Parallel Algorithms: Theory and Practice

CS260 – Lecture 5
Yan Gu

Deterministic Parallelism

Last week – Sorting algorithms

- **Parallel quicksort**
  - Key: partition elements based on the pivot in parallel
  - Parallel filtering/packing algorithm – $O(n)$ work and $O(\log n)$ depth
  - $O(n \log n)$ work and $O(\log^2 n)$ depth

- **Parallel mergesort**
  - Key: merge two sorted arrays into another sorted array in parallel
  - Parallel merging algorithm – $O(n)$ work and $O(\log n)$ depth
  - $O(n \log n)$ work and $O(\log^2 n)$ depth

Last week – Sorting algorithms

- **Parallel selection sort**
  - $O(\log n)$ depth but $O(n^2)$ work

- **List ranking – random mate**
  - Determine in a linked list, the rank of each node
  - Using randomization to filter out (on expectation) ¼ nodes in each round
  - Reduce problem size and recursively apply the algorithm
  - Expand the list back and restore the information

Parallel Algorithms: Theory and Practice

CS260 – Lecture 5

Yan Gu

Deterministic Parallelism

CS260: Parallel algorithms Lecture 5

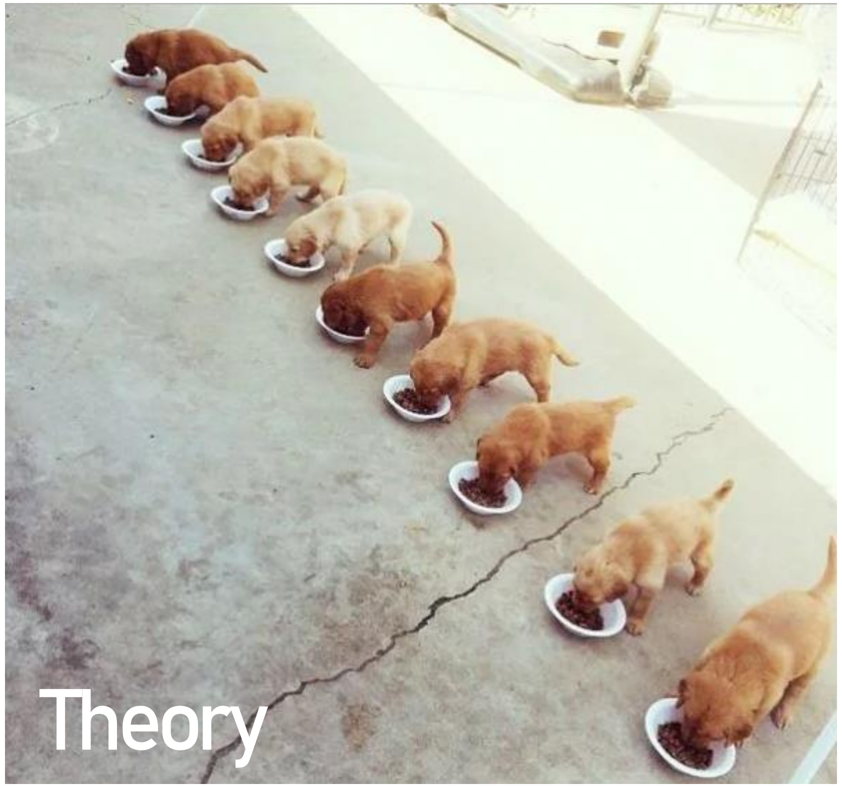
Race

Deterministic Parallelism

Some materials are from 6.172 Performance Engineering of Software Systems, credits to Charles Leiserson

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!

