Graph Algorithms

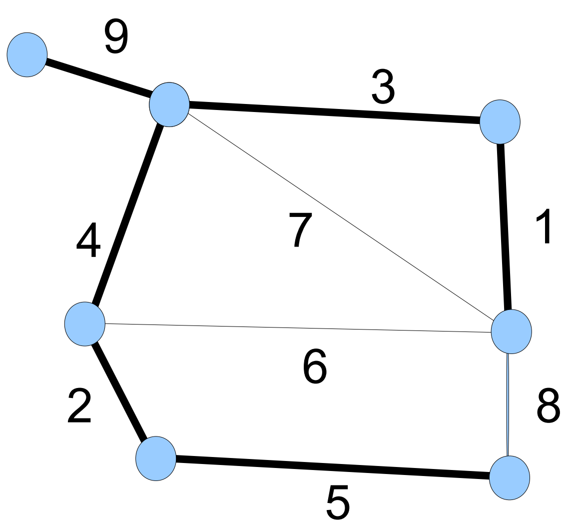
Today: Minimum spanning trees and Shortest paths

Minimum Weight Spanning Tree

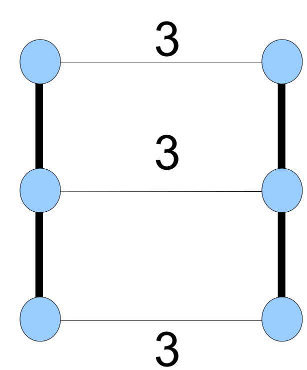
`G=(V,E)` is an undirected graph containing **`n=|V|`** vertices and **`m=|E|`** weighted edges

Problem: Compute a spanning tree of minimum weight

Cut property:
Partition the vertices into two groups A and V-A. Then the lightest edge between a vertex in A and one in V-A belongs to the MST (assuming all edge weights unique)



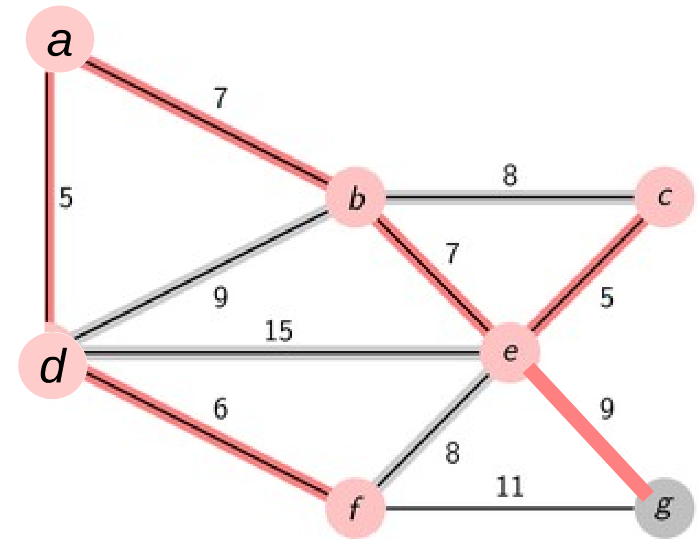
Multiple edges of minimum weight:
→ Pick any one across cut



Minimum Weight Spanning Tree

Prim's Algorithm:
Idea: Expand a partial connected spanning tree **T** as cheaply as possible

Choose any vertex $v \in V$
$V_T = \{v\}$, $E_T = $ ˆ
Repeat $n-1$ times
    Pick vertex $w \notin V_T$ closest to some vertex $u \in V_T$
    Add w to V_T and $\{u, w\}$ to E_T
    Update minimum distance to each vertex not in V_T



Implementation: Maintain a data-structure (heap or list) of vertices not in **T** so that one easily can find the vertex with minimum distance to a node in **T**

Worst case running time: $\Theta((m+n)\lg(n))$ (heap) or $\Theta(n^2)$ (list)
Vertices must be chosen in correct order
    → Sequential algorithm

Parallelizing Prim's algorithm

Selection of tree edges is inherently sequential.

