Outline

- Timeline
- Parallel graph algorithms
  - Applications
  - Designing parallel graph algorithms: DFS, BFS
  - Case studies:

    A)Shortest Paths: Classic, Delta-stepping

    B)Maximal Independent Sets: Luby's algorithm

    C)Graph traversals: Breadth-first search

    D)Strongly Connected Components

Graph reminder

- Define: Graph G = (V,E): a set of vertices V and a set of edges E between vertices
- n=|V| (number of vertices)
- m=|E| (number of edges)
- D=diameter (max #hops between any pair of vertices)
- Edges can be directed or undirected, weighted or not.
- They can even have attributes (i.e. semantic graphs)
- Sequences of edges $<u_1,u_2>$, $<u_2,u_3>$, ... ,$<u_{n-1},u_n>$ is a *path* from u_1 to u_n. Its *length* is the sum of its weights.

Many types of graphs

- Lines and trees
- Completely regular grids
- Planar graphs (no edges need cross)
- Low-dimensional Euclidean
- Power law graphs
- ...

Algorithms are not one-size-fits-all!

Applications (1)

- Routing in transportation networks (route planning with costs)
- Internet and the WWW
  - The world-wide web can be represented as a directed graph
  - Web search and crawl: traversal
  - Link analysis, ranking: Page rank
  - Document classification and clustering
  - Internet topologies (router networks) are naturally modeled as graphs

Applications (2): Large Graphs in Scientific Computing

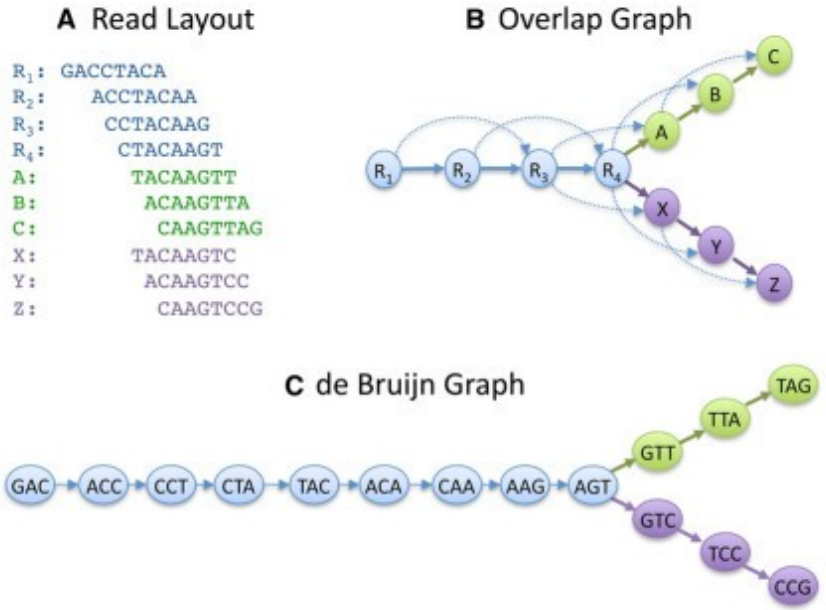
- Graph partitioning: Dynamic load balancing in parallel simulations
- Problem size: as big as the sparse linear system to be solved or the simulation to be performed

Applications (3): Large-scale data analysis

- Graph abstractions are very useful to analyze complex data sets.
- Sources of data: simulations, experimental devices, the Internet, sensor networks
- Challenges: data size, heterogeneity, uncertainty, data quality
- Examples:
  - Astrophysics: massive datasets, temporal variations
  - Bioinformatics: data quality, heterogeneity
  - Social Informatics: new analytics challenges, data uncertainty