Race

Determinacy Races

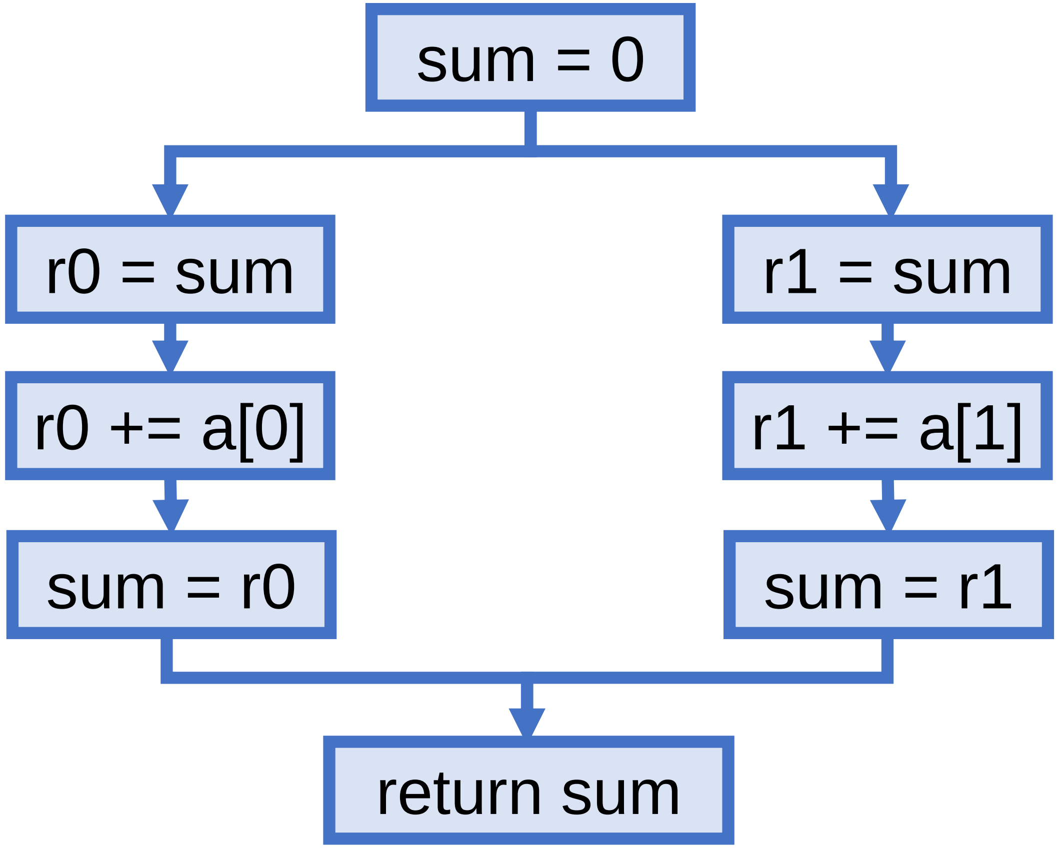
- Definition: a **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

```
direct_reduce(A, n) {
  parallel_for (i=0;i<n;i++)
    sum = sum + a[i];
  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.

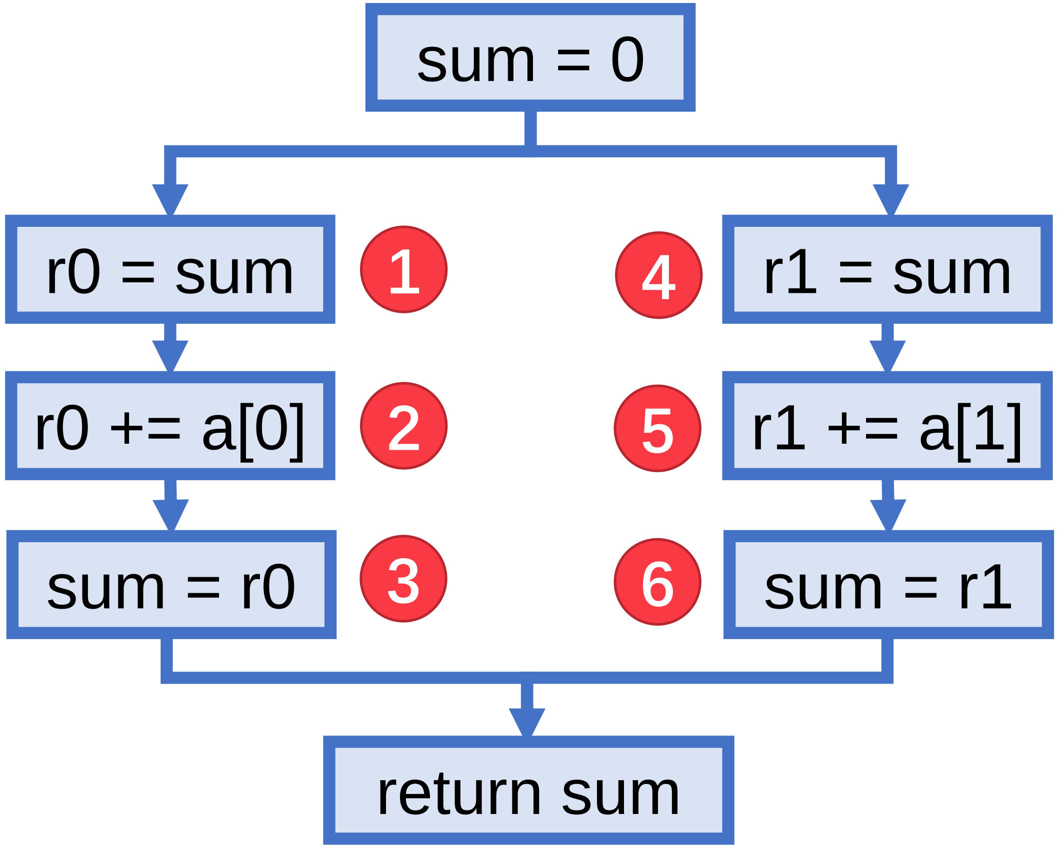
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + a[i];
  return sum;
}
```



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.

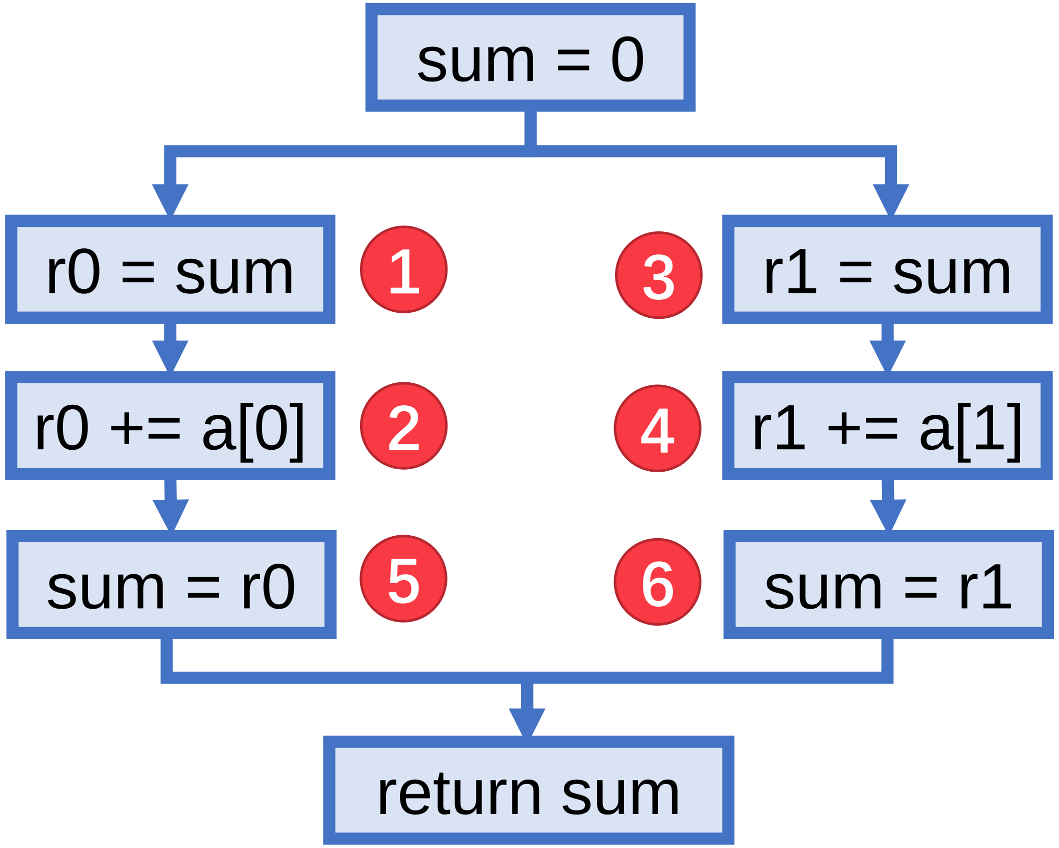
```
direct_reduce(A, n) {
  parallel_for (i=0;i<2;i++)
    sum = sum + a[i];
  return sum;
}
```



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 + a[i];
  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 two calls should be independent, including code executed by further in_parallel tasks

```
reduce(A, n) {
    if (n == 1) return A[0];
    In parallel:
        L = reduce(A, n/2);
        R = reduce(A + n/2, n-n/2);
    return L+R;
}
```