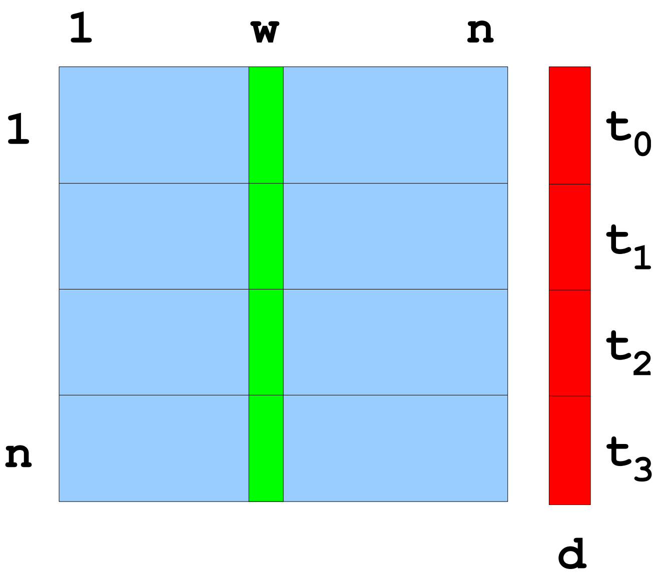
What can be done in parallel?

- Select vertex **w** to be added to **T**
- Update distances

Assume complete graph and **p<n** threads.
Array **d[1..n]** holds distances from **T** to each vertex

Running time per iteration:
Sequential: Finding **min** and updating: $\Theta(n)$
Parallel: $\Theta(n/p)$
Overhead: Reduction for overall **min** : $\Theta(lg(p))$

- $T_s \in \Theta(n^2)$
- $T_p \in \Theta(n^2/p + n lg(p))$



Weight matrix

Parallelizing Prim's algorithm

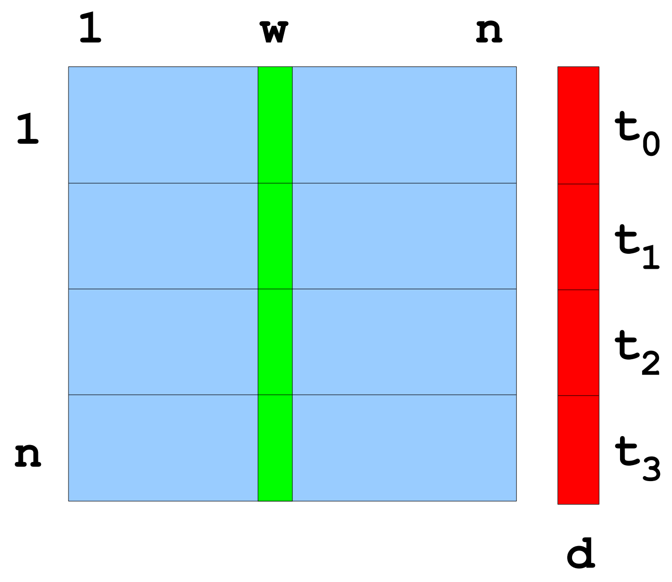
Selection of tree edges is inherently sequential.

What can be done in parallel?

- Select vertex **w** to be added to **T**
- Update distances

Assume complete graph and **p<n** threads.
Array **d[1..n]** holds distances from **T** to each vertex

Running time per iteration:
Sequential: Finding **min** and updating: $\Theta(n)$
Parallel: $\Theta(n/p)$
Overhead: Reduction for overall **min** : $\Theta(\lg(p))$



Weigth matrix

- $T_s \in \Theta(n^2)$
- $T_p \in \Theta(n^2/p + n\lg(p))$

Works for Dijkstra's SP-algorithm as well!

Other parallel MST algorithms

Kruskal's algorithm: Sort edges by increasing weight, pick edges that do not cause a cycle.

