Lecture 6
Parallel Algorithms I

Dr. Wilson Rivera

ICOM 6025: High Performance Computing
Electrical and Computer Engineering Department
University of Puerto Rico

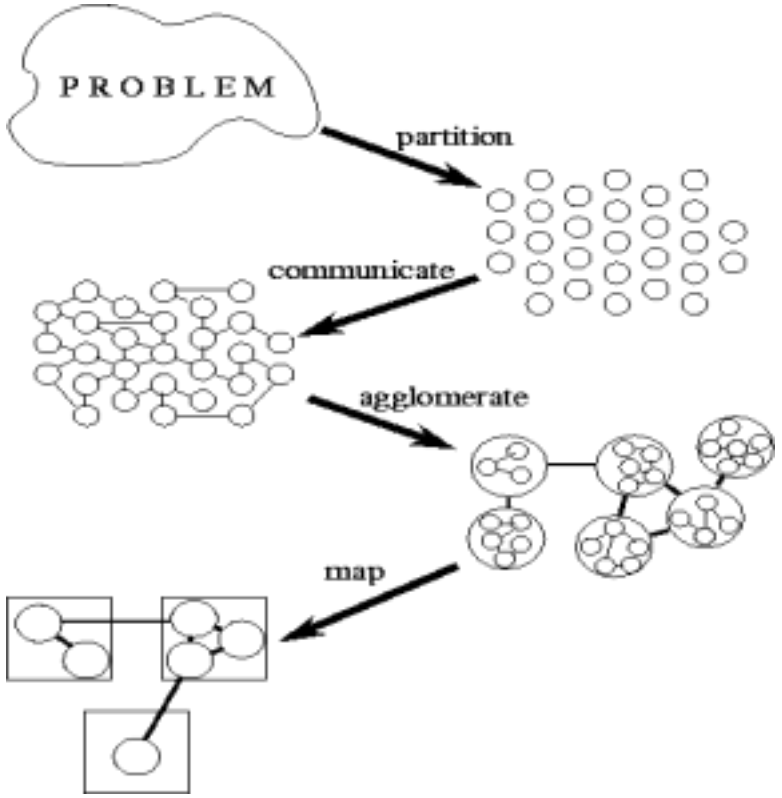
Original slides from Introduction to Parallel Computing (Grama et al.)
Modify by Wilson Rivera

Outline

- Goal: Understand and implement parallel algorithms
  - Methodological design
    - Partitioning, Communication, Agglomeration, mapping
  - Graph definitions and representation
    - Minimum Spanning Tree
      - Prim's Algorithm
    - Single-Source Shortest Path
      - Dijkstra's Algorithm
    - All-Pairs Shortest Path
      - Dijkstra's Algorithm formulations
      - Floyd's Algorithm
    - Algorithms for Sparse Graphs
      - Johnson's Algorithm
    - Large scale graphs

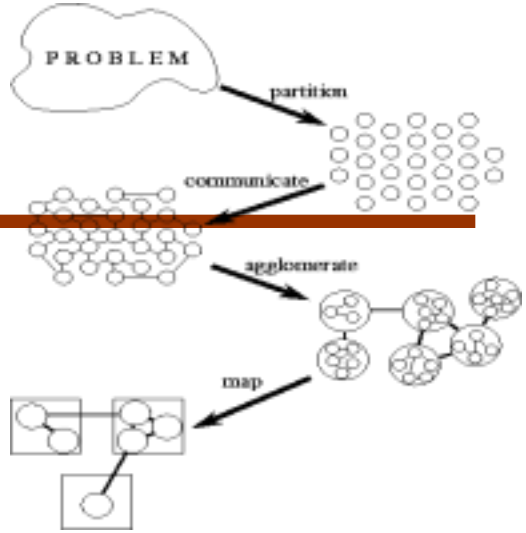
Methodological Design

- Partition
  - Task/data decomposition
- Communication
  - Task execution coordination
- Agglomeration
  - Evaluation of the structure
- Mapping
  - Resource assignment



I. Foster, "Designing and Building Parallel Programs," Addison-Wesley, 1995. Book is online, see webpage.

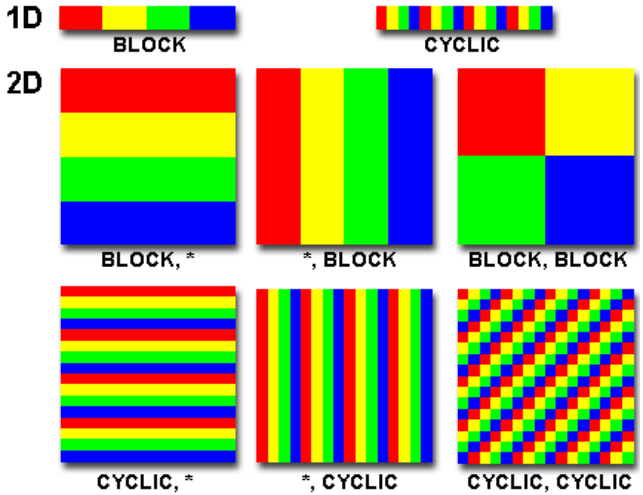
Partitioning



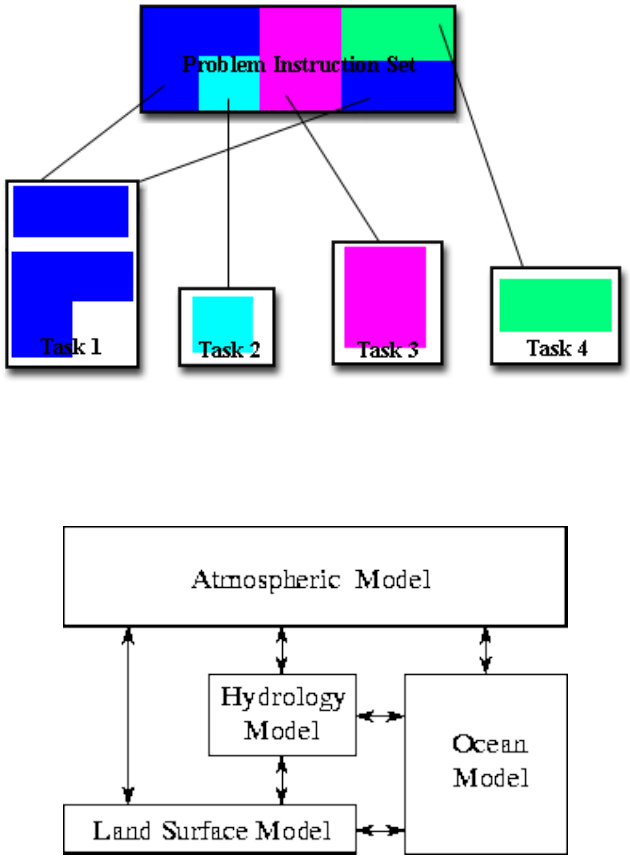
- Focus on defining large number of small task to yield a fine-grained decomposition of the problem
- A good partition divides into small pieces both the computational *tasks* associated with a problem and the *data* on which the tasks operates
  - *Domain decomposition* focuses on data
  - *Functional decomposition* focuses on tasks
  - Mixing domain/functional decomposition is possible

Partitioning

Domain decomposition



Functional decomposition



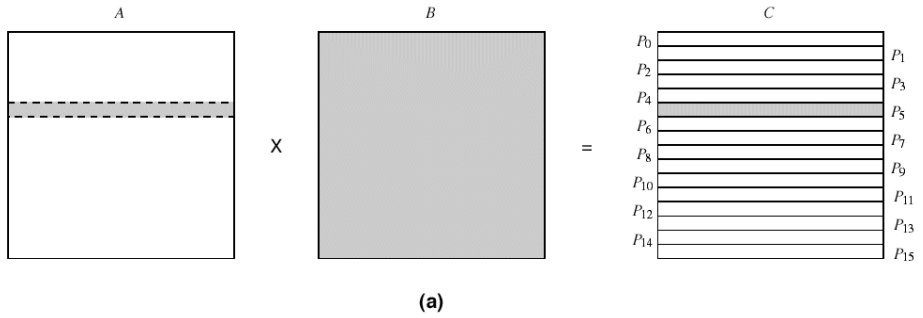
Partitioning

- A x B = C
- A[i,:] · B[:,j] = C[i,j]

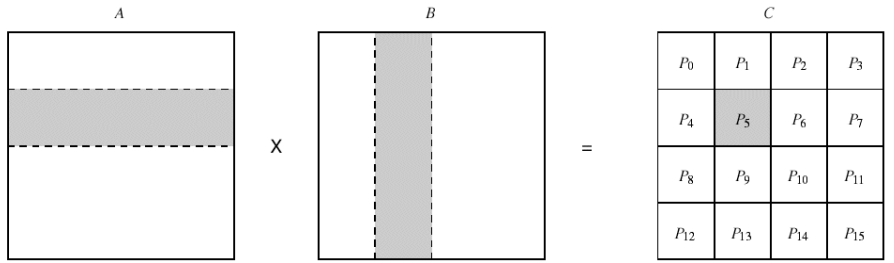
- Row partitioning
  - *N* tasks

- Block partitioning
  - *N*N/B* tasks

- Shading shows data sharing in B matrix



(a)

