CS260 – Lecture 9*

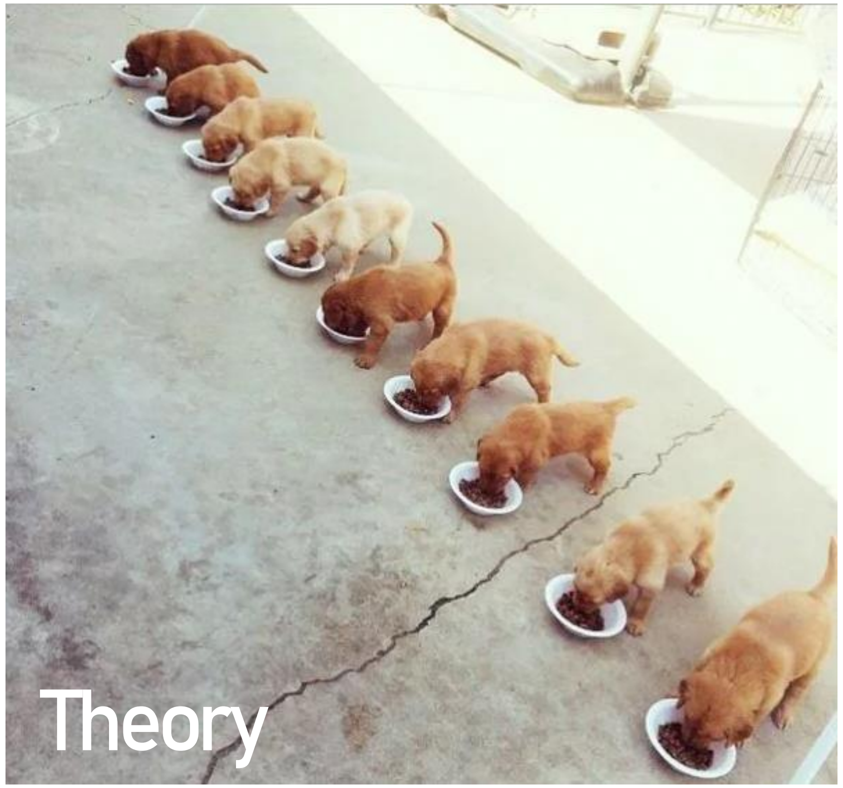
Yan Gu

Parallel Algorithms: Theory and Practice

Race

Why is parallelism "hard"?

Non-determinism!!



Why is parallelism "hard"?

Non-determinism!!

- Scheduling is unknown
- Relative ordering for operations is unknown

- Hard to debug
  - Bugs can be **non-deterministic!**
  - Bugs can be different if you rerun the code
  - Referred to as race hazard / condition

Race hazard can cause severe consequences

- Therac-25 radiation therapy machine — killed 3 people and seriously injured many more (between 1985 and 1987). https://en.wikipedia.org/wiki/Therac-25

- North American Blackout of 2003 — left 50 million people without power for up to a week. https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

- Race bugs are notoriously difficult to discover by conventional testing!

