Introduction to Parallel & Distributed Computing

Parallel Graph Algorithms

Lecture 16, Spring 2014

Instructor: 罗国杰

gluo@pku.edu.cn

In This Lecture …

- **Parallel formulations of some important and fundamental graph algorithms**
  - **Graph theory plays an important role in CS**
  - **It provides an easy and systematic way to model many problems**
  - **Many problems can be expressed in terms of graphs, and can be solved using standard graph algorithms**

Example: Route Planning Problem

- How to find an "optimal" (fastest, shortest, etc.) route from point A to point B in a map?
- Dijkstra's algorithm does work, but with a couple of modifications
  - Bidirectional search
    - Expand both ends until meeting in the middle
  - Hierarchical search
    - Maintain "highway" layer and "local" layer
  - …
  - A survey paper
    - [D. Delling, Engineering Route Planning Algorithms, 2009]

Topic Overview

- **Minimum Spanning Tree: Prim's Algorithm**
- **Single-Source Shortest Paths: Dijkstra's Algorithm**
- **All-Pairs Shortest Paths**
- **Transitive Closure**
- **Connected Components**
- **Algorithms for Sparse Graphs**

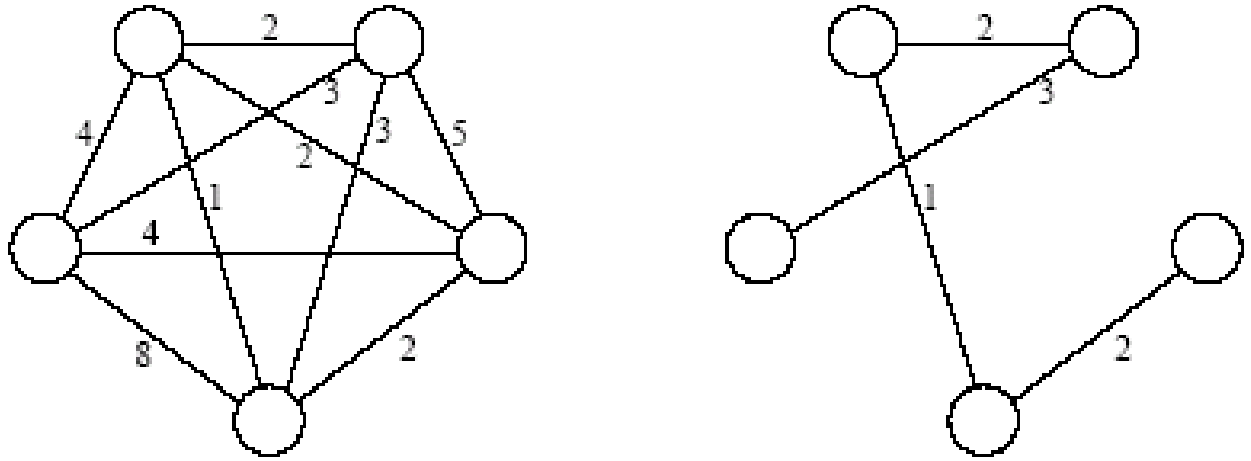
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

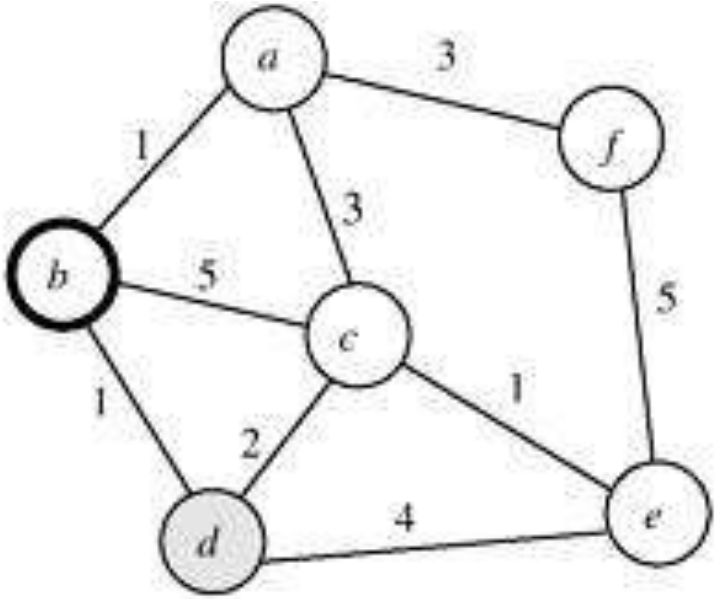
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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.**

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003

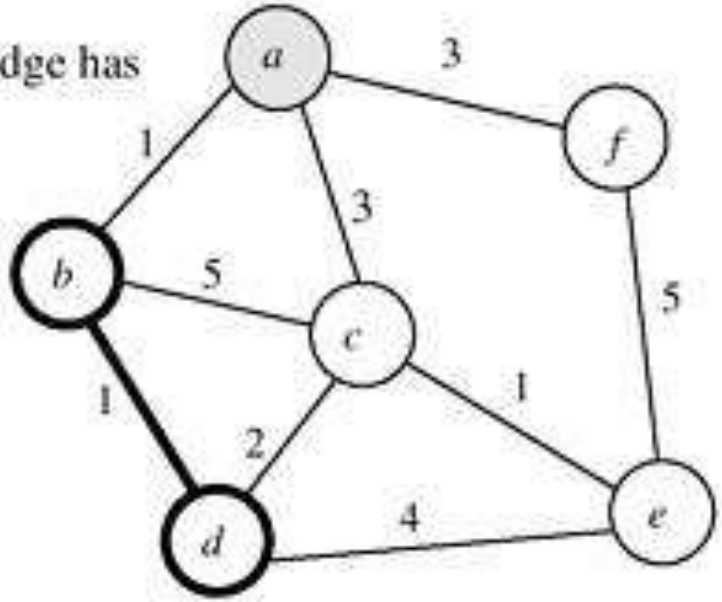
(a) Original graph



	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

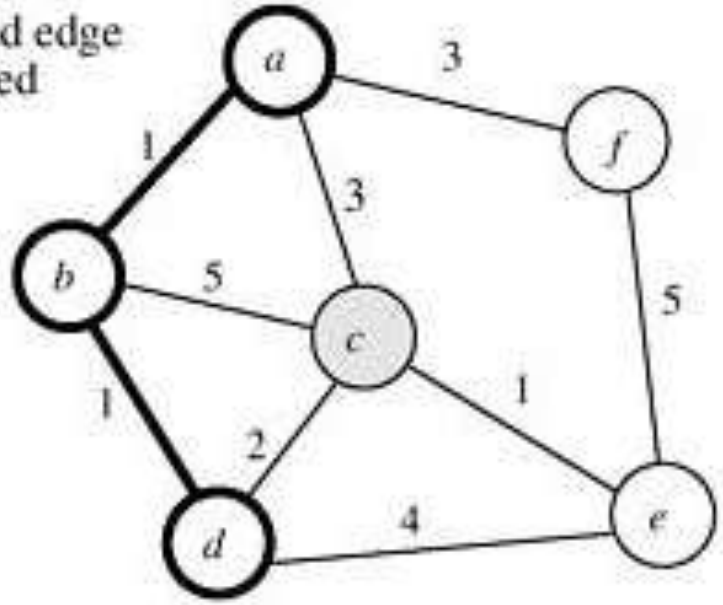
(b) After the first edge has been selected