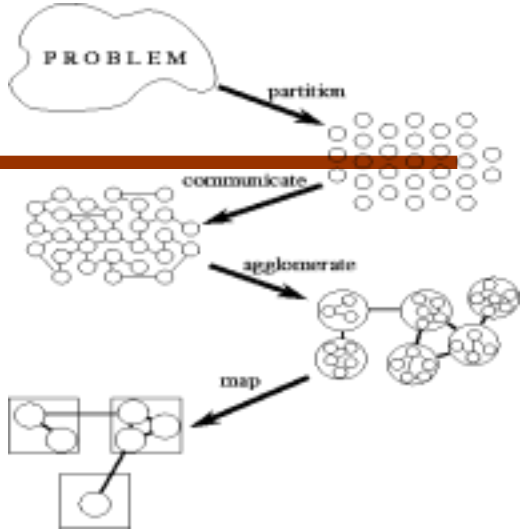


(b)

Partitioning Checklist

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, may loose design flexibility.
- Does your partition avoid redundant computation and storage requirements? If not, may not be scalable.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- Does the number of tasks scale with problem size? If not may not be able to solve larger problems with more processors

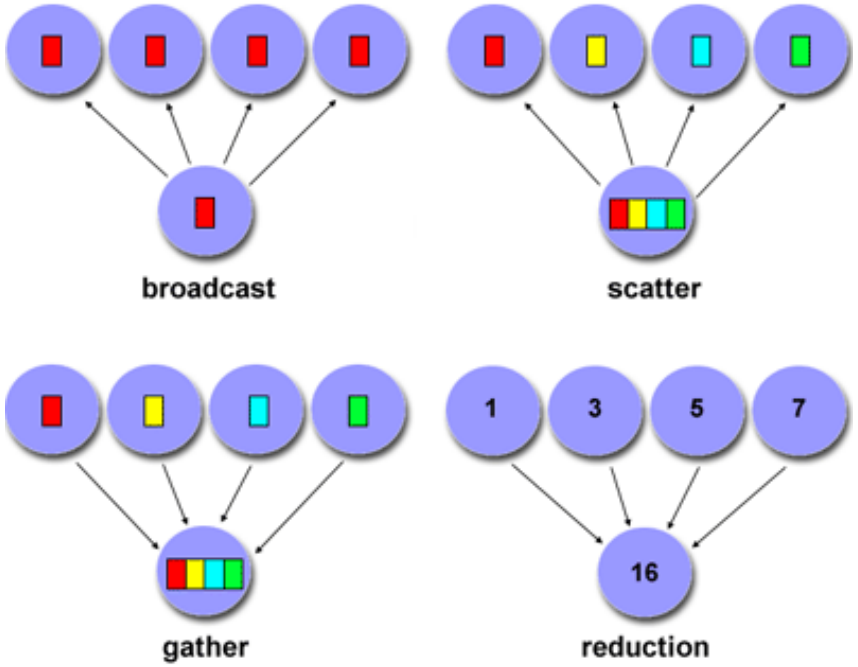
Communication



- Tasks generated by a partition must interact to allow the computation to proceed
  - Information flow: data and control
- Types of communication
  - *Local* vs. *Global*: locality of communication
  - *Structured* vs. *Unstructured*: communication patterns
  - *Static* vs. *Dynamic*: determined by runtime conditions
  - *Synchronous* vs. *Asynchronous*: coordination degree
- Granularity and frequency of communication
  - Size of data exchange
- Think of communication as interaction and control
  - Applicable to both shared and distributed memory parallelism

Types of Communication

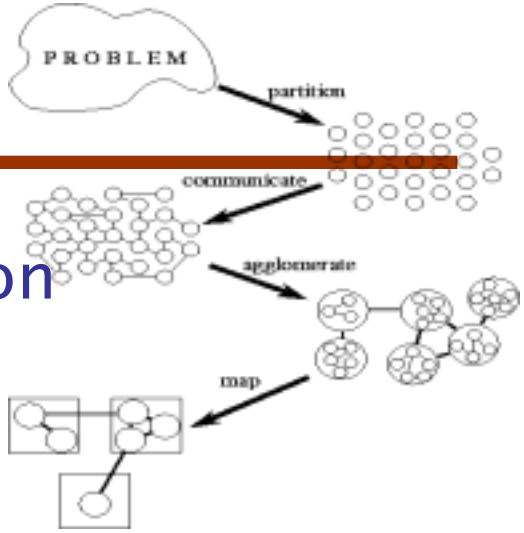
- Point-to-point
- Group-based
- Hierarchical
- Collective



Communication Design Checklist

- Is the distribution of communications equal?
  - Unbalanced communication may limit scalability
- What is the communication locality?
  - Wider communication is more expensive
- What is the degree of communication concurrency?
  - Communication operations may be parallelized
- Is computation associated with different tasks able to proceed concurrently? Can communication be overlapped with computation?
  - Try to reorder computation and communication to expose opportunities for parallelism

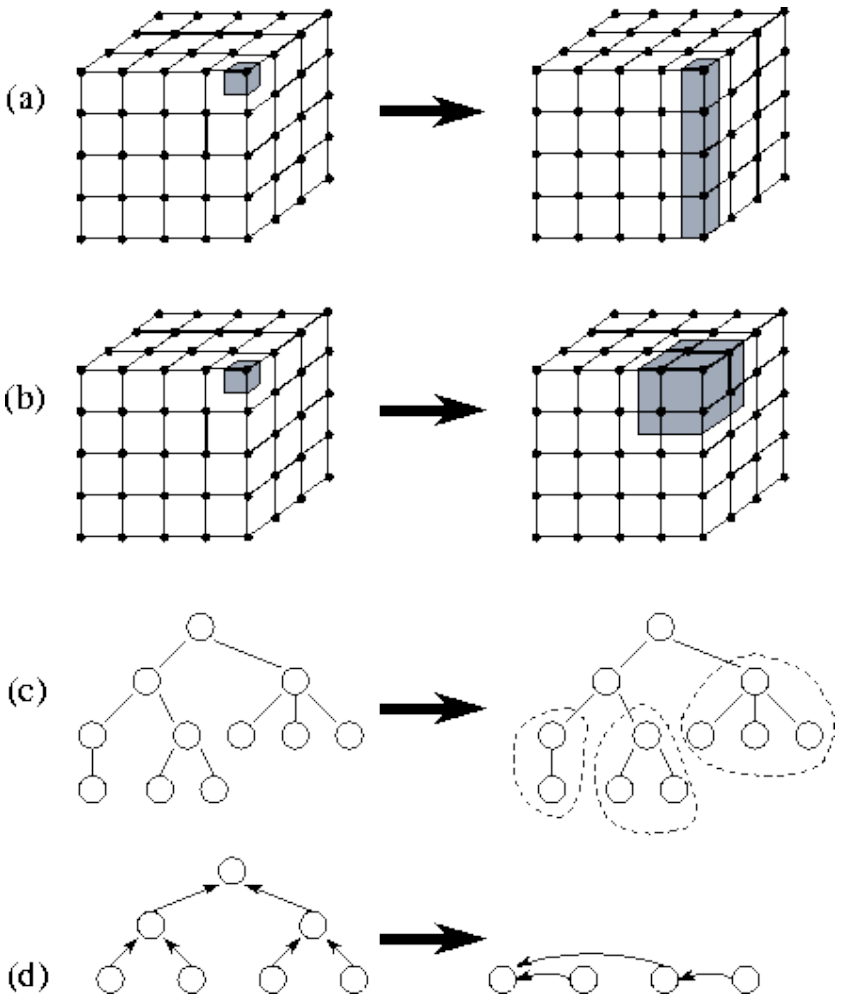
Agglomeration



- Revisit partitioning and communication
  - View to efficient algorithm execution
- Is it useful to *agglomerate*?
  - What happens when tasks are combined?
- Is it useful to *replicate* data and/or computation?
- Changes important algorithm and performance ratios
  - *Surface-to-volume*: reduction in communication at the expense of decreasing parallelism
  - *Communication/computation*: which cost dominates
- Replication may allow reduction in communication
- Maintain flexibility to allow overlap

Types of Agglomeration

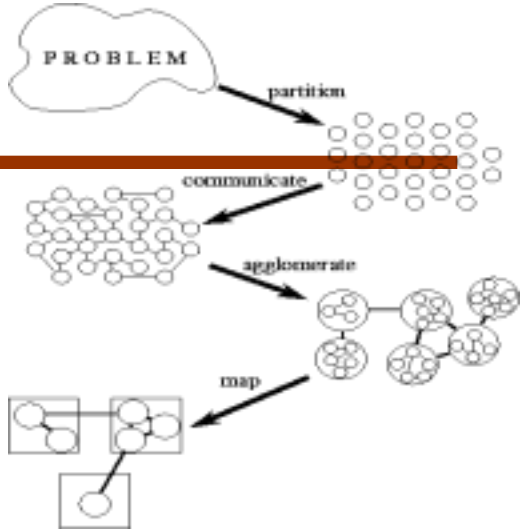
- Element to column
- Element to block
  - Better surface to volume
- Task merging
- Task reduction
  - Reduces communication



Agglomeration Design Checklist

- Has increased locality reduced communication costs?
- Is replicated computation worth it?
- Does data replication compromise scalability?
- Is the computation still balanced?
- Is scalability in problem size still possible?
- Is there still sufficient concurrency?
- Is there room for more agglomeration?
- Fine-grained vs. coarse-grained?

