CPSC 4240/5240: Parallel Programming Techniques

Lecture 14: Graphs and ParlayLib

Lecturer: Quanquan C. Liu

Spring 2026

Some slides courtesy of Aydin Buluç, CS267, 2016

More Parallel Graph Algorithms

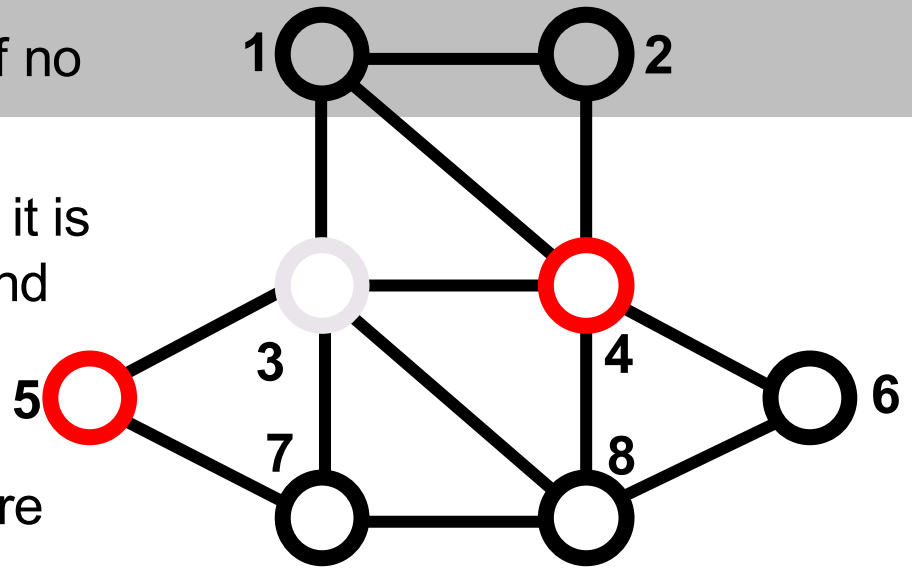
Some have real-life applications!

Parallel Maximal Independent Set

Another example

Maximal Independent Set

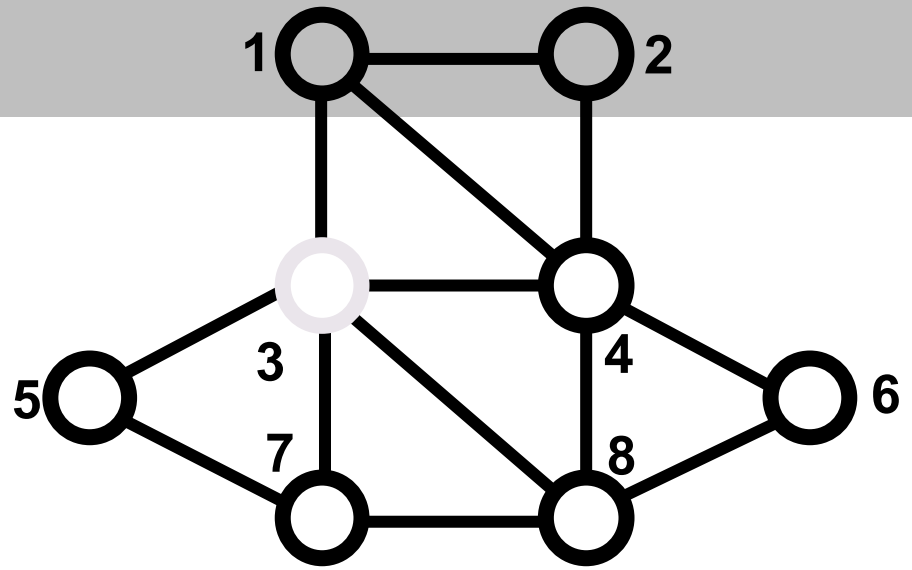
- Graph with vertices V = {1,2,…,n}
- A set S of vertices is independent if no two vertices in S are neighbors.
- An independent set S is maximal if it is impossible to add another vertex and stay independent
- An independent set S is maximum if no other independent set has more vertices
- Finding a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



The set of red vertices
S = {4, 5} is *independent*
and is *maximal*
but not *maximum*

Sequential Maximal Independent Set

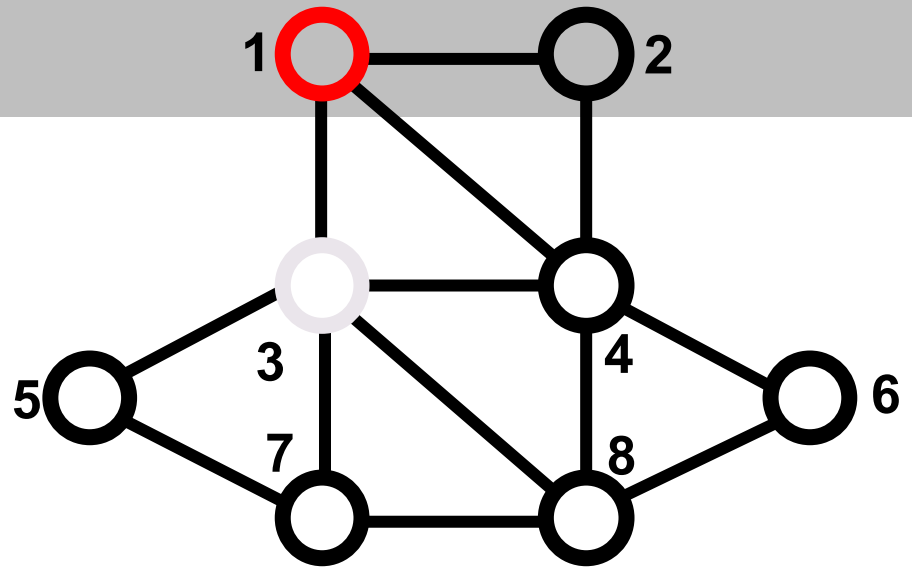
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { }

Sequential Maximal Independent Set

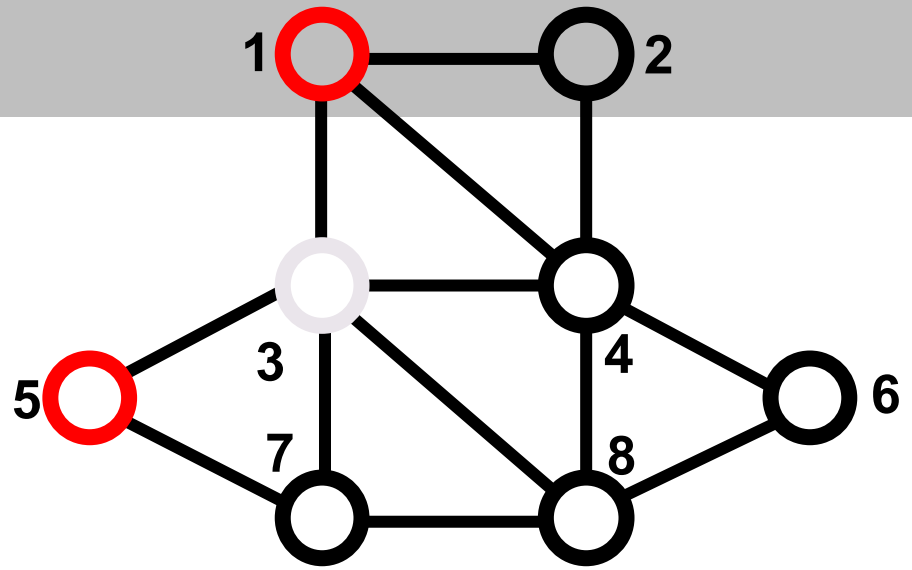
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1 }

Sequential Maximal Independent Set

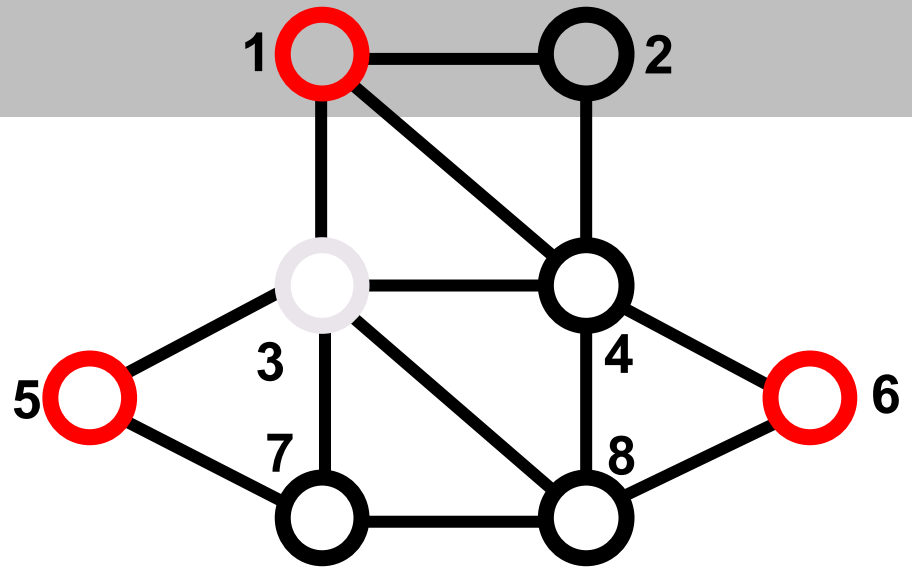
```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```



S = { 1, 5 }

Sequential Maximal Independent Set