	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

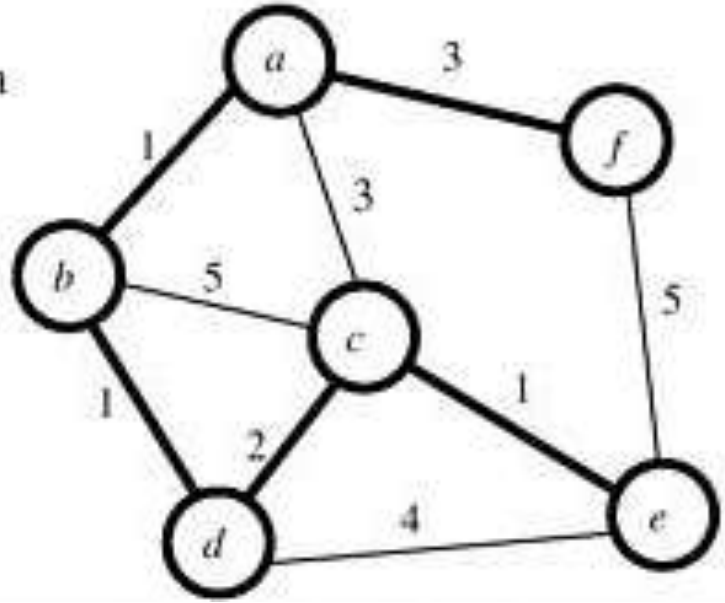
(c) After the second edge has been selected



	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	2	∞	∞	∞	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	2	∞	∞	∞	5	0

Minimum Spanning Tree: Prim's 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 sequential minimum spanning tree algorithm.

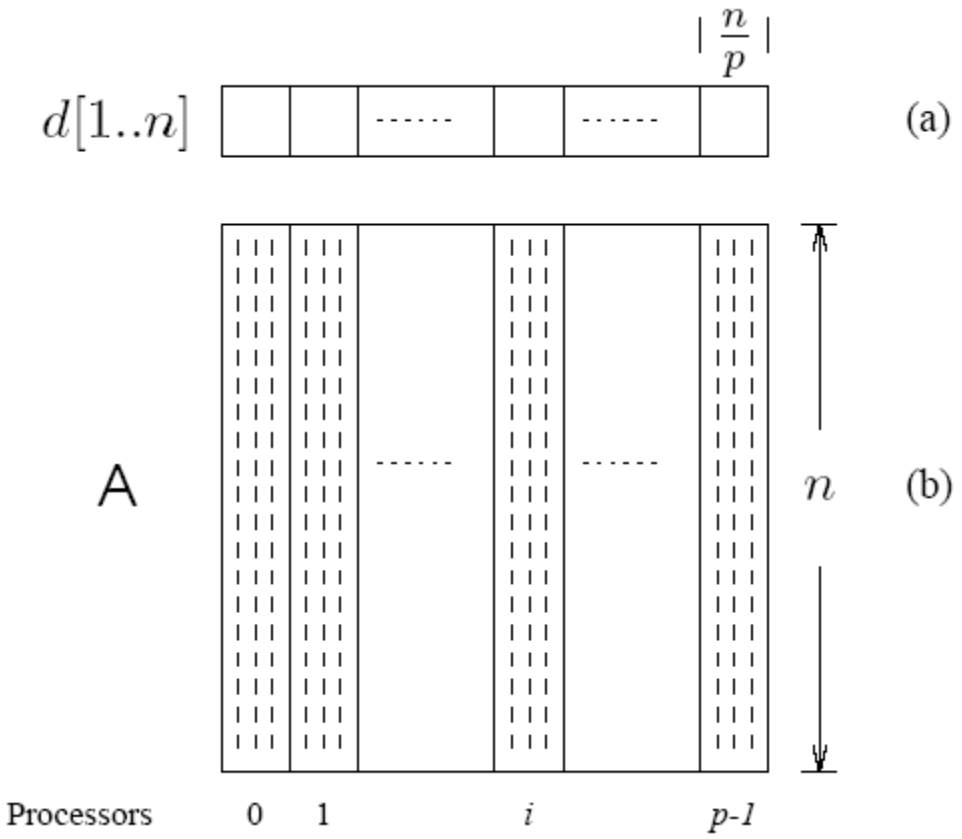
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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, 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 broadcast to all processors.**
- **Each processor updates its part of the d vector locally.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Prim's Algorithm: Parallel Formulation



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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Prim's Algorithm: Parallel Formulation

- **The parallel time per iteration is $O(n/p + \log p)$.**
  - **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 total parallel time is given by $O(n^2/p + n \log p)$.**

- **ATTENSION**
  - **$O(n/p)$ for minimum selection and local update is only estimated for the algorithm we use here, which can be improved by a better algorithm and advance data structures (e.g., heap)**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Topic Overview