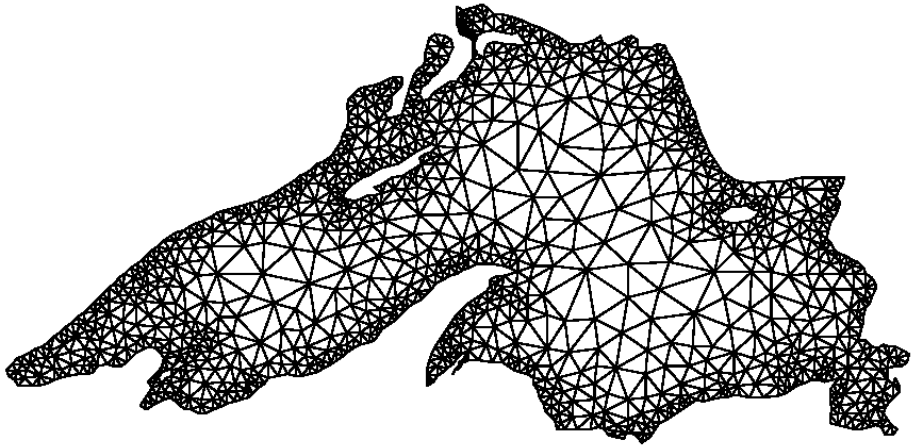
Mapping



- Specify where each task is to execute
  - Less of a concern on shared-memory systems
- Attempt to minimize execution time
  - Place concurrent tasks on different processors to enhance physical concurrency
  - Place communicating tasks on same processor, or on processors close to each other, to increase locality
  - Strategies can conflict!
- Mapping problem is *NP-complete*
  - Use problem classifications and heuristics

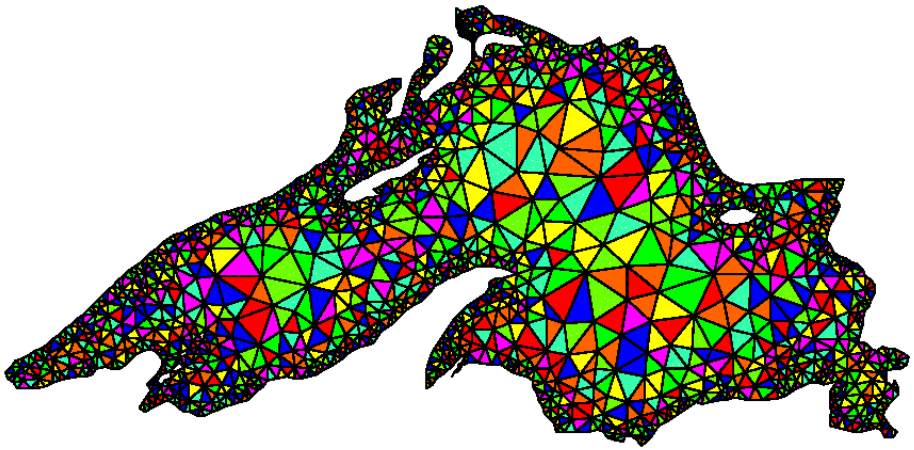
Mapping

- Mesh model of Lake Superior
- How to assign mesh elements to processors

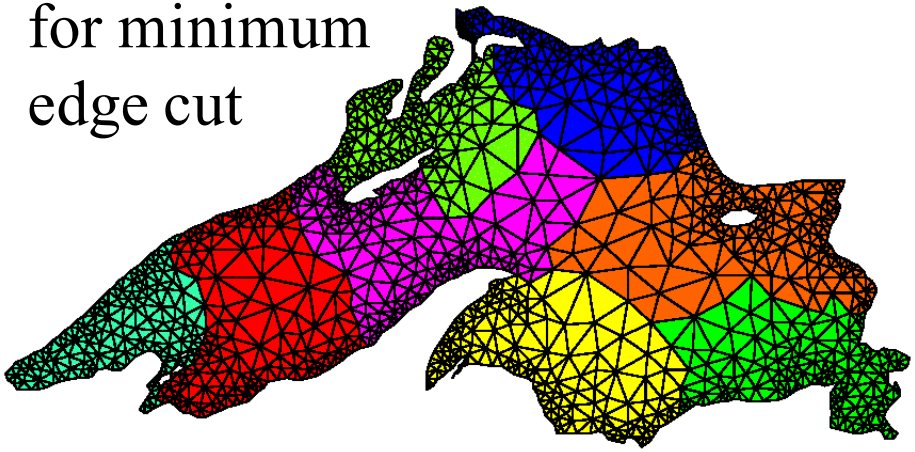


- Distribute onto 8 processors

randomly



graph partitioning for minimum edge cut



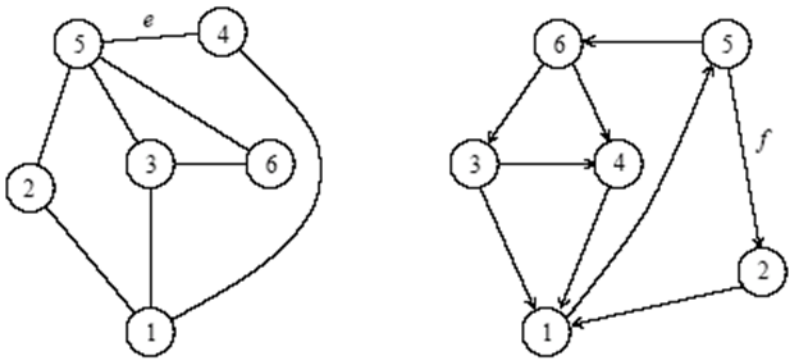
Mapping Design Checklist

- Is static mapping too restrictive and non-responsive?
- Is dynamic mapping too costly in overhead?
- Does centralized scheduling lead to bottlenecks?
- Do dynamic load-balancing schemes require too much coordination to re-balance the load?
- What is the tradeoff of dynamic scheduling complexity versus performance improvement?
- Are there enough tasks to achieve high levels of concurrency? If not, processors may idle.

Graph Definitions

- 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*.

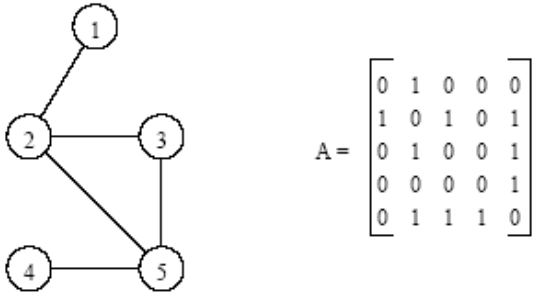


(a) (b)

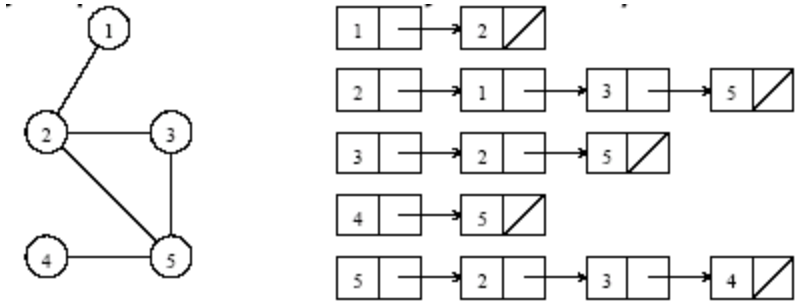
Graph Definitions

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

Definitions and Representation



An undirected graph and its adjacency matrix representation



An undirected graph and its adjacency list representation.

For a grapn with n nodes, adjacency matrices take $\Theta(n^2)$ space and adjacency list takes $\Theta(|E|)$ space.

