SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Parallel Programming Examples

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Parallel Programming

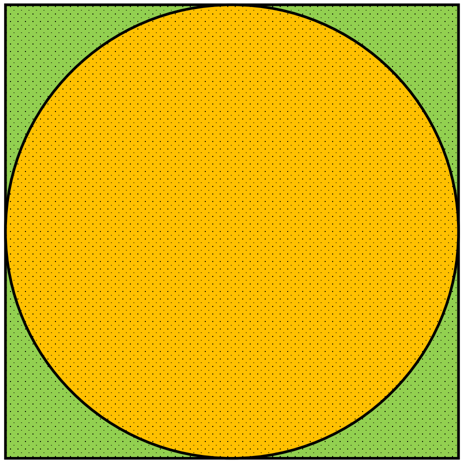
- Goal: Parallelize some simple sequential programs
- Understand some of the challenges involved
- Not tied to any specific language

- Homework
  - Implement each parallel program and check performance
  - Try to improve performance with better parallelization strategies

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

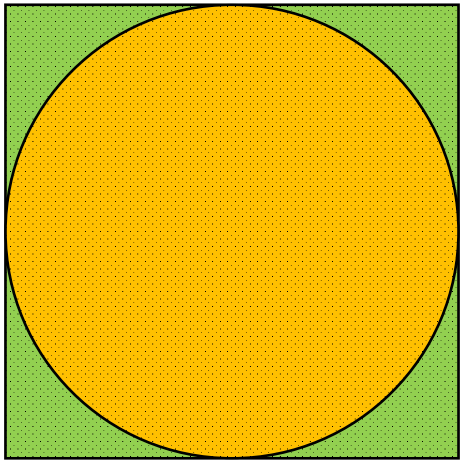
```
circle_count = 0;
for (uint i = 0; i < n; i++) {
    x = get_random(0, 1);
    y = get_random(0, 1);
    if ((x, y) inside circle)
        ++circle_count;
}
pi_value = 4.0 * circle_count / n;
```



Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

```
circle_count = 0;
create T threads
for each thread in parallel {
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            ++circle_count;
    }
}
pi_value = 4.0 * circle_count / n;
```

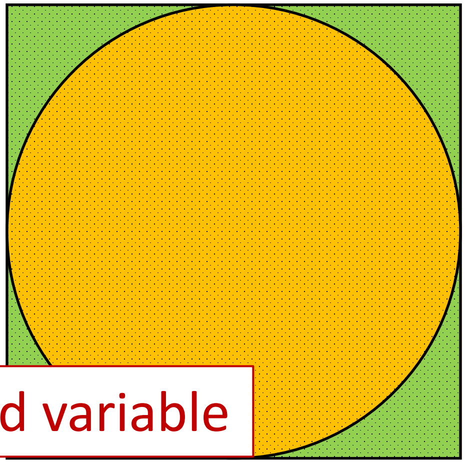


Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

```
circle_count = 0;
create T threads
for each thread in parallel {
    for (uint i = 0;
          i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            ++circle_count;
    }
}
pi_value = 4.0 * circle_count / n;
```

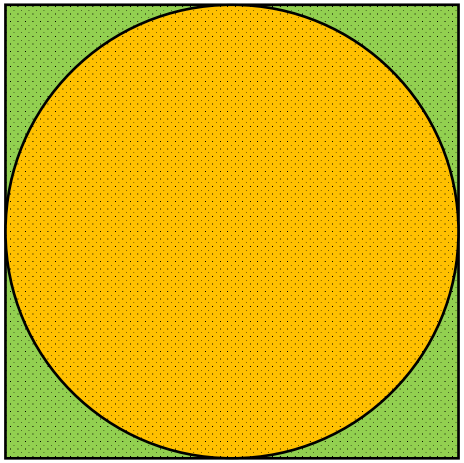
shared variable



Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

```
circle_count = 0;
create T threads
for each thread in parallel {
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            lock();
            ++circle_count;
            unlock();
    }
}
pi_value = 4.0 * circle_count / n;
```

