Parallel Graph Algorithms

Grama et. al.
Introduction to Parallel Computing, chapter 10
Mikael Rännar after material by
Robert Granat, Isak Jonsson och Erik Elmroth
7/5 2010

Overview

- Graphs - definitions, properties, representation
- Minimal spanning tree
  - Prim's algorithm
- Shortest path (1-to-all)
  - Dijkstra's algorithm
- Shortest path (all pairs)
  - Algorithm based on matrix multiplication
  - Dijkstra's algorithm
    - Source partitioned
    - Source parallel
  - Floyd's algorithm
- Transitive closure
- Connected components

Graphs - definitions

Undirected graph:

- G(V, E): V is the set of vertices and E is the set of edges
- An edge e ∈ E is an unordered pair (u, v) where u,v ∈ V

Directed graph:

- An edge e ∈ E is an ordered pair (u, v) (from u to v) where u,v ∈ V
- **A path** from u to v is a sequence (u, ..., v) of vertices where consecutive vertices in the sequence corresponds to an edge in the graph
- Simple path: all vertices in the path are distinct
- Cycle: u = v
- Acyclic: contains no cycles

Examples of graphs

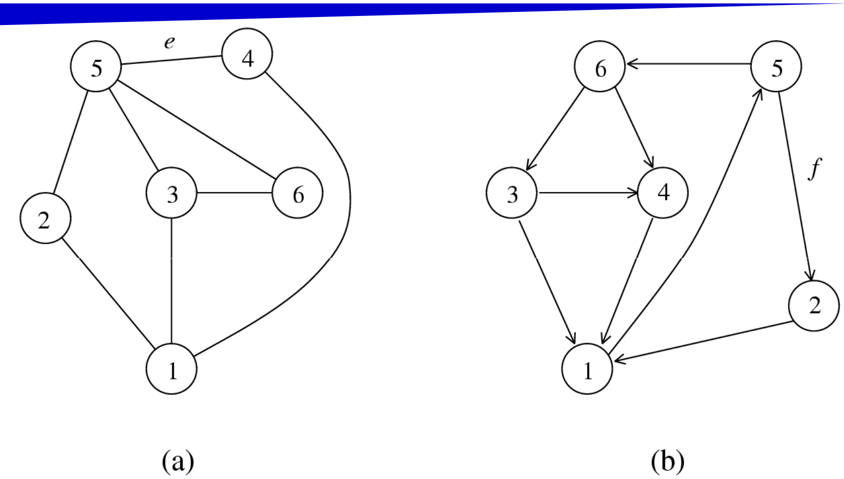


Figure 7.1 (a) An undirected graph and (b) a directed graph.

Graphs - properties

- A graph is *connected* if it exists a path between every pair of vertices
- A graph is *complete* if it exists an edge between every pair of vertices
- G′(V′, E′) is a *subgraph* of G(V, E) if V′ ∈ V and E′ ∈ E
- A *tree* is a connected acyclic graph
- A *forest* consists of several trees
- A graph G(V, E) is sparse if |E| is much smaller than O(|V|^2)
  - Corresponds to a sparse Adj-matris (se nedan)

Weighted graphs:

- G(V, E, w), where w is a real valued function defined on E (every existing edge has a value)
- The weight of the graph is the sum of the weights of its edges

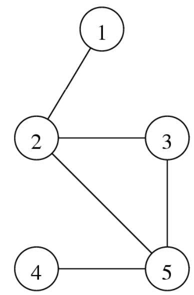
Matrix Representation of Graphs

Non weighted graph

$$a_{i,j} = \begin{cases} 1 & \text{om } (v_i, v_j) \in E \\ 0 & \text{annars} \end{cases}$$

Weighted graph

$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{om } (v_i, v_j) \in E \\ 0 & \text{om } i = j \\ \infty & \text{annars} \end{cases}$$



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

Suitable for dense graphs

Figure 7.2 An undirected graph and its adjacency matrix representation.