Boruvka's algorithm:
Start with every node being a separate component in **T**
While **T** is not connected
    For each component **C**
        Pick cheapest edge **e** connecting **C** with another component **C'**
        Add **e** to **T**, merge **C** and **C'**

Running time (Boruvka):
Each pass goes through all the components touching each edge at most twice: $\Theta(m)$
Number of passes:
Number of components is halved each time: $O(\lg(n))$ iterations
$T_s \in O(m \lg(n))$

Parallel advantage (Boruvka):

- Can work on several components simultaneously
- Must be careful when components merge:
  Can use hash table to compute new edges

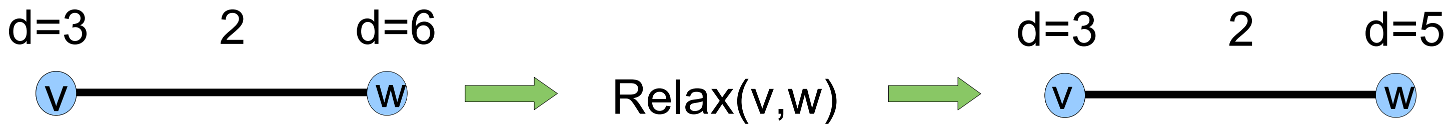
Shortest path computations

G=(V,E) is an undirected graph containing **n=|V|** vertices and **m=|E|** weighted edges

Problem: Compute shortest path from a starting vertex s to every other vertex in G

Relaxing an edge:
Let d[v] be the current length of a shortest path from s to v.

Relax(v,w)
If d[v] + weight(v,w) < d[w]
d[w] = d[v] + weight(v,w)



Fact:
If no further relax operations are possible, then the d-values give the length of the shortest paths from s to every vertex.

Single source shortest path

Perform Relax() from vertex closest to s:

Dijkstra´s algorithm
Sufficient with Relax() operations once from each vertex:
    O(n) iterations, Total cost: O(m log(n)) or O(n^2)   Heap vs Array for storing d-values

Perform Relax() on every edge until convergence:

Bellman – Ford´s algorithm   Handles negative weights
    O(n) iterations, Total cost O(nm)

Perform Relax from a set of closest vertices:

Δ – stepping algorithm
    Place v in bucket: trunk(d[v]/Δ).
    Perform Relax() on first non-emtpy bucket
    First on edges of length < Δ, then on longer edges

Only consider vertices where neighbor value changed in previous iteration

What can be done in parallel for each of these algorithms?

All-to-all shortest path

Problem: Given complete weighted graph `G`:
Compute shortest path between each vertex pair `(i,j)`

All-to-all shortest path

Recall:

Algorithm Prim for MST {

V_T = {1}, E_T = ˆ, for **v=2,...,n {d[v]=c[1][v];close[v]=1}**

Repeat **n-1** times {

Pick some vertex **w ∊ argmin{d[v]:v∉V_T}**

Add **w** to **V_T** and **{close[w],w}** to **E_T**

for **v ∉ V_T** if **(c[w][v]<d[v]){d[v]=c[w][v];close[v]=w}**

}

}

Running time: **Θ(n^2)**

All-to-all shortest path

Recall:

Algorithm Prim for MST {
  V_T = {1}, E_T = ˆ, for v=2,...,n {d[v]=c[1][v];close[v]=1}
  Repeat n-1 times {
    Pick some vertex w ∊ argmin{d[v]:v∉V_T}
    Add w to V_T and {close[w],w} to E_T
    for v ∉ V_T if (c[w][v]<d[v]){d[v]=c[w][v];close[v]=w}
  }
}

Algorithm Dijkstra for SPT rooted at vertex 1 {
  V_T = {1}, E_T = ˆ, for v=2,...,n {d[v]=c[1][v];close[v]=1}
  Repeat n-1 times {
    Pick some vertex w ∊ argmin{d[v]:v∉V_T}
    Add w to V_T and {close[w],w} to E_T
    for v ∉ V_T if (d[w]+c[w][v]<d[v]){d[v]=d[w]+c[w][v];close[v]=w}
  }
}

Running time: $\Theta(n^2)$

All-to-all shortest path

Problem: Given complete weighted graph **`G`**:
Compute shortest path between each vertex pair **`(i,j)`**

All-to-all shortest path

Problem: Given complete weighted graph **G**:
Compute shortest path between each vertex pair **(i,j)**

Method 1, **p<n**:
Run Dijkstra starting from every node. $T_s(n) \in \Theta(n^3)$

Parallel version: Each process runs Dijkstra starting from **n/p** vertices

$$T(n,p) \in \Theta(n^3/p)$$

Requires no communication

All-to-all shortest path

Problem: Given complete weighted graph G:
Compute shortest path between each vertex pair (i,j)

