertex s by means of vertices in V_T; Prim's algorithm stores d [u], the cost of the minimum-cost edge connecting a vertex in V_T to u.

Single-Source Shortest Paths: 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
            $l[v] := \min\{l[v], l[u] + w(u, v)\}$;
    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 1-D block mapping.
  - Each of the p processes is assigned n/p consecutive columns of the weighted adjacency matrix, and computes n/p values of the array l.
  - During each iteration, all processes perform computation and communication similar to that performed by the parallel formulation of Prim's algorithm.
  - 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.
- The parallel performance of Dijkstra's algorithm is identical to that of Prim's algorithm.

innovate achieve lead

BITS Pilani
Pilani Campus

All-Pairs Shortest Paths

All-Pairs Shortest Paths

- Instead of finding the shortest paths from a single vertex v to every other vertex, we are sometimes interested in finding the shortest paths between all pairs of vertices
- Formally, 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 ∈ V such that i <>j.
- For a graph with n vertices, the output of an all-pairs shortest paths algorithm is an n x n matrix D = ($d_{i,j}$) such that $d_{i,j}$ is the cost of the shortest path from vertex v_i to vertex v_j.

All-Pairs Shortest Paths

- Two algorithms to solve the all-pairs shortest paths problem
- The first algorithm uses Dijkstra's single-source shortest paths algorithm, and the second uses Floyd's algorithm.
- Dijkstra's algorithm requires non-negative edge weights, whereas Floyd's algorithm works with graphs having negative-weight edges provided they contain no negative-weight cycles

Dijkstra's Algorithm

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

Dijkstra's Algorithm: Parallel Formulation

- Two parallelization strategies
  - 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: Partition the vertices across processors
  - Works well if p<=n; No communication
  - Can at best use only n processors
- Source-parallel formulation: Parallelize SSSP for a vertex across a subset of processors
  - p processes are divided into p/n subsets
  - Each of the n-single source shortest paths problem is solved by a subset

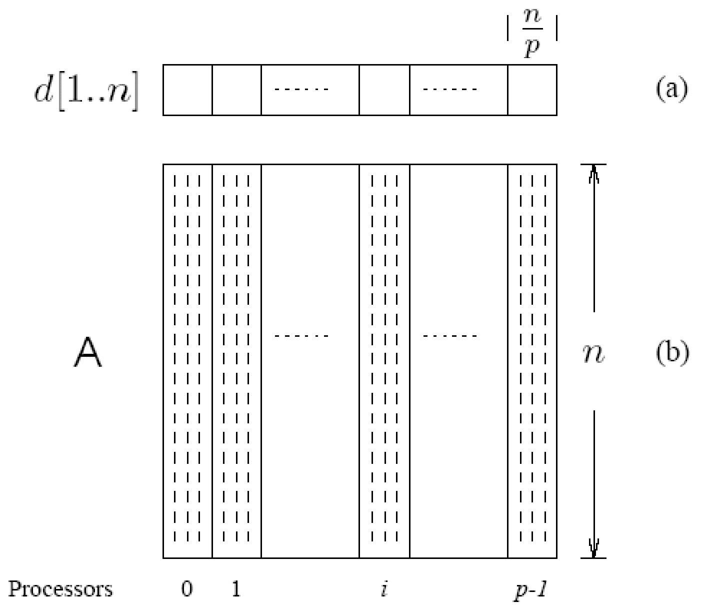
Dijkstra's Algorithm: Source Partitioned Formulation

- 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: $\Theta(n^2)$.
- While the algorithm is cost optimal, it can only use n processors.

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 time:
  - = time taken by a single group
  - = n iterations * (n2/p) computations

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



p log p/n^2 = O (1). Hence, this formulation can use up to O (n^2/log n) processes efficiently

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

Dijkstra's Algorithm: Parallel Formulation

- Comparing the two parallel formulations of Dijkstra's all-pairs algorithm
  - we see that the source-partitioned formulation performs no communication, can use no more than n processes, and solves the problem in time Q(n^2).
  - In contrast, the source-parallel formulation uses up to n^2/log n processes, has some communication overhead, and solves the problem in time Q(n log n) when n^2/log n processes are used.
  - Thus, the source-parallel formulation exploits more parallelism than does the source-partitioned formulation.