List Representation of Graphs

- G(V, E) is represented by the list Adj[1..|V|] of lists
- For each v ∈ V is Adj[v] a linked list of all vertices that has an edge in common with v

Suitable for sparse graphs

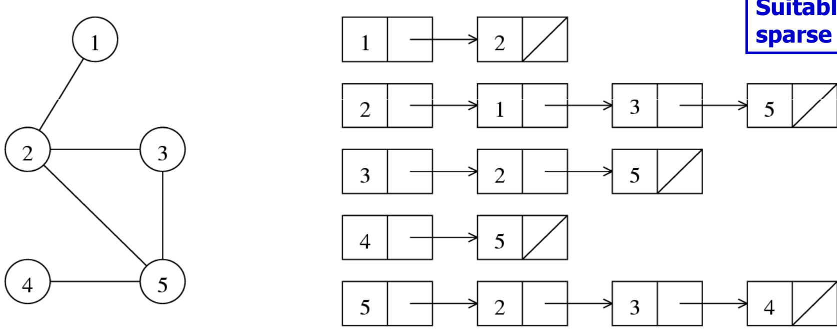


Figure 7.3 An undirected graph and its adjacency list representation.

Minimum spanning tree (MST)

- A **spanning** tree contains all the vertices of the graph
- MST for a weighted graph is a spanning tree with minimum weight
- If G is not connected it can not have a MST (instead it has a minimum spanning forest)
- Assume from now on that G is connected (if not we can find connected components and apply the MST algorithm on each to get a minimum spanning forest)

Prim's algorithm for minimum spanning tree

Repeat until all vertices are included

- Select an arbitrary vertex u
- Chooses vertex v such that the edge (u, v) is guaranteed to be in the MST
- Let A = (a_{ij}) be the matrix representation of G = (V, E, w)
- Let V_T be the set of included vertices in the MST
- Let d[1..n] be a vector where d[v] for each v $\in$ (V - V_T) is the weight for the edge with the least weight from any vertex in V_T to v
- For each iteration a new vertex v is chosen such that d[v] is minimal

Cost:

- The while-loop is executed n-1 times
- min-operation (line 10) and the for-loop (line 12-13) each takes O(n) steps

In Total: $\Theta(n^2)$ steps

Prim's algoritm

```
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
```

Initialization: vertices with edge to r (initial vertex) is given cost = weight of the edge. All other vertices is given the cost = inf

Of all vertices outside V_T with an edge to V_T the edge with smallest weight is chosen

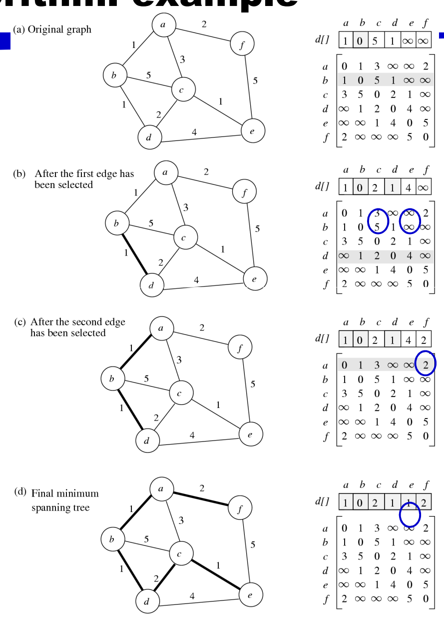
Include the vertex

Update d-values

Program 7.1 Prim's sequential minimum spanning tree algorithm.