Method 1, $p<n$:
Run Dijkstra starting from every node. $T_s(n) \in \Theta(n^3)$

Parallel version: Each process runs Dijkstra starting from n/p vertices

$$T(n,p) \in \Theta(n^3/p)$$

Requires no communication

Method 2, $p>n$:
For each vertex v, $k=p/n$ processes collaborate on Dijkstra starting from v

$$T(n,k) = T_{comp} + T_{comm} \in \Theta(n^2/k + n\ \lg(k))$$

$$T(n,p) \in O(n^3/p + n\ \lg(p/n))$$

For p large, communication will dominate

→ no further speedup

Floyd's Algorithm

$r_{ij}(k)$ = Length of shortest i-j-path avoiding vertices $\{k+1,...,n\}-\{i,j\}$

Example: $r_{ij}(0) = c[i][j]$ Direct edge

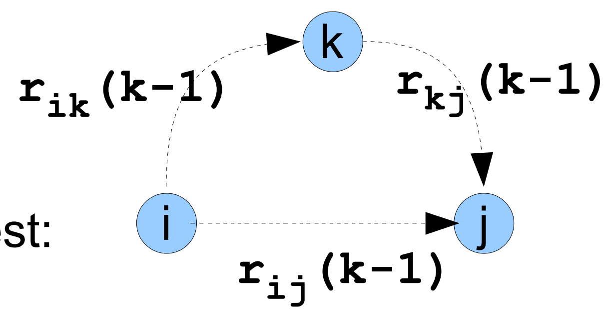
Assume we know $r_{ij}(k-1)$ for all i and j. Compute $r_{ij}(k)$

Idea: Either

- the i-j-path intersects vertex k or
- the i-j-path avoids vertex k

Compute the shortest path for both alternatives and pick smallest:

$$r_{ij}(k) = \min\ \{r_{ik}(k-1)+r_{kj}(k-1),\ r_{ij}(k-1)\}$$



Floyd's algorithm:

```
for (i,j=1,...,n) r[i][j] = c[i][j]
for (k=1,...,n) {
  for (i,j=1,...,n) oldr[i][j] = r[i][j]   // avoid by using pointers
  for (i,j=1,...,n)
    r[i][j] = min {oldr[i][k]+oldr[k][j], oldr[i][j]}
}
```

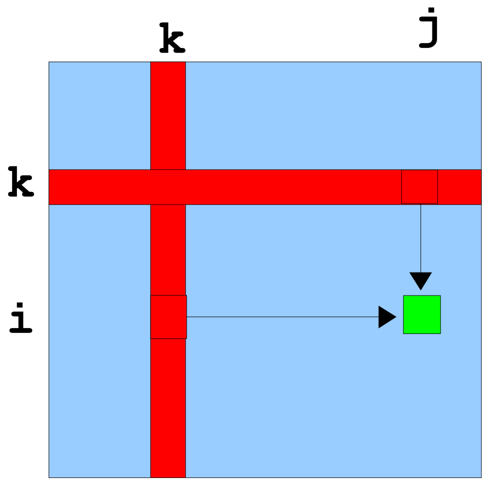
Running time: $T_s(n) \in \Theta(n^3)$ Memory $\in \Theta(n^2)$

Parallelizing Floyd's algorithm

Observations:

- Inherently sequential: Iteration **`k`** of Floyd's algorithm depends on iteration **`k-1`**
- Given **`oldr[1..n][1..n]`** we can compute **`r[1..n][1..n]`** in parallel!

To compute **`r[i][j]`**: Need **`oldr[i][k],oldr[k][j]`**, and **`oldr[i][j]`**



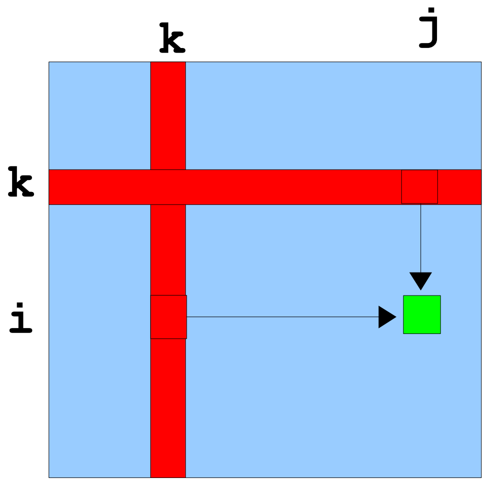
- In iteration **`k`**: Use **`k`**th row and **`k`**th column
- Structurally similar to Gaussian elimination

Parallelizing Floyd's algorithm

Observations:

- Inherently sequential: Iteration **`k`** of Floyd's algorithm depends on iteration **`k-1`**
- Given **`oldr[1..n][1..n]`** we can compute **`r[1..n][1..n]`** in parallel!