```
1.  S = empty set;
2.  for  vertex v = 1 to n {
3.      if (v has no neighbor in S) {
4.          add v to S
5.      }
6.  }
```

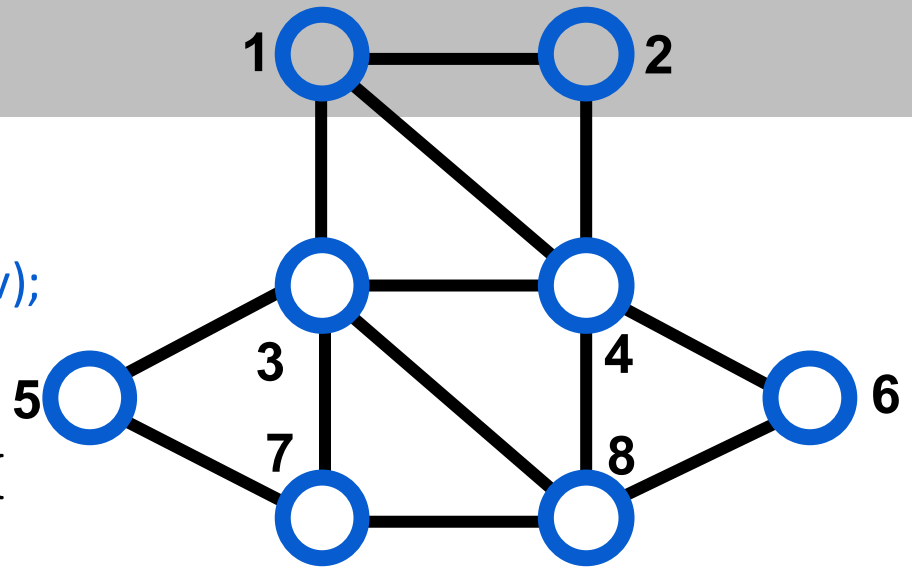


S = { 1, 5, 6 }

work **O(n), but** ***span*** **O(n)**

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



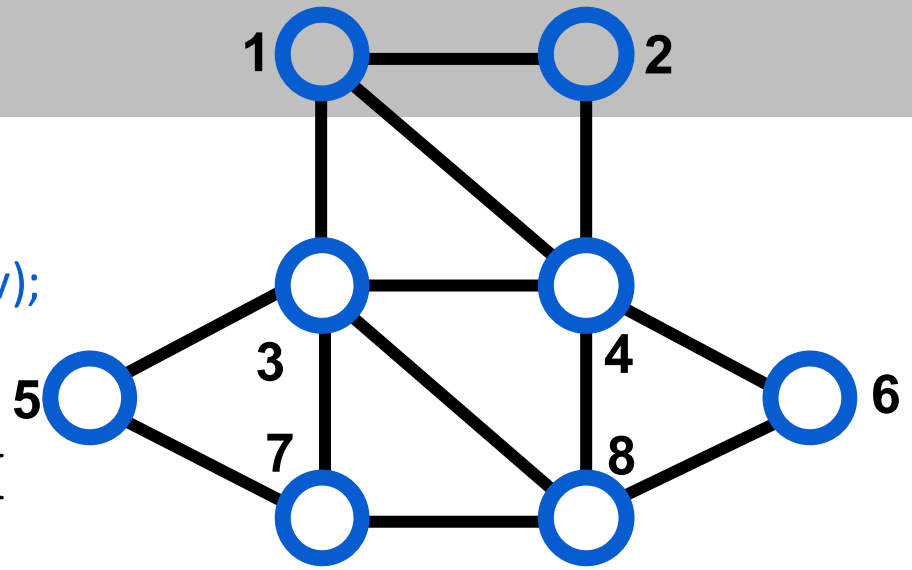
S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

M. Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". *SIAM Journal on Computing* **15** (4): 1036–1053, 1986

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

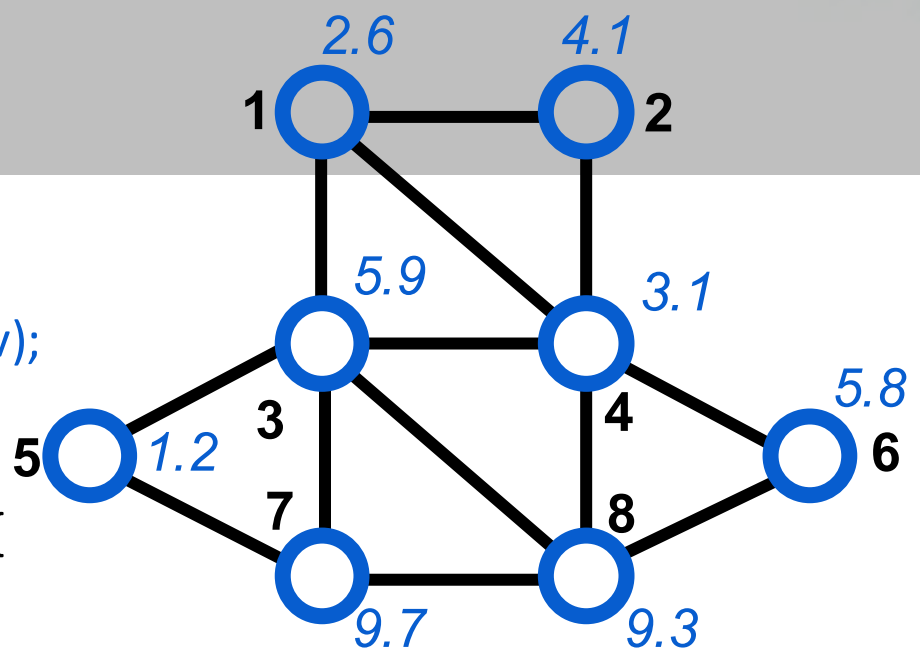


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

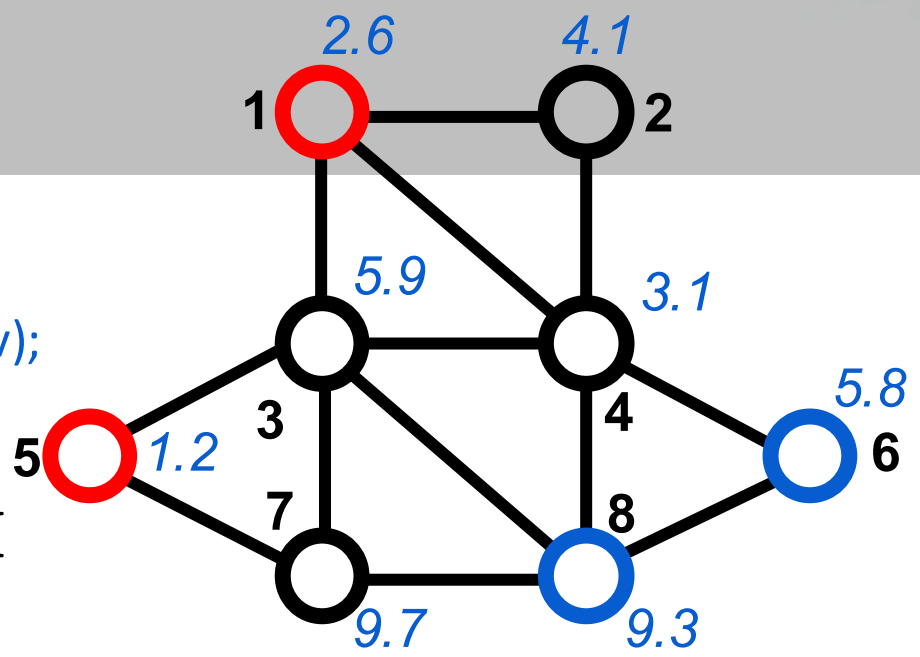


S = { }

C = { 1, 2, 3, 4, 5, 6, 7, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

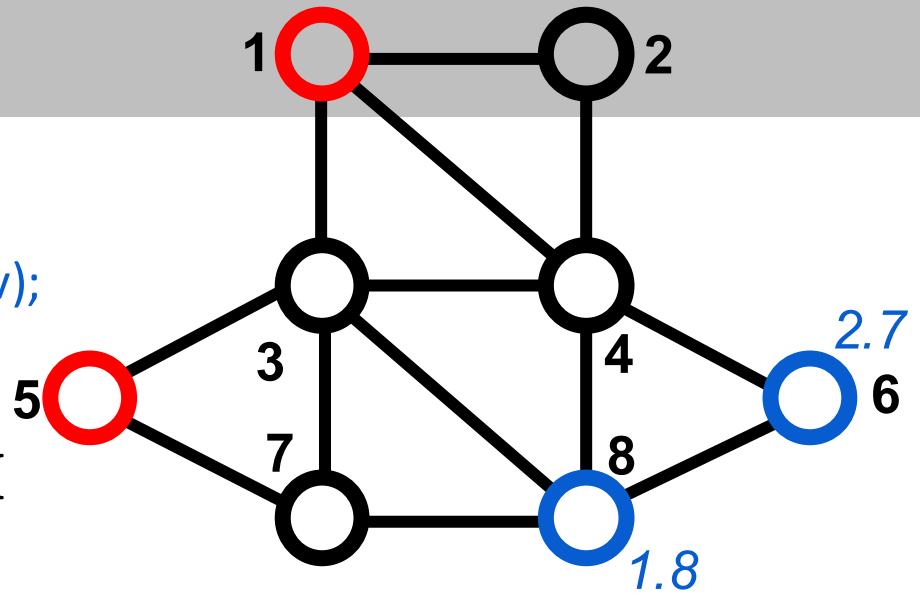


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

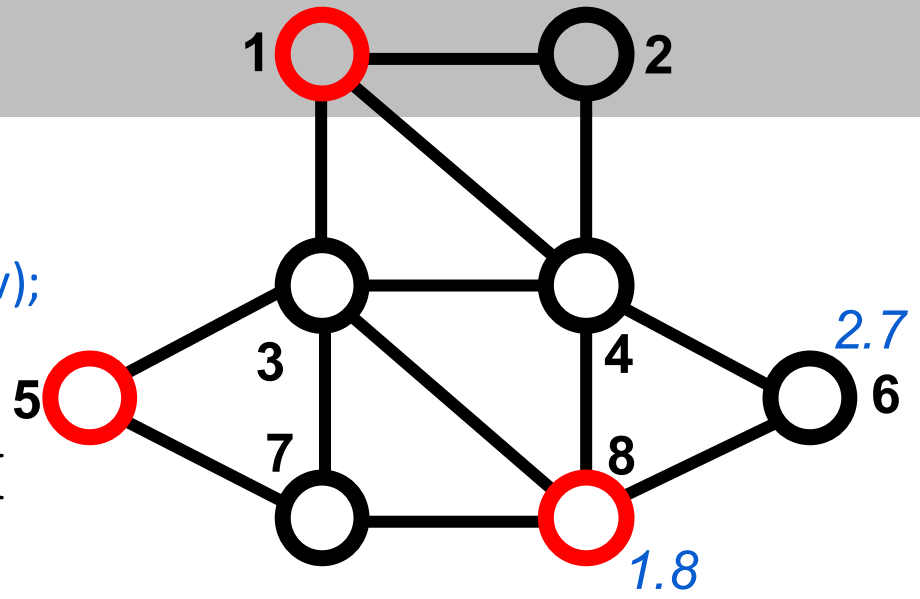


S = { 1, 5 }

C = { 6, 8 }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```

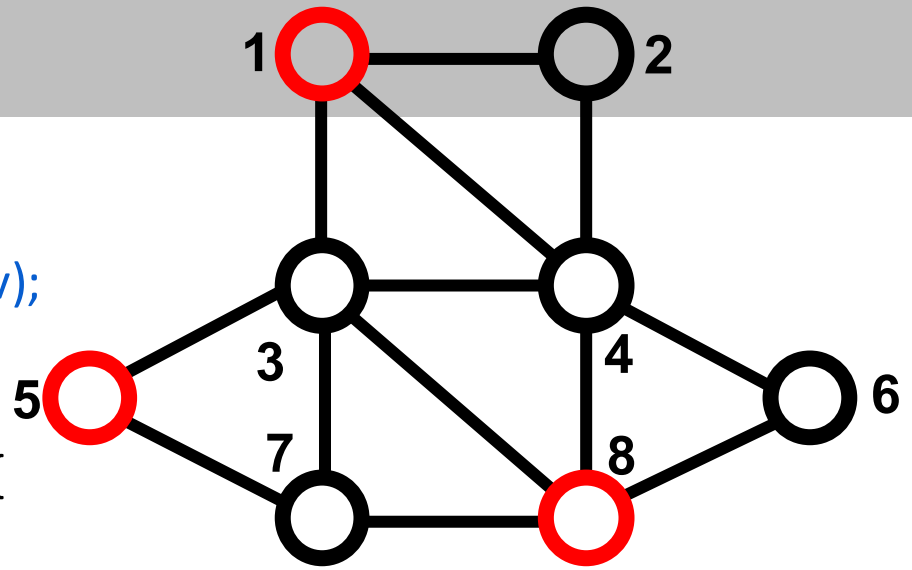


S = { 1, 5, 8 }

C = { }

Parallel, Randomized MIS