Prim's algorithm: example

Start node: b



Parallelizing Prim's algorithm

- d[v] is updated for all v in every iteration
  → can not choose 2 vertices at the same time
  → can not parallelize the while-loop
  Instead we parallelize the for-loop!
- Every processor holds a block-column (n/p columns) of A and corresponding part of d
- V_i is the subset of vertices belonging to P_i
- Every processor computes d[u] for its vertices
- Global minimum for d[u] is computed by a all-to-one-reduction
- The processor holding the global minima broadcasts the new vertex u
- The processor responsible for u marks that u belongs to V_T and all processors updates d[v] for their local vertices

Matrix partitioning for Prim's algorithm

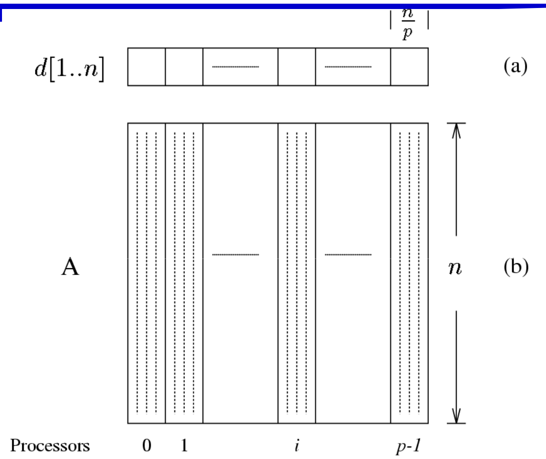


Figure 7.6 The partitioning of the distance array d and the adjacency matrix A among p processors. Copyright (r) 1994 Benjamin/Cummings Publishing Co.

Analysis of the parallel Prim's algorithm

- Since every node is to compute d[v] for all their vertices, they have to have the whole columns for their vertices in Adj
- Cost to update the d-values for each processor is $\Theta(n/p)$
- Comm.cost.: reduction + broadcast
- Total Tp = $\Theta(n^2/p) + \Theta(n \log p)$

In every iteration

<u>Updating:</u>
...
for all $v \in (V - V_T)$ that I own
  d[v] = min{d[v], w(u, v)}
end

Shortest Path (from 1 to all)

Dijkstra's algorithm: (from starting node s)

- Basically Prim's algorithm but..
- Instead of storing d[u] Dijkstra's store l[u] which is the total weight from s till u (i.e., the minimum cost to reach vertex u from vertex s by means of vertices in V_T)

Parallel Dijkstra's algorithm like parallel Prim's with the change above. The analysis is identical!

Shortest Path (all pairs)

- Find the shortest path between all pairs of vertices
- The resultat is an n x n-matrix D = d_{ij}, where d_{ij} is the shortest path from vertex v_i to vertex v_j

Algorithm based on matrix multiplication

- Let G = (V, E, w) be represented by the matrix A
- Let d_{ij}^k represent the shortest path from v_i to v_j that contains a maximum of k edges
- Let v_m be a vertex in that path
- Then $d_{ij}^k = \min\{d_{im}^{k-1} + w(v_m, v_j)\}$ (where the minimizing is done over m from 1 to n and $d_{ij}^1 = a_{ij}$)
- We create a matrix D for each maximum path length: $D^k = (d_{ij}^k)$
- Since the shortest path between two vertices always is maximum n-1 then D^{n-1} contains all pair's shortest paths

Matrix multiplication algorithm (cont.)

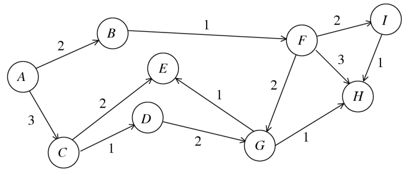
- D^k is computed from D^{k-1} with a modified matrix multiplication:

$$c_{ij} = \min_{k=1}^{k=n} a_{ik} + b_{kj}$$

(Find k that gives minimal c_{ij})

- Since D^1 = A then $D^k = A^k$ is computed by the modified matrix multiplication
- Compute the result D_{n-1} by computing A^2, A^4, A^8,... A^{n-1}, with the modified matrix multiplication in log n steps
- The complexity $\Theta(n^3)$ for matrix multiplication gives $\Theta(n^3 \log n)$ in total
- Parallelization: the same parallel algorithms as in the usual matrix multiplication, e.g. in $\Theta(\log n)$ time with the DNS-algorithm

Matrix multiplication algorithm (example)



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

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

Figure 7.7 An example of the matrix-multiplication-based all-pairs shortest paths algorithm.