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths

- **For a weighted graph $G = (V,E,w)$, the *single-source all-sinks 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.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths: 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
```

Dijkstra's sequential single-source shortest paths algorithm.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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 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.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Topic Overview

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

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.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm

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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm: Parallel Formulation

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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm (Source Partitioned)

- **Source-by-source parallelism**
- **Use p processes to execute the Dijkstra's SSSP algorithm on n vertices ($p \leq n$)**
  - **no interprocess communication needed, assuming the adjacency matrix is replicated at all processes**
  - **The parallel runtime is $\Theta(n^3/p)$**
- **The maximum #proc is $\Theta(n)$**
  - **with runtime is $\Theta(n^2)$**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Dijkstra's Algorithm (Source Parallel)

- **What if $p > n$ …**
  - **each of the SSSP is further executed in parallel**
  - **parallel execution of SSSP on *n* vertices**
  - **use *p/n* processes to execute SSSP for each vertex**
- **Using previous results, this takes total time:**

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

  - **(*log p* is a pessimistic estimate of the broadcasting time)**
- **The maximum #proc is $\Theta(n^2/\log n)$**
  - **with runtime $\Theta(n \log n)$**
  - **when the efficiency is not degraded**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm

The following recurrence relation follows:

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

Computed for each pair of nodes and for $k = 1, \ldots, n$.

The serial complexity is $O(n^3)$.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

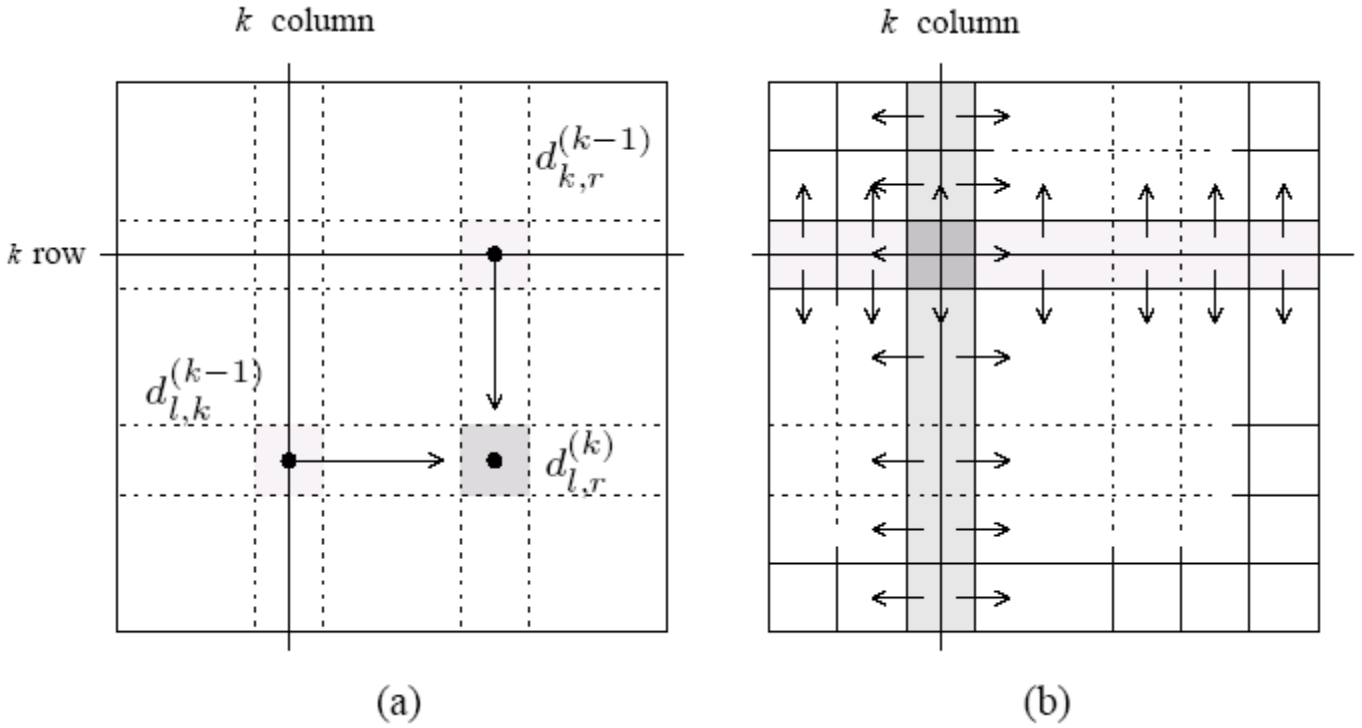
Floyd's Algorithm

```
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 ( d_{i,j}^(k-1), d_{i,k}^(k-1) + d_{k,j}^(k-1) );
8.  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.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm: Communications



(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.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (1-D Block Mapping)

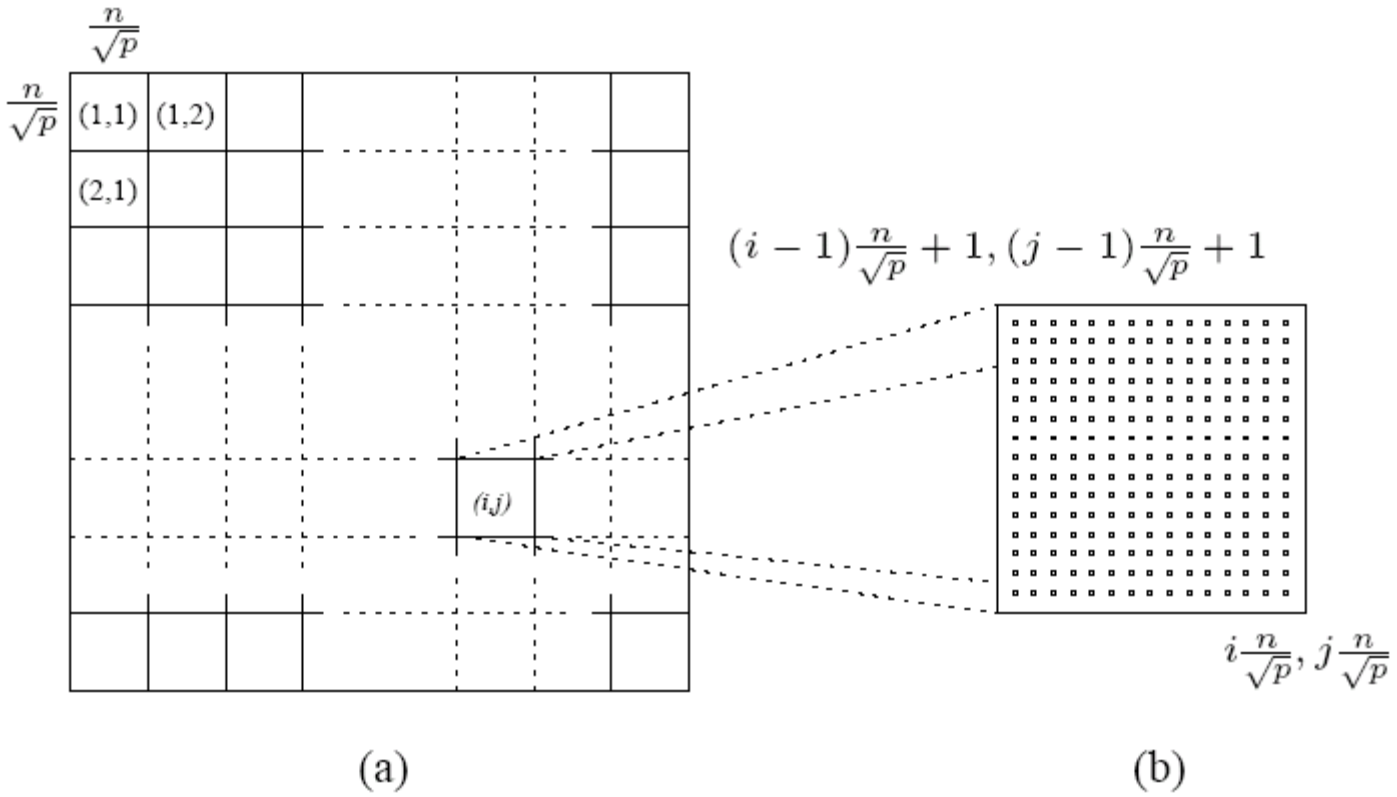
- **Matrix $D^{(k)}$ is divided into p blocks with size $(n/p) \times n$**
  - **Each process update its part of the matrix ($p \leq n$)**
- **To compute** $d_{i,j}^{(k)} = \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right)$
  - $d_{i,j}^{(k-1)}$ **and** $d_{i,k}^{(k-1)}$ **is owned by** P_i **itself**
  - $d_{k,j}^{(k-1)}$ **is owned by** P_k
- **In general, during the k^{th} iteration, the processes containing the k^{th} row is responsible for broadcast**
- **Parallel runtime is $\Theta(n^3/p + n^2 \log p)$**
- **The maximum #proc is $\Theta(n/\log n)$**
  - **with runtime $\Theta(n^2 \log n)$**
  - **when the efficiency is not degraded**

