Lecture 05: Limits of Parallelism and Locks

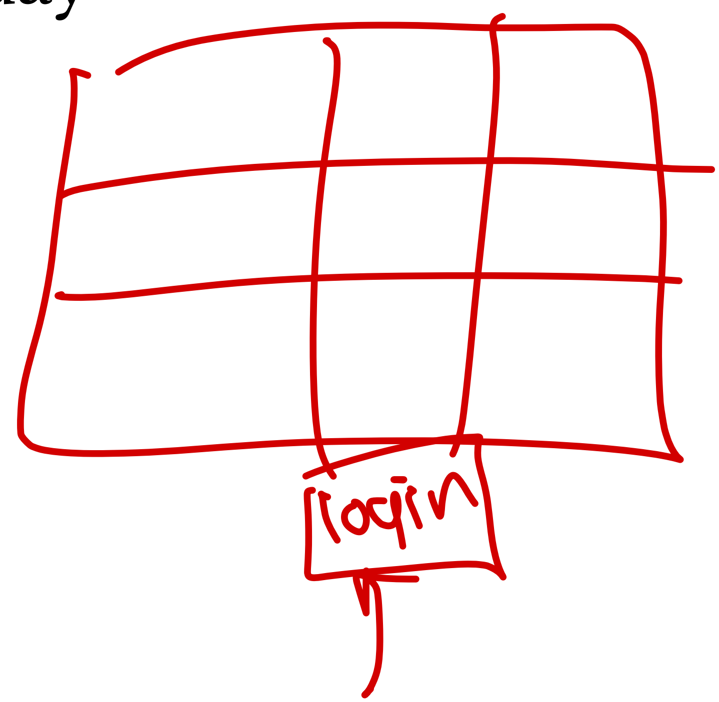
COSC 272: Parallel and Distributed Computing

Spring 2023

Announcement

1. Lab Assignment 01 Due Today
2. Written Homework 01 Posted Sunday
   - due next Friday

Don't answer cluster Q's



Outline

1. Limitations of Parallelism
2. Mutual Exclusion (locks)

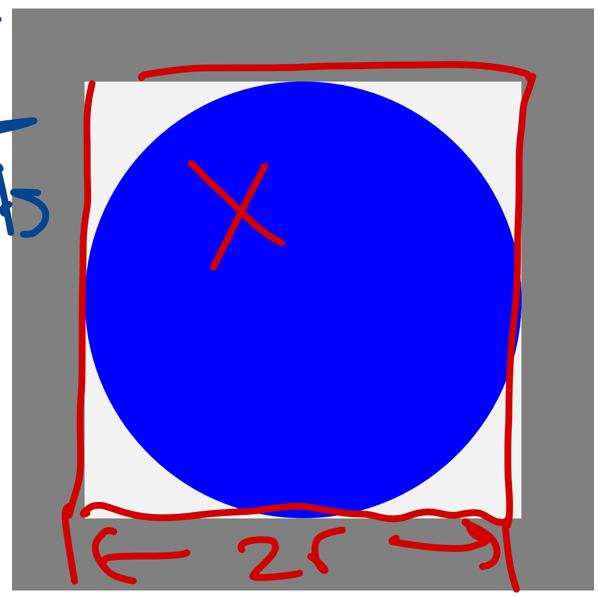
Last Time

Embarrassingly Parallel Problems

- can be broken into many **simple** computations, (almost) all of which can be performed in parallel

Example: Monte Carlo Estimation

$$\frac{\pi}{4} \approx \frac{\#\text{ darts hit circ}}{\text{total }\#\text{ darts}}$$



A square $4r^2$

Prob dart hits circle

$$= \frac{A\text{ circ}}{A\text{ sq}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Area of a disk: $A = \pi r^2$; estimate π!

Question

Why is Monte Carlo estimation embarrassingly parallel?

Want : 1,000,000 trials

k : # processes (threads)

- each performs $\frac{1M}{k}$ trials
- record # hits for each process

when done **aggregate** # hits

Another Question

How much performance increase with k cores?

T = 1 core time

expect $\frac{T}{k}$ time for k cores

ignores overhead of aggregation

Another Question

How much performance increase with k cores?

