COL380

Introduction to
Parallel & Distributed Programming

Agenda

- Sample clock
- Basic synchronization primitives and their properties

Synchronization

- Do Operation X at time T

Synchronization

- Do Operation X at time T

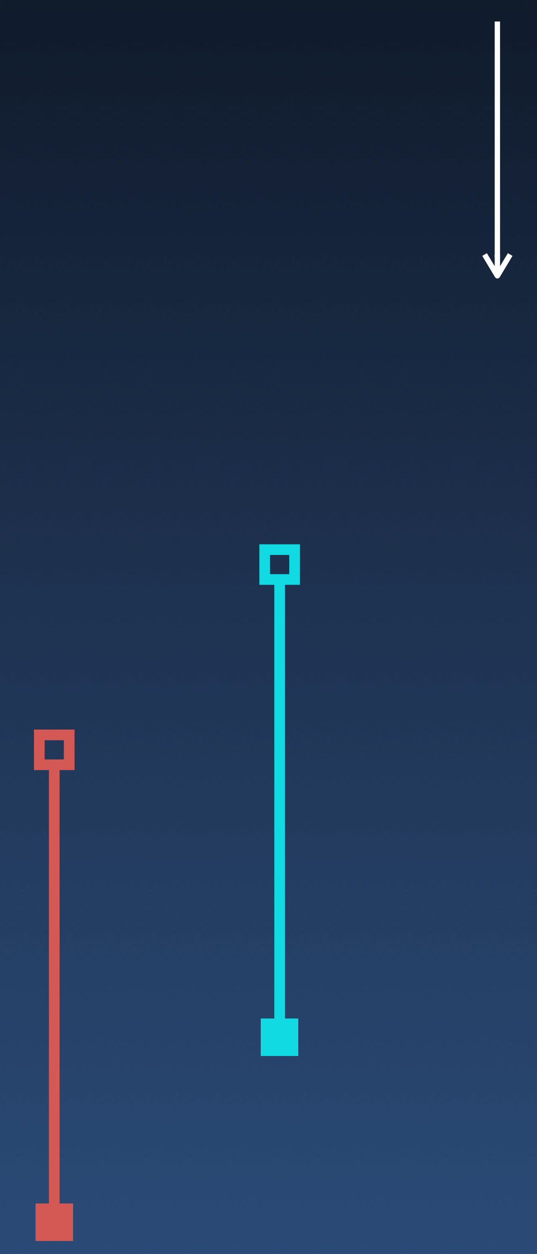
- Do we need a precise notion of time to make progress?
  - ➡ Always increasing
  - ➡ Shared view or individual view?

Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
  - Always increasing
  - Shared view or individual view?
- Basic synchronization
  - Two (or a set of) events should happen together
  - Any two (from a set of) events should NOT happen together
  - ★ Event A should happen after event B

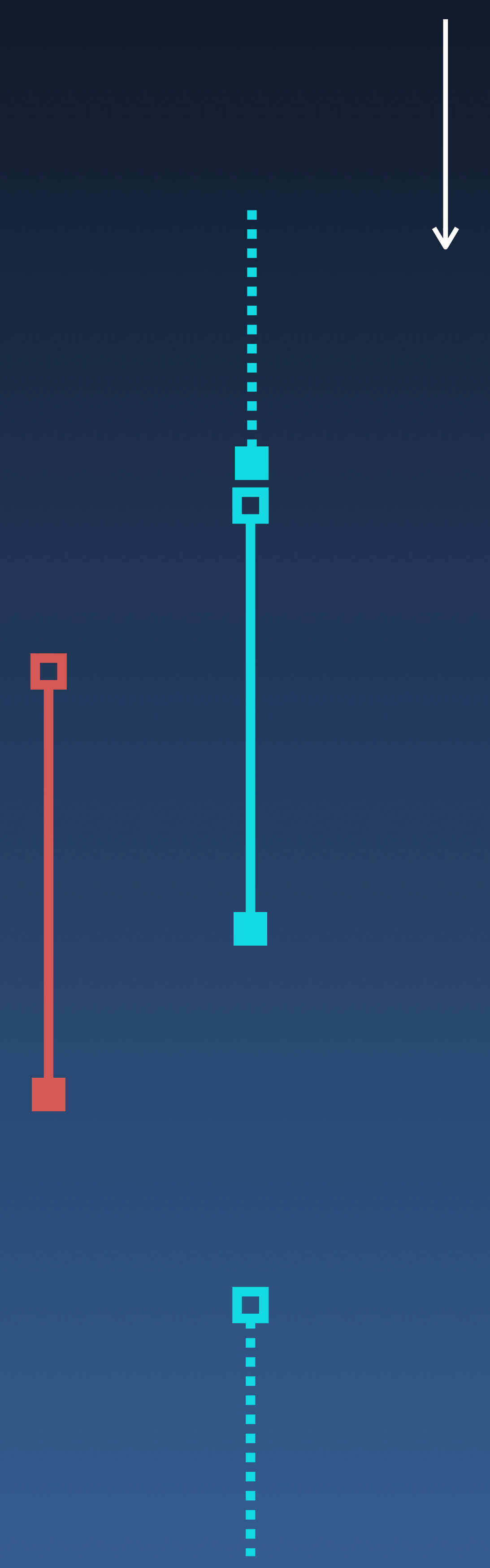
Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
  - ➡ Always increasing
  - ➡ Shared view or individual view?
- Basic synchronization
  - ➡ Two (or a set of) events should happen together
  - ➡ Any two (from a set of) events should NOT happen together
  - ★ Event A should happen after event B