```
1.  S = empty set;  C = V;
2.  while  C  is not empty {
3.      label each v in C with a random r(v);
4.      for all v in C in parallel {
5.          if r(v) < min( r(neighbors of v) ) {
6.              move v from C to S;
7.              remove neighbors of v from C;
8.          }
9.      }
10. }
```



Theorem: This algorithm "very probably" finishes within O(log n) rounds.

work **O(n log n), but** ***span*** **O(log n)**

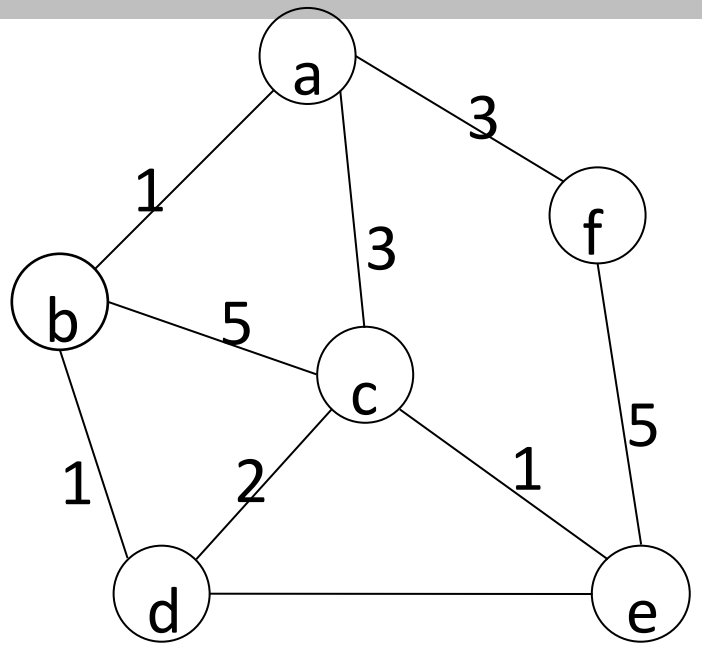
Minimum Spanning Tree

Parallel Prim's Algorithm

Minimum Spanning Tree Problem

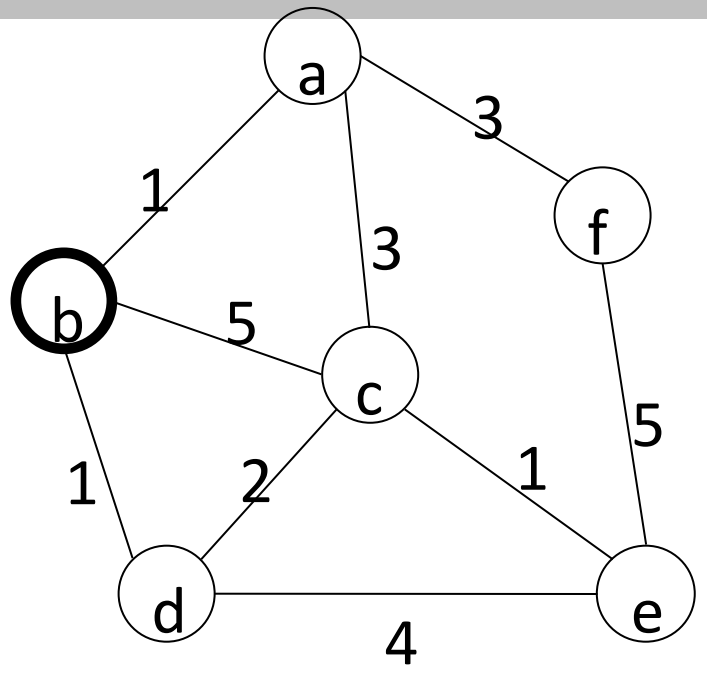
- A **Minimum Spanning Tree (MST)** of a weighted, connected, and undirected graph is a subset of edges that:
  - Connects **all vertices** in the graph (forms a spanning tree)
  - Has the **minimum possible total edge weight**
  - Contains **no cycles**
- **Applications:**
  - Network design: laying out fiber optic cables with minimal cost
  - **Clustering** in machine learning
  - **Circuit design** in VLSI

Example: Prim's Algorithm



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

Root is node b
(Prim's)



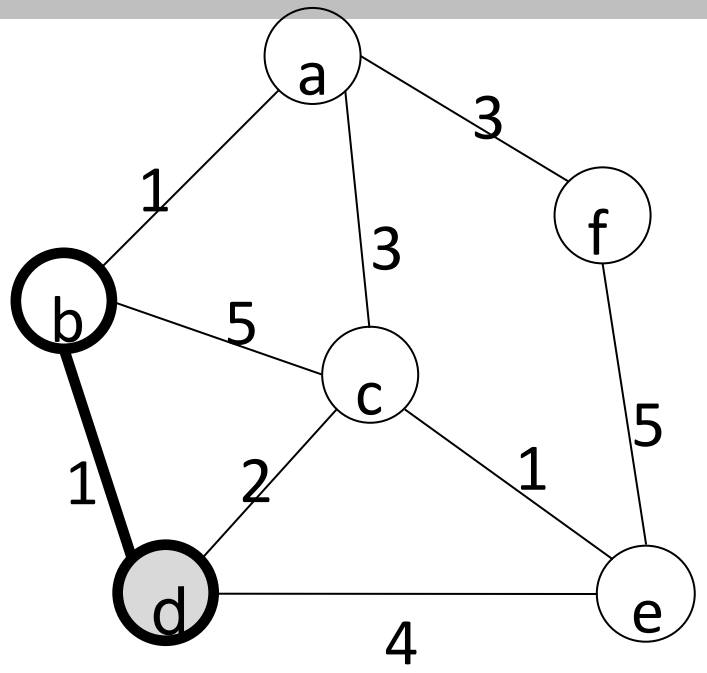
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

Initialize

$$d[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 5 & 1 & \infty & \infty \\ \hline \end{array}$$

Since d[b, d] = 1, add the edge b to d and consider node d next

Next consider node d
(Prim's)



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

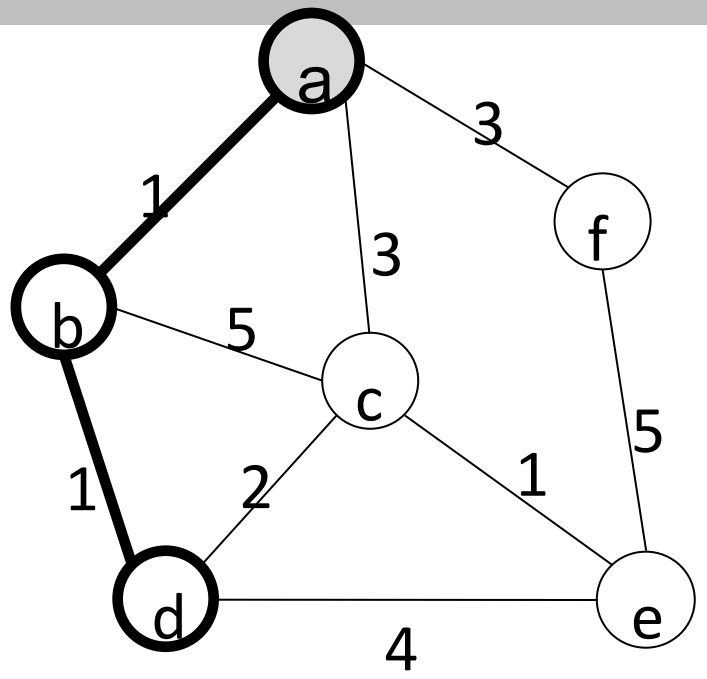
Take Minimum Weight Edge

$$d[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 2 & 1 & 4 & \infty \end{array}$$

Since d[(b, d), a] = 1, add the edge b to a and consider node a next

Next consider node a

(Prim's)

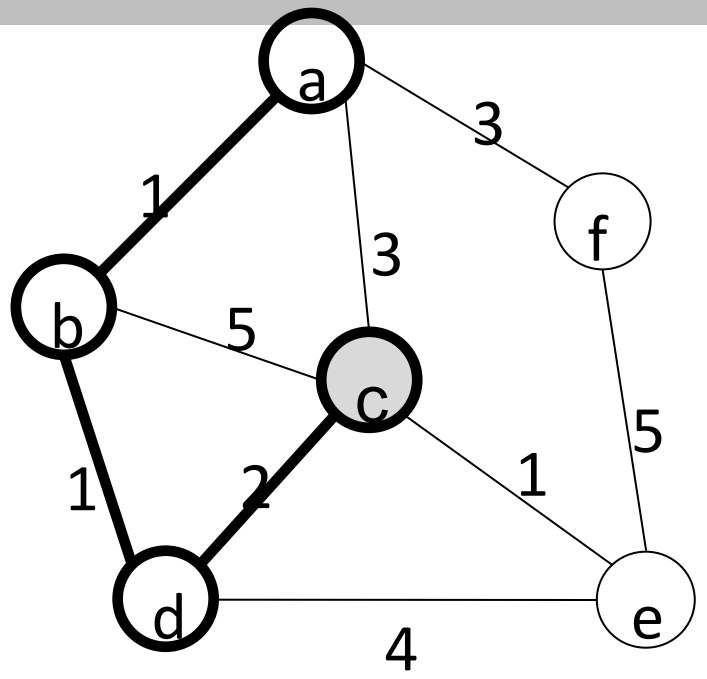


$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$d[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 2 & 1 & 4 & 3 \end{array}$$