Floyd's Algorithm (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 $d_{i,j}^{(k)}$ processor $P_{i,j}$ must get $d_{i,k}^{(k-1)}$ and $d_{k,j}^{(k-1)}$.**
- **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.**
  - **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.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (2-D Block Mapping)



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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (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.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (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**

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

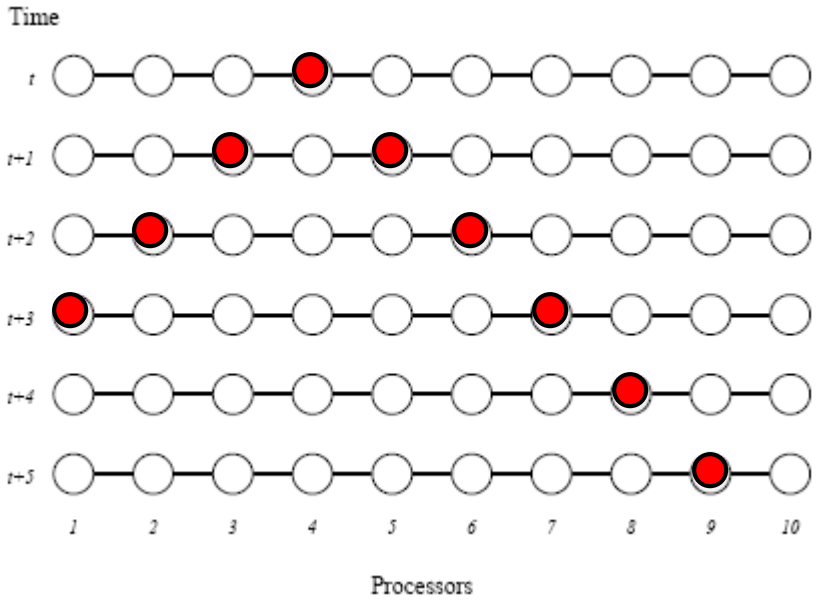
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (2-D + 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 k^{th} iteration as soon as it has computed the $(k-1)^{th}$ iteration and has the relevant parts of the $D^{(k-1)}$ matrix.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (2-D + 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.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (2-D + Pipelining)

- **In each step, $n/\sqrt{p}$ elements of the first row are sent from process $P_{i,j}$ to $P_{i+1,j}$.**
- **Similarly, elements of the first column are sent from process $P_{i,j}$ to process $P_{i,j+1}$.**
- **Each such step takes time $\Theta(n/\sqrt{p})$.**
- **After $\Theta(\sqrt{p})$ steps, process $P_{\sqrt{p},\sqrt{p}}$ gets the relevant elements of the first row and first column in time $\Theta(n)$.**
- **The values of successive rows and columns follow after time $\Theta(n^2/p)$ in a pipelined mode.**
- **Process $P_{\sqrt{p},\sqrt{p}}$ finishes its share of the shortest path computation in time $\Theta(n^3/p) + \Theta(n)$.**
- **When process $P_{\sqrt{p},\sqrt{p}}$ has finished the $(n-1)^{th}$ iteration, it sends the relevant values of the n^{th} row and column to the other processes.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Floyd's Algorithm (2-D + Pipelining)

- **The overall parallel run time of this formulation is**

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

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

All-pairs Shortest Path: Comparison

- **The performance and scalability of the all-pairs shortest paths algorithms on various architectures with bisection bandwidth. Similar run times apply to all cube architectures, provided that processes are properly mapped to the underlying processors.**

	Maximum Number of Processes for $E = \Theta(1)$	Corresponding Parallel Run Time
Dijkstra source-partitioned	$\Theta(n)$	$\Theta(n^2)$
Dijkstra source-parallel	$\Theta(n^2/\log n)$	$\Theta(n \log n)$
Floyd 1-D block	$\Theta(n/\log n)$	$\Theta(n^2 \log n)$
Floyd 2-D block	$\Theta(n^2/\log^2 n)$	$\Theta(n \log^2 n)$
Floyd pipelined 2-D block	$\Theta(n^2)$	$\Theta(n)$

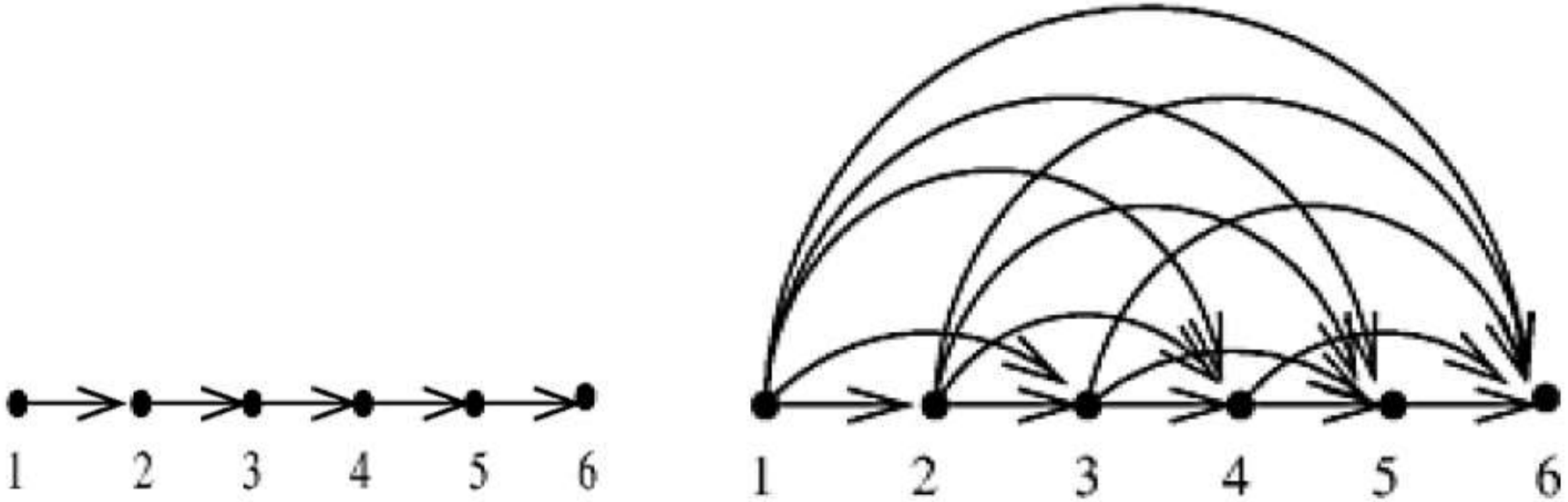
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Topic Overview