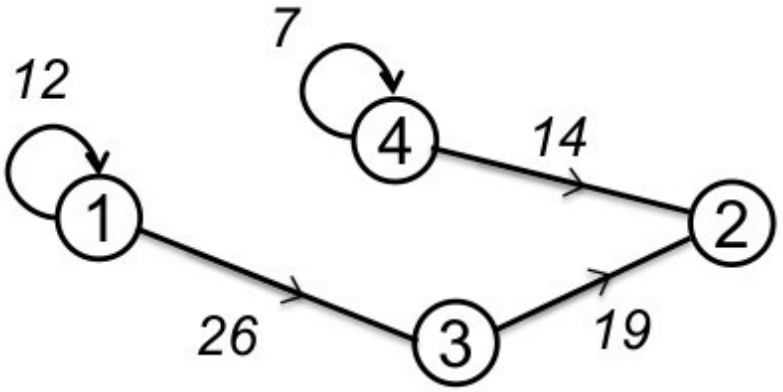
Applications (4): Large Graphs in Biology

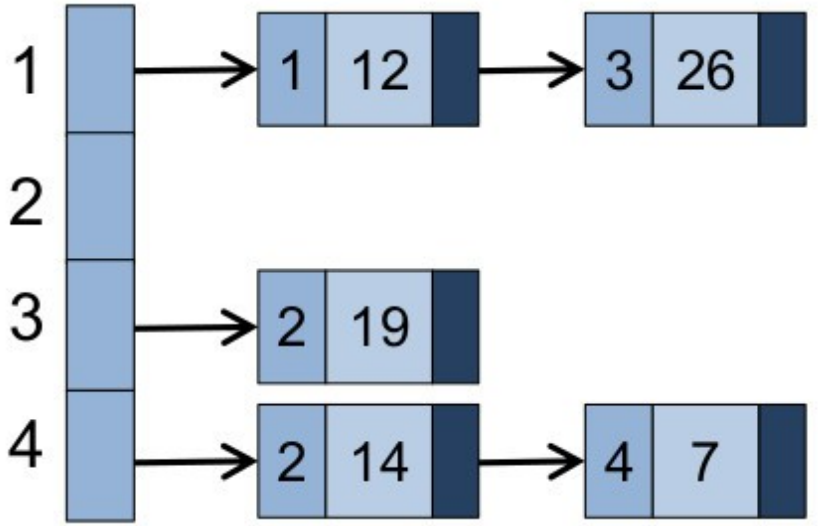
- Graph Theoretical analysis of Brain Connectivity
- Whole genome assembly
  - 26 billion (8B of which are non-erroneous) unique k-mers (vertices) in the hexaploit wheat genome W7984 for k=51



Graph representations

Compressed sparse rows (CSR) = cache-efficient adjacency lists





Index into adjacency array

1	3	3	4	6

(row pointers in CSR)

Adjacencies	1	3	2	2	4
Weights	12	26	19	14	7

(column ids in CSR)