Avoiding races

- Iterations of a parallel_for loop should be independent
- Between two in_parallel tasks, the code of the two calls should be independent, including code executed by further in_parallel tasks

```
reduce(A, n) {
  if (n == 1) return A[0];
  if (n is odd) n=n+1;
  parallel_for i=1 to n/2
    B[i]=A[2i]+A[2i+1];
  return reduce(B, n/2);
}
```

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;
```

This is not the end...

- Consider a hash table
- A key-value pair is inserted to a random location based on the key
- No guarantee that no two keys will not be inserted to the same location

Lock-based solution (critical section)

- **Lock the memory location for each write**

- **A correct solution**

- **Very poor performance**
  - No guarantee for execute order
  - Bad scalability (worse performance for more cores)
  - Risk of no progress

- **Need better solutions**

```
direct_reduce(A, n) {
  parallel_for (i=0;i<n;i++) {
    getLock(&sum);
    sum = sum + a[i];
    releaseLock(&sum);
  }
  return sum;
}
```

Atomic primitives (Lecture 2)

- **Compare-and-swap (CAS):**
  - bool CAS(value* p, value vold, value vnew)
  - Compare the value stored in the pointer p with value *vold*, if they are equal, try to change p's value to *vnew*. If successful, return true. Otherwise, return false.
- **Test-and-set (TAS):**
  - bool TAS(bool* p)
  - Determine if the Boolean value stored at p is false, if so, try to set it to true. If successful, return true. Otherwise, return false.
- **Fetch-and-add (FAA):**
  - integer FAA(integer* p)
  - Add integer p's value by 1, and return the old value

Atomic primitives (Lecture 2)

- Use CAS to implement reduce
- Relatively better performance
  - Guarantee to proceed
  - Implemented by hardware (relatively faster, bad in this case)
- Main challenge:

```
direct_reduce(A, n) {
  parallel_for (i=0;i<n;i++) {
    old = sum;
    while (!CAS(&sum, old, old+a[i]))
      old = sum;
    }
  return sum;
}
```

Implementations are racy, still hard to debug!

Deterministic Parallelism

High-level idea

- Some additional restrictions, but weaker than race-free
- A parallel algorithm can be racy, but the parallel execution must match the sequential execution
- When debugging:
  - First guarantee the sequential execution is correct
  - Then check if the parallel execution is the same
    - E.g., printing out all intermediate states

Random Permutation

a	b	c	d	e	f	g	h

Random Permutation

f	a	e	g	h	c	d	b

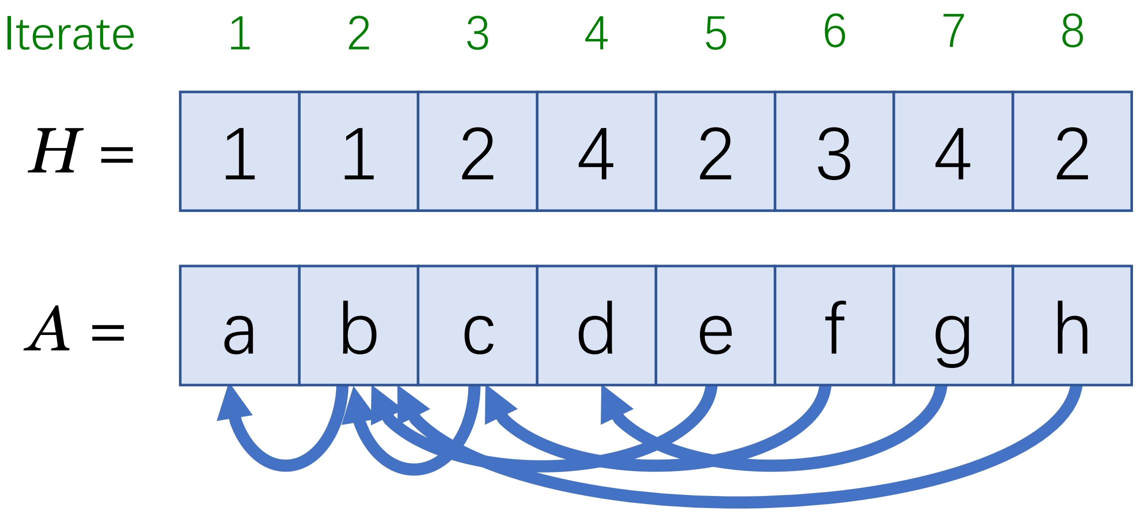
- Generating random permutation is a fundamental building block in parallel algorithms

- But for decades, we don’t know how to randomly permute elements in parallel efficiently both theoretically and pratically

Sequential Random Permutation [Durstenfeld64, Knuth69]

KNUTHSHUFFLE(A, H)
  for $i \leftarrow n$ to 1 **do**
    swap($A[H[i]]$, $A[i]$)

$H[i]$ is randomly drawn between 1 and i



Sequential Random Permutation [Durstenfeld64, Knuth69]

```
KnuthShuffle(A, H)
  for i ← n to 1 do
    swap(A[H[i]], A[i])
```

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	b	c	d	e	f	g	h

Sequential Random Permutation [Durstenfeld64, Knuth69]

```
KnuthShuffle(A, H)
  for i ← n to 1 do
    swap(A[H[i]], A[i])
```

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	c	d	e	f	g	b

Sequential Random Permutation [Durstenfeld64, Knuth69]

KnuthShuffle(A, H)
for $i \leftarrow n$ to 1 **do**
swap($A[H[i]]$, $A[i]$)

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	c	d	e	f	g	b

Sequential Random Permutation [Durstenfeld64, Knuth69]

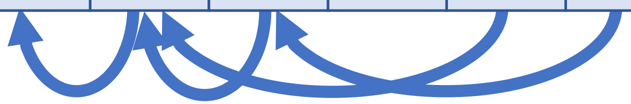
KnuthShuffle(A, H)
for $i \leftarrow n$ to 1 **do**
swap($A[H[i]]$, $A[i]$)

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	c	g	e	f	d	b

Sequential Random Permutation [Durstenfeld64, Knuth69]