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

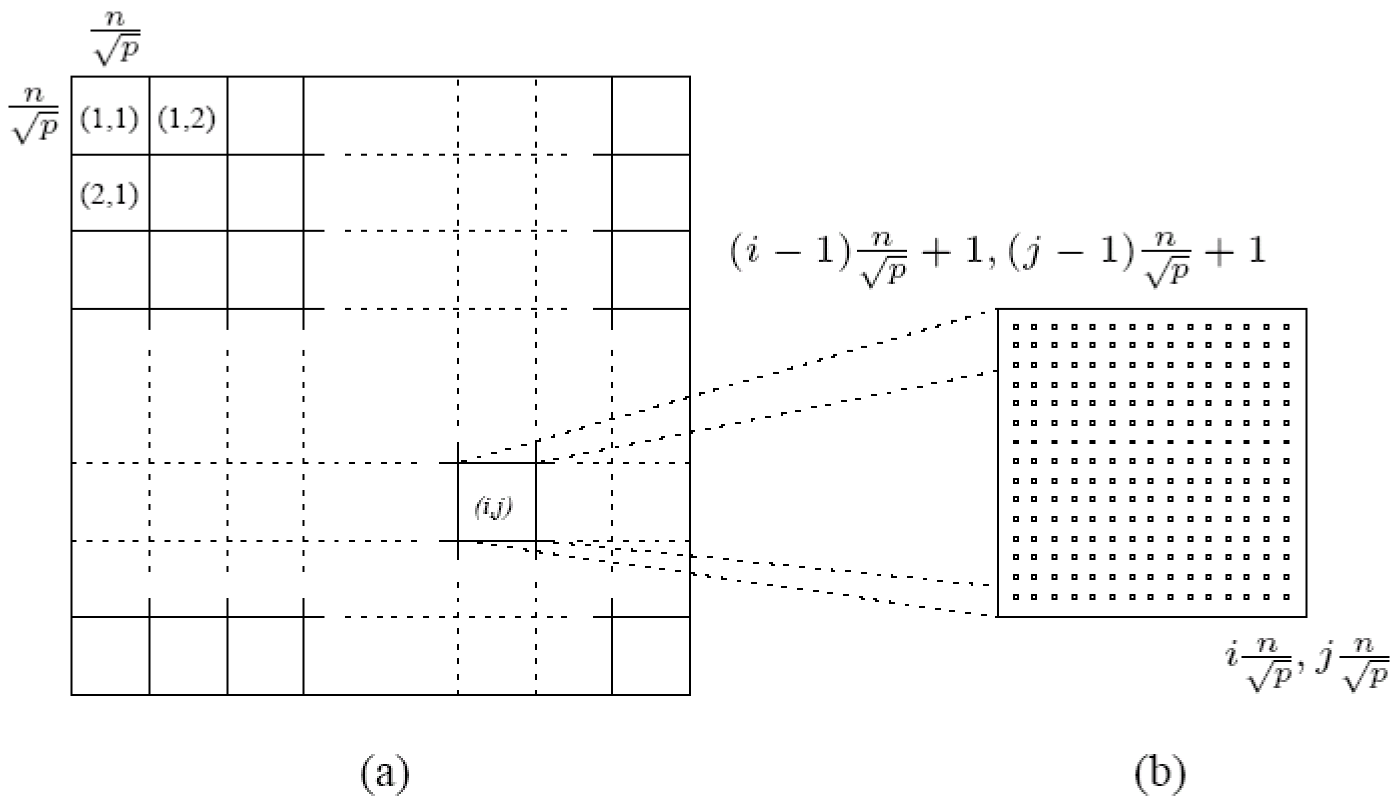
```
procedure FLOYD_ALL_PAIRS_SP($A$)
begin
    $D^{(0)} = A$;
    for $k := 1$ to $n$ do
        for $i := 1$ to $n$ do
            for $j := 1$ to $n$ do
                $d_{i,j}^{(k)} := \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right);$
end FLOYD_ALL_PAIRS_SP
```

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

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

- 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 $d_{l,k}^{(k-1)}$ 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 $\sqrt{p}$ 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.