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Transitive Closure: Example



G *Transitive closure G**

Transitive Closure

- **Problem**
  - **Determine if two vertices in a graph are connected**
- **Transitive closure of G**
  - **The graph $G^* = (V, E^*)$ where**
  - **$E^* = \{ (v_i, v_j)$ | there is a path from v_i to v_j in $G\}$**
- **Connectivity matrix $A^* = (a^*_{i,j})$**
  - **$a^*_{i,j} = 1$ if there is a path from v_i to v_j or $i = j$**
  - **$a^*_{i,j} = 0$ otherwise**

Transitive Closure

- **Method 1**
  - **Assign weights of *1* to each edge**
  - **Use Floyd's algorithm to obtain distance matrix *D***
  - **Obtain *A** from *D* by**
    - $a^*_{i,j} = 1$ **if** $d_{i,j} > 0$ **or** $i = j$
    - $a^*_{i,j} = 0$ **if** $d_{i,j} = \infty$
- **Method 2**
  - **Modify Floyd's algorithm**
    - **Initially**

      $a^*_{i,j} = 1$ **if** $i = j$ **or** (i,j) **is an edge**

      $a^*_{i,j} = 0$ **otherwise**
    - **Replace "min" by "or"**
    - **Replace "+" by "and"**

Modified 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.                  a_{i,j}^(k) := a_{i,j}^(k-1) or (a_{i,k}^(k-1) and a_{k,j}^(k-1))
8.  end FLOYD_ALL_PAIRS_SP
```

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

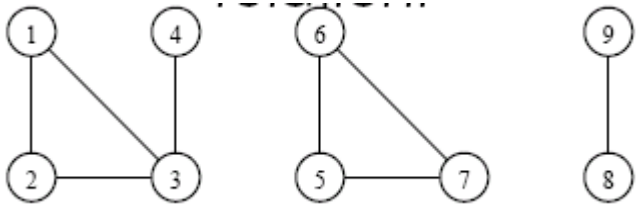
Topic Overview

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components

- **The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation.**

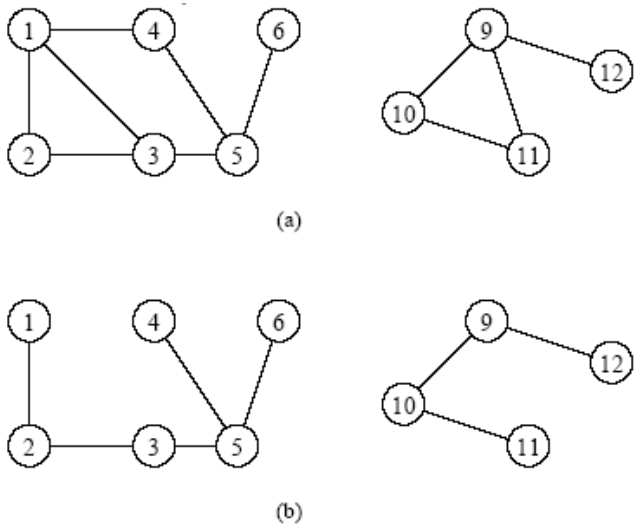


A graph with three connected components: *{1,2,3,4}*, *{5,6,7}*, and *{8,9}*.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components: Depth-First Search Based Algorithm

- **Perform DFS on the graph to get a forest - each tree in the forest corresponds to a separate connected component.**



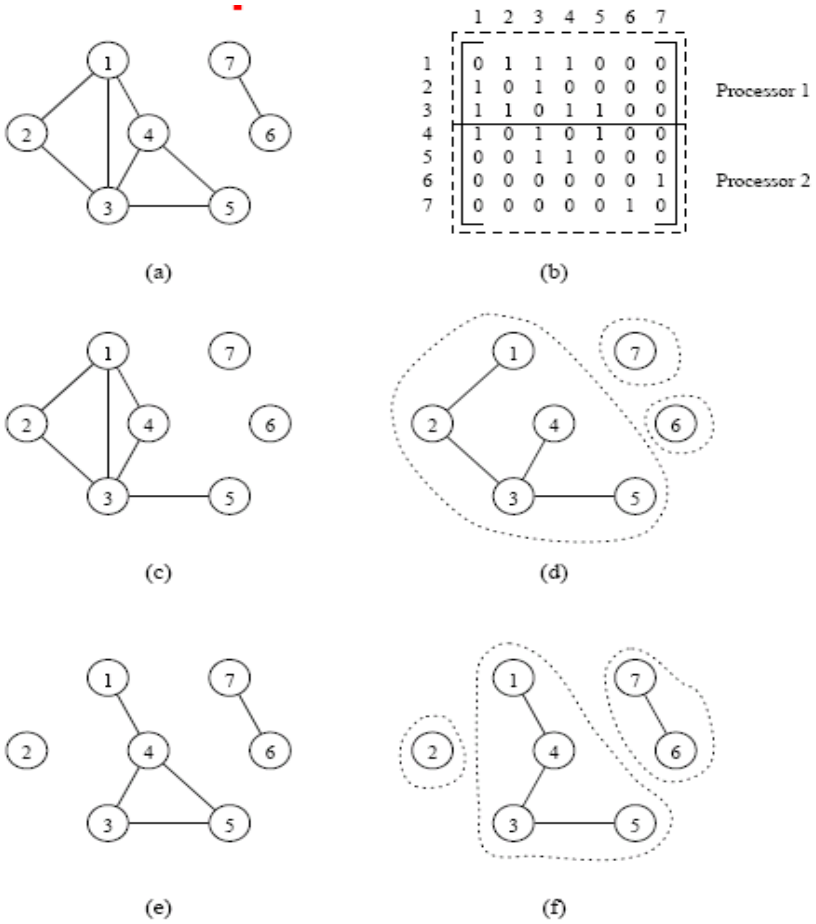
Part (b) is a depth-first forest obtained from depth-first traversal of the graph in part (a). Each of these trees is a connected component of the graph in part (a).

Connected Components: Parallel Formulation

- **Partition the graph across processes and run independent connected component algorithms on each processor. At this point, we have *p* spanning forests.**
- **In the second step, spanning forests are merged pairwise until only one spanning forest remains.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components: Parallel Formulation



Computing connected components in parallel. The adjacency matrix of the graph *G* in (a) is partitioned into two parts (b). Each process gets a subgraph of *G* ((c) and (e)). Each process then computes the spanning forest of the subgraph ((d) and (f)). Finally, the two spanning trees are merged to form the solution.

Connected Components: Parallel Formulation

- **To merge pairs of spanning forests efficiently, the algorithm uses disjoint sets of edges.**
- **We define the following operations on the disjoint sets:**
  - *find*(*x*)
    - **returns a pointer to the representative element of the set containing *x*. Each set has its own unique representative.**
  - *union*(*x*, *y*)
    - **unites the sets containing the elements *x* and *y*. The two sets are assumed to be disjoint prior to the operation.**
  - **Can be implemented as a disjoint-set forest**