Definitions and Representation

- G=(V,E) is sparse if $|E| <<< |V|^2$
  - Otherwise dense

- Adjacent matrix works for dense graphs
- Adjacent list works better for sparse graphs

Minimum Spanning Tree

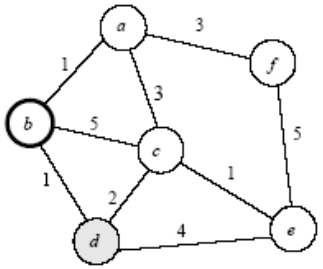
- A *spanning tree* of an undirected graph *G* is a subgraph of *G* that is a tree containing all the vertices of *G*.

- 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: Prim's Algorithm

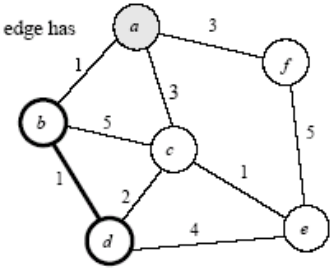
(a) Original graph



	a	b	c	d	e	f
d[]	1	0	5	1	∞	∞

	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

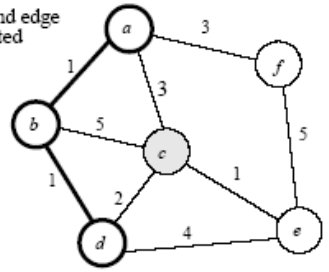
(b) After the first edge has been selected



	a	b	c	d	e	f
d[]	1	0	2	1	4	∞

	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

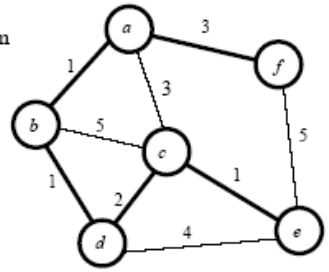
(c) After the second edge has been selected



	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

Prim's minimum spanning tree algorithm.

Minimum Spanning Tree: Prim's Algorithm

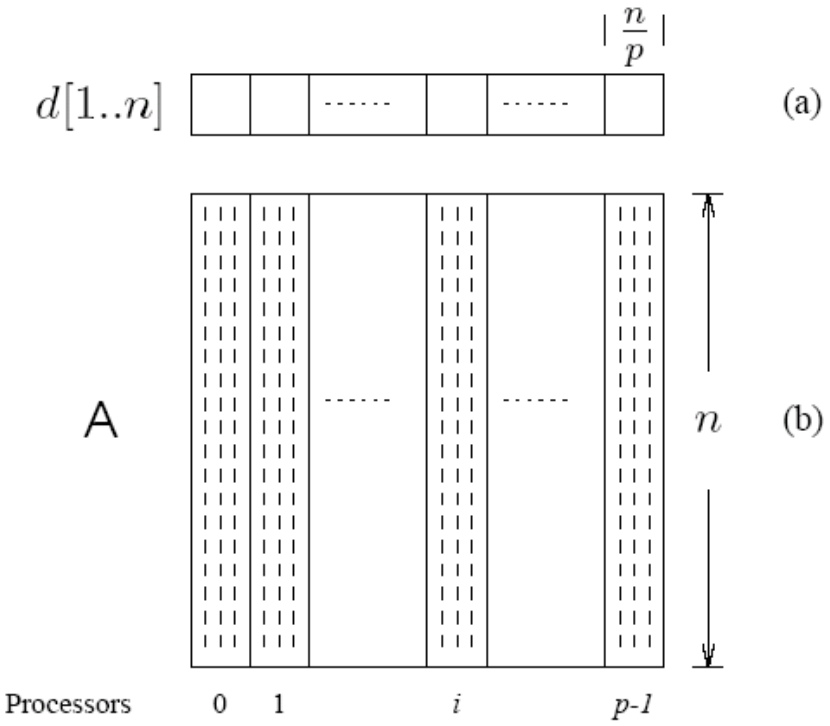
```
1.    procedure PRIM_MST(V, E, w, r)
2.    begin
3.        V_T := {r};
4.        d[r] := 0;
5.        for all v ∈ (V − V_T) do
6.            if edge (r, v) exists set d[v] := w(r, v);
7.            else set d[v] := ∞;
8.        while V_T ≠ V do
9.        begin
10.           find a vertex u such that 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.

Prim's Algorithm: Parallel Formulation

- The algorithm works in *n* outer (while) iterations
  - it is hard to execute these iterations concurrently.
- The inner loop is relatively easy to parallelize.
  - The adjacency matrix is partitioned in a 1-D block fashion, with distance vector *d* partitioned accordingly.
  - In each step, a processor selects the locally closest node, followed by a global reduction to select globally closest node.
  - This node is inserted into MST, and the choice broadcast to all processors.
  - Each processor updates its part of the *d* vector locally.

Prim's Algorithm: Parallel Formulation



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

Prim's Algorithm: Parallel Formulation

$$T_p = \Theta(n^2 / p) + \Theta(n \log p)$$

Computation **Communication**

Single-Source Shortest Paths

- For a weighted graph $G = (V,E,w)$, the *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V.

- Dijkstra's algorithm is similar to Prim's algorithm. It maintains a set of nodes for which the shortest paths are known.
  - Open Shortest Path First (OSPF)
  - A* Algorithm for Path finding

Single-Source Shortest Paths: Dijkstra's Algorithm

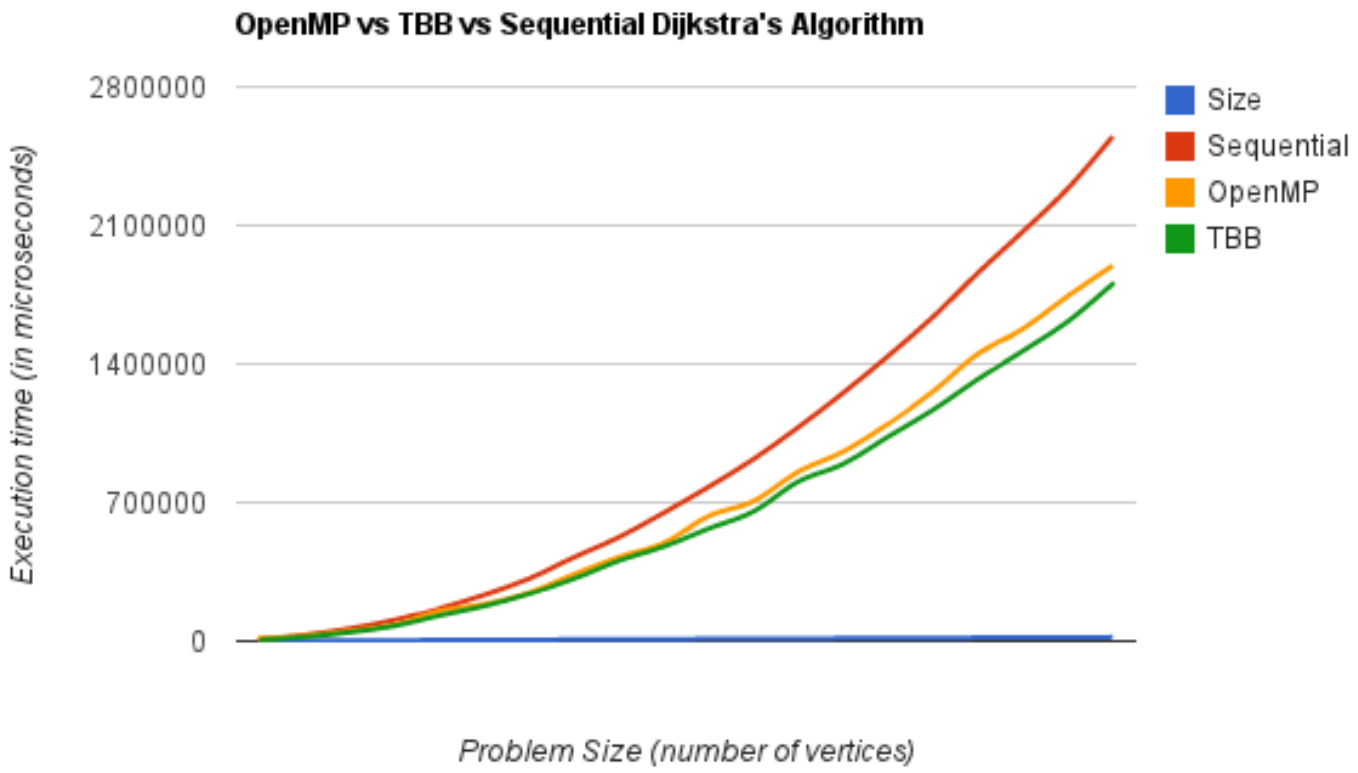
```
1.   procedure DIJKSTRA_SINGLE_SOURCE_SP(V, E, w, s)
2.   begin
3.       V_T := {s};
4.       for all v ∈ (V − V_T) do
5.           if (s, v) exists set l[v] := w(s, v);
6.           else set l[v] := ∞;
7.       while V_T ≠ V do
8.       begin
9.           find a vertex u such that l[u] := min{l[v]|v ∈ (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.