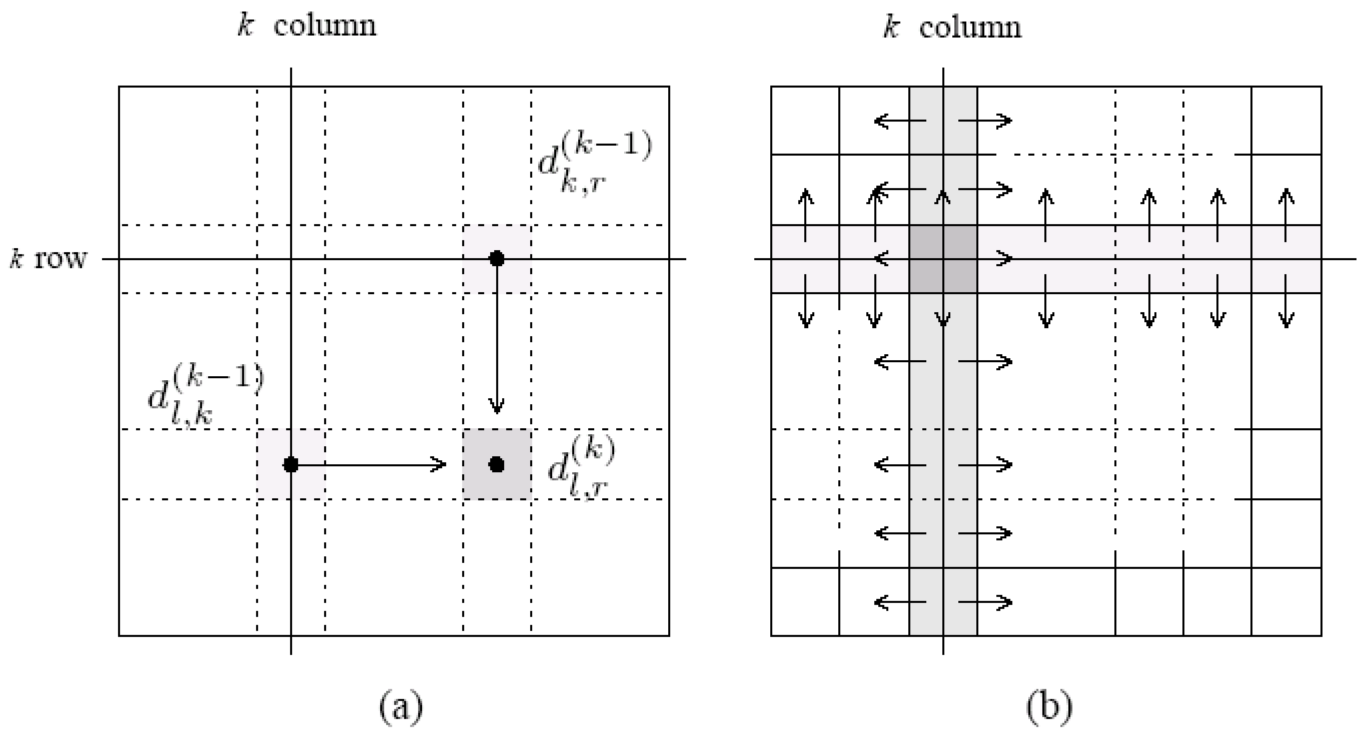
Floyd's Algorithm: Parallel Formulation Using 2-D Block Mapping



$\frac{n}{\sqrt{p}}$
$\frac{n}{\sqrt{p}}$
(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 2-D Block Mapping



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

Floyd's Algorithm: Parallel Formulation Using 2-D 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 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.

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 $\Theta(n^2/p)$.
- Each process runs <u>n iterations</u>. 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: Parallel Formulation Using 2-D Block Mapping

- The above formulation can use $O(n^2 / \log^2 n)$ processors cost-optimally.

$$S = \frac{\Theta(n^3)}{\Theta(n^3/p) + \Theta((n^2 \log p)/\sqrt{p})}$$

$$E = \frac{1}{1 + \Theta((\sqrt{p} \log p)/n)}$$

  - $\Theta\left(\frac{\sqrt{p}logp}{n}\right) = O(1)$
  - → p=O$(n^2/log^2 n)$
- This algorithm can be further improved by relaxing the strict synchronization after each iteration.

References

BITS Pilani

- Chapter 10.1-4 of text book.

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You