Since d[(a, b, d), c] = 2, add the edge d to c and consider node c next

Next consider node c
(Prim's)

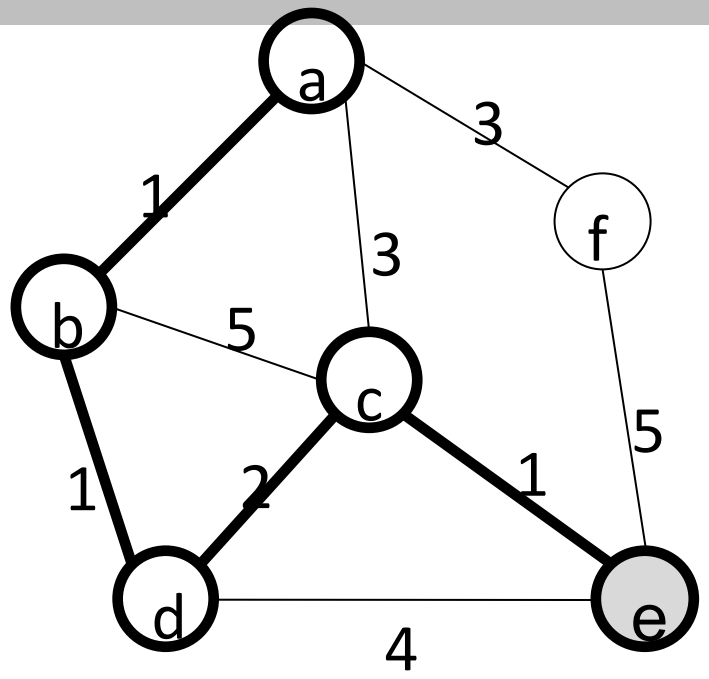


$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$d[] = \begin{array}{|c|c|c|c|c|c|} \hline a & b & c & d & e & f \\ \hline 1 & 0 & 2 & 1 & 1 & 3 \\ \hline \end{array}$$

Since d[(a, b, c, d), e] = 1, add the edge c to e and consider node e next

Next consider node e
(Prim's)

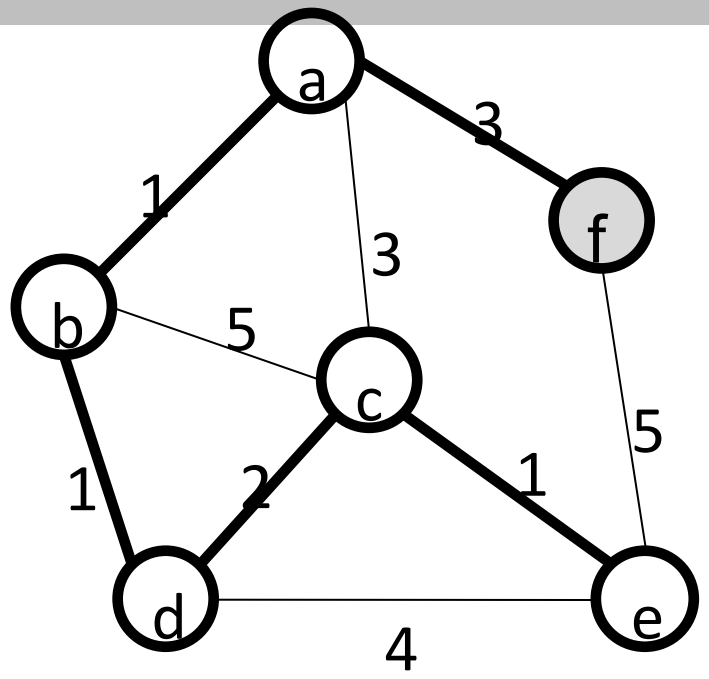


$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$d[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 2 & 1 & 1 & 3 \end{array}$$

Since d[(a, b, c, d, e), f] = 3, add the edge a to f and consider node f next

Next consider node f
(Prim's)



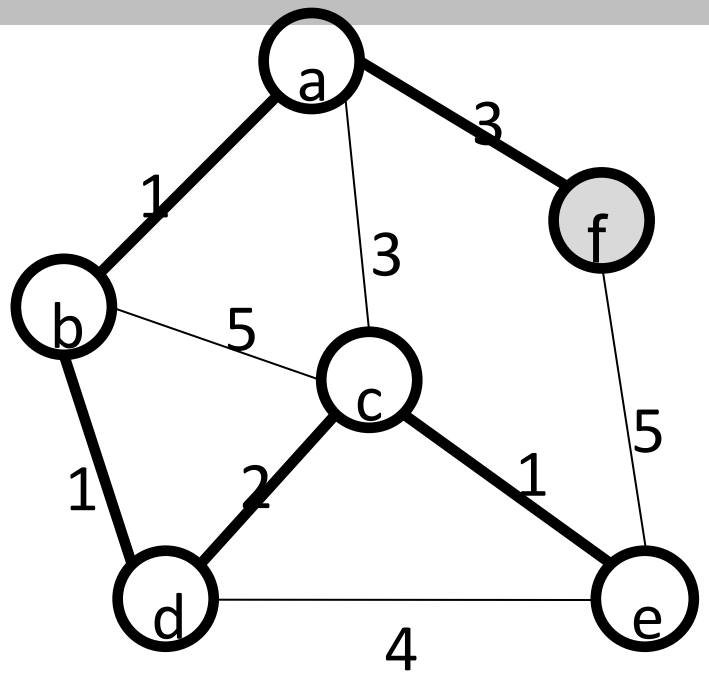
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$d[] = \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 2 & 1 & 1 & 3 \\ \hline \end{array} \end{array}$$

All vertices reached so stop

Next consider node f
(Prim's)

What is the runtime?

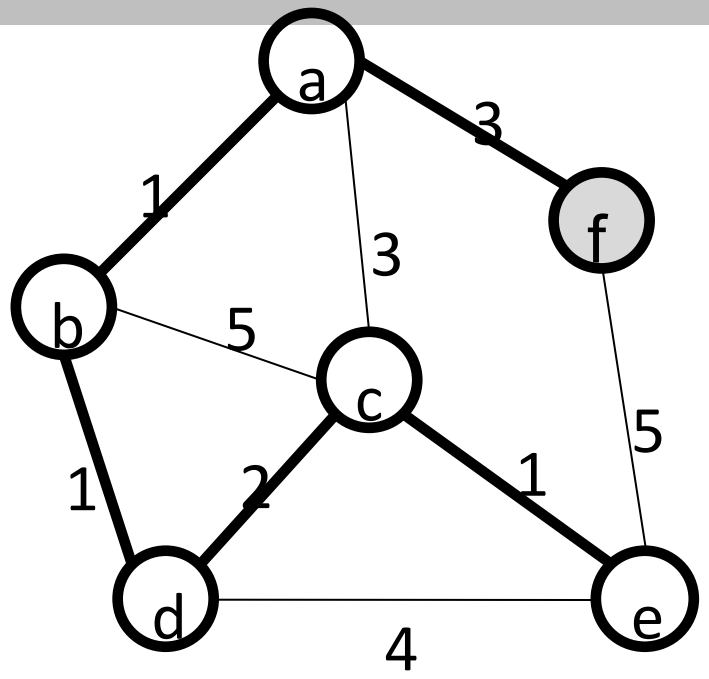


$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$d[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 2 & 1 & 1 & 3 \end{array}$$

All vertices reached so stop

Next consider node f
(Prim's)



$$A = \begin{matrix} a \\ b \\ c \\ d \\ e \\ f \end{matrix} \begin{bmatrix} 0 \\ 1 \\ 3 \\ \infty \\ \infty \\ 3 \end{bmatrix}$$

$d[] =$

All vertices reached so stop

What is the runtime?

$O(n^2)$ with adjacency matrix

$O(m \log n)$ with binary heap

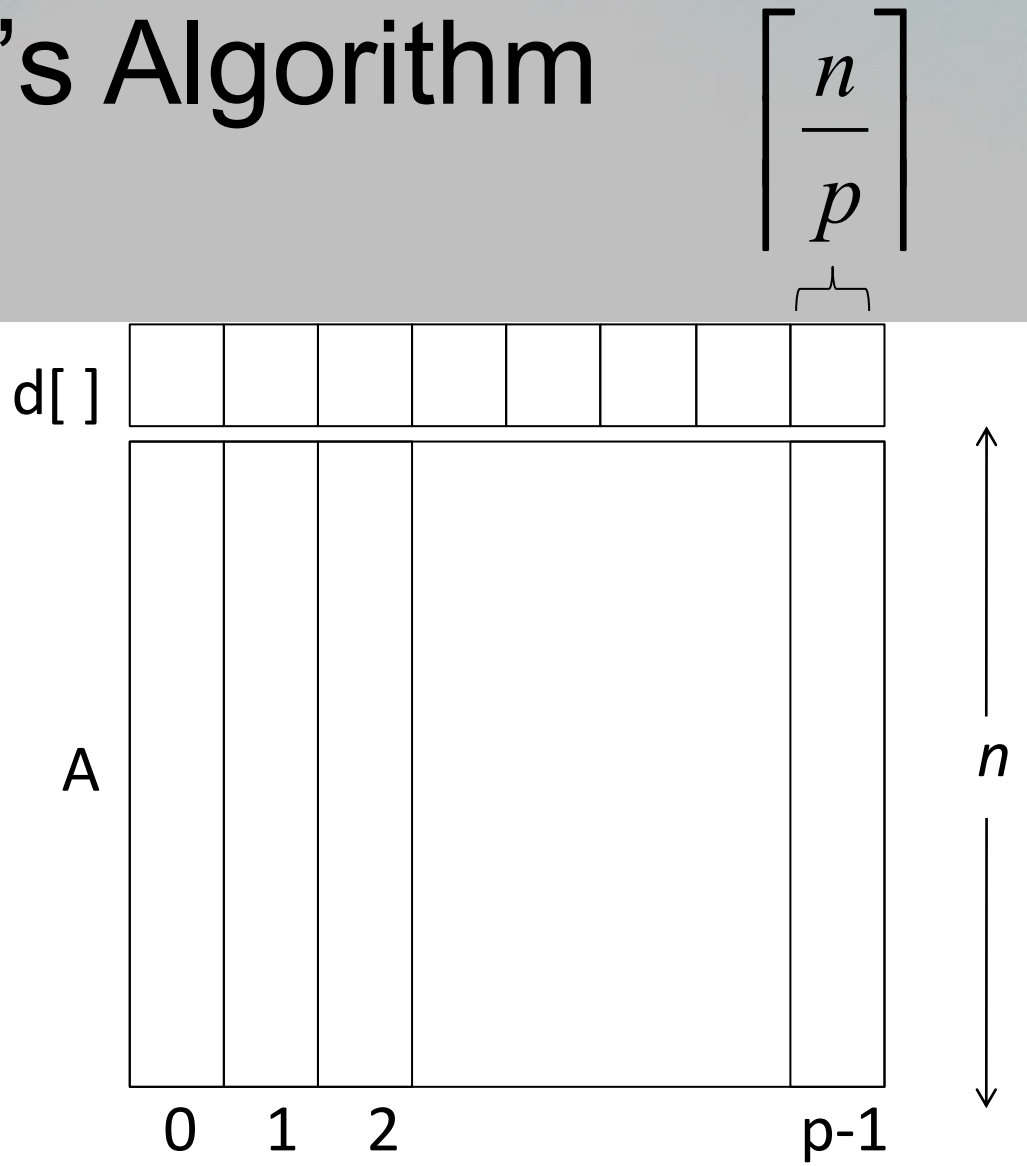
$O(m + n \log n)$ with Fibonacci heap

Parallelizing Prim's Algorithm

- We can't just simply execute the procedure for finding the smallest weight edge in parallel because the d[] array changes with each selection of a vertex
- We have to update values in d[] from all processors after each iteration
- Suppose we have *n* vertices in the graph and *p* processors