Dijkstra's Algorithm: Parallel Formulation

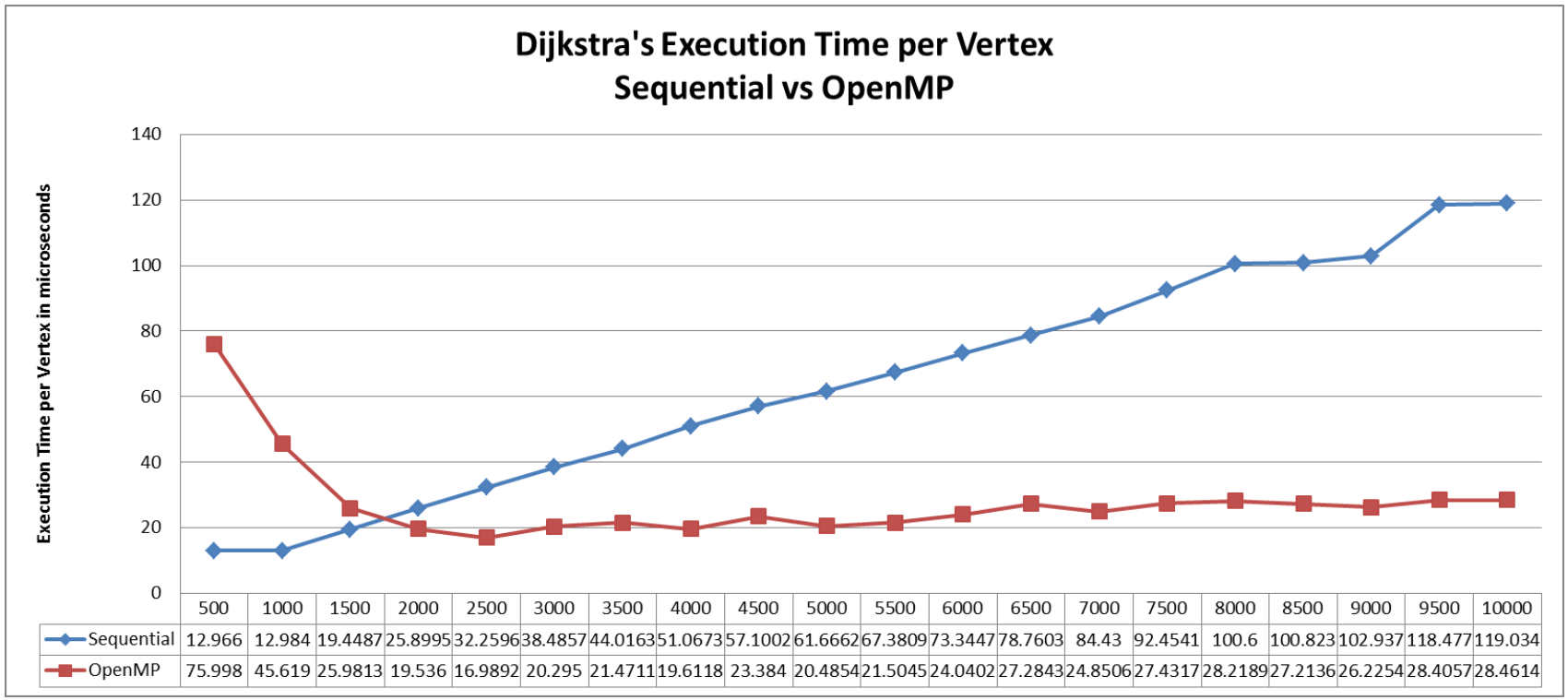
- Very similar to the parallel formulation of Prim's algorithm for minimum spanning trees.
  - The weighted adjacency matrix is partitioned using the 1-D block mapping.
  - Each process selects, locally, the node closest to the source, followed by a global reduction to select next node.
  - The node 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.

Dijkstra's Algorithm: Parallel Formulation



Source: Samuel Rodriguez

Dijkstra's Algorithm: Parallel Formulation



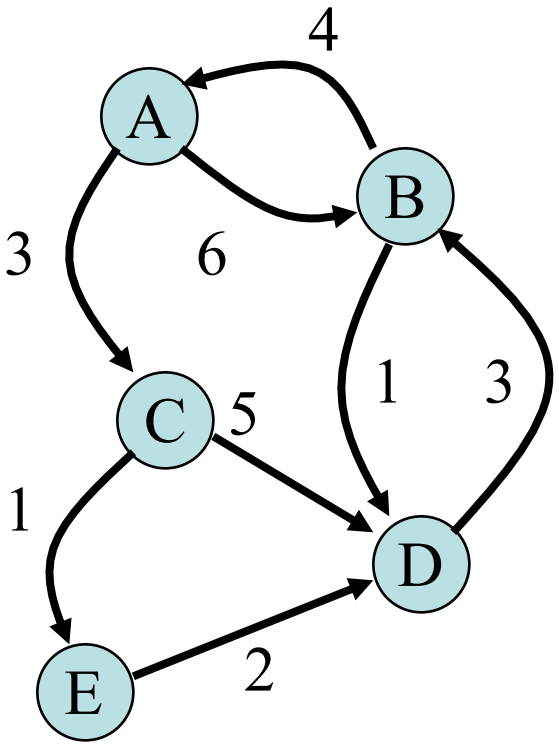
	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500	8000	8500	9000	9500	10000
Sequential	12.966	12.984	19.4487	25.8995	32.2596	38.4857	44.0163	51.0673	57.1002	61.6662	67.3809	73.3447	78.7603	84.43	92.4541	100.6	100.823	102.937	118.477	119.034
OpenMP	75.998	45.619	25.9813	19.536	16.9892	20.295	21.4711	19.6118	23.384	20.4854	21.5045	24.0402	27.2843	24.8506	27.4317	28.2189	27.2136	26.2254	28.4057	28.4614

Source: Omar Soto

All-Pairs Shortest Paths Problem

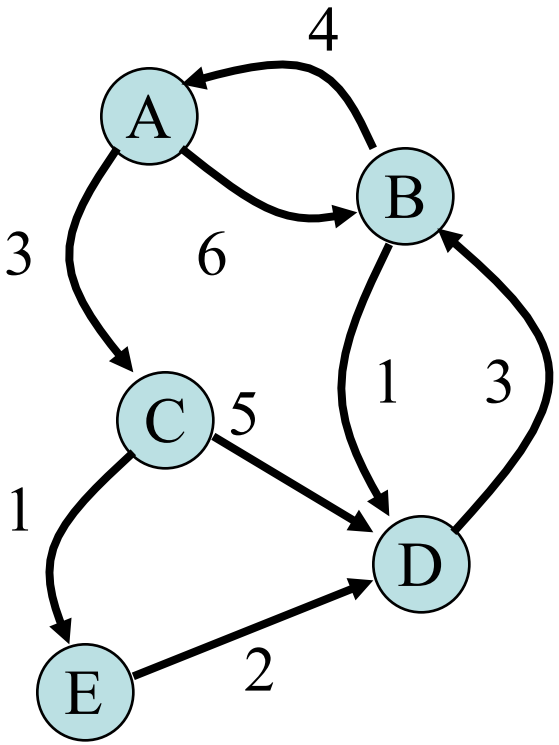
- Given a weighted graph *G(V,E,w)*, the *all-pairs shortest paths* problem is to find the shortest paths between all pairs of vertices $v_i, v_j \in V$.

All-pairs Shortest Path Problem



	A	B	C	D	E
A	0	6	3	∞	∞
B	4	0	∞	1	∞
C	∞	∞	0	5	1
D	∞	3	∞	0	∞
E	∞	∞	∞	2	0

All-pairs Shortest Path Problem



	A	B	C	D	E
A	0	6	3	6	4
B	4	0	7	1	8
C	12	6	0	3	1
D	7	3	10	0	11
E	9	5	12	2	0

Dijkstra's Algorithm: Parallel Formulation

- Two parallelization strategies
  - Source partitioned
    - A sequential Dijsktra's shortest path algorithm is used to solve the problem on each vertex
  - Source Parallel
    - A parallel Dijkstra's shortest path algorithm is used to solve the problem on each vertex

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 inter-process communication (provided that the adjacency matrix is replicated at all processes).
  - The parallel run time of this formulation is: $\Theta(n^2)$.

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.
- This takes time

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

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
- There are two options
  - vertex v_k is not in the shortest path
  - Vertes v_k is in the shortest path, then we can break *the path* into two paths – one from v_i to v_k and one from v_k to v_j .

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 $O(n^3)$.

