COL380