Parallelizing Prim's Algorithm

- Partition the adjacency matrix and the distance array (d) across processors



Parallelizing Prim's Algorithm

- Each processor computes the next vertex from among its vertices; this is the **local minimum**
- A parallel reduction is done on the distance array (d) to find the **global minimum**
- **Update d[] array in parallel** using the partitions

Parallel Prim's Algorithm

What is the work?

- Span of parallel algorithm:

$$O\left(\frac{n^2}{p}\right) + O(n \log p)$$

- Each reduction takes $\log p$ time, but we have to do up to n of them
- Setting $p = n$ gives $O(n \log n)$ span

Parallel Prim's Algorithm

- Span of parallel algorithm:

$$O\left(\frac{n^2}{p}\right) + O(n \log p)$$

What is the work?

Work is still $O(n^2)$

- Each reduction takes $\log p$ time, but we have to do up to n of them
- Setting $p = n$ gives $O(n \log n)$ span

Parallel Prim's Algorithm

- Spa

What is the work?

Work is still $O(n^2)$

- Eac
the

But we get $O(n \log n)$ span, better than Fibonacci heap and only need adjacency matrix

- Setting $p = n$ gives $O(n \log n)$ span

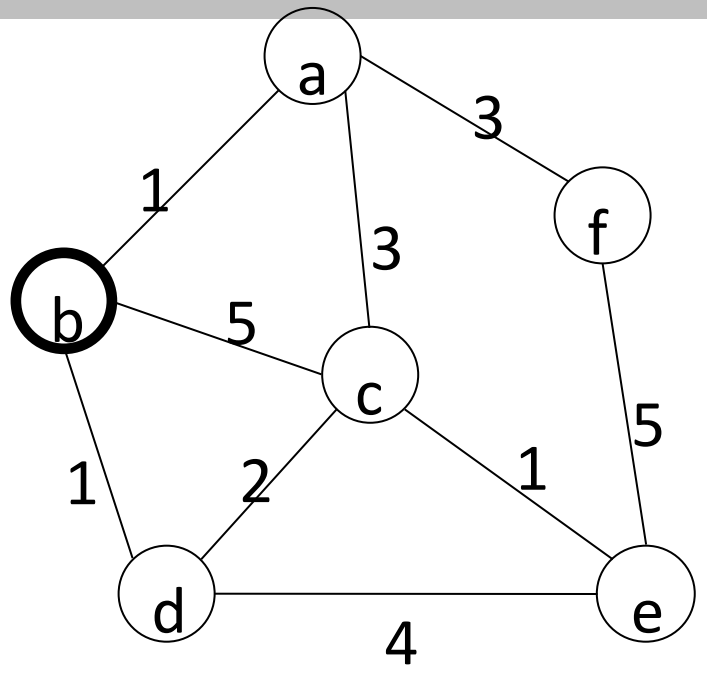
Dijkstra’s Algorithm

Parallel shortest paths algorithms.

Dijkstra's Algorithm for Single-Source Shortest Path

- Given a source node, what is the shortest distance to each other node?

Source Node is node b
(Dijkstra's)



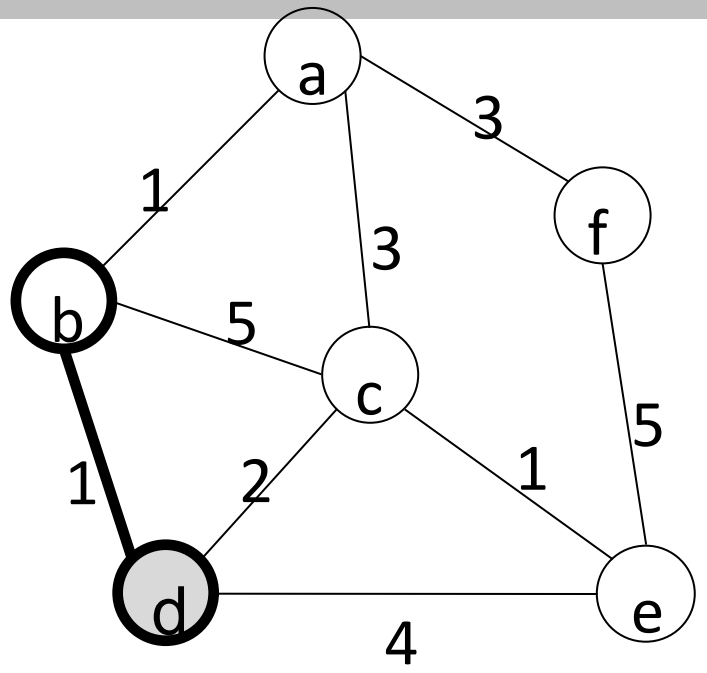
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

Initialize

$$l[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 5 & 1 & \infty & \infty \end{array}$$

Since l[b, d] = 1, consider node d next

Next consider node d
(Dijkstra's)



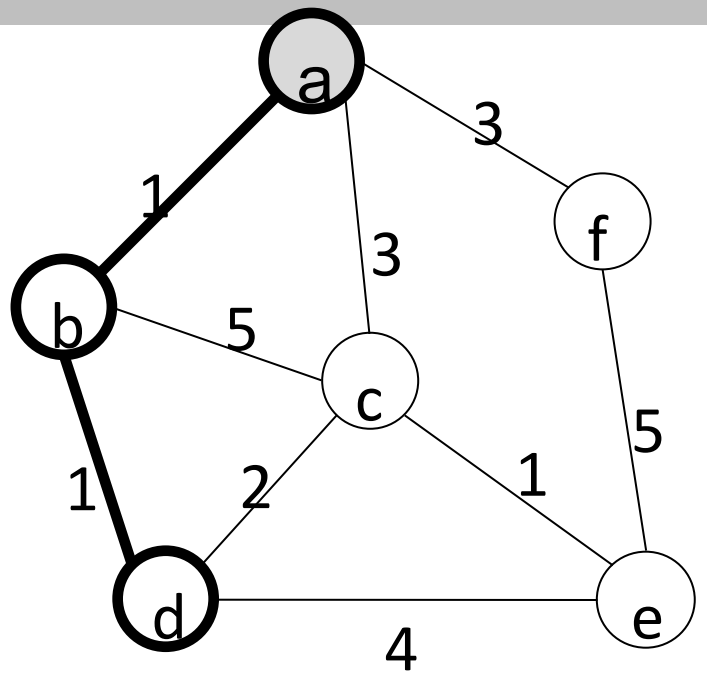
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$l[] =$

a	b	c	d	e	f
1	0	3	1	5	∞

Since l[b, a] = 1, consider node a next

Next consider node a
(Dijkstra's)



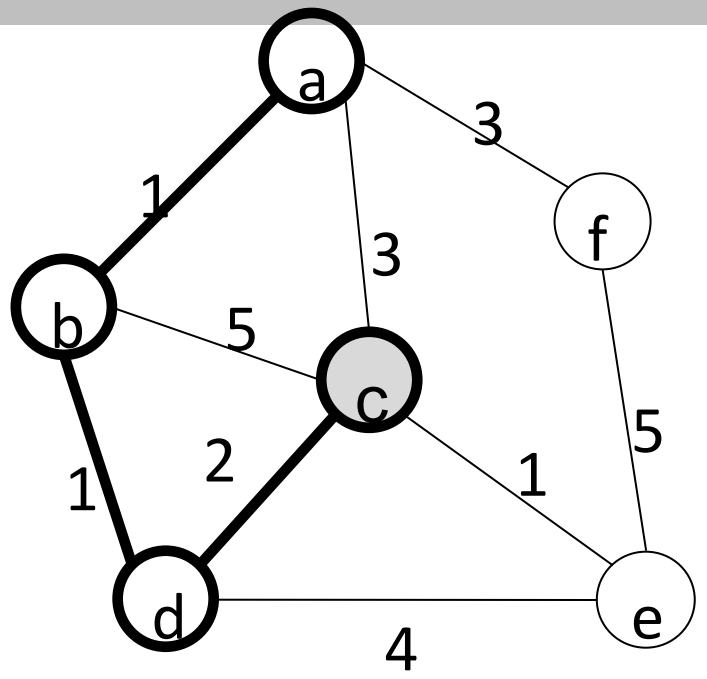