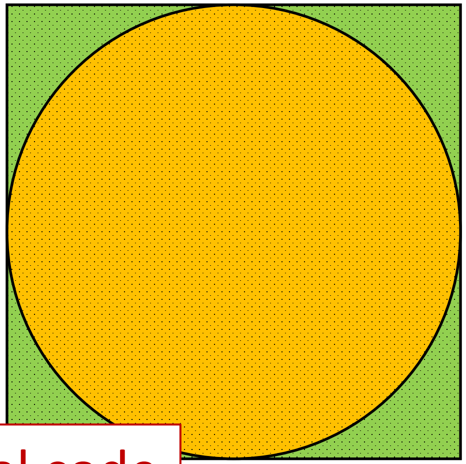


Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

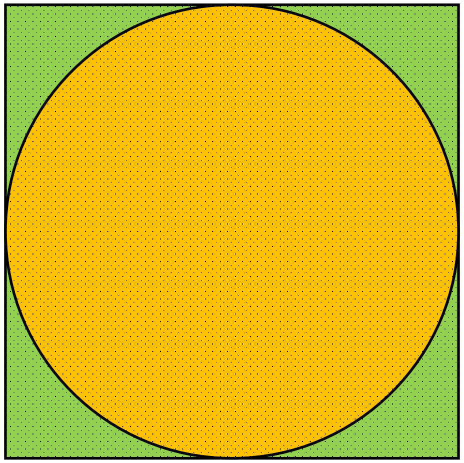
```
circle_count = 0;
create T threads
for each thread in parallel {
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            lock();
            ++circle_count;
            unlock();
    }
}
pi_value = 4.0 * circle_count / n;
```

serial code



Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

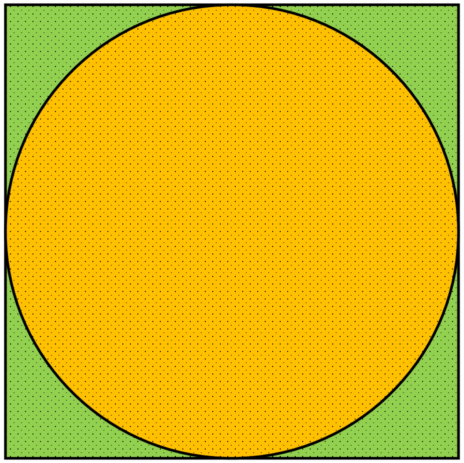


```
circle_count = 0;
create T threads
for each thread in parallel {
    local_circle_count = 0;
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            ++local_circle_count;
    }
    lock();
    circle_count += local_circle_count;
    unlock();
}
pi_value = 4.0 * circle_count / n;
```

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

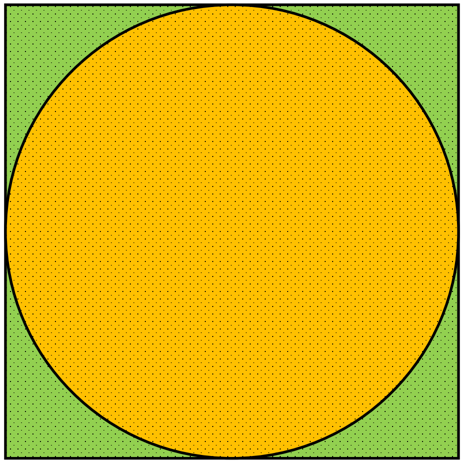
```
circle_count = 0;
create T threads
for each thread in parallel {
    local_circle_count = 0;
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            ++local_circle_count;
    }
    lock();
    circle_count += local_circle_count;
    unlock();
}
pi_value = 4.0 * circle_count / n;
```



use atomics to eliminate locks

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$



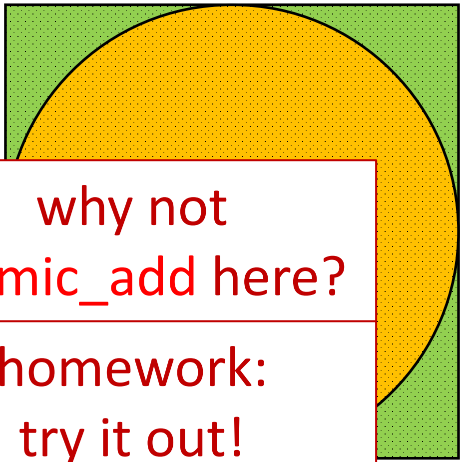
```
circle_count = 0;
create T threads
for each thread in parallel {
    local_circle_count = 0;
    for (uint i = 0;
            i < approx(n/T); i++) {
          x = get_random(0, 1);
          y = get_random(0, 1);
          if ((x, y) inside circle)
              ++local_circle_count;
    }
    atomic_add(circle_count, local_circle_count);
}
pi_value = 4.0 * circle_count / n;
```