```
KnuthShuffle(A, H)
  for i ← n to 1 do
    swap(A[H[i]], A[i])
```

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	c	g	e	f	d	b



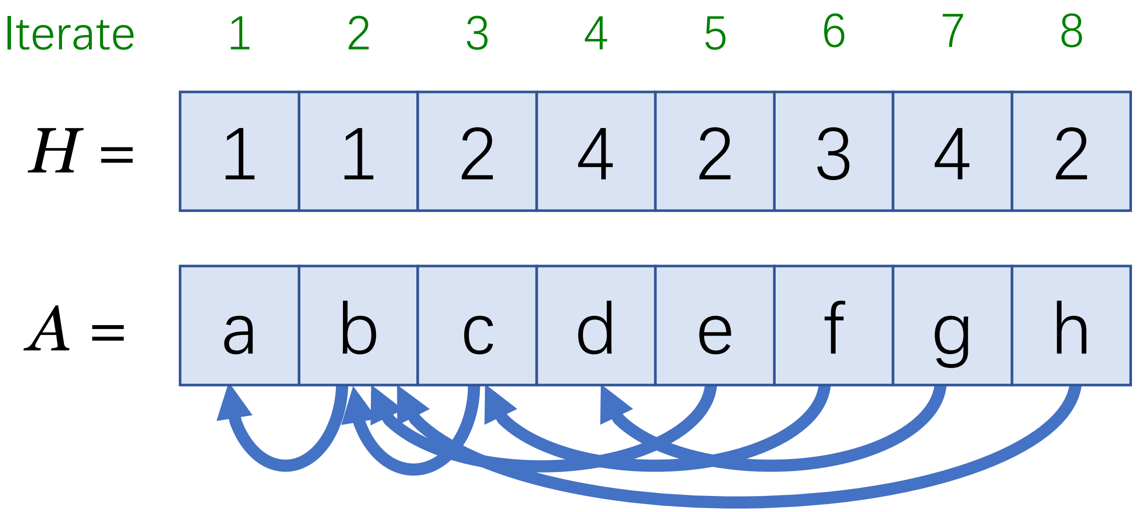
Sequential Random Permutation [Durstenfeld64, Knuth69]