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$l[] =$

a	b	c	d	e	f
1	0	3	1	5	4

Since l[b, c] = 3, consider node c next

Next consider node c
(Dijkstra's)



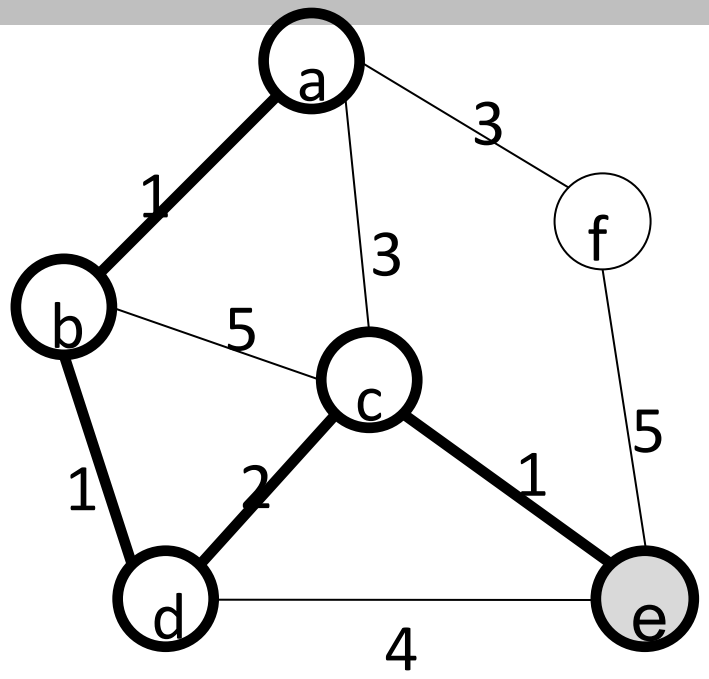
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$l[] =$

a	b	c	d	e	f
1	0	3	1	4	4

Since l[b, e] = 4, consider node e next

Next consider node e
(Dijkstra's)



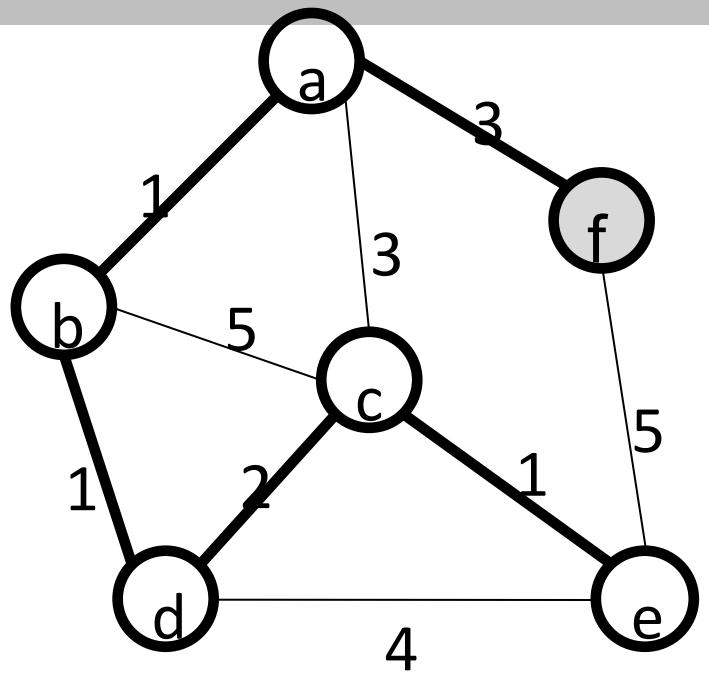
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$l[] =$

a	b	c	d	e	f
1	0	3	1	4	4

Since d[b, f] = 4, add the edge a to f

Next consider node f
(Dijkstra's)



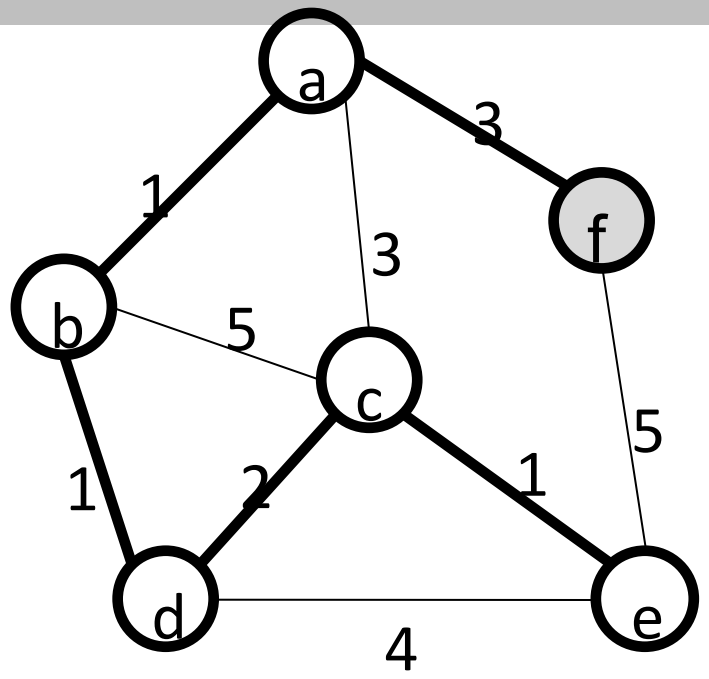
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$$l[] = \begin{array}{cccccc} a & b & c & d & e & f \\ \hline 1 & 0 & 3 & 1 & 4 & 4 \end{array}$$

All nodes have been visited

Next consider node f
(Dijkstra's)

How do we parallelize this algorithm?



$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 3 & \infty & \infty & 3 \\ 1 & 0 & 5 & 1 & \infty & \infty \\ 3 & 5 & 0 & 2 & 1 & \infty \\ \infty & 1 & 2 & 0 & 4 & \infty \\ \infty & \infty & 1 & 4 & 0 & 5 \\ 3 & \infty & \infty & \infty & 5 & 0 \end{bmatrix} \end{array}$$

$l[] =$

a	b	c	d	e	f
1	0	3	1	4	4

All nodes have been visited

Parallelizing Dijkstra's Algorithm

- Since Dijkstra's Algorithm and Prim's Algorithm are essentially the same, we can parallelize them the same way!
- Complexity of parallel algorithm:

$$O\left(\frac{n^2}{p}\right) + O(n \log p)$$

- If we have *n* processors, this becomes:

$$O(n \log n)$$

Using Libraries of Parallel Data Structures and Primitives

Why is it better than using OpenMP and threads for more complicated parallel algorithms?

Our algorithms achieve good parallel scalability

Usage of **shared-memory** parallel data structures

Instead of splitting different instructions on different threads

Putting instructions on different threads vs. using (provably efficient) parallel methods