    - **Using the "union by rank" and "path compression" techniques**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components: Parallel Formulation

- **For merging forest *A* into forest *B***
  - **for each edge *(u,v)* of *A*, a *find* operation is performed to determine if the two vertices *u* and *v*, are in the same tree of *B*.**
    - **If not, then the two trees (sets) of *B* containing *u* and *v* are united by a *union* operation.**
    - **Otherwise, no *union* operation is necessary.**
- **Hence, merging *A* and *B* requires at most *2(n-1)* *find* operations and *(n-1)* *union* operations.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components: Parallel 1-D Block Mapping

- The $n \times n$ adjacency matrix is partitioned into p blocks.
- Each processor can compute its local spanning forest in time $\Theta(n^2/p)$.
  - DFS or BFS takes $\Theta(|E|+|V|)$, when $|E|$ is $\Theta(n^2/p)$ in the worst case
- Merging is done by embedding a logical tree into the topology. There are $\log p$ merging stages, and each takes time $\Theta(n)$. Thus, the cost due to merging is $\Theta(n \log p)$.
- During each merging stage, spanning forests are sent between nearest neighbors. Recall that $\Theta(n)$ edges of the spanning forest are transmitted.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Connected Components: Parallel 1-D Block Mapping

- **The parallel run time of the connected-component algorithm is**

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

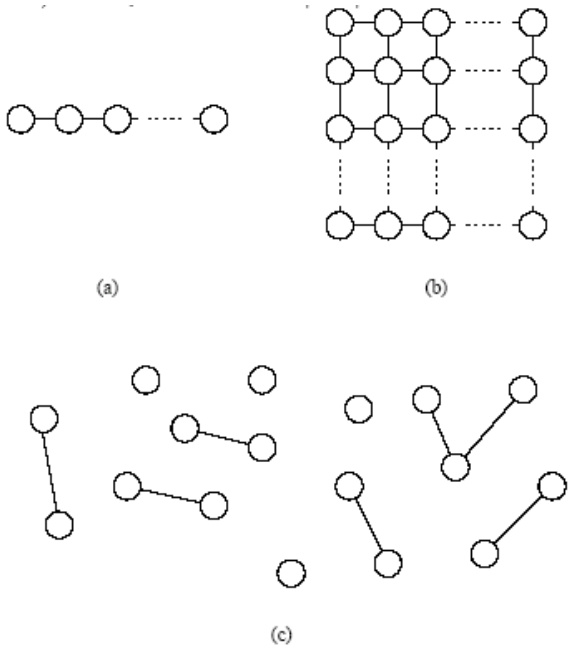
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Topic Overview

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

Algorithms for Sparse Graphs

- **A graph $G = (V,E)$ is sparse if $|E|$ is much smaller than $|V|^2$.**



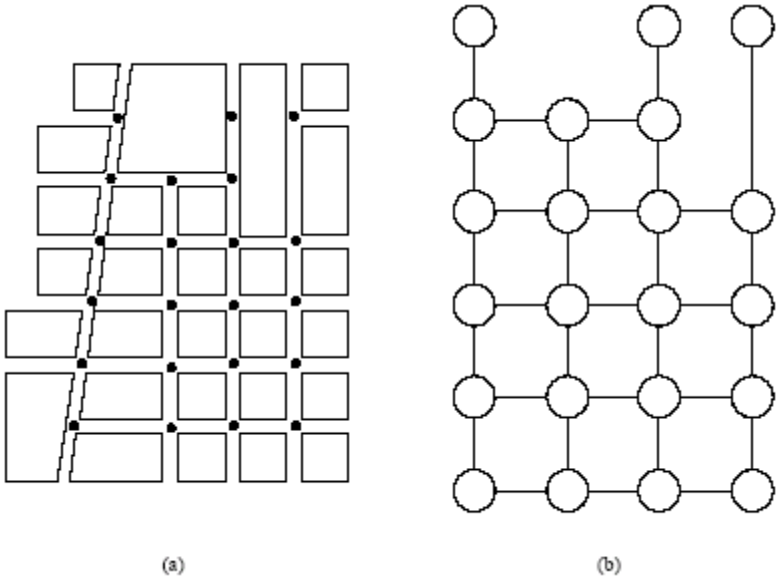
(a) a linear graph, in which each vertex has two incident edges;
(b) a grid graph, in which each vertex has four incident vertices;
(c) a random sparse graph.

Algorithms for Sparse Graphs

- **Dense algorithms can be improved significantly if we make use of the sparseness.**
  - **For example, the run time of Prim's minimum spanning tree algorithm can be reduced from $\Theta(n^2)$ to $\Theta(|E| \log n)$.**
- **Sparse algorithms use adjacency list instead of an adjacency matrix.**
- **Partitioning adjacency lists is more difficult for sparse graphs - do we balance number of vertices or edges?**
- **Parallel algorithms typically make use of graph structure or degree information for performance.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Algorithms for Sparse Graphs

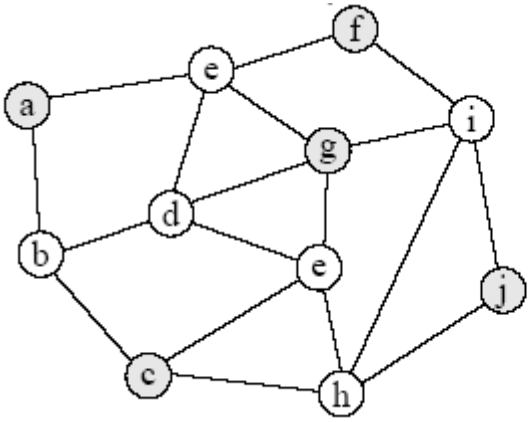


A street map (a) can be represented by a graph (b). In the graph shown in (b), each street intersection is a vertex and each edge is a street segment. The vertices of (b) are the intersections of (a) marked by dots.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finding a Maximal Independent Set (极大独立集)

- **A set of vertices $I \subset V$ is called *independent* if no pair of vertices in I is connected via an edge in G. An independent set is called *maximal* if by including any other vertex not in I, the independence property is violated.**



{a, d, i, h} is an independent set

{a, c, j, f, g} is a maximal independent set

{a, d, h, f} is a maximal independent set