```
KnuthShuffle(A, H)
  for i ← n to 1 do
    swap(A[H[i]], A[i])
```

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	f	a	e	g	h	c	d	b

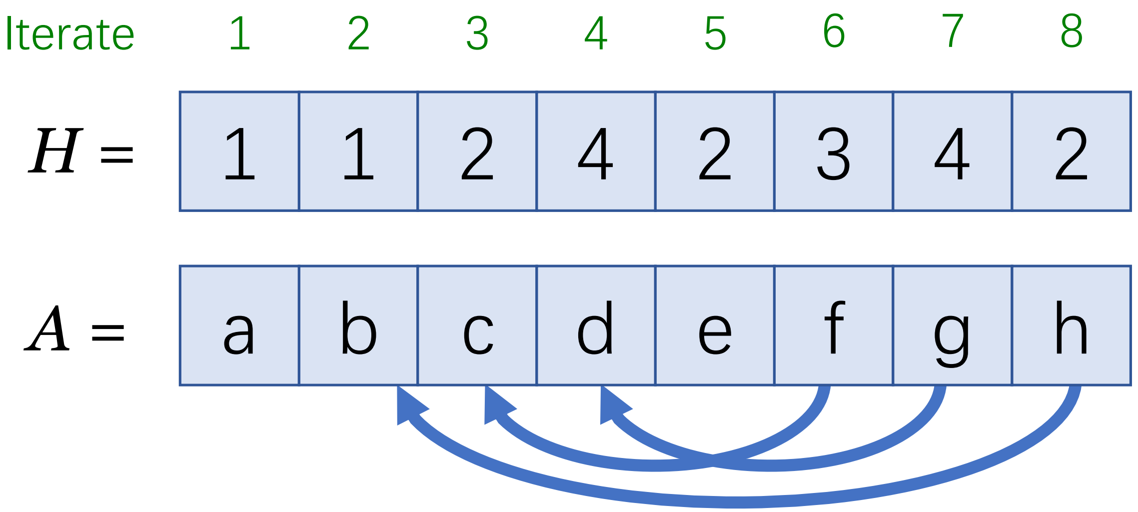
Can this simple sequential algorithm be parallelized?

KnuthShuffle(A, H)
for $i \leftarrow n$ to 1 **do**
swap($A[H[i]]$, $A[i]$)



Can this simple sequential algorithm be parallelized?

KNUTHSHUFFLE(A, H)
for $i \leftarrow n$ to 1 **do**
swap($A[H[i]], A[i]$)



Can this simple sequential algorithm be parallelized?

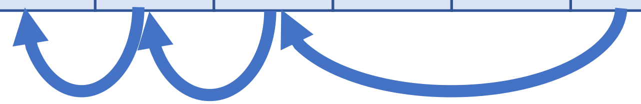
```
KnuthShuffle(A, H)
  for i ← n to 1 do
    swap(A[H[i]], A[i])
```