```
int calculate_sum(const vector<vector<int>>& matrix) {
    int sum = 0;
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            sum += matrix[i][j];
        }
    }
    return sum;
}
```

Sequential: 0.13672 sec.

Setup:

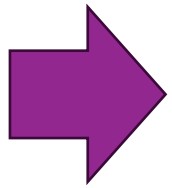
- 10^6 vectors
- $7.5 \cdot 10^5$ have 20 elements
- $2.5 \cdot 10^5$ have 10^3 elements

Sum up all elements using **10 threads**

https://github.com/qqliu/parlay_examples/blob/main/matrix_sum.cpp

Putting instructions on different threads vs. using (provably efficient) parallel methods

```
int calculate_sum(const vector<vector<int>>& matrix) {
    int sum = 0;
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            sum += matrix[i][j];
        }
    }
    return sum;
}
```



```
int parallel_matrix_sum(const vector<vector<int>>& matrix, int num_threads) {
    int rows = matrix.size();

    int rows_per_thread = rows / num_threads;
    int extra_rows = rows % num_threads;

    vector<future<int>> futures;
    int total_sum = 0;

    for (int i = 0; i < num_threads; ++i) {
        int start_row = i * rows_per_thread;
        int end_row = start_row + rows_per_thread;
        if (i == num_threads - 1) {
            end_row += extra_rows;
        }

        futures.push_back(async(launch::async, calculate_row_sum,
                    ref(matrix), start_row, end_row));
    }

    for (auto& future : futures) {
        int partial_sum = future.get();
        total_sum += partial_sum;
    }

    return total_sum;
}
```

Sequential: 0.13672 sec.

Divide vectors among threads: 0.06304 sec.

https://github.com/qqliu/parlay_examples/blob/main/matrix_

Putting instructions on different threads vs. using (provably efficient) parallel methods

```cpp
int calculate_sum(const vector<vector<int>>& matrix) {
    int sum = 0;
    for (int i = 0; i < matrix.size(); i++) {
        for (int j = 0; j < matrix[i].size(); j++) {
            sum += matrix[i][j];
        }
    }
    return sum;
}
```

2.17x speedup

```cpp
int parallel_matrix_sum(const vector<vector<int>>& matrix, int num_threads) {
    int rows = matrix.size();

    int rows_per_thread = rows / num_threads;
    int extra_rows = rows % num_threads;

    vector<future<int>> futures;
    int total_sum = 0;

    for (int i = 0; i < num_threads; ++i) {
        int start_row = i * rows_per_thread;
        int end_row = start_row + rows_per_thread;
        if (i == num_threads - 1) {
            end_row += extra_rows;
        }

        futures.push_back(async(launch::async, calculate_row_sum,
                    ref(matrix), start_row, end_row));
    }

    for (auto& future : futures) {
        int partial_sum = future.get();
        total_sum += partial_sum;
    }

    return total_sum;
}
```

Sequential: 0.13672 sec.

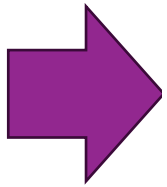
Divide vectors among threads: 0.06304 sec.

https://github.com/qqliu/parlay_examples/blob/main/matrix_

Putting instructions on different threads vs. using (provably efficient) parallel methods

```cpp
int parallel_matrix_sum(const vector<vector<int>>& matrix, int num_threads) {
    int rows = matrix.size();

    int rows_per_thread = rows / num_threads;
    int extra_rows = rows % num_threads;

    vector<future<int>> futures;
    int total_sum = 0;

    for (int i = 0; i < num_threads; ++i) {
        int start_row = i * rows_per_thread;
        int end_row = start_row + rows_per_thread;
        if (i == num_threads - 1) {
            end_row += extra_rows;
        }

        futures.push_back(async(launch::async, calculate_row_sum,
                    ref(matrix), start_row, end_row));
    }

    for (auto& future : futures) {
        int partial_sum = future.get();
        total_sum += partial_sum;
    }

    return total_sum;
}
```

https://github.com/qqliu/parlay_examples/blob/main/matrix_sum.cpp



```cpp
int parlay_scan_matrix_sum(const vector<vector<int>>& matrix) {
    parlay::sequence row_sums(matrix.size(), 0);

    parlay::parallel_for(0, matrix.size(), [&] (int i) {
        row_sums[i] = parlay::scan(matrix[i], parlay::plus<int>()).second;
    });

    int sum = parlay::scan(row_sums, parlay::plus<int>()).second;
    return sum;
}
```

https://github.com/cmuparlay/parlaylib

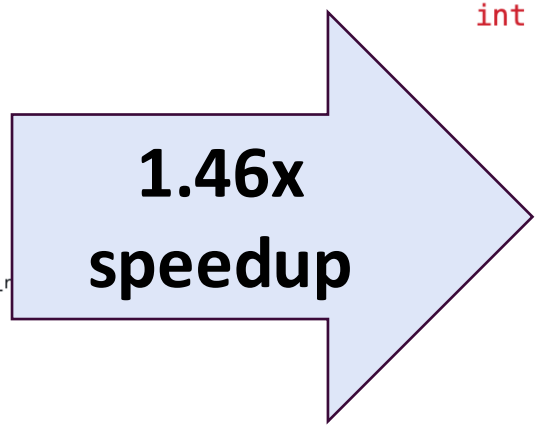
Divide vectors among threads:
0.06304 sec.

Using parallel scan:
0.0432 sec.

Putting instructions on different threads vs. using (provably efficient) parallel methods