Dijkstra's algorithm for shortest path: all pairs

- The problem is solved for all pairs by applying the 1-vertex-algorithm on each vertex in the graph
- The complexity of the 1-vertex-algorithm $\Theta(n^2)$ gives $\Theta(n^3)$ for all pairs
- 2 parallel approaches - source partitioned & source parallel

Source-partitioned

- Every processor get one vertex and solves the 1-vertex-problem sequentially for that vertex
- No communication
- Is optimal from a communication point of view, but can only use n processors
- Also requires p times as much memory!

Dijkstra's, all pairs (approach 2)

Source-parallel

- The processors are divided into n partitionseach with p/n processors (p > n)
- Each partition solves the 1-vertex-problem for 1 vertex with the parallel 1-vertex-algorithm $\rightarrow$ 2 levels of parallelism:
  - Coarse grain: Every node in the graph is handled independently of the others
  - Fine grain: p/n processors share the work with one vertex

On a 2-dimensionell mesh:

- $p^{1/2}$ x $p^{1/2}$ mesh is divided into partitions each with $(p/n)^{1/2}$ x $(p/n)^{1/2}$ processors
- No communication between processor groups
- The parallel 1-vertex-algorithm for a 2-dimensionell mesh within each group

Shortest Path (all pairs): Floyd's algorithm

- Let V = $\{v_1, v_2, ..., v_n\}$ (all vertices in G)
  let $V^k = \{v_1, v_2, ..., v_k\}$, $k \leq n$ be a subset of V.
- For any pair $v_i, v_j \in V$, consider all paths whose intermediate vertices belongs to V^k.
- Let p_{ij}^k be the shortest of these paths (with weight d_{ij}^k)
- If the vertex v_k is not in the path then p_{ij}^k is the same as p_{ij}^{k-1}
- If v_k is in the path then the path can be split into two paths: One from v_1 to v_k and one from v_k to v_j where both paths uses vertices in $V^{k-1} = \{v_1, v_2, ..., v_{k-1}\}$
- I that case the the weight of the path is $d_{ij}^k = d_{ij}^{k-1} + d_{kj}^{k-1}$

Floyd's algorithm (cont...)

Thus we get the recursion:

$$d_{ij}^{k} = \begin{cases} w(v_i, v_j) & \text{om } k = 0 \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\} & \text{om } k \geq 1 \end{cases}$$

And the 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

Program 7.3 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.

Parallel Floyd's algorithm (block-checkerboard...)

- D^k is partitioned into p block of size $(n/p^{1/2})$ x $(n/p^{1/2})$

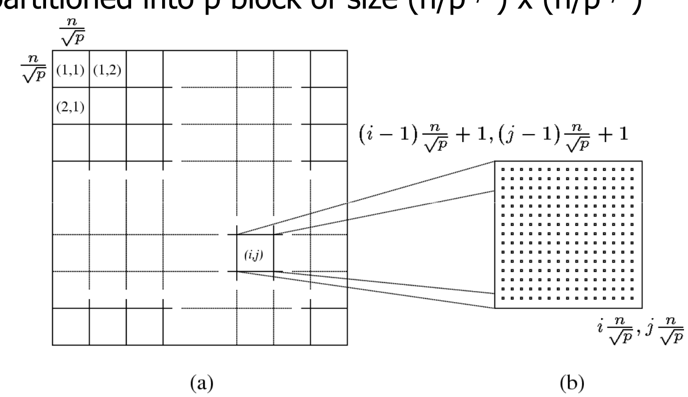


Figure 7.9 (a) Matrix $D^{(k)}$ partitioned by block checkerboarding into $\sqrt{p} \times \sqrt{p}$ subblocks, and (b) the square subblock of $D^{(k)}$ assigned to processor $P_{i,j}$.