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	g	f	e	c	d	b

Which swaps **cannot** run in parallel?

KnuthShuffle(A, H)
  for $i \leftarrow n$ to 1 **do**
    swap($A[H[i]]$, $A[i]$)

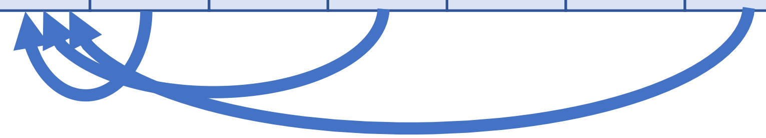
Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	b	c	d	e	f	g	h



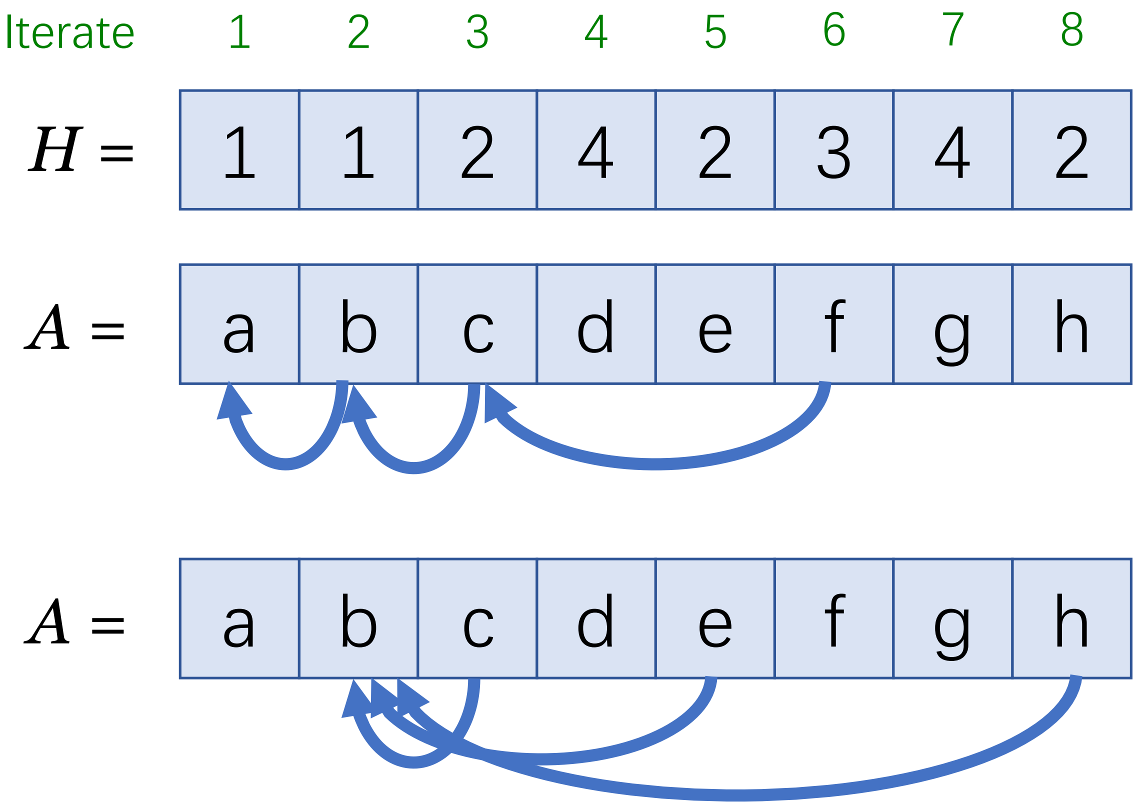
Which swaps **cannot** run in parallel?

KnuthShuffle(A, H)
  for $i \leftarrow n$ to 1 **do**
    swap($A[H[i]]$, $A[i]$)

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	b	c	d	e	f	g	h

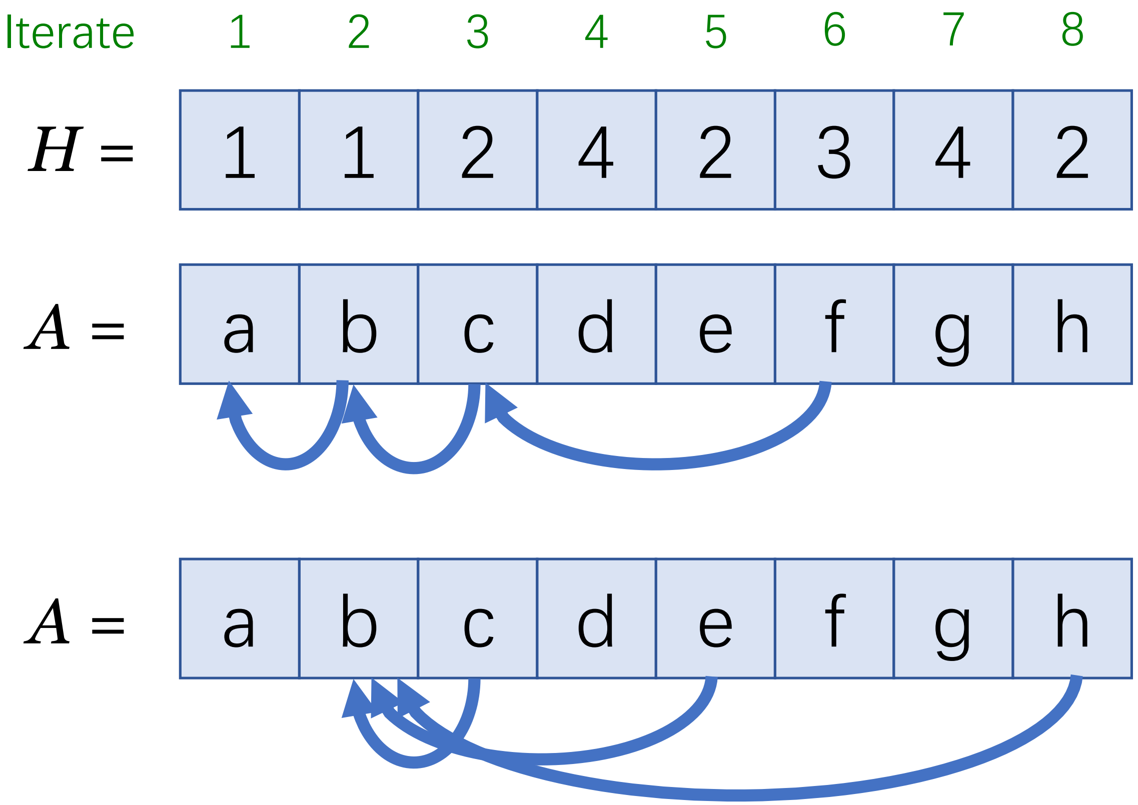


A simple parallel algorithm



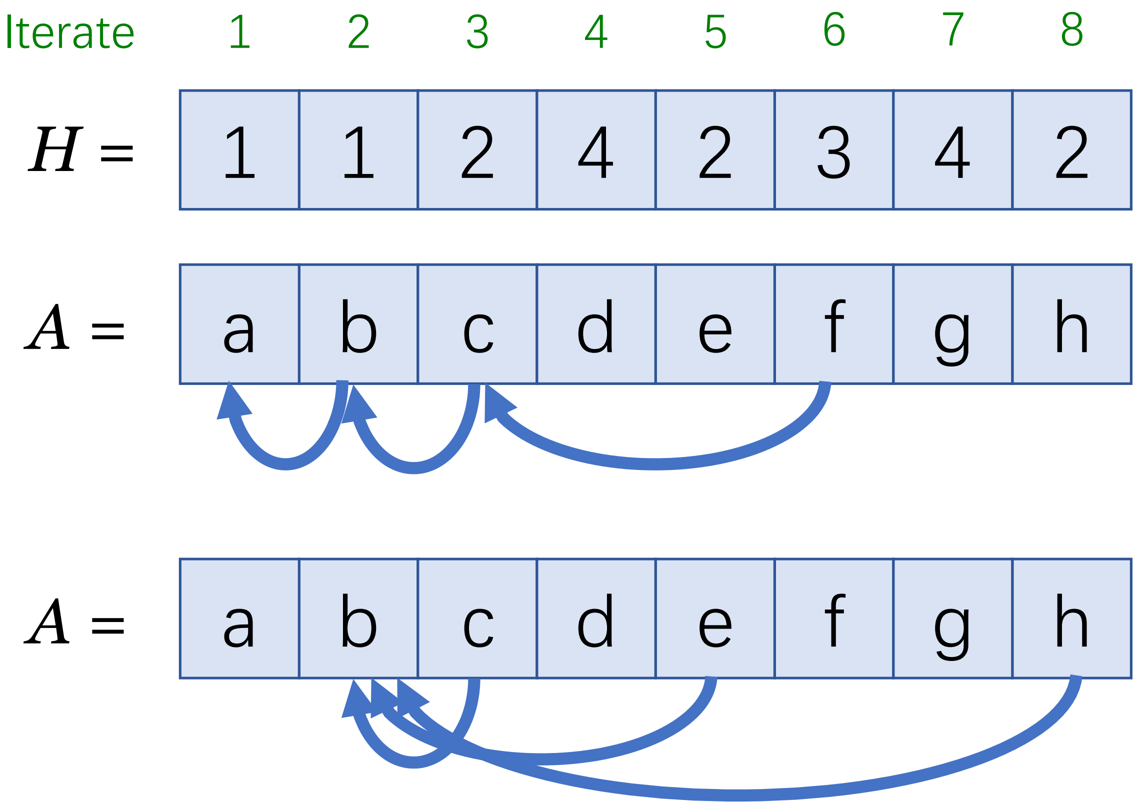