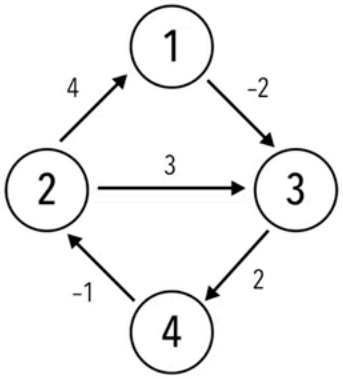
Floyd's Algorithm

```
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*.

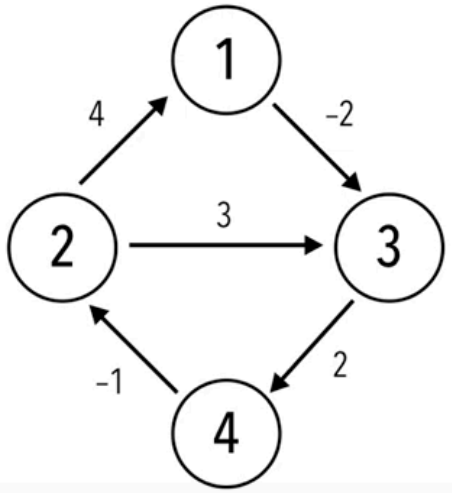
Floyd's Algorithm

```
let V = number of vertices in graph
let dist = V × V array of minimum distances
for each vertex v
  dist [v][v] ← 0
for each edge (u,v)
  dist [u][v] ← weight(u,v)
for k from 1 to V
  for i from 1 to V
    for j from 1 to V
      if dist [i][j] > dist [i][k] + dist [k][j]
        dist [i][j] ← dist [i][k] + dist [k][j]
      end if
```



	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

Floyd's Algorithm

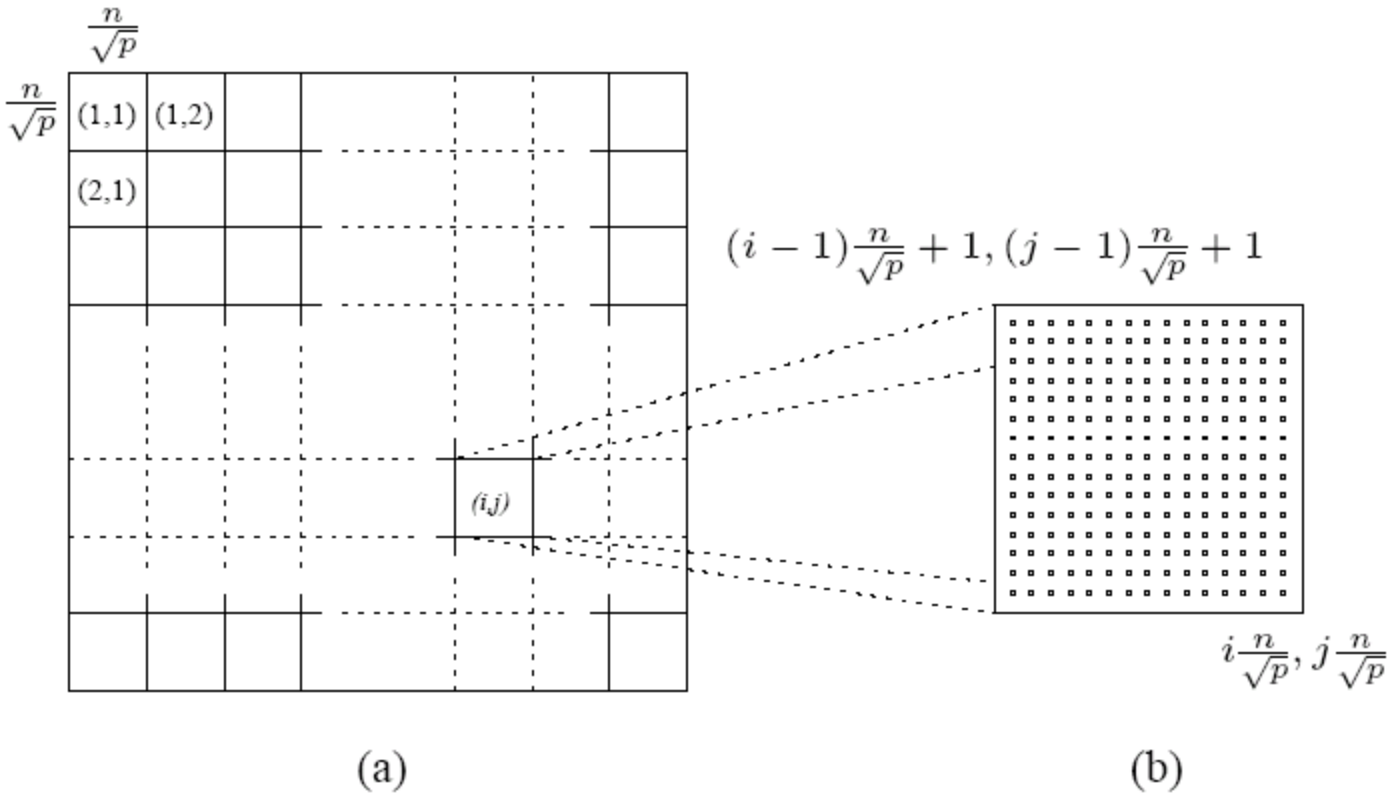


	1	2	3	4
1	0	-1	-2	0
2	4	0	2	4
3	5	1	0	2
4	3	-1	1	0

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

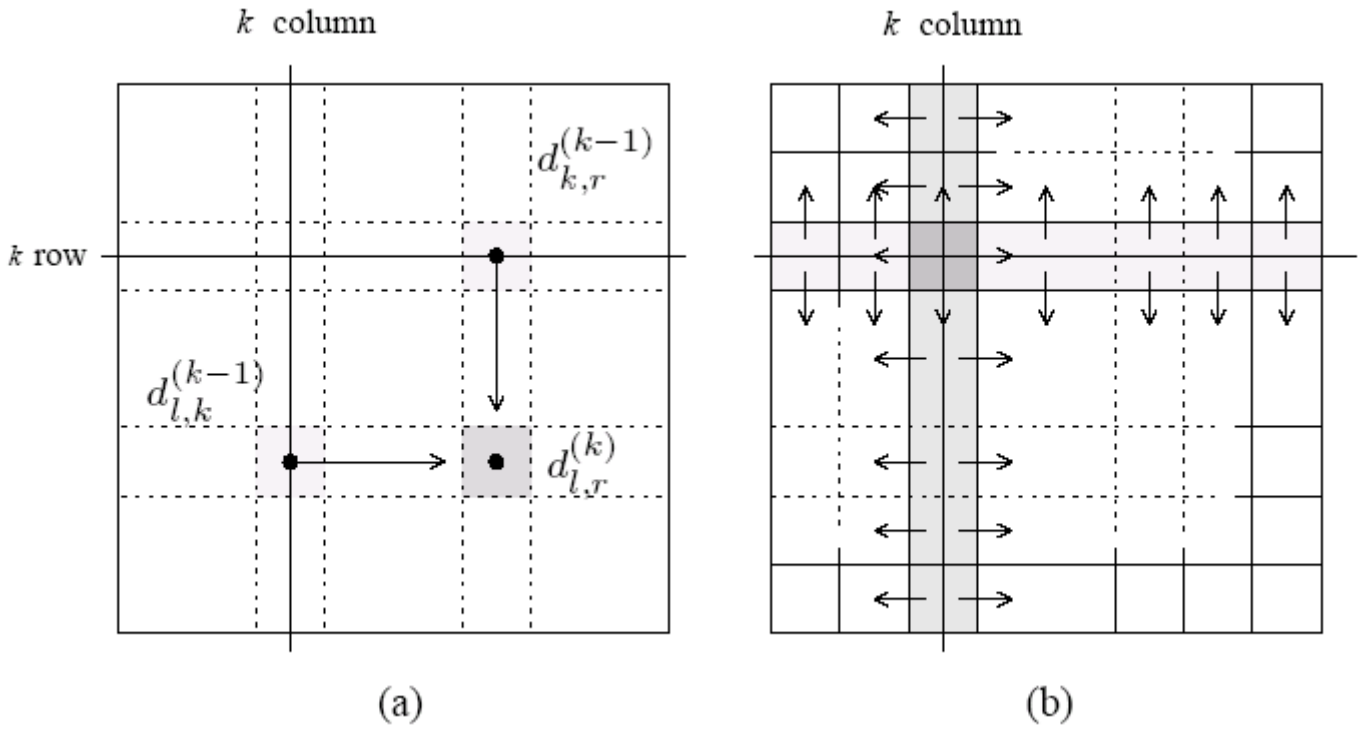
- Matrix $D^{(k)}$ is divided into p blocks of size $(n / \sqrt{p}) \times (n / \sqrt{p})$.
- Each processor updates its part of the matrix during each iteration.
- To compute $d_{l,k}^{(k-1)}$ processor $P_{i,j}$ must get $d_{l,k}^{(k-1)}$ and $d_{k,r}^{(k-1)}$.
- In general, during the k^{th} iteration, 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



(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



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 Algorithm: Parallel Formulation Using 2-D Block Mapping

- During each iteration of the algorithm, the k^{th} row and k^{th} column of processors perform a one-to-all broadcast along their rows/columns.
- The size of this broadcast is $n/\sqrt{p}$ elements, taking time $\Theta((n \log p)/\sqrt{p})$.
- The synchronization step takes time $\Theta(\log p)$.
- The computation time is $\Theta(n^2/p)$.
- The parallel run time of the 2-D block mapping formulation of Floyd's algorithm is

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

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

- The above formulation can use
  - $O(n^2 / \log^2 n)$ processors cost-optimally.
- The isoefficiency of this formulation is
  - $\Theta(p^{1.5} \log^3 p)$.

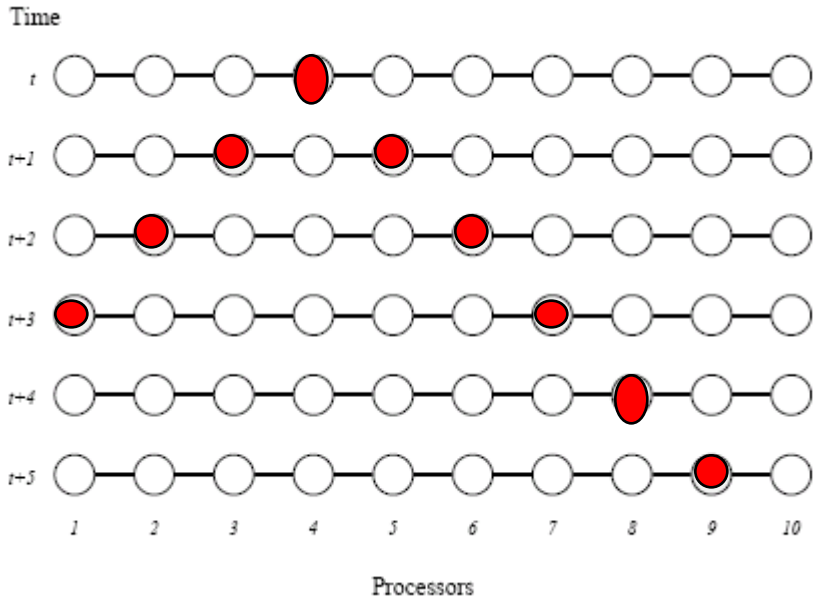
- This algorithm can be further improved by relaxing the strict synchronization after each iteration.

Floyd's Algorithm: Speeding Things Up by Pipelining

- The synchronization step in parallel Floyd's algorithm can be removed without affecting the correctness of the algorithm

- A process starts working on the 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.

Floyd's Algorithm: Speeding Things Up by Pipelining



- 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 $t + 1$
- Similarly, processes farther away from process 4 receive the segment later
- Process 1 (at the boundary) does not forward the segment after receiving it.

Floyd's Algorithm: Speeding Things Up by Pipelining

- The overall parallel run time of this formulation is

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

- The pipelined formulation of Floyd's algorithm uses up to $O(n^2)$ processes efficiently.
- The corresponding isoefficiency is $\Theta(p^{1.5})$.

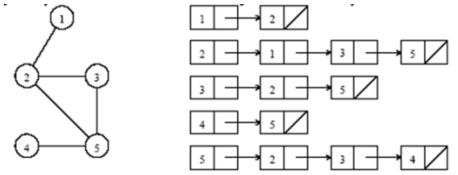
All-pairs Shortest Path: Comparison

	Maximum Number of Processes for $E = \Theta(1)$	Corresponding Parallel Run Time	Isoefficiency Function
Dijkstra source-partitioned	$\Theta(n)$	$\Theta(n^2)$	$\Theta(p^3)$
Dijkstra source-parallel	$\Theta(n^2/\log n)$	$\Theta(n \log n)$	$\Theta((p \log p)^{1.5})$
Floyd 1-D block	$\Theta(n/\log n)$	$\Theta(n^2 \log n)$	$\Theta((p \log p)^3)$
Floyd 2-D block	$\Theta(n^2/\log^2 n)$	$\Theta(n \log^2 n)$	$\Theta(p^{1.5} \log^3 p)$
Floyd pipelined 2-D block	$\Theta(n^2)$	$\Theta(n)$	$\Theta(p^{1.5})$

Algorithms for Sparse Graphs

- Graph 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
  - Parallel algorithms for sparse graphs typically make use of graph structure or degree information for better performance.



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 $v \in (V - V_T)$.

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
```