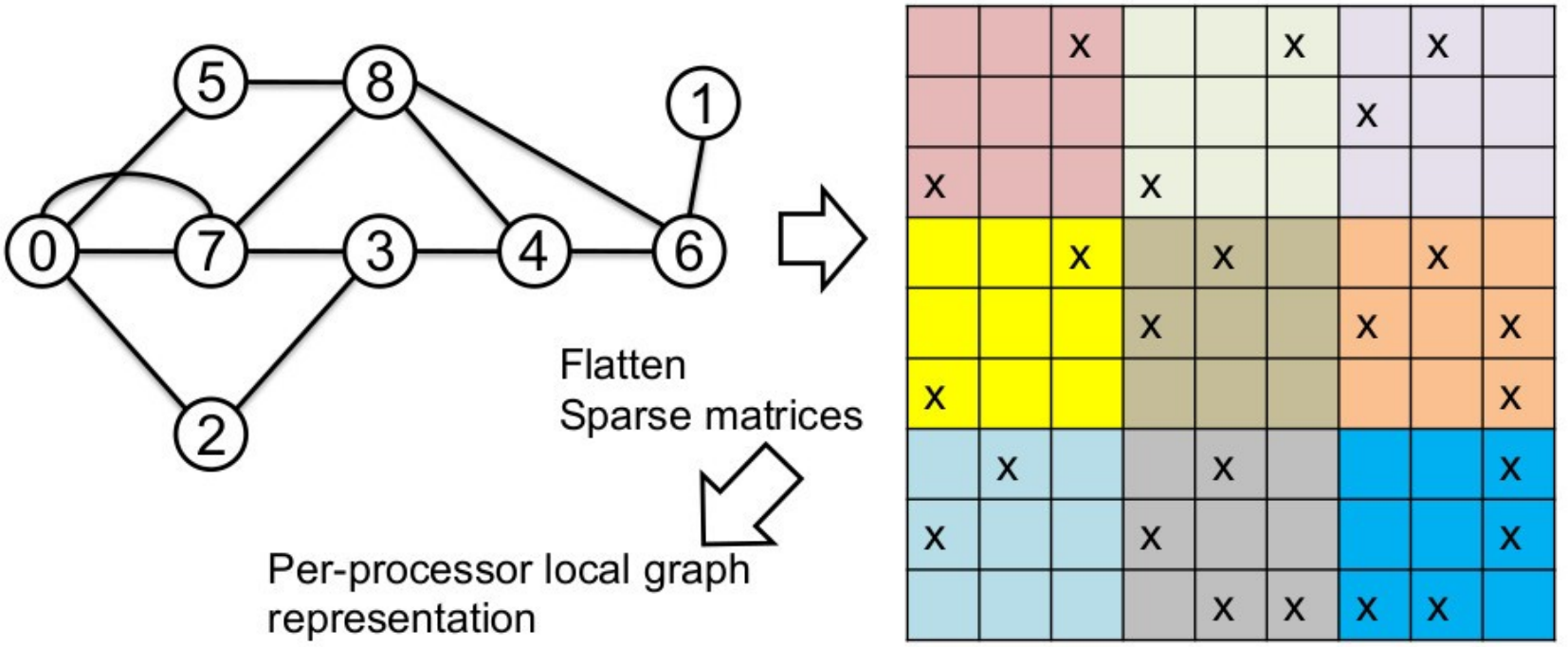
(numerical values in CSR)

Distributed graph representations

- Each processor stores the entire graph (“full replication”)
- Each processor stores n/p vertices and all adjacencies out of these vertices (“1D partitioning”)
  - How to create these “p” vertex partitions?
    - Graph partitioning algorithms: recursively optimize for conductance (edge cut/size of smaller partition)
    - Randomly shuffling the vertex identifiers ensures that edge count/processor are roughly the same

2D checkerboard distribution

- Consider a logical 2D processor grid (p_r * p_c = p) and the matrix representation of the graph
- Assign each processor a sub-matrix (i.e, the edges within the sub-matrix)



Memory Bandwidth Problems

- Many graph operations are
  - Computationally cheap (per node or edge)
  - Bad for locality
- Consider:
  - 323 million in US (fits in 32-bit int)
  - About 350 Facebook friends each
  - Compressed sparse row: about 450 GB
- What representation?

Memory Bandwidth Issues

- Adjacency Matrix
  - Pro: efficient for dense graphs
  - Con: wasteful for sparse case…
- Coordinates: Tuples: (i,j,w_{ij})
  - Pro: Easy to update
  - Con: Slow for multiply
- Adjacency list: Linked lists of adjacent nodes
  - Pro: Still easy to update
  - Con: May cost more to store than coordinates?

Distributed Graph Algorithm Design

- DFS and BFS components
- Distribute to processors
- Minimize communication

Graph traversal: Depth-first search (DFS)

```
procedure DFS(vertex v)
    v.visited = true
    previsit (v)
    for all v s.t. (v, w) ∈ E
       if (!w.visited) DFS(w)
    postvisit (v)
```

Graph traversal: Parallel Depth-first search (DFS)

- Each processor maintains a frontier of vertices
- As in sequential DFS, a frontier stores the subset of visited vertices, whose outgoing edges have not yet been explored.
- When a processor discovers a new vertex, it attempts to visit the vertex by using an *atomic read-modify-write* operation and, if it succeeds, adds the vertex to its frontier.
- Must perform load balancing to keep all the processors busy.
- A naive approach to this end would be to generate one thread for each vertex in the frontier.