```
while swaps unfinished do
  par-for each swap (i, H[i]) do
    if no other swaps to i and
        i is the last swap to H[i]
      process the swap
  pack the unfinished swaps
```

A simple parallel algorithm



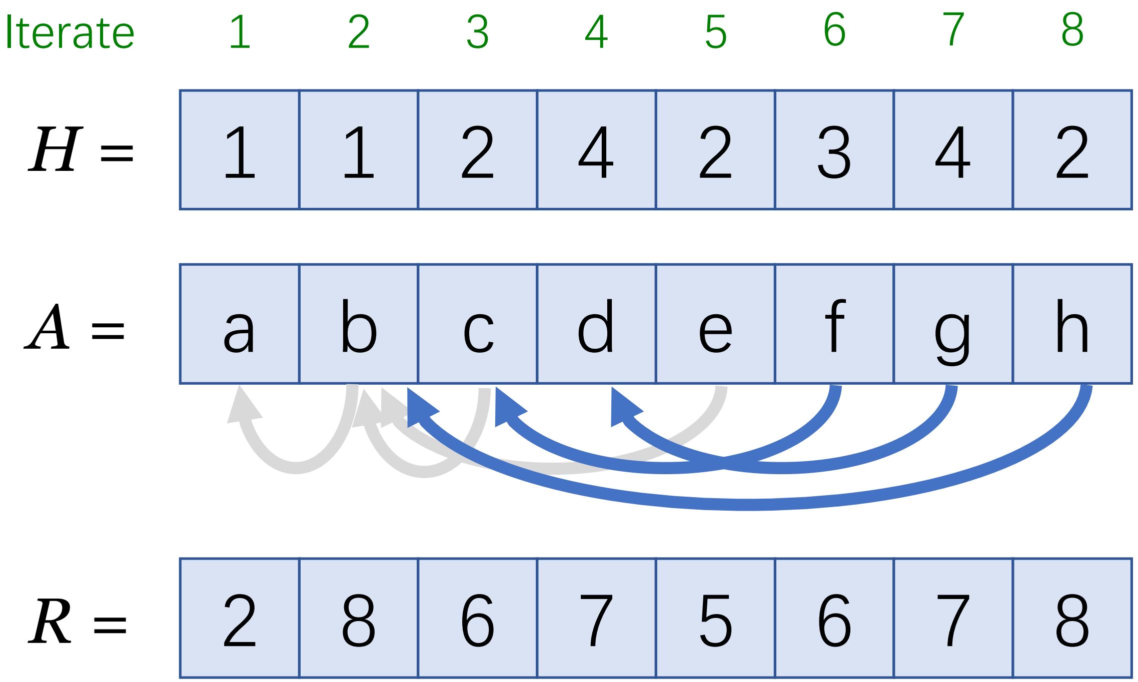
while swaps unfinished **do**
  par-for each swap (i, $H[i]$) **do**
    if no other swaps to i **and**
      i is the last swap to $H[i]$
      process the swap
  pack the unfinished swaps

A simple parallel algorithm



while swaps unfinished **do**
  parafor each swap $(i, H[i])$ **do**
    $R[i] \leftarrow \max(R[i], i)$
    $R[H[i]] \leftarrow \max(R[H[i]], i)$
  parafor each swap $(i, H[i])$ **do**
    if $R[i] = i$ and $R[H[i]] = i$
      $\text{swap}(A[H[i]], A[i])$
  pack the swaps

The first round



while swaps unfinished **do**
  parafor each swap $(i, H[i])$ **do**
    $R[i] \leftarrow \max(R[i], i)$
    $R[H[i]] \leftarrow \max(R[H[i]], i)$
  parafor each swap $(i, H[i])$ **do**
    if $R[i] = i$ and $R[H[i]] = i$
      swap$(A[H[i]], A[i])$
  pack the swaps

The first round

Iterate	1	2	3	4	5	6	7	8
H =	1	1	2	4	2	3	4	2
A =	a	h	g	f	e	d	c	b

while swaps unfinished **do**

  parafor each swap $(i, H[i])$ **do**

    $R[i] \leftarrow \max(R[i], i)$

    $R[H[i]] \leftarrow \max(R[H[i]], i)$

  parafor each swap $(i, H[i])$ **do**

    if $R[i] = i$ and $R[H[i]] = i$

      swap$(A[H[i]], A[i])$

  pack the swaps

Example of Deterministic Parallelism

- Not race-free (atomic updates needed)

- Relative ordering of the swaps is consistent with sequential execution
  - When debugging, first check the sequential execution, then check if the destinations of the swaps are the same in the parallel execution

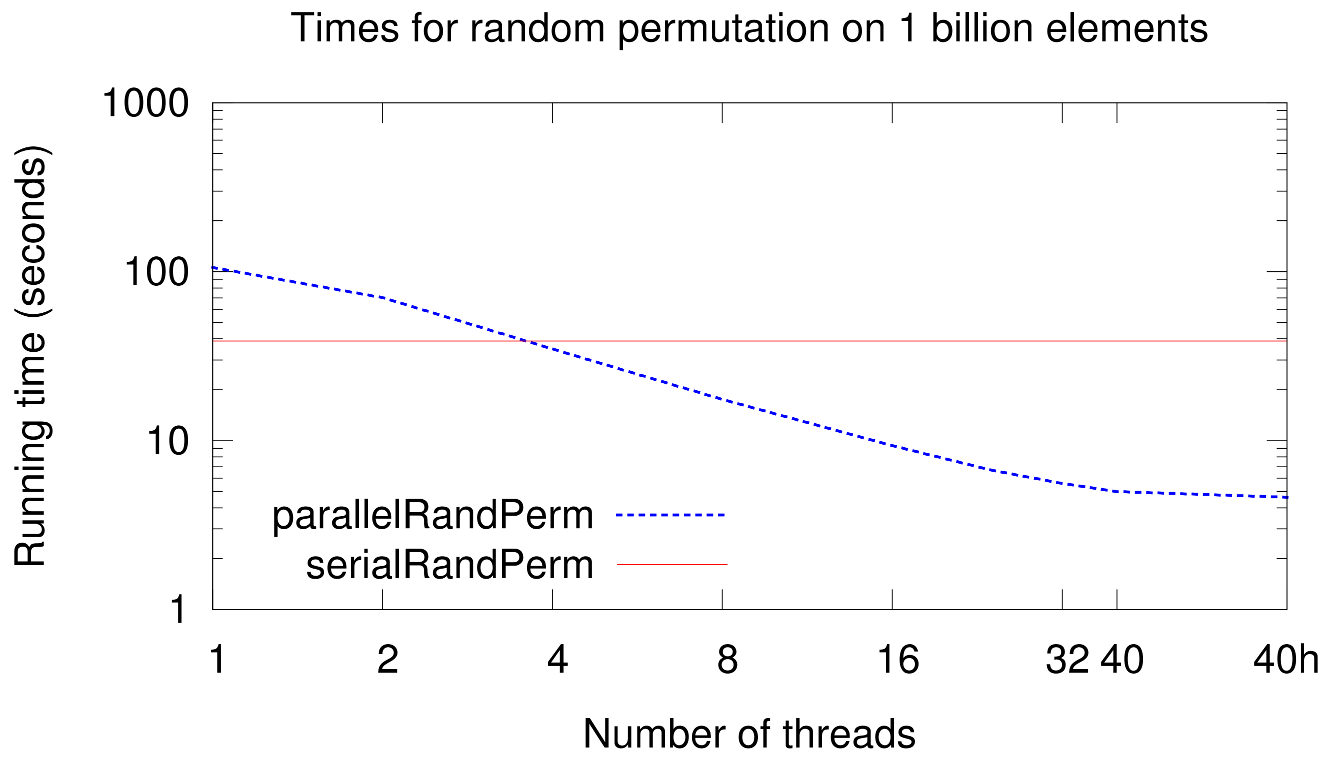
- Execution is "deterministic"
  - Output is always the same for different executions
  - Input/output of each operation is always the same for different executions

- Determinism is supported by "priority updates"

Work-depth analysis for random permutation

- The number of rounds **is $\Theta(\log n)$** w.h.p.
  - Very simple proof
- This algorithm uses $O(n)$ **work** and $O(\log n)$ **span** w.h.p., and is **optimal** under certain assumptions
- Good performance in practice

Good practical performance



- **Good performance in practice**
  - Can outperform the sequential algorithm on 4 cores
  - 8.5x faster than sequential on 40 cores
  - Almost perfect self-relative speedup (35-40x)

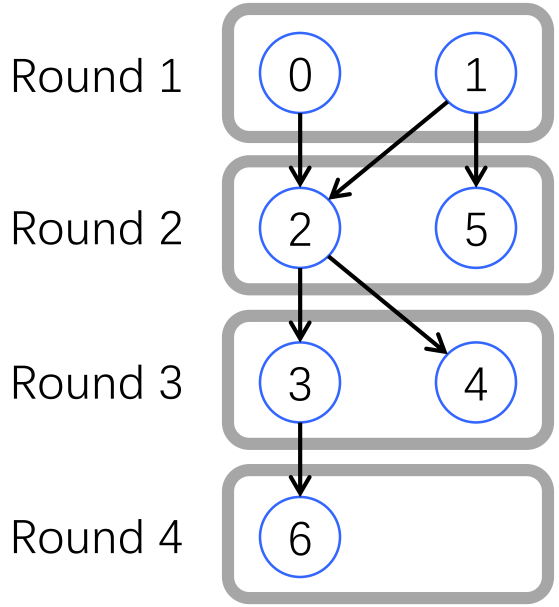
Many sequential iterative algorithms are already parallel

- Random permutation (Knuth shuffle) [SODA15, manuscript]
- List contraction [SODA15, manuscript]
- Tree contraction [SODA15, manuscript]
- Comparison sort [SPAA16a]
- Incremental convex hull [SPAA16a]
- Incremental Delaunay triangulation [SPAA16a]
- Strongly connected component [SPAA16a]
- Least-element lists [SPAA16a]

0
1
2
3
4
5
6

Many sequential iterative algorithms are already parallel

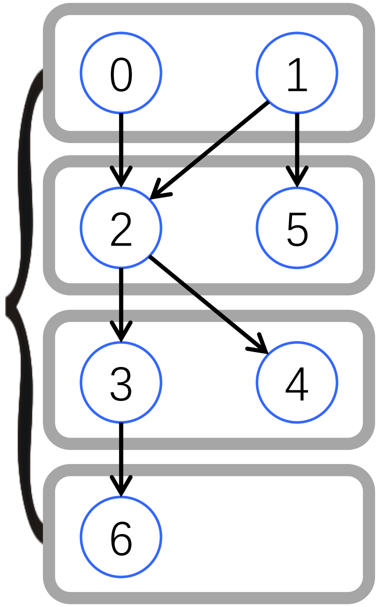
- Random permutation (Knuth shuffle) [SODA15, manuscript]
- List contraction [SODA15, manuscript]
- Tree contraction [SODA15, manuscript]
- Comparison sort [SPAA16a]
- Incremental convex hull [SPAA16a]
- Incremental Delaunay triangulation [SPAA16a]
- Strongly connected component [SPAA16a]
- Least-element lists [SPAA16a]



Many sequential iterative algorithms are already parallel

- Random permutation (Knuth shuffle) [SODA15, manuscript]
- List contraction [SODA15, manuscript]
- Tree contraction [SODA15, manuscript]
- Comparison sort [SPAA16a]
- Incremental convex hull [SPAA16a]
- Incremental Delaunay triangulation [SPA
- Strongly connected component [SPAA16a]
- Least-element lists [SPAA16a]

$O(\log n)$ rounds w.h.p. for all these problems!



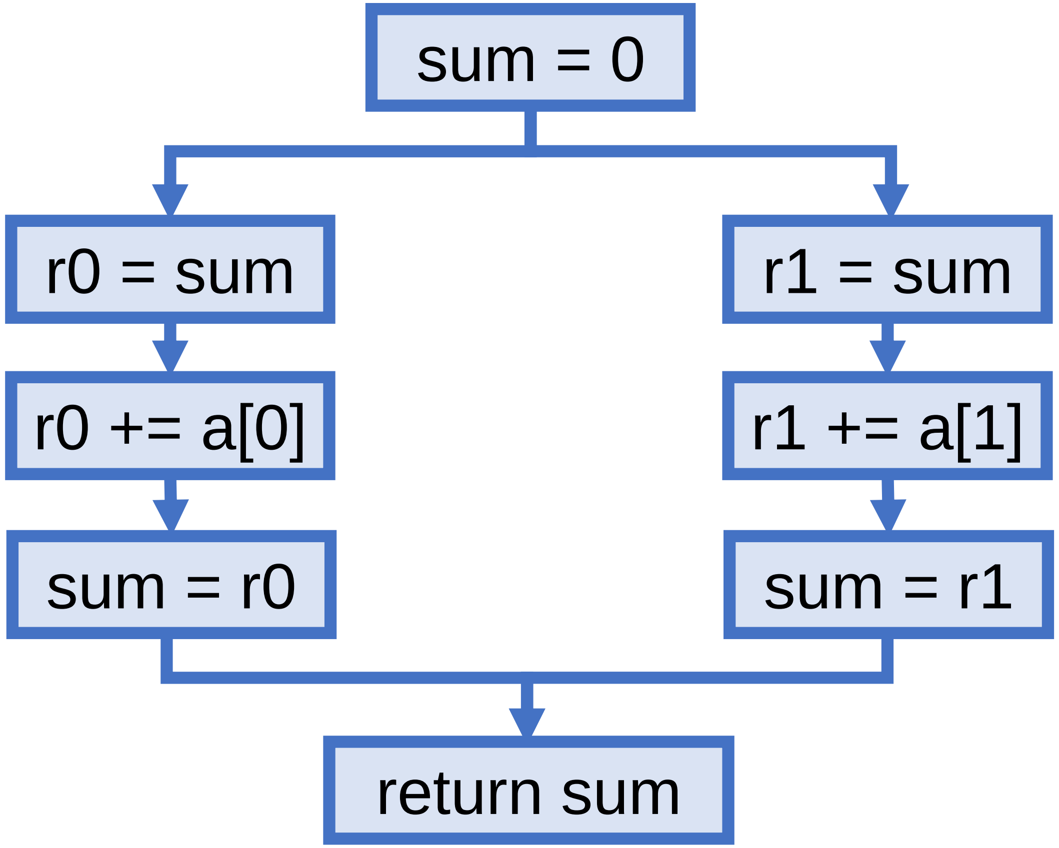
Simple, efficient both theoretically and practically

Wrap up

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 + a[i];