More on Graph Algorithms

- K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak, "An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances"
  - Parallel priority queues: relaxed heaps
  - Ullman-Yannakakis randomized approach
  - Δ - stepping algorithm
    - U. Meyer and P. Sanders. .-stepping: a parallelizable shortest path algorithm. J. Algs., 49(1):114-152, 2003.

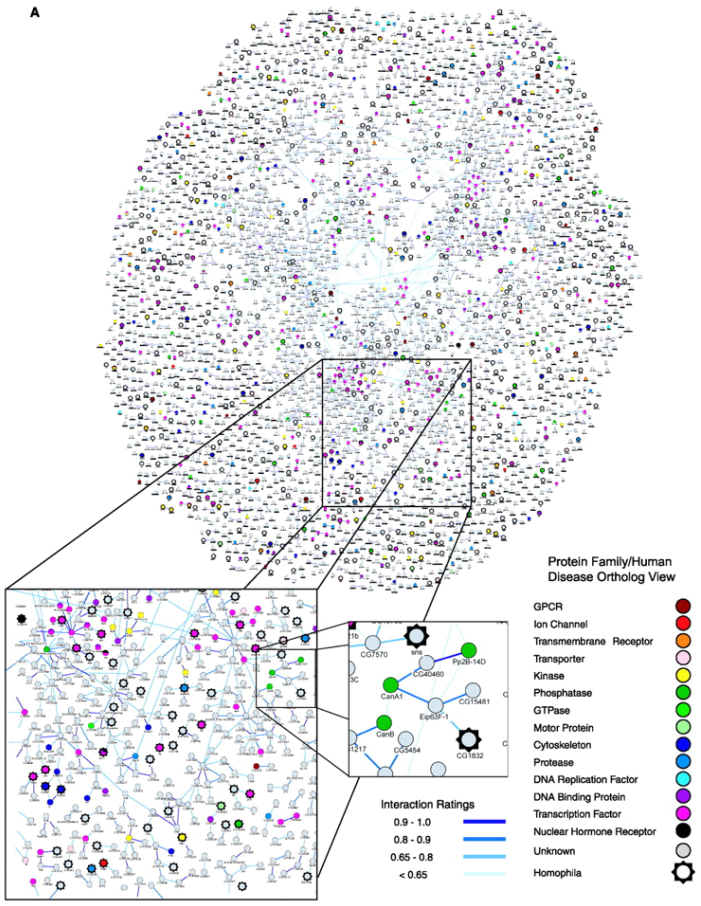
Parallel Graph Frameworks

- SNAP
- Multithreaded Graph Library
- Parallel Boost Graph Library
- Giraph
- Knowledge Discovery Toolkit
- STINGER

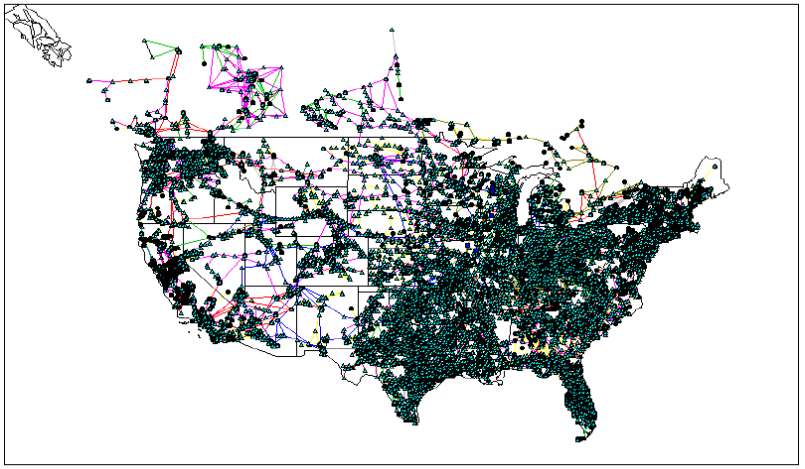
Parallel Graph Frameworks

Software	Interface	Parallelization
Pregel	C++	On disk
MTLG	C++	Shared
PBGL	C++	In memory
KDT	Python	In memory
Pegasus	Hadoop	On disk
SNAP	C	Shared (openMP)
GraphCT	C	Shared