Graph traversal: Serial Breadth-first search (BFS)

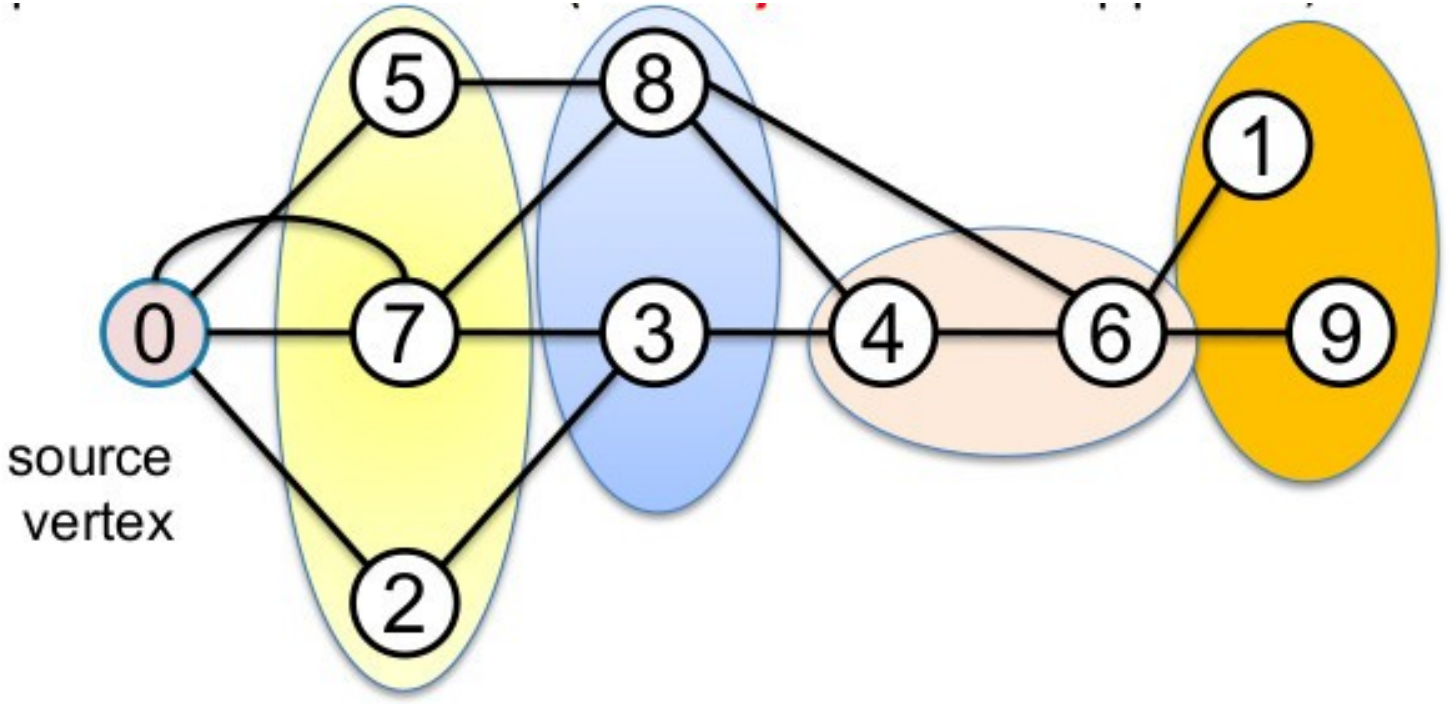
Push seed node onto queue and mark

While queue nonempty:

- Pop node from queue
- Visit node
- Push unmarked neighbors on queue
- Mark all neighbors