check out FAA, CAS

CAS: https://en.wikipedia.org/wiki/Compare-and-swap
FAA: https://en.wikipedia.org/wiki/Fetch-and-add
C++11 std::atomic: https://en.cppreference.com/w/cpp/atomic

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$



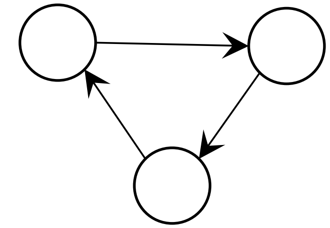
```
circle_count = 0;
create T threads
for each thread in parallel {
    local_circle_count = 0;
    for (uint i = 0;
            i < approx(n/T); i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
            ++local_circle_count;
    }
    atomic_add(circle_count, local_circle_count);
}
pi_value = 4.0 * circle_count / n;
```

why not atomic_add here?

homework: try it out!

CAS: https://en.wikipedia.org/wiki/Compare-and-swap
FAA: https://en.wikipedia.org/wiki/Fetch-and-add
C++11 std::atomic: https://en.cppreference.com/w/cpp/atomic

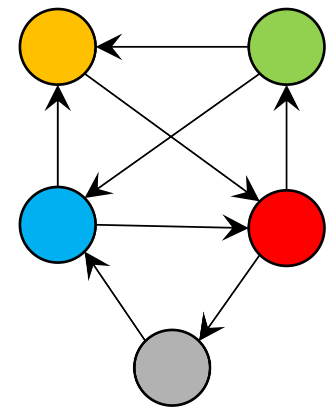
Triangle Counting



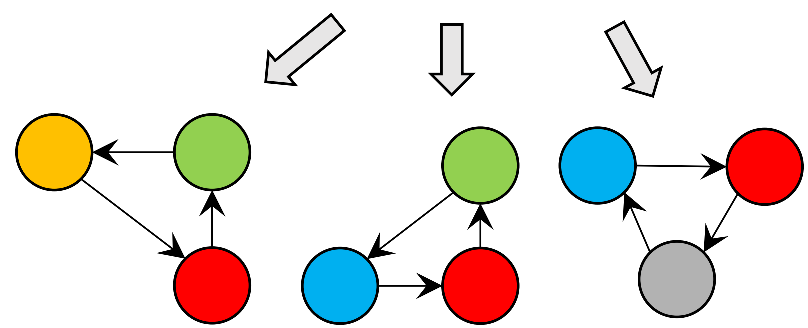
Triangle

```
triangles(u, v) {
    return |outNeighbors(v) ∩ inNeighbors(u)|;
}

count = 0;
for u in V {
    for v in outNeighbors(u) {
        count += triangles(u, v);
    }
}
count = count / 3;
```