  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 two calls should be independent, including code executed by further in_parallel tasks

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 and reason.
  - Check the correctness of the sequential execution
  - Check if the parallel execution is the same as the sequential one

Atomic primitives (Lecture 2)

- **Compare-and-swap (CAS):**
  - bool CAS(value* p, value vold, value vnew)
  - Compare the value stored in the pointer p with value *vold*, if they are equal, try to change p's value to *vnew*. If successful, return true. Otherwise, return false.
- **Test-and-set (TAS):**
  - bool TAS(bool p)
  - Determine if the Boolean value stored at p is false, if so, try to set it to true. If successful, return true. Otherwise, return false.
- **Fetch-and-add (FAA):**
  - integer FAA(integer* p)
  - Add integer p's value by 1, and return the old value

Deterministic Parallelism

- Not race-free (atomic updates needed)

- Relative ordering of the operations is consistent with sequential execution
  - When debugging, first check the sequential execution, then check if the destinations of the swaps are the same in the parallel execution

- Execution is "deterministic"
  - Output is always the same for different executions
  - Input/output of each operation is always the same for different executions

- Determinism is supported by "priority updates"

Determinism is transitive

- If all subcomponents in an algorithm are race-free, then this algorithm is race-free
- If all subcomponents in an algorithm are deterministic, then this algorithm is deterministic

Parallel thinking

- When taking CS 141, 218 (classic algorithm courses) or reading CLRS, an algorithm is a list of operations

- Quicksort?
- Mergesort?
- Red-black tree?
- Suffix-tree?

- Algorithms become complicated in the parallel setting, so this is no longer a good abstraction

Parallel thinking

- Consider subroutines as primitives / functions / building blocks. An algorithm is the combination of a set of subroutines

- Quicksort: find a pivot, apply partition (rely on filter (rely on scan (rely on reduce))), then recurse
- Mergesort: first solve two subproblems, then use parallel merge
- Red-black tree: don't use RB-tree, use P-tree that is based on join
- Suffix-tree: design a parallel primitive to merge two trees

- Often use divide-and-conquer or reduce or similar techniques for inductively solving the subproblems with smaller sizes
- Conceptually simpler to understand
- Properties are transitive (race-free, deterministic, persistence, etc.)

Software Crisis

- In 1960s, programming was in assembly language

```
_start:
        mov     $start,%r15
        mov     $10,%r12

loop:
        mov     %r15,%r14

        /* Find out if index is 1
        mov     %r14,%rax
        mov     $0,%rdx
        div     %r12

        /* Ones decimal */
        mov     %rdx,%r13
        add     $0x30,%r13
        mov     %r13b,msg+7

        /* Check if there is a ten
        cmp     $0,%rax
        je      noTens

        /* There is a tens decimal
```

```
for (int i = 1; i <= V - 1; i++) {
    for (int j = 0; j < E; j++) {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}
```

Software Crisis

- In 1980s, programmers realized that it is almost impossible to writing very long C code

```
for (int i = 1; i <= V - 1; i++) {
    for (int j = 0; j < E; j++) {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            dist[v] = dist[u] + weight;
    }
}
```

- New concept of OOP and programming languages

New Software Crisis caused by Parallelism

- **Algorithms and programming become even more sophisticated**

- **Non-determinism can be a huge problem even for very simple applications**
  - Hard to debug
  - Hard to guarantee correctness

- **Use ideas from PL and algorithm research**
  - Functional programming
  - Race, deterministic parallelism

Homework 2

- HW 2 out tonight
- Due Feb 19th – You have 3 weeks