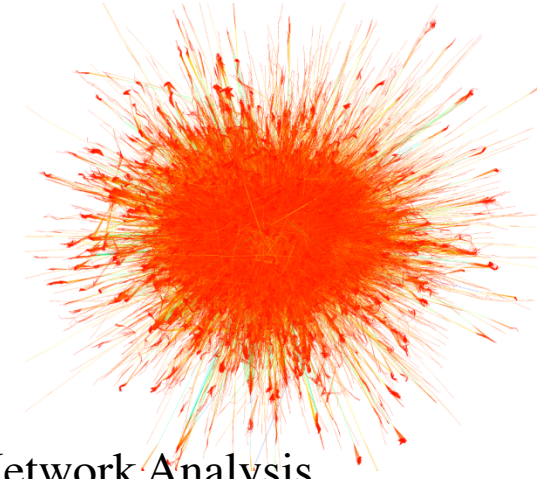
Large Scale Graphs



Protein interaction map

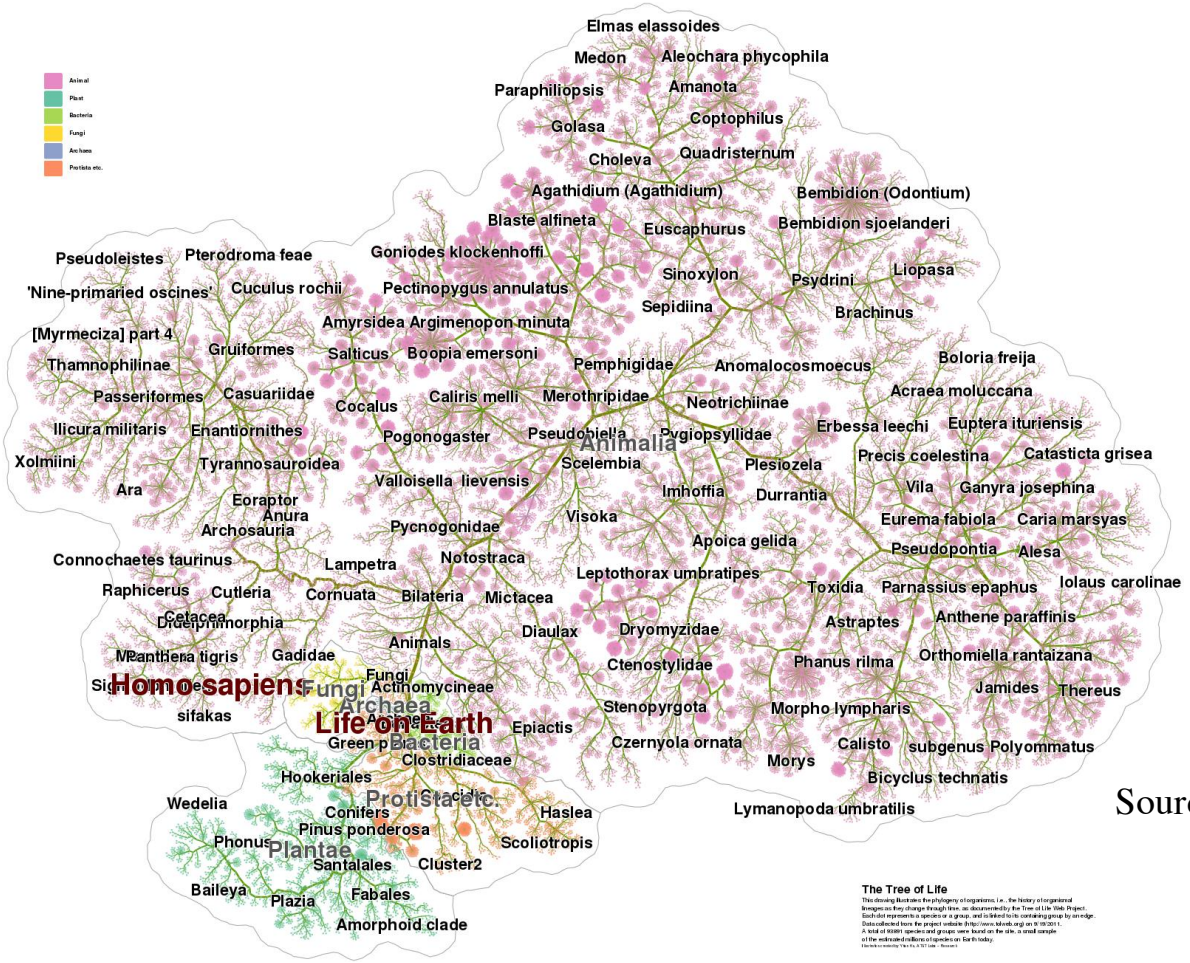


US High Voltage Transmission Grid



Social Network Analysis

Visualizing the Tree of Life

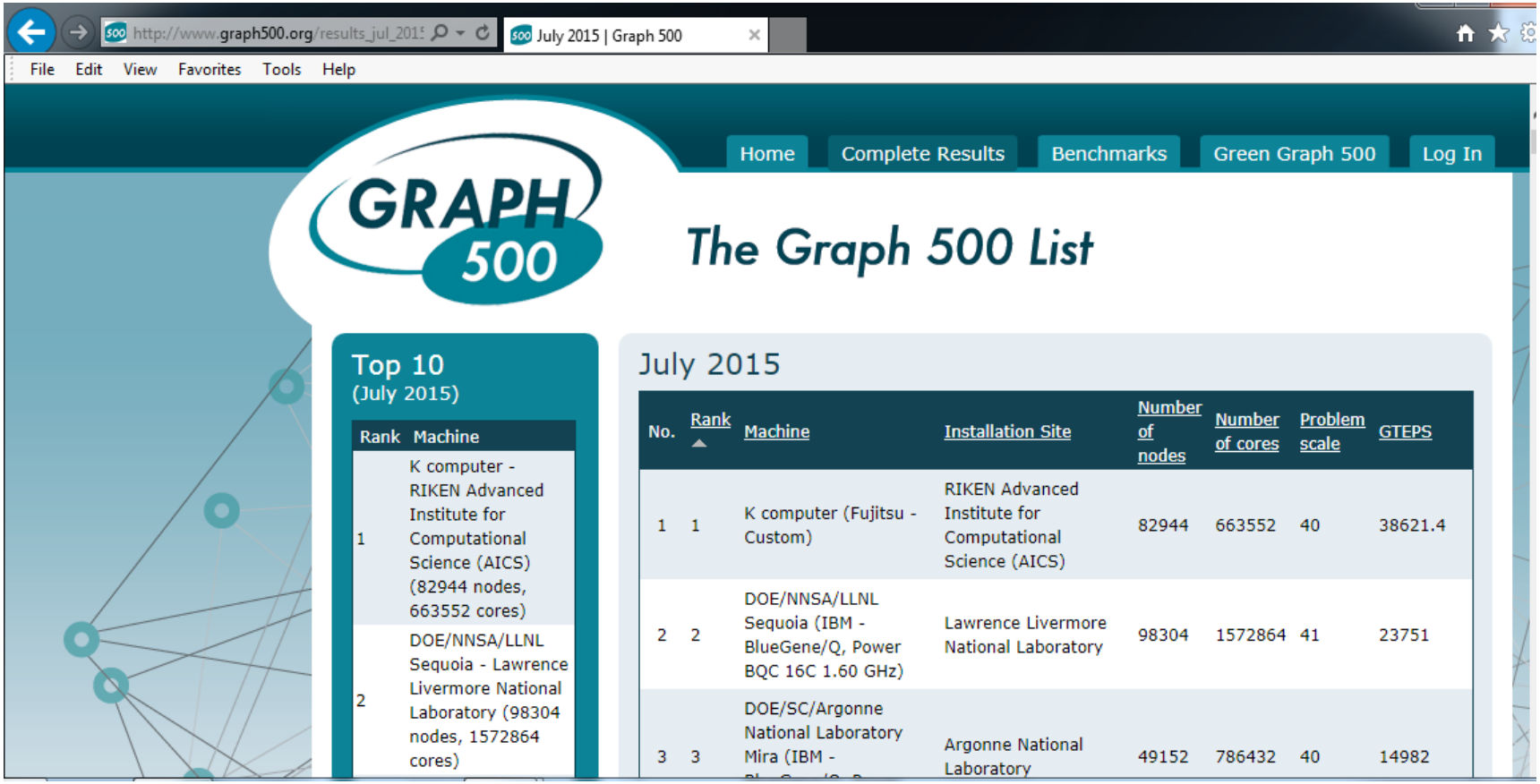


Source: Dr. Jifan Hu

Facebook Friendship

Source: Paul Butler

Graph 500 List



Summary

- Minimum Spanning Tree
  - Prim's Algorithm
- Single-Source Shortest Path
  - Dijkstra's Algorithm
- All-Pairs Shortest Path
  - Dijkstra's Algorithm formulations
  - Floyd's Algorithm
- Algorithms for Sparse Graphs
  - Johnson's Algorithm
- Large scale graphs