```
int parallel_matrix_sum(const vector<vector<int>>& matrix, int num_threads) {
    int rows = matrix.size();

    int rows_per_thread = rows / num_threads;
    int extra_rows = rows % num_threads;

    vector<future<int>> futures;
    int total_sum = 0;

    for (int i = 0; i < num_threads; ++i) {
        int start_row = i * rows_per_thread;
        int end_row = start_row + rows_per_thread;
        if (i == num_threads - 1) {
            end_row += extra_rows;
        }

        futures.push_back(async(launch::async, calculate_r
                    ref(matrix), start_row, end_row));
    }

    for (auto& future : futures) {
        int partial_sum = future.get();
        total_sum += partial_sum;
    }

    return total_sum;
}
```

https://github.com/qqliu/parlay_examples/blob/main/matrix_sum.cpp




```
int parlay_scan_matrix_sum(const vector<vector<int>>& matrix) {
    parlay::sequence row_sums(matrix.size(), 0);

    parlay::parallel_for(0, matrix.size(), [&] (int i) {
        row_sums[i] = parlay::scan(matrix[i], parlay::plus<int>()).second;
    });

    int sum = parlay::scan(row_sums, parlay::plus<int>()).second;
    return sum;
}
```

https://github.com/cmuparlay/parlaylib

Divide vectors among threads:
0.06304 sec.

Using parallel scan:
0.0432 sec.

Putting instructions on different threads vs. using (provably efficient) parallel methods

https://github.com/qqliu/parlay_examples/blob/main/matrix_sum.cpp

Why?

1.46x speedup

```
int parlay_scan_matrix_sum(const vector<vector<int>>& matrix) {
    parlay::sequence row_sums(matrix.size(), 0);

    parlay::parallel_for(0, matrix.size(), [&] (int i) {
        row_sums[i] = parlay::scan(matrix[i], parlay::plus<int>()).second;
    });

    int sum = parlay::scan(row_sums, parlay::plus<int>()).second;
    return sum;
}
```

https://github.com/cmuparlay/parlaylib

Divide rows among threads: 0.06304 sec.

Using parallel scan: 0.0432 sec.

Putting instructions on different threads vs. using (provably efficient) parallel methods

https://github.com/qqliu/parlay_examples/blob/main/matrix_sum.cpp

Why?

Some vectors **smaller than others**; **uneven** distribution of work

1.46x speedup

```
int parlay_scan_matrix_sum(const vector<vector<int>>& matrix) {
    parlay::sequence row_sums(matrix.size(), 0);

    parlay::parallel_for(0, matrix.size(), [&] (int i) {
        row_sums[i] = parlay::scan(matrix[i], parlay::plus<int>()).second;
    });

    int sum = parlay::scan(row_sums, parlay::plus<int>()).second;
    return sum;
}
```

https://github.com/cmuparlay/parlaylib

Divide rows among threads: 0.06304 sec.

Using parallel scan: 0.0432 sec.

Putting instructions on different threads vs. using (provably efficient) parallel methods

Why?

Some vectors **smaller than others**; **uneven** distribution of work

1.46x speedup

Irregular data:

Graphs are highly irregular!

Divide rows among threads:
0.06304 sec.

Using parallel scan:
0.0432 sec.

Packages that contain parallel primitives

- GraphBLAS (https://graphblas.org/)
  - High-performance linear algebra-based framework
  - Graph computations
  - Sparse graph processing
  - Computations on matrices
- **Parlaylib (https://github.com/cmuparlay/parlaylib)**
  - C++ library for parallel algorithms and data structures
  - Shared-memory multicore
  - Particularly well-suited for irregular computations

https://cmuparlay.github.io/parlaylib/

A Toolkit for Programming Parallel Algorithms on Shared-Memory Multicore Machines

What is ParlayLib?

- A C++ library providing a **suite of parallel algorithms and data structures**
- Designed to simplify writing efficient parallel code **on multi-core shared-memory architectures**
- **Goal:**
  - Enable developers to express parallelism at a high level while abstracting low-level thread management

Key Features of Parlaylib

- **High-Level Abstractions:**
  - Data-parallel primitives such as parallel sort, scan, and reduce
- **Ease of Integration**:
  - Can be integrated into existing C++ projects with minimal changes
- **Github page**:
  - https://github.com/cmuparlay/parlaylib

Components of ParlayLib

- **Underlying model**:
  - **Task-based parallelism**:
    - Overall workload divided into smaller, often independent (different) **tasks**
    - Tasks can be functions, operations, or routines
    - **Dynamic scheduling** runtime scheduler schedules tasks dynamically
    - Algorithms to manage concurrency and synchronization
    - Good for irregular workloads
  - **Data-based parallelism**: performing same operation on each element of dataset simultaneously

Components of ParlayLib

- **Algorithms:**
  - Parallel versions of common operations:
    - Sorting
    - Scanning
    - Sums
    - Filters
- **Data structures:**
  - Arrays, sequences, hash tables, and other containers for parallel operations
- **Easy to integrate** with other C++ libraries

ParlayLib abstracts away the need for lower level thread management; **high-level constructs you can directly use**

How to Use ParlayLib

How to Use ParlayLib

- Instructions for use given at https://cmuparlay.github.io/parlaylib/installation.html
- For our class assignments, download ParlayLib as a submodule
- We'll include the folder in the assignment
- Include files from ParlayLib as you would include any other files
- ParlayLib is a **header-only library** so easy integration into your programs

How to Use ParlayLib

Include files from ParlayLib as you would normal header files

```
#include "parlaylib/include/parlay/parallel.h"
#include "parlaylib/include/parlay/primitives.h"
#include "parlaylib/include/parlay/sequence.h"
#include "parlaylib/examples/mergesort.h"

int main() {
   // Use parlay sequence to create sequence of integers
   parlay::sequence<int> data = {5, 2, 9, 1, 5, 6};

   // Perform parallel merge sort
   merge_sort(data);

   for (int x : data) {
      std::cout << x << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

How to Use ParlayLib

Use ParlayLib-specific data structures: allows you to perform parallel/concurrent operations safely

```
#include "parlaylib/include/parlay/parallel.h"
#include "parlaylib/include/parlay/primitives.h"
#include "parlaylib/include/parlay/sequence.h"
#include "parlaylib/examples/mergesort.h"

int main() {
   // Use parlay sequence to create sequence of integers
   parlay::sequence<int> data = {5, 2, 9, 1, 5, 6};

   // Perform parallel merge sort
   merge_sort(data);

   for (int x : data) {
      std::cout << x << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

How to Use ParlayLib

Perform parallel merge_sort to sort the data

```
#include "parlaylib/include/parlay/parallel.h"
#include "parlaylib/include/parlay/primitives.h"
#include "parlaylib/include/parlay/sequence.h"
#include "parlaylib/examples/mergesort.h"

int main() {
   // Use parlay sequence to create sequence of integers
   parlay::sequence<int> data = {5, 2, 9, 1, 5, 6};

   // Perform parallel merge sort
    merge_sort(data);

   for (int x : data) {
      std::cout << x << " ";
   }
   std::cout << std::endl;
   return 0;
}
```

How to Use ParlayLib

Now let's sort 10^8 integers

ParlayLib Implementation

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use parlay sequence to create sequence of integers
parlay::sequence<int> data(n);

for (size_t i = 0; i < n; i++) {
   data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
merge_sort(data);
t.next("mergesort time");
```

How to Use ParlayLib

Now let's sort 10^8 integers

ParlayLib implementation versus std::sort

ParlayLib Implementation

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use parlay sequence to create sequence of integers
parlay::sequence<int> data(n);

for (size_t i = 0; i < n; i++) {
   data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
merge_sort(data);
t.next("mergesort time");
```

How to Use ParlayLib

Now let's sort 10^8 integers

ParlayLib implementation versus std::sort

std::sort Implementation

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use std::vector to create sequence of integers
std::vector<int> data(n);

for (size_t i = 0; i < n; i++) {
   data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
std::sort(data.begin(), data.end());
t.next("mergesort time");
```

How to Use ParlayLib

ParlayLib: 1.1720
std::sort: 5.5806

I only have 8 cores on my laptop!

std::sort Implementation

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use std::vector to create sequence of integers
std::vector<int> data(n);

for (size_t i = 0; i < n; i++) {
    data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
std::sort(data.begin(), data.end());
t.next("mergesort time");
```

How to Use ParlayLib

Test it yourself!

```
g++ -std=c++17 -O3 parallel_sort.cpp -o parallel_sort -pthread
```

How to Use ParlayLib

Test it yourself!

```
g++ -std=c++17 -O3 parallel_sort.cpp -o parallel_sort -pthread
```

https://github.com/qqliu/CPSC424 (Lecture 15)

Using ParlayLib with OpenMP

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use parlay sequence to create sequence of integers
parlay::sequence<int> data(n);

#pragma omp parallel for
for (size_t i = 0; i < n; i++) {
   data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
merge_sort(data);
t.next("mergesort time");
```

Using ParlayLib with OpenMP

Only need to add the flag

-fopenmp

```
const size_t n = 100000000;
const unsigned seed = 42;

std::mt19937 gen(seed);
std::uniform_int_distribution<int> dist(0, 100000000);

// Use parlay sequence to create sequence of integers
parlay::sequence<int> data(n);

#pragma omp parallel for
for (size_t i = 0; i < n; i++) {
    data[i] = dist(gen);
}

parlay::internal::timer t("Time");

t.start();
merge_sort(data);
t.next("mergesort time");
```