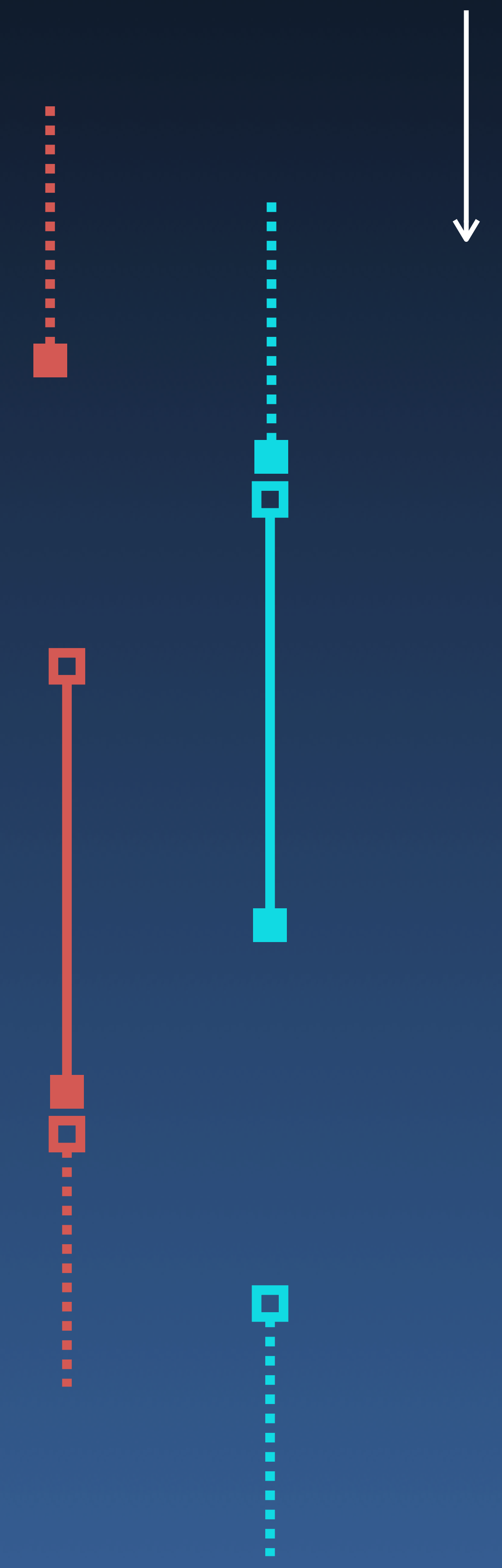
Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
  - ➡ Always increasing
  - ➡ Shared view or individual view?
- Basic synchronization
  - ➡ Two (or a set of) events should happen together
  - ➡ Any two (from a set of) events should NOT happen together
  - ★ Event A should happen after event B



Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
    - ➡ Always increasing
    - ➡ Shared view or individual view?
- Basic synchronization
    - ➡ Two (or a set of) events should happen together
    - ➡ Any two (from a set of) events should NOT happen together
    - ★ Event A should happen after event B