parallelize

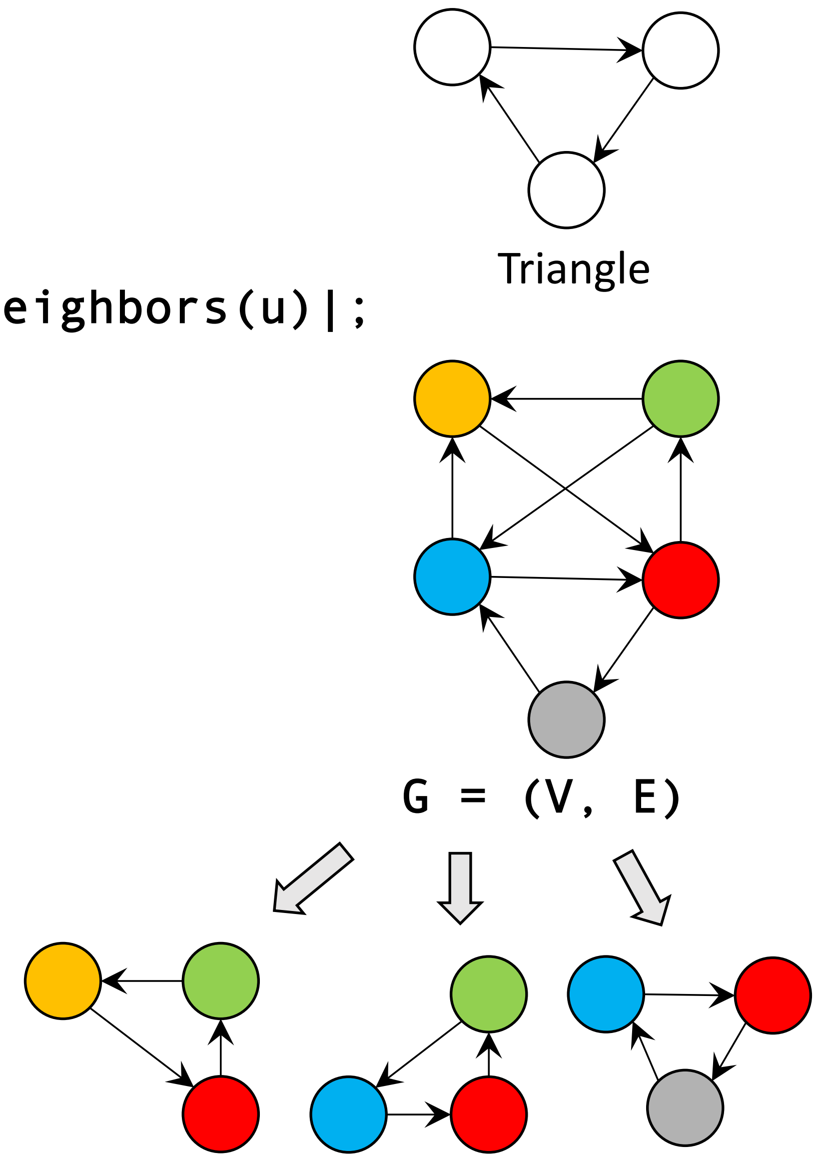


G = (V, E)



Triangles

Triangle Counting

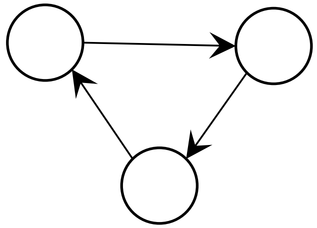


Triangles

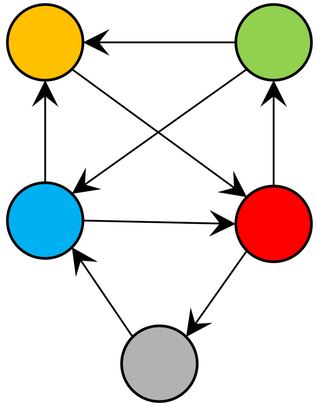
```
triangles(u, v) {
    return |outNeighbors(v) ∩ inNeighbors(u)|;
}

count = 0;
for u in V {
    for v in outNeighbors(u) {
        count += triangles(u, v);
    }
}
count = count / 3;
```

write shared

Triangle Counting



Triangle

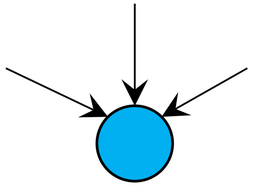


```
triangles(u, v) {
    return |outNeighbors(v) ∩ inNeighbors(u)|;
}

count = 0;
create T threads
for each thread in parallel {
    local_count = 0;
    for u in subset(V, tid) {
        for v in outNeighbors(u) {
            local_count += triangles(u, v);
        }
    }
    atomic_add(count, local_count);
}
count = count / 3;
```

solution similar to π estimation

PageRank

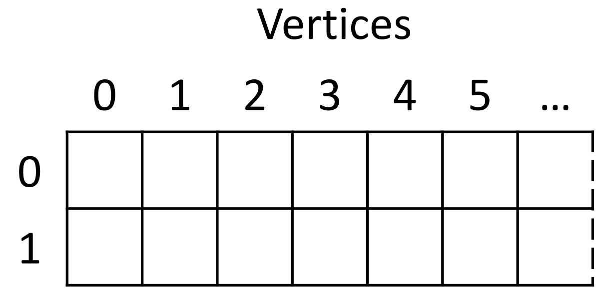
$$PR[V] = 0.15 + 0.85 \times \sum_{u \in inNeighbors(v)} \frac{PR[u]}{degree(u)}$$