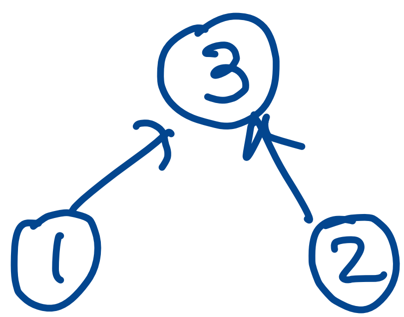
- What if $k \approx$ number of samples taken?

computation per core
is really short,
but agg. might take
much longer

Not So Parallel

Dependencies?

```
a1 = b1 + c1;
a2 = b2 - c2;
d = a1 * a2
```

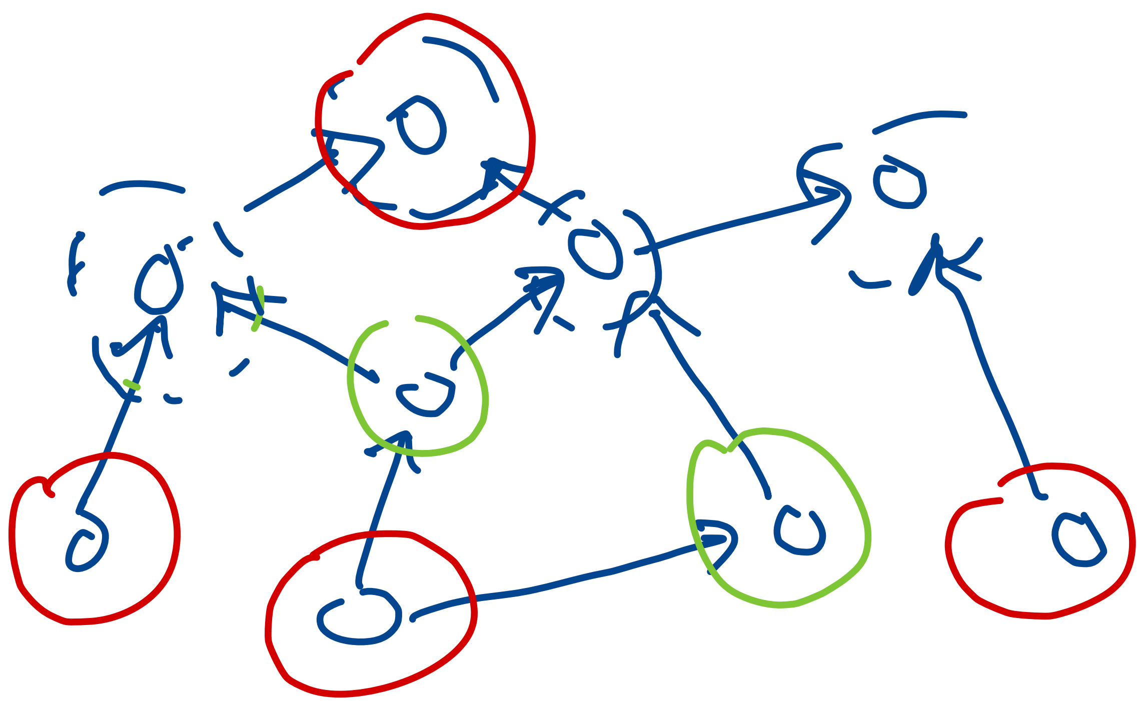


Not So Parallel

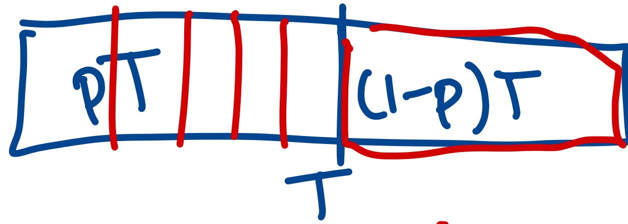
Dependencies?

```
a1 = b1 + c1;
a2 = b2 - c2;
d = a1 * a2
```

Dependency relation: directed acyclic graph (DAG)



More Generally



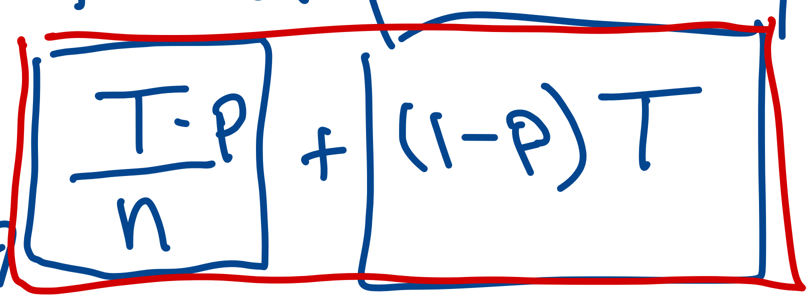
Consider a program that requires

- N elementary operations
- T time to run sequentially

Suppose

- a p-fraction of operations can be performed in parallel
- $1 - p$ fraction must be performed sequentially

Question: how long could program take with n parallel machines?