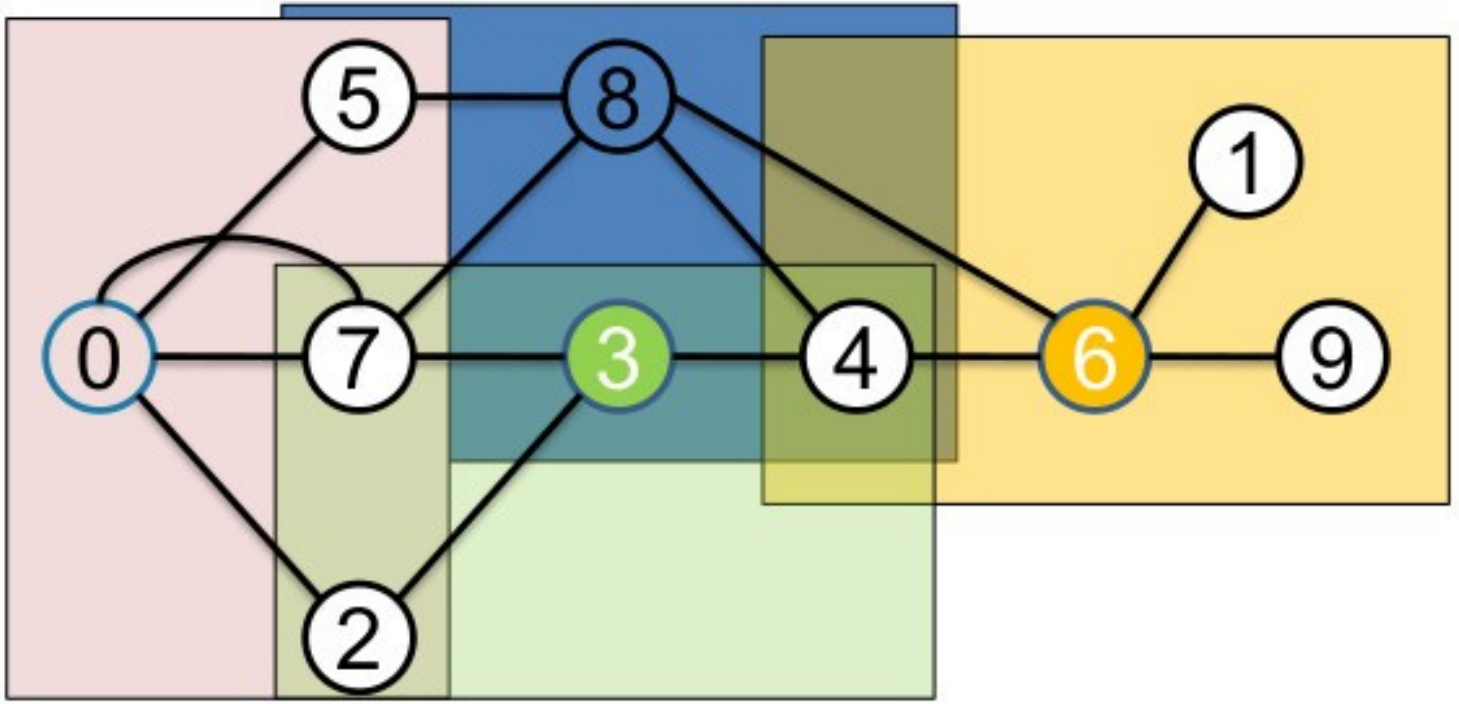
Graph traversal: Parallel Breadth-first search (BFS)

- Expand current frontier (level-synchronous approach, suited for low diameter graphs)
  - O(D) parallel steps
  - Adjacencies of all vertices in current frontier are visited in parallel



Graph traversal: Parallel Breadth-first search (BFS)

- Stitch together multiple concurrent traversals (Ullman-Yannakakis approach, suited for high-diameter graphs)
  - path-limited searches from “super vertices”
  - All-pairs shortest path between “super vertices”



Serial BFS: Bottom up

- Classical (top-down) algorithm is optimal in worst case, but pessimistic for low-diameter graphs
- Direction Optimization:
  - Switch from top-down to bottom-up search
  - When the majority of the vertices are discovered.
- Top down: For all v in frontier attempt to ``parent'' all neighbors(v)
- Bottom up: For all v in unvisited find any parent (neighbor(v) in frontier)

Serial BFS: Bottom up

Set $d[v]=\infty$ for all vertices

Set $d[s]=0$ for seed s

Until d stops changing

For each $u \in V$

$d[u]=\min(d[u],\min_{w \in N(u)} d[w]+1)$

Graph Algorithm Case Studies

- Shortest Paths: Delta-stepping, Floyd-Warshall
- Maximal Independent Sets: Luby’s algorithm
- Graph traversals: Breadth-first search
- Strongly Connected Components