- Used by Google to rank webpages
- Estimates importance of a webpage by counting the number and quality of links to the page
- Iterative algorithm
  - Compute ranks over and over again to account for changes in incoming links' qualities
  - PageRank values stop changing over time

PageRank: https://en.wikipedia.org/wiki/PageRank

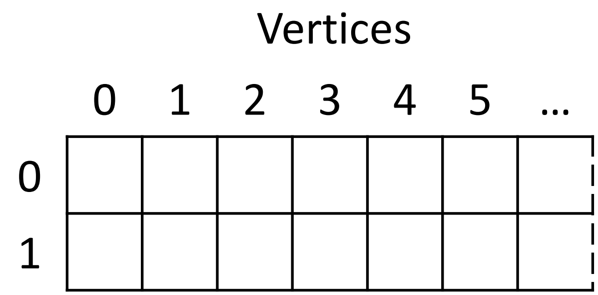
PageRank




```
#define CURR i % 2
#define NEXT (i+1) % 2


for (i=0; i<k; ++i) {
    for v in V {
        sum = 0;
        for u in inNeighbors(v) {
            sum += pagerank[CURR][u] / out_degree(u);
        }
        pagerank[NEXT][v] = 0.15 + 0.85 x sum;
    }
}
```

parallelize

PageRank




```
#define CURR i % 2
#define NEXT (i+1) % 2


for (i=0; i<k; ++i) {
    for v in V {
        sum = 0;
        for u in inNeighbors(v) {
            sum += pagerank[CURR][u] / out_degree(u);
        }
        pagerank[NEXT][v] = 0.15 + 0.85 x sum;
    }
}
```

shared data?

separate read and write locations

PageRank

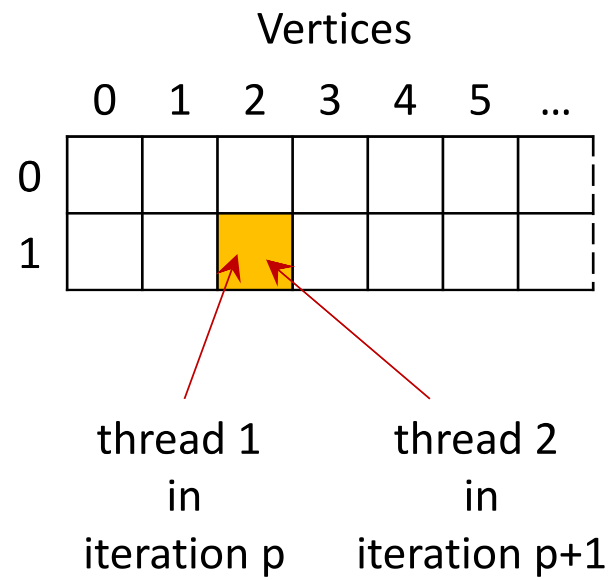
Vertices

	0	1	2	3	4	5	…
0							
1							

```
#define CURR i % 2
#define NEXT (i+1) % 2

create T threads
for each thread in parallel {
    for (i=0; i<k; ++i) {
        for v in subset(V, tid) {
            sum = 0;
            for u in inNeighbors(v) {
                sum += pagerank[CURR][u] / out_degree(u);
            }
            pagerank[NEXT][v] = 0.15 + 0.85 x sum;
        }
    }
}
```

read and write locations still different?