To compute **`r[i][j]`**: Need **`oldr[i][k]`**, **`oldr[k][j]`**, and **`oldr[i][j]`**



- In iteration **`k`**: Use **`k`**th row and **`k`**th column
- Structurally similar to Gaussian elimination

- Trivially parallelizable using OpenMP
- Before nested **`i,j`**-loop insert:

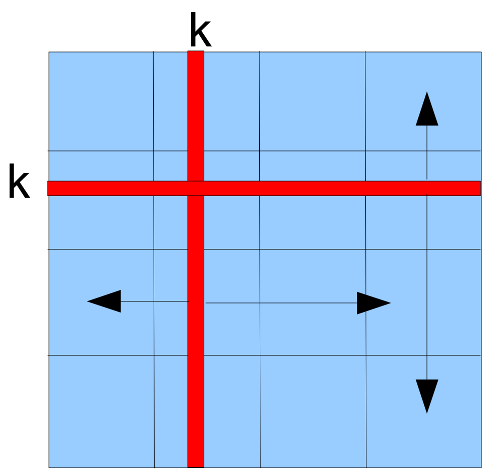
```
#pragma omp parallel for
```

$T(n,p) \in \Theta(n^3/p)$

Floyd on Distributed Memory

Block-wise partitioning ($s^2=p$):

sxs processes, each holding an **(n/s)x(n/s)**-block



Broadcast **k**th row and **k**th column

Each broadcast message

- involves **s** processors
- has length **n/s**

$T_{comp} \in \Theta(n^3/p)$

$T_{comm} = n*(2t_0 + 2t_1 n/s) = 2t_0 n + 2t_1 n^2/s$

where t_0 is the cost to start communication and t_1 is the cost to send one word.

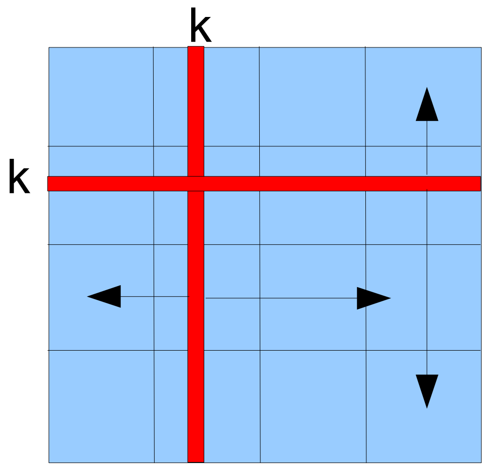
For $p<n^2$, T_{comp} will be (asymptotically) larger than T_{comm}

For $p=n$, we get $T_{comp} \in \Theta(n^2)$ and $T_{comm} \in \Theta(n^{3/2})$

Floyd on Distributed Memory

Block-wise partitioning (s^2=p):

sxs processes, each holding an (n/s)x(n/s)-block



Broadcast kth row and kth column

Each broadcast message

- involves s processors
- has length n/s

$T_{comp} \in \Theta(n^3/p)$

$T_{comm} = n*(2t_0 + 2t_1n/s) = 2t_0n + 2t_1n^2/s$

For n^2>p, T_{comp} will be (asymptotically) larger than T_{comm}

For p=n, we get $T_{comp} \in \Theta(n^2)$ and $T_{comm} \in \Theta(n^{3/2})$

Observation: Floyd's algorithm also solves the transitive closure problem:

Given a digraph D, compute D^* such that

(i,j) is an arc in D^* ⇔ there is a path from i to j in D:

$t_{ij}(k) = (t_{ik}(k-1)$ && $t_{kj}(k-1))$ || $t_{ij}(k-1)$ where t_{ij} is a Boolean variable