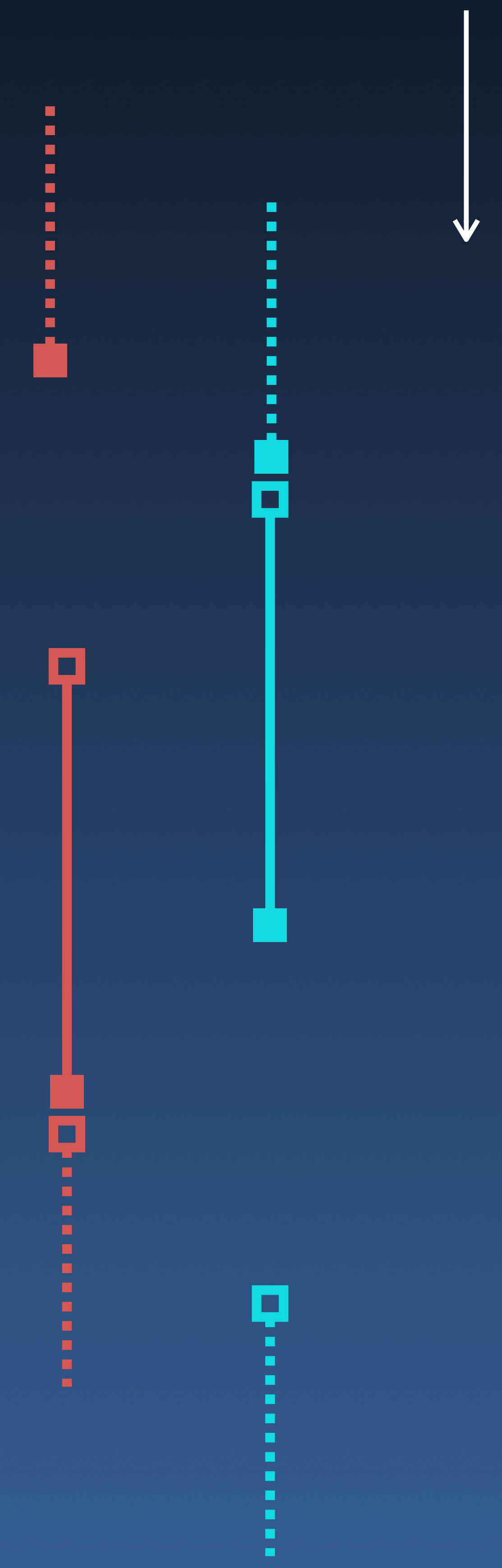


Subodh Kumar

Synchronization

- Do Operation X at time T
- Do we need a precise notion of time to make progress?
  - ➡ Always increasing
  - ➡ Shared view or individual view?
- Basic synchronization
  - ➡ Two (or a set of) events should happen together
  - ➡ Any two (from a set of) events should NOT happen together

★ Event A should happen after event B Stop and Go



Subodh Kumar

Causal Ordering

- Define a partial order
  - Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$
  - Inverse is not required to be true

Causal Ordering

- Define a partial order
  - Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$

    Same as: $\text{Time}(A) < \text{Time}(B) \Rightarrow B \not\rightarrow A$
  - Inverse is not required to be true

    Means they can are concurrent

Causal Ordering

- Define a partial order
  - Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$

    Same as: $\text{Time}(A) < \text{Time}(B) \Rightarrow B \not\rightarrow A$
  - Inverse is not required to be true

    Means they can are concurrent

Strong causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$ && $\text{Time}(A) < \text{Time}(B) \Rightarrow A \rightarrow B$

Causal Ordering

- Define a partial order
  - ➡ Causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$

    Same as: $\text{Time}(A) < \text{Time}(B) \Rightarrow B \nrightarrow A$

  - ➡ Inverse is not required to be true

    Means they can are concurrent

Strong causality: $A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)$ && $\text{Time}(A) < \text{Time}(B) \Rightarrow A \rightarrow B$

- Clocks at least must support partial ordering of events
  - ➡ Can construct total ordering (e.g., by using Process-ID to break tie)
  - ➡ Possible to build “counters” that can support total order (strong causality)

Logical-Clock

- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace

$$\boxed{A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)}$$

Subodh Kumar

Logical-Clock

- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace
- Interaction between entities is through messages
  - ➡ Data + counter

$$\boxed{A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)}$$

Logical-Clock

- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace
- Interaction between entities is through messages
  - ➡ Data + counter
- On message receipt:
  - ➡ If recipient counter < received counter
    - ▸ Increase local counter to received counter
    - ▸ Receive is an event, so increment by one

$$\boxed{A \rightarrow B \Rightarrow \mathrm{Time}(A) < \mathrm{Time}(B)}$$

Logical-Clock
[Lamport’s Timestamp algorithm]