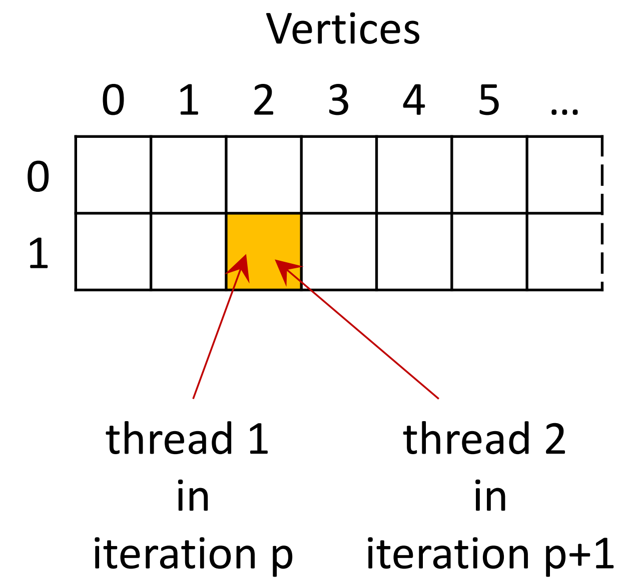
PageRank




```
#define CURR i % 2
#define NEXT (i+1) % 2

create T threads
for each thread in parallel {
    for (i=0; i<k; ++i) {
        for v in subset(V, tid) {
            sum = 0;
            for u in inNeighbors(v) {
                sum += pagerank[CURR][u] / out_degree(u);
            }
            pagerank[NEXT][v] = 0.15 + 0.85 x sum;
        }
    }
}
```

read and write locations still different?

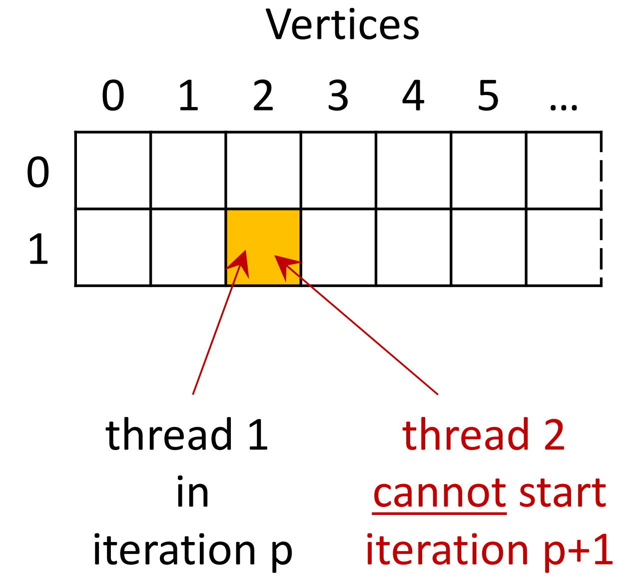
PageRank




```
#define CURR i % 2
#define NEXT (i+1) % 2

create T threads
for each thread in parallel {
    for (i=0; i<k; ++i) {
        for v in subset(V, tid) {
            sum = 0;
            for u in inNeighbors(v) {
                lock();
                sum += pagerank[CURR][u] / out_degree(u);
                unlock();
            }
            lock();
            pagerank[NEXT][v] = 0.15 + 0.85 x sum;
            unlock();
        }
    }
}
```

locks don’t control thread progress

PageRank




```
#define CURR i % 2
#define NEXT (i+1) % 2

create T threads
for each thread in parallel {
    for (i=0; i<k; ++i) {
        for v in subset(V, tid) {
            sum = 0;
            for u in inNeighbors(v) {
                sum += pagerank[CURR][u] / out_degree(u);
            }
            pagerank[NEXT][v] = 0.15 + 0.85 x sum;
        }
        barrier();
    }
}
```

blocks threads until all threads arrive here

Barrier: https://en.wikipedia.org/wiki/Barrier_(computer_science)

Dijkstra's SSSP Algorithm

```
pq.enqueue(<0, source>);   // Priority Queue
while(!pq.empty()) {
    u = pq.dequeue().second;
    for v in outNeighbors(u) {
        if(d[u] + w(u, v) < d(v)) {
            d[v] = d[u] + w(u, v);
            pq.enqueue(<d[v], v>);
        }
    }
}
```

parallelize

Dijkstra's SSSP Algorithm

```
pq.enqueue(<0, source>);   // Priority Queue
while(!pq.empty()) {
    u = pq.dequeue().second;
    for v in outNeighbors(u) {
        if(d[u] + w(u, v) < d(v)) {
            d[v] = d[u] + w(u, v);
            pq.enqueue(<d[v], v>);
        }
    }
}
```

shared data

Dijkstra’s SSSP Algorithm

Note: concurrent data structures without locks later in this course

```
pq.enqueue(<0, source>);
while(!pq.empty()) {
    u = pq.dequeue().second;
    for v in outNeighbors(u) {
        if(d[u] + w(u, v) < d(v)) {
            d[v] = d[u] + w(u, v);
            pq.enqueue(<d[v], v>);
        }
    }
}
```

```
enqueue() {
    lock();
    . . .
    unlock();
}

dequeue() {
    lock();
    . . .
    unlock();
}

empty() {
    lock();
    . . .
    unlock();
}
```