Examples of independent and maximal independent sets.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finding a Maximal Independent Set (MIS)

- **Simple algorithms start by MIS I to be empty, and assigning all vertices to a candidate set C.**
- **Vertex v from C is moved into I and all vertices adjacent to v are removed from C.**
- **This process is repeated until C is empty.**
- **This process is inherently serial!**

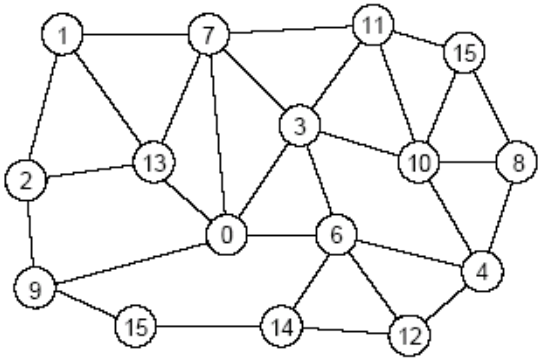
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finding a Maximal Independent Set (MIS)

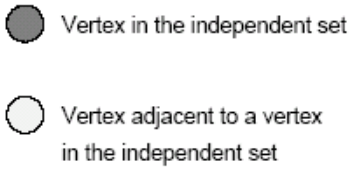
- **Parallel MIS algorithms use randomization to gain concurrency (Luby's algorithm for graph coloring).**
- **Initially, each node is in the candidate set C. Each node generates a (unique) random number and communicates it to its neighbors.**
- **If a node's number is smaller than that of all its neighbors, it joins set I. All of its neighbors are removed from C.**
- **This process continues until C is empty.**
- **On average, this algorithm converges after $O(log|V|)$ such steps.**

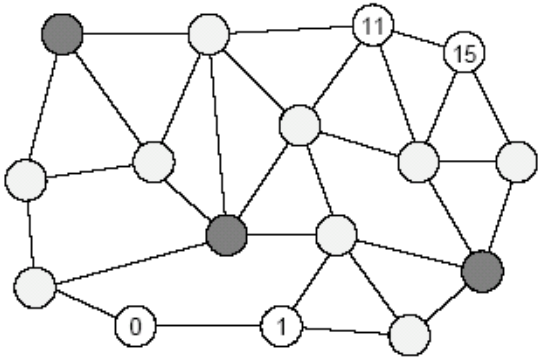
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finding a Maximal Independent Set (MIS)

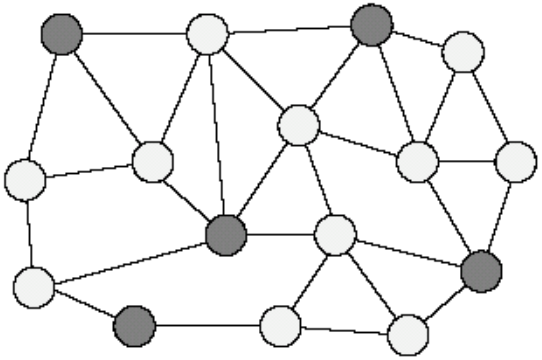


(a) After the 1st random number assignment





(b) After the 2nd random number assignment



(c) Final maximal independent set

The different augmentation steps of Luby's randomized maximal independent set algorithm. The numbers inside each vertex correspond to the random number assigned to the vertex.

Finding a Maximal Independent Set (MIS): Parallel Formulation (Shared Addr. Space)

- **We use three arrays, each of length n**
  - **I, which stores nodes in MIS**
  - **C, which stores the candidate set, and**
  - **R, the random numbers.**
- **Partition C across p processes. Each process generates the corresponding values in the R array, and from this, computes which candidate vertices can enter MIS.**
- **The C array is updated by deleting all the neighbors of vertices that entered *MIS*.**
- **The performance of this algorithm is dependent on the structure of the graph.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Finding a Maximal Independent Set (MIS): Parallel Formulation (Distributed Memory)

- **Each vertex is assigned a single random number**
- **After communication, each vertex determines the *number* of its adjacent vertices that have smaller and greater random numbers**
- **Loop**
  - **each vertex waits to receive the color values of its adjacent vertices that have smaller random numbers**
  - **Once all these colors have been received, the vertex selects a consistent color, and sends it to all of its adjacent vertices with greater random number**
- **Terminates when all vertices have been colored**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths

- **Dijkstra's algorithm, modified to handle sparse graphs is called Johnson's algorithm.**
- **The modification accounts for the fact that the minimization step in Dijkstra's algorithm needs to be performed only for those nodes adjacent to the previously selected nodes.**
- **Johnson's algorithm uses a priority queue Q to store the value $l[v]$ for each vertex $v \in (V - V_T)$.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths: Johnson's Algorithm

```
1.   procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
2.   begin
3.       Q := V;
4.       for all v ∈ Q do
5.           l[v] := ∞;
6.       l[s] := 0;
7.       while Q ≠ ∅ do
8.       begin
9.           u := extract_min(Q);
10.          for each v ∈ Adj[u] do
11.              if v ∈ Q and l[u] + w(u, v) < l[v] then
12.                  l[v] := l[u] + w(u, v);
13.      endwhile
14.  end JOHNSON_SINGLE_SOURCE_SP
```

Johnson's sequential single-source shortest paths algorithm.

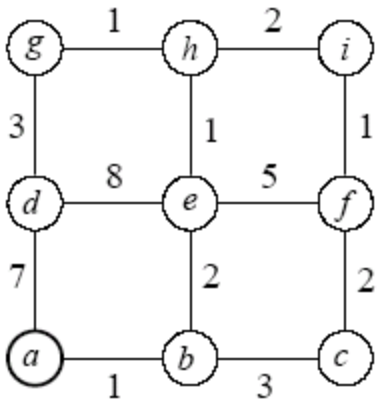
A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths: Parallel Johnson's Algorithm

- **Maintaining strict order of Johnson's algorithm generally leads to a very restrictive class of parallel algorithms.**
- **We need to allow exploration of multiple nodes concurrently. This is done by simultaneously extracting *p* nodes from the priority queue, updating the neighbors' cost, and augmenting the shortest path.**
- **If an error is made, it can be discovered (as a shorter path) and the node can be reinserted with this shorter path.**

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

Single-Source Shortest Paths: Parallel Johnson's Algorithm



Priority Queue

(1) *b:1, d:7, c:inf, e:inf, f:inf, g:inf, h:inf, i:inf*

(2) *e:3, c:4, g:10, f:inf, h:inf, i:inf*

(3) *h:4, f:6, i:inf*

(4) *g:5, i:6*

Array l[]

a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
0	1	∞	7	∞	∞	∞	∞	∞
0	1	4	7	3	∞	10	∞	∞
0	1	4	7	3	6	10	4	∞
0	1	4	7	3	6	5	4	6

An example of the modified Johnson's algorithm for processing unsafe vertices concurrently.

A. Grama et al., "Introduction to Parallel Computing," Addison Wesley, 2003

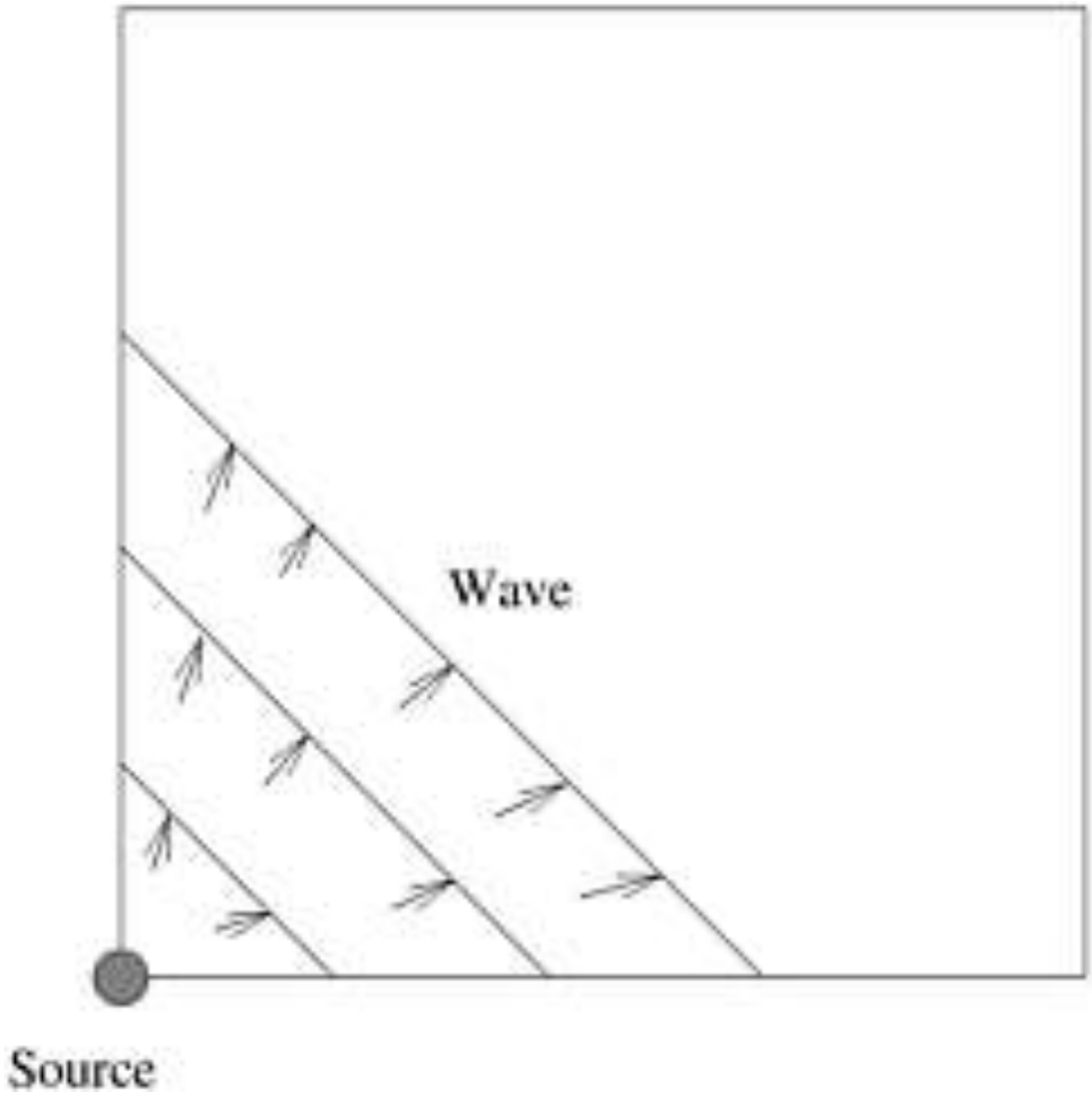
Johnson's Algorithm: Distributed Memory Formulation

- Idea: remove the bottleneck of working with a single priority queue
- V is partitioned among p processes
- P_i has:
  - a local priority queue Q_i
  - an array sp[] that store the shortest path from source to the vertices assigned to P_i
- sp[v] is updated from l[v] each time v is extracted from the queue
  - Initially sp[v] = ∞, sp[s] = 0
  - Each process executes Johnson's algorithm locally
  - At the end sp[v] stores the shortest path from source to vertex v

Johnson's Algorithm: Distributed Memory Formulation

- How to maintain the queues?
  - Assume (u,v) is an edge, P_i has just extracted u from Q_i
  - P_i sends l[u]+w(u,v) to P_j
  - P_j receives the message and sets the values of l[v] in Q_j to min{l[v], l[v]+w(u,v)}
  - P_j might have already computed sp[v] ⇒ two cases
    - If sp[v] ≤ l[u]+w(u,v) ⇒ longer path ⇒ P_j does nothing
    - If sp[v] > l[u]+w(u,v) ⇒ P_j must insert v back into Q_j with l[v] = l[u]+w(u,v) and disregards the value sp[v]
- The algorithm terminates when all the queues are empty

Johnson's Algorithm: Distributed Memory Formulation



Summary

- Minimum Spanning Tree: Prim's Algorithm
- Single-Source Shortest Paths: Dijkstra's Algorithm
- All-Pairs Shortest Paths
- Transitive Closure
- Connected Components
- Algorithms for Sparse Graphs

A. Grama et al., “Introduction to Parallel Computing,” Addison Wesley, 2003