Idea

With n parallel machines:

- perform p-fraction of parallelizable ops in parallel on all n machines
  - total time $\frac{T \cdot p}{n}$
- perform remaining ops sequentially on a single machine
  - total time $T \cdot (1 - p)$

Total time: $T \cdot (1 - p) + T \cdot \frac{p}{n} = T \cdot \left(1 - p + \frac{p}{n}\right)$

How Much Improvement?

The **speedup** is the ratio of the original time T to the parallel time $T \cdot \left(1 - p + \frac{p}{n}\right)$:

- $S = \frac{1}{1-p+\frac{p}{n}}$

times faster n processor solution is than 1 processor

Theoretical upper bound

This relation is called **Amdahl's Law**

How Much Improvement?

The **speedup** is the ratio of the original time T to the parallel time $T \cdot \left(1 - p + \frac{p}{n}\right)$:

- $S = \frac{1}{1-p+\frac{p}{n}}$

This relation is called **Amdahl's Law**

This is the best performance improvement possible **in principle**

- may not be achievable in practice!

Example

1 person can chop 1 onion per minute

Recipe calls for:

- chop 6 onions ← Parallelizable
- saute onions for 4 minutes ← sequential

Note:

- chopping onions can be done in parallel
- sauteing
  - takes 4 minutes no matter what
  - must be accomplished after chopping

$T = 10$ min.

$P = \frac{6}{10} = 0.6$

Example (continued)

How much can the cooking process be sped up by n cooks?

Example (continued)

- For one chef, $T = 6 + 4 = 10$
- Only chopping onions is parallelizable, so $p = 6/10 = 0.6$
- Amdahl's Law:
  - $S = \frac{1}{1-p-\frac{p}{n}} = \frac{1}{0.4+\frac{1}{n}0.6}$
- So:
  - $n = 2 \implies S = 1.43$
  - $n = 3 \implies S = 1.67$
  - $n = 6 \implies S = 2$
- Always have $S < 1/(1-p) = 2.5$

Speedup Improvement by Adding More Processors

- Second processor: 43%
- Third processor: 17%
- Fourth processor: 9%
- Fifth processor: 6%
- Sixth processor 4%

Latency vs Number of Processors

How does latency T scale with n?

- Adding more processors has *declining marginal utility*:
  - each additional processor has a smaller effect on total performance
  - at some point, adding more processors to a computation is wasteful
- Another consideration:
  - after parallel ops have been performed, extra processors are idle (potentially wasteful!)

Remarks

The proportion of parallelizable operations p is not always obvious from problem statement

Remarks

The proportion of parallelizable operations p is not always obvious from problem statement

- Amdahl's law a valuable heuristic for general phenomena:
  1. an n-fold increase in parallel processing power does not typically give an n-fold speedup in computations
  2. adding new parallel processors becomes less helpful the more parallel processors you already have

Remarks

The proportion of parallelizable operations p is not always obvious from problem statement

- Amdahl's law a valuable heuristic for general phenomena:
  1. an n-fold increase in parallel processing power does not typically give an n-fold speedup in computations
  2. adding new parallel processors becomes less helpful the more parallel processors you already have
- Often helpful to think about scheduling subtasks (not individual operations)
- May have relationships between tasks (e.g., one must be performed before another)

Locks

Back to `Counter` Example

The problem with

```
public void increment () {
    ++count;
}
```

The operation `++count` is **not** atomic

- consists of:
  1. `read` count value
  2. increment value in register
  3. `write` updated value
- these operations can be *interleaved* for concurrent executions

A Strategy

Fix the issue by *locking* the `count`

To increment the `Counter`:

1. check if `Counter` is locked
   - if so, wait until it is unlocked
2. lock the `Counter`
   - no other thread can modify while locked
3. increment the counter
4. unlock the `Counter`

An Attempt

```
public class LockedCounter {
    long count = 0;
    boolean locked = false;
    public long getCount () { return count; }
    public void increment () { count++; }
    public void reset () { count = 0; }
    public void lock (int id) {
        while (locked) { }
        locked = true;
    }
    public void unlock () { locked = false; }
    public boolean isLocked () { return locked; }
```

Running the Locked Counter

```
public void run () {
    for (long i = 0; i < times; i++) {
        counter.lock(id);
        try {
            counter.increment();
        }
        finally {
            counter.unlock();
        }

    }
```

Will It Work?

`LockedCounterTester`

Demo!

Question

What happened? Can we make the locked counter idea work?

Morals

1. Empirical testing is not enough!
2. Must understand correctness **formally**

Next Week

Two threads:

- Mutual Exclusion
- Locality of Reference