Dijkstra’s SSSP Algorithm

Note: concurrent data structures without locks later in this course

```
pq.enqueue(<0, source>);
while(!pq.empty()) {
    u = pq.dequeue().second;
    for v in outNeighbors(u) {
        if(d[u] + w(u, v) < d(v)) {
            d[v] = d[u] + w(u, v);
            pq.enqueue(<d[v], v>);
        }
    }
}
```

shared data

- Violating “order” is okay for SSSP because intermediate values are not necessarily incorrect
- However, no “update” should be missed
- d[·] values will get refined and corrected out
- Hence, don’t need to order threads (i.e., no barrier)
- This is different from PageRank

```
enqueue() {
    lock();
    . . .
    unlock();
}

dequeue() {
    lock();
    . . .
    unlock();
}

empty() {
    lock();
    . . .
    unlock();
}
```

Dijkstra’s SSSP Algorithm

```
pq.enqueue(<0, source>);
create T threads
for each thread in parallel {
    while(!pq.empty()) {
        u = pq.dequeue().second;
        if(u == NULL) continue;
        for v in outNeighbors(u) {
            if(atomic_min(d[v],
                          d[u] + w(u, v))) {
                pq.enqueue(<d[v], v>);
            }
        }
    }
}
```

to safeguard against race

- returns true if d[v] got updated as d[u] + w(u, v)
- returns false if d[v] already has value <= d[u] + w(u, v)
- can be implemented using CAS (homework)

Dijkstra’s SSSP Algorithm

```
pq.enqueue(<0, source>);
create T threads
for each thread in parallel {
    while(!pq.empty()) {
        u = pq.dequeue().second;
        if(u == NULL) continue;
        for v in outNeighbors(u) {
            if(atomic_min(d[v],
                              d[u] + w(u, v))) {
                pq.enqueue(<d[v], v>);
            }
        }
    }
}
```

What about termination?

thread 1 gets pq.empty() = true, but thread 2 calls pq.enqueue() just after that

adding barrier() will not help (why?)

Dijkstra’s SSSP Algorithm

```
done = 0;
pq.enqueue(<0, source>);
create T threads
for each thread in parallel {
  while(true) {
    if(pq.empty()) {
      atomic_add(done, 1);
      while(pq.empty()) {
        if(done == T) break;
      }
      if(done == T) break;
      atomic_add(done, -1);
      continue;
    }
```

termination logic

```
    u = pq.dequeue().second;
    if(u == NULL) continue;
    for v in outNeighbors(u) {
      if(atomic_min(d[v],
                    d[u] + w(u, v))) {
        pq.enqueue(<d[v], v>);
      }
    }
  } // end of while
} // end of for-thread
```

Parallel Algorithm Design

- Identifying independent tasks is not enough
  - Expressing the independent tasks so that they collaboratively accomplish the goal can be challenging
  - Often involves restructuring algorithm
- Correctness based on algorithm needs
  - E.g., ordering threads in PageRank v/s SSSP
- Sub-components need to be made thread-safe
  - E.g., priority queue in SSSP
- Termination detection
  - Especially important when logic lacks synchrony (e.g., no barriers)
- Synchronization
  - locks, barriers
  - Atomics for simple operations are usually much faster

Homework

- Implement each parallel program and check performance
- Try to improve performance with better parallelization strategies

Reading (for Next Class)

- [AMP] Chapter 3
  - Upto 3.7

- [Paper] Linearizability: A Correctness Condition for Concurrent Objects: https://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf
  - Upto section 3

- [Paper] How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/How-to-Make-a-Multiprocessor-Computer-That-Correctly-Executes-Multiprocess-Programs.pdf