Determinacy Races

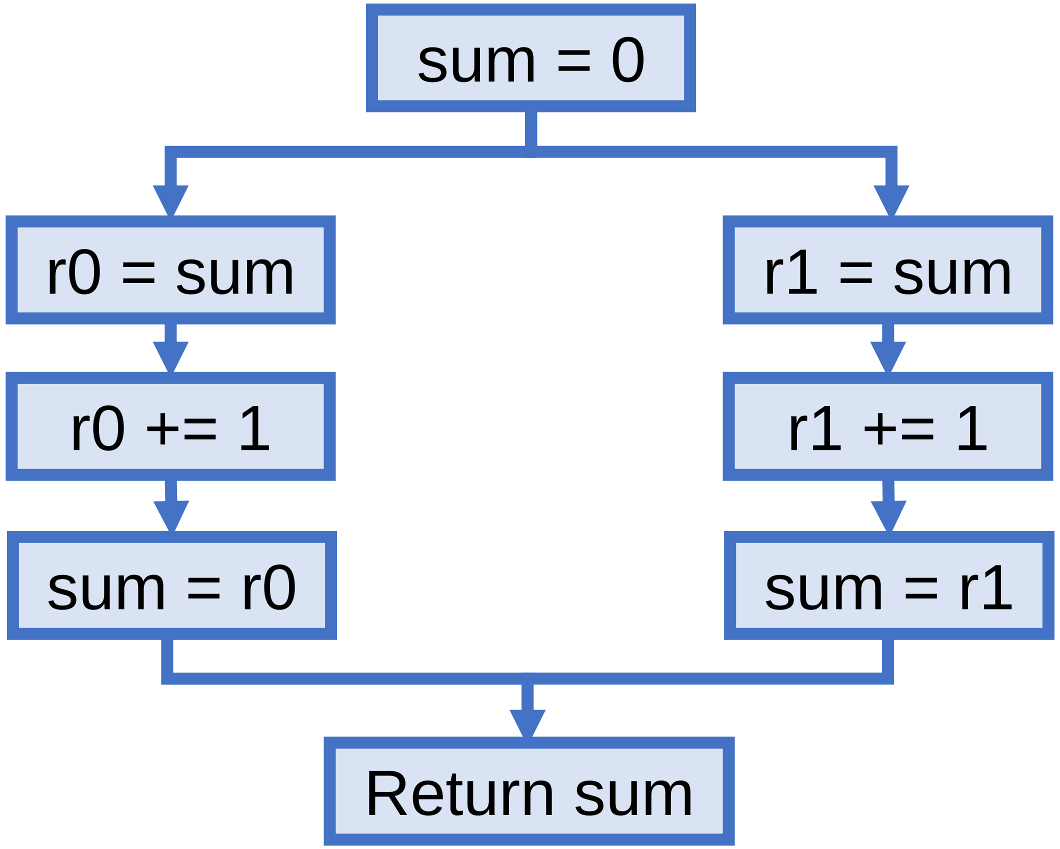
- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

```
direct_reduce(A, n) {
  parallel_for (i=0;i<n;i++)
    sum = sum + 1;
  return sum;
}
```

Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

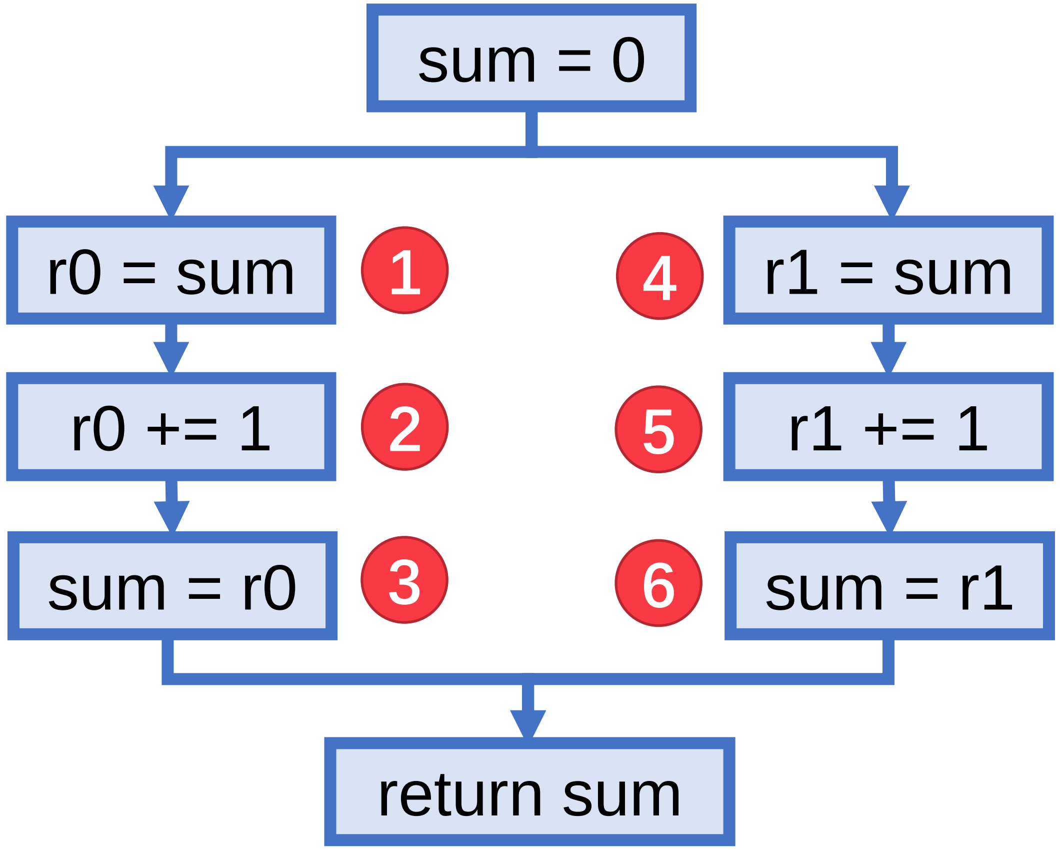
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + 1;
  return sum;
}
```



Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

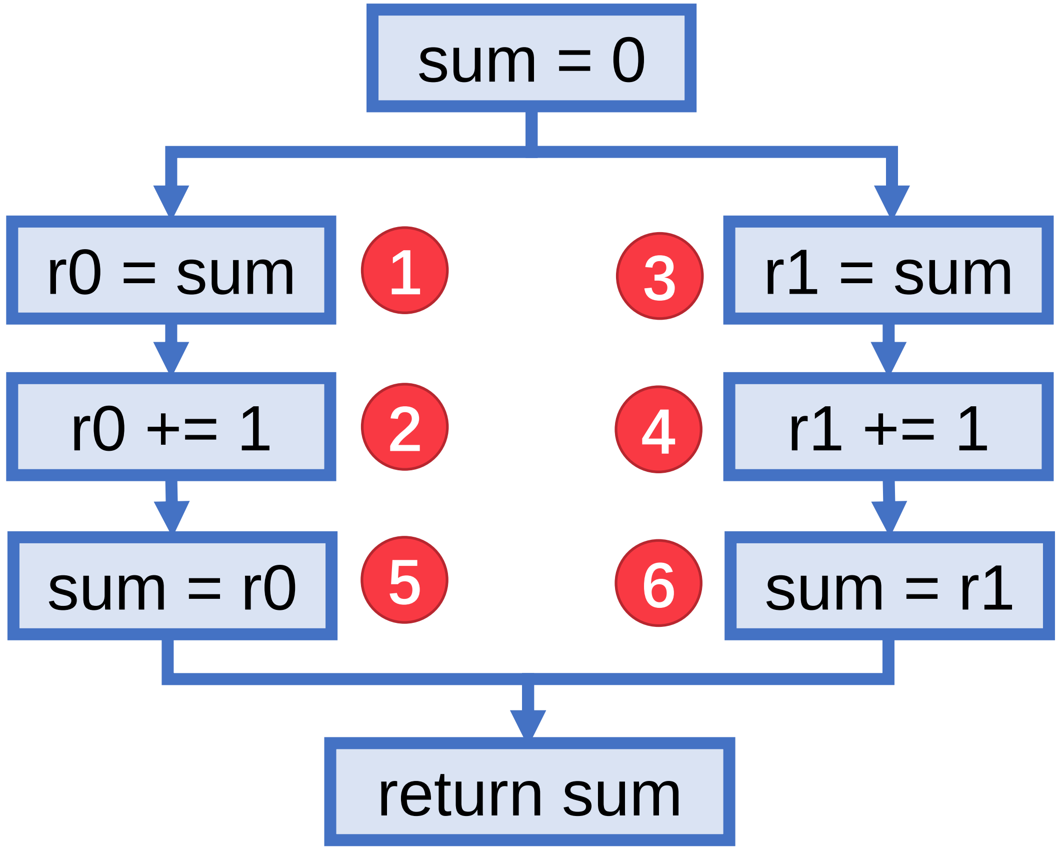
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + 1;
  return sum;
}
```



Determinacy Races

- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + 1;
  return sum;
}
```



Types of Races

- Suppose that instruction A and instruction B both access a location x, and suppose that A∥B (A is parallel to B).

A	B	Race Type
Read	Read	No race
Read	Write	Read race
Write	Read	Read race
Write	Write	Write race

- Two sections of code are **independent** if they have no determinacy races between them.

Avoiding races

- Iterations of a parallel_for loop should be independent
- Between two in_parallel tasks, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children

Benefit of being race-free

- Scheduling is still unknown
- Relative ordering for operations is still unknown

- However, the computed value of each instruction is deterministic! This is easy to debug.
  - Check the correctness of the sequential execution
  - Check if the parallel execution is the same as the sequential one

- Race detection: given a DAG, show all the races
- False sharing: nasty related effect
  - E.g., updating x.a and x.b in parallel is safe but can be inefficient

```
Struct {
  char a, b;
} x;
```