- Each entity (process) maintains a counter
  - ➡ increments every ‘event,’ at its own pace
- Interaction between entities is through messages
  - ➡ Data + counter
- On message receipt:
  - ➡ If recipient counter < received counter
    - ▶ Increase local counter to received counter
    - ▶ Receive is an event, so increment by one

$\boxed{A \rightarrow B \Rightarrow \text{Time}(A) < \text{Time}(B)}$

Synchronization Concepts

- Synchronization primitives
    - ➡ Lock/Mutex, Condition variables, Monitor
    - ➡ Atomics, Critical section, Barrier, Wait, Order
- Properties of Synchronization
    - ➡ Safety, Liveness
    - ➡ Blocking, Starvation-free, Deadlock-free, Lockfree, Waitfree
- Central authority?
    - ➡ OS scheduler, Runtime

Lock

- Object: lock
- Actions: Lock and Unlock

Lock

```
omp_lock_t *lockA;
omp_init_lock (lockA);
…
omp_destroy_lock (lockA);
```

- Object: lock
- Actions: Lock and Unlock

Critical Section

```
OperateA(object *A)
{
    omp_set_lock(lockA);
    Operate_Exclusively(A)
    omp_unset_lock (lockA);
}
```

Lock

```
omp_lock_t *lockA;
omp_init_lock (lockA);
…
omp_destroy_lock (lockA);
```

- Object: lock
- Actions: Lock and Unlock

Critical Section

```
OperateA(object *A)
{
    omp_set_lock(lockA);
    Operate_Exclusively(A)
    omp_unset_lock (lockA);
```

```
OperatePlus(object *A)
{
    omp_set_lock(lockA);
    if(! A->initalized())
        OperateA(A);
    omp_unset_lock(lockA)
}
```

- ➡ Reentrant
- ➡ Recursive
- ➡ Timed
- ➡ Exclusive
- ➡ Shared

Lock

```
omp_lock_t *lockA;
omp_init_lock (lockA);
…
omp_destroy_lock (lockA);
```

- Object: lock
- Actions: Lock and Unlock

➡ Reentrant
➡ Recursive
➡ Timed
➡ Exclusive
➡ Shared

Critical Section

```
OperateA(object *A)
{
   omp_set_lock(lockA);
   Operate_Exclusively(A)
   omp_unset_lock (lockA);
```

```
OperatePlus(object *A)
{
    omp_set_lock(lockA);
    if(! A->initalized())
        OperateA(A);
    omp_unset_lock(lockA)
}
```

See
   omp_test_lock_t
   omp_init_test_lock
   omp_set_test_lock
   omp_unset_test_lock

Lock

```
omp_lock_t *lockA;
_lock (lockA);

omp_destroy_lock (lockA);
```

See:

```
    int omp_test_lock (omp_lock_t *);
```

- Object: lock
- Actions: Lock and Unlock

➡ Reentrant
➡ Recursive
➡ Timed
➡ Exclusive
➡ Shared

Critical Section

```
OperateA(object *A)
{
    omp_set_lock(lockA);
    Operate_Exclusively(A)
    omp_unset_lock (lockA);
```

```
OperatePlus(object *A)
{
    omp_set_lock(lockA);
    if(! A->initalized())
        OperateA(A);
    omp_unset_lock(lockA)
}
```

See

```
    omp_test_lock_t
    omp_init_test_lock
    omp_set_test_lock
    omp_unset_test_lock
```

Critical Section

- Block of code
- Criticality context

Critical Section

- Block of code
- Criticality context

```
#pragma omp critical (a_name)
{
        mutually_excluded_code();
}
```

Condition Variable

- Raise the condition
- Wait for a condition to ‘hold’

```
Produce();
acv.notify_one();
```

```
std::condition_variable acv;
…

std::unique_lock<std::mutex> alock(amutex);
acv.wait(alock);
.. Condition Holds Now ..
Consume();
```

Barrier

- A group of entities
- Wait for all

```
ParallelInput();
#pragma omp barrier
ParallelProcess();
#pragma omp barrier
ParallelOutput();
```

Review

- Logical clocks
  - ➡ Implementation
- Synchronization primitives
  - ➡ Lock, Critical section, Condition variables, Barrier