Parallel Single-source Shortest Paths (SSSP) algorithms

- Famous serial algorithms:
  - Bellman-Ford : label correcting - works on any graph
  - Dijkstra : label setting – requires nonnegative edge weights
- No known PRAM algorithm that runs in sub-linear time and O(m+n log n) work
- Ullman-Yannakakis randomized approach
- Meyer and Sanders, Δ - stepping algorithm

Classic Dijkstra

- Dequeue *closest point* to frontier, expand frontier
- Update priority queue of distances (in parallel)
- Repeat

Classic Bellman-Ford

- Initialize d[u] with distance over-estimates to source

  d[s]=0

  Repeatedly relax $d[u]:=\min_{(v,u) \in E} d[v]+w(v,u)$
- Converges (eventually) as long as all nodes visited repeatedly, updates are atomic

Δ - stepping algorithm

- Label-correcting algorithm: Can relax edges from unsettled vertices also
- “approximate bucket implementation of Dijkstra”
- For random edge weighs [0,1], runs in O(n + m + D · L) where L = max distance from source to any node
- Vertices are ordered using buckets of width Δ
- Each bucket may be processed in parallel
- Basic operation: Relax (e(u,v)):
  - d(v) = min { d(v), d(u) + w(u, v) }
- Δ < min w(e) : Degenerates into Dijkstra
- Δ > max w(e) : Degenerates into Bellman-Ford

Maximal Independent Sets (MIS)

- Graph with vertices V = {1,2,...,n}
- A set S of vertices is *independent* if no two vertices in S are neighbors.
- An independent set S is *maximal* if it is impossible to add another vertex and stay independent
- An independent set S is *maximum* if no other independent set has more vertices
- Finding a maximum independent set is intractably difficult (NP-hard)
- Finding a maximal independent set is easy, at least on one processor.

Sequential (Greedy) Maximal Independent Set Algorithm

```
S = empty set;
for vertex v = 1 to n
    if (v has no neighbor in S)
        add v to S
```

Luby's Parallel Algorithm

```
S = empty set; C = V;
while C is not empty
    label each v in C with a random r(v);
    for all v in C in parallel
        if r(v) < min( r(neighbors of v) )
            move v from C to S;
            remove neighbors of v from C
```

- "Probably" finishes in O(log n) rounds

Strongly Connected Components

- Strongly connected components (SCCs): all maximal strongly connected sub-graphs in a large directed graph.
- Sequential algorithm: use depth-first search (Tarjan); work=O(m+n) for m=|E|, n=|V|, but DFS seems to be inherently sequential.
- Parallel algorithm: divide-and-conquer and BFS (Fleischer et al.); worst-case span O(n) but good in practice on many graphs.

Fleischer/Hendrickson/Pinar algorithm

- Partition the given graph into three disjoint subgraphs
- Each can be processed independently/recursively
- FW(v): vertices reachable from vertex v (ForWard).
- BW(v): vertices from which v is reachable (BackWard).

Fleischer/Hendrickson/Pinar algorithm

1) Select maximally connected node, called the *pivot*
2) Find all vertices that can be reached from the pivot (descendant (D))
3) Find all vertices that can reach the pivot (predecessor (P))
4) Intersection of those two sets is an SCC ($S = P \cap D$)
5) Now have three distinct sets leftover ($D \setminus S$), ($P \setminus S$), and remainder (R)

   $V \setminus X$ denotes the subset of vertices in V which are not in a subset X

Fleischer/Hendrickson/Pinar algorithm

procedure FW-BW(V)

if V = ∅ then return ∅

Select a pivot u ∈ V

D ← BFS(G(V, E(V)), u)

P ← BFS(G(V, E′(V)), u)

R ← (V \ (P ∪ D)

S ← (P ∩ D)

new task do FW-BW(D \ S)

new task do FW-BW(P \ S)

new task do FW-BW(R)