Parallel Floyd's algorithm (cont.)

- In iteration k processor P_{ij} needs data from column k and row k in the D^{k-1}-matrix

Example: to compute d_{lr}^{k} it needs d_{lk}^{k-1} and d_{kr}^{k-1}

- In iteration k all p processors holding data from row k sends these elements to the other processors in the same processor column

- Analogously processors storing elements from column k sends these elements to the other processors in the same processor row

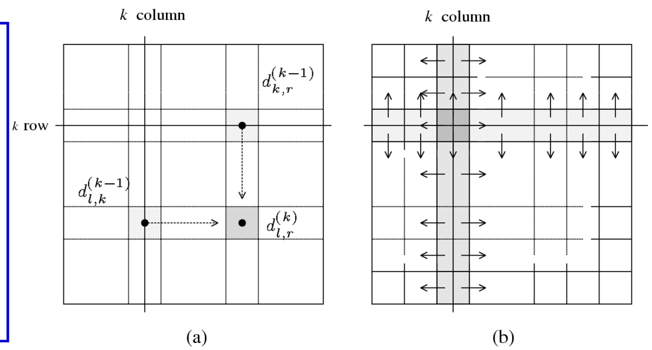


Figure 7.10 (a) Communication patterns used in the block-checkerboard partitioning. When computing $d^{(k)}_{i,j}$, information must be

Parallel Floyd's algorithm on a hypercube

- The 2D mesh is mapped to the hypercube such that each processor row and processor column in the mesh corresponds to a subcube

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


Program 7.4 Floyd's parallel formulation using the block-checkerboard partitioning. $P_{*,j}$

Transitive closure

- If G = (V, E) is a graph then its transitive closure is a graph G^* = (V, E^*), where $E^* = \{(v_i, v_j) |$ exists a path from v_i to v_j in G$\}$
- Compute the connectivity-matrix A^* such that $a_{ij} = 1$ if i = j or a path from v_i to v_j exists and $a_{ij} = \infty$ otherwise.

Method 1:

- Set the weights in G to 1 and compute the shortest path between all pairs. Interpret the resultat D such that

$d_{ij} = \infty \quad \rightarrow \quad a_{ij} = \infty$ and

$i = j$ or $d_{ij} > 0 \quad \rightarrow \quad a_{ij} = 1 \qquad (\Rightarrow A^*)$

Method 2:

- Modify Floyd's algorithm by changing min and+ on line 7 to logical *or* and logical *and*. $d_{ij}^{(k)} = d_{ij}^{(k-1)}$ OR $(d_{ik}^{(k-1)}$ AND $d_{kj}^{(k-1)})$

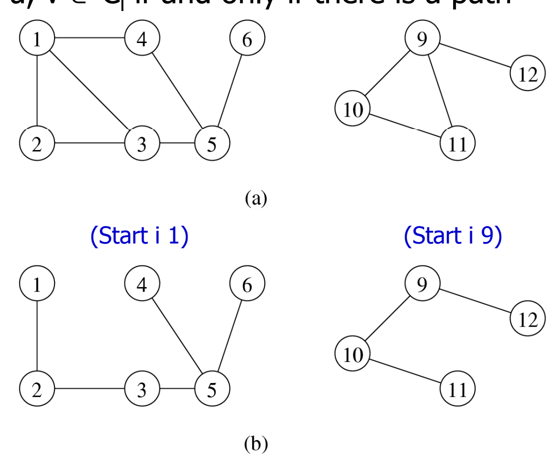
Connected Components

- The connected components of G = (V, E) = maximum disjoint sets $C_1, C_2, ..., C_k$ such that $V = C_1 \cup C_2 \cup ... \cup C_k$ and $u, v \in C_i$ if and only if there is a path between u and v

Depth-first-search based algorithm:

- Given a forest of depth-first trees

- Each tree only contains components that does not belong to another tree



// algorithm for connected components

- Let G = (V, E) be represented by the matrix A
- Distribute A with 1 part on each process
  $\rightarrow$ each process has a subgraph $G_i = (V, E_i)$ of G

All processes has all vertices but not all edges

Step 1:

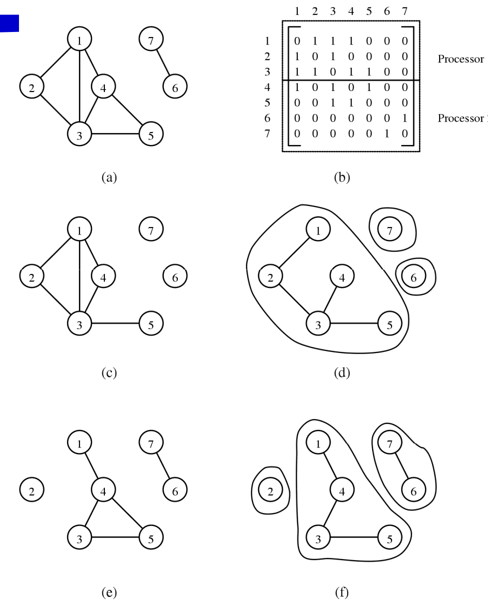
- All processes computes a depth-first spanning forest for their subgraph

Step 2:

- All the spanning forests are pairwise united to one forest:

  Given the spanning forests A and B is the following done for each edge (u, v) in A:
  - If the vertices u and v are in the same tree in B – do nothing
  - Otherwise, take B's tree containing u and unite with B's tree with v

// algorithm for connected components



Algorithms for sparse graphs

- G = (V, E) is sparse if $|E| << |V|^2$
- Algorithms for dense graphs still works for sparse, but they are often ineffective
  - For example, Prim's algorithm for minimum spanning tree takes $\Theta(n^2)$ time, regardless of the number of edges
  - With a list representation the algorithm can be modified to $O(|E| \log n)$ (more efficient when $|E| = O(n^2/\log n)$)
- The complexity for the algorithm with matrix storing is normally $W(n^2)$ while for the list representation it normally is $W(n + |E|)$

Work distribution

Matrix representation

- The matrix is evenly distributed on the processors
  - Balanced work load
  - Little communication (each processor have consecutive rows/columns)

List representation

- The number of lists are distributed evenly on the proc.
  - Can give an unbalanced work load as the lists often are of different length
- The number of edges are distributed evenly on the proc.
  - May demand that the lists are shared between processors ==> increased communication

$\rightarrow$

- Hard to make good algorithms for general sparse graphs
- Special algorithms for different types of sparse graphs
- Sparse graphs with mesh structure (grid graphs)

Shortest path (1-to-all) Johnson's algorithm

Repetition of Dijkstra's algorithm

- Finds vertices u ∈ (V - V_T) such that
  l[u] = min{l[v] | v ∈ (V - V_T)} and puts them into V_T
- For each v ∈ (V - V_T) compute
  l[v] = min{l[v] | l[u] + w(u,v)}

Modification into Johnson's algorithm

- Priority queue Q for all v ∈ (V - V_T) sorted by size of l[v] (least first)
- Initially all l[v] = ∞, except starting vertex s which has l[s] = 0
- For each step the vertex u with the smallest value l[u] is taken from the queue, u's list is traversed and for each edge (u, v) the l[v] is updated. (Only vertices in the same list need to be investigated.) After that the queue is sorted.

Johnson's algorithm

```
procedure JOHNSON_SINGLE_SOURCE_SP(V, E, s)
begin
    Q := V;
    for all v ∈ Q do
        l[v] := ∞;
    l[s] := 0;
    while Q ≠ ∅ do
    begin
        u := extract_min(Q);
        for each v ∈ Adj[u] do
            if v ∈ Q and l[u] + w(u, v) < l[v] then
                l[v] := l[u] + w(u, v);
    endwhile
end JOHNSON_SINGLE_SOURCE_SP
```

Program 7.5 Johnson's sequential single-source shortest paths algorithm.

Parallel Johnson's (central queue)

- 1 processor keeps the priority queue
- All other updates l[v] for v ∈ (V - V_T) and sends new values to the processor with the priority queue
- In each iteration the algorithm updates roughly
  |E|/|V| vertices ==> a maximum of |E|/|V| proc. can have work
- For each new edge the queue is updated in
  O(log n) time (take the old l[v] away and put a new value in)
- A total of |E| edges requires O(|E| log n) time
- The same order as the sequential algorithm

Parallel Johnson's (distributed queue)

- Distribute V on the processors in p disjoint sets such that P_i has V_i
- Each processor
  - has a priority queue Q_i for their own vertices
  - has a vector sp, where in the end sp[v] will be the cost for the shortest path from s to v
  - executes Johnson's algorithm on their own subgraph
- Initially l[v] = ∞ for all vertices except s
  (l[s] = 0)
- Each time a vertex v is removed from the queue set sp[v] = l[v]

Dependencies in Johnson's algorithm

- If $u \in V_i$ and $v \in V_j$ when P_i removes u from Q_i then P_i sends information to P_j about that l[v] possibly could have the a value l[u] + w(u,v)
- P_j sets l[v] = min{l[v], l[u] + w(u,v)}
- Since P_j also executes Johnson's algorithm P_j may already have removed v from the queue Q_j. Then we can have two cases:
  - If l[u] + w(u,w) ≥ sp[v] then the already found path is shorter. Then P_j does not have to do anything.
  - If sp[v] > l[u] + w(u,w) then the path via u is shorter than the shortest path found so far.
    P_j lets l[v] = l[u] + w(u,w), removes the value of sp[v] and puts back v into Q_j.
- The algorithm is not terminated until all queues are empty!
- Note, that due to the distributed queue the removal of values from the queue is not always done in the "right" order and then "unnecessary" work is done by the parallel algorithm.

Example of extra work

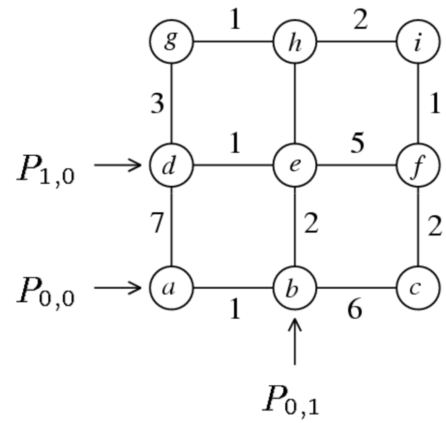


Figure 7.17 A grid graph.
Copyright (r) 1994 Ben-

Wave of activity in the priority queues

- Initially only the processor with the source vertex has a non-empty queue
- Wavefront of activity in the queues
- A processor has no work before the wavefront has arrived and after it has passed

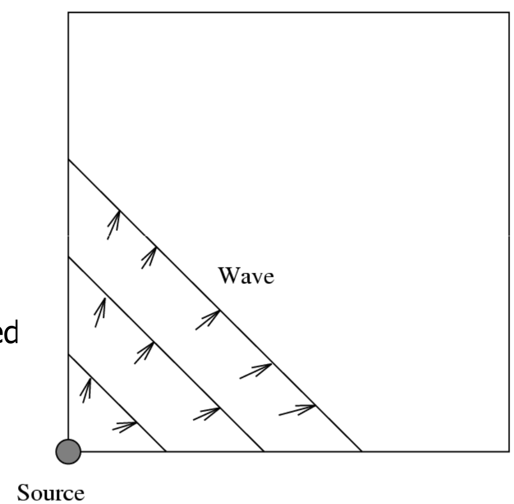


Figure 7.18 The wave of activity

Block checkerboard partitioning

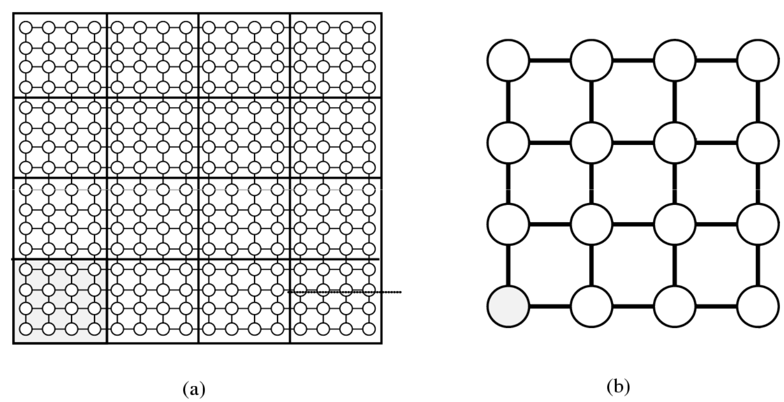


Figure 7.19 Mapping the grid graph (a) onto a mesh (b) by using the block-checkerboard mapping. In this example, $n = 16$ and $\sqrt{p} = 4$. The shaded vertices are mapped onto the shaded processor.

Cyclic checkerboard partitioning

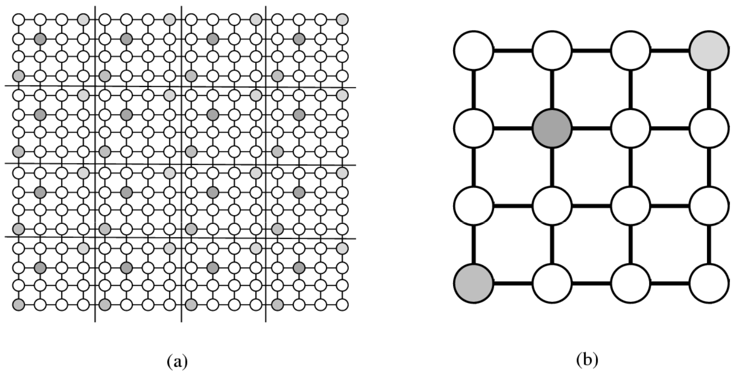


Figure 7.20 Mapping the grid graph (a) onto a mesh (b) by using the cyclic-checkerboard mapping. In this example, $n = 16$ and $\sqrt{p} = 4$. The shaded graph vertices are mapped onto the correspondingly shaded mesh processors.

Block column partitioning

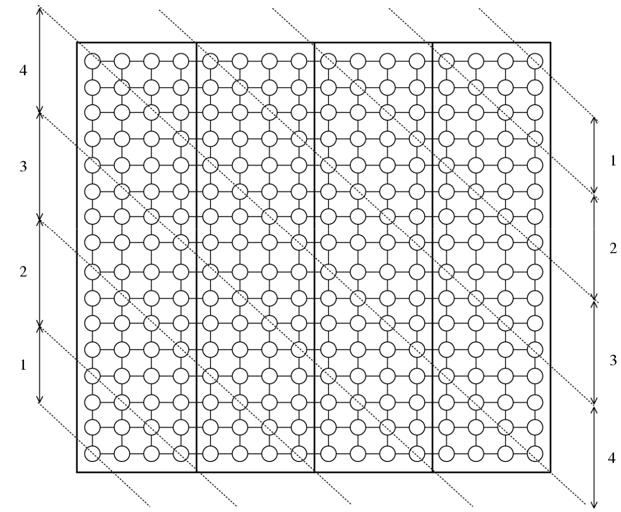


Figure 7.22 The number of busy processors as the computational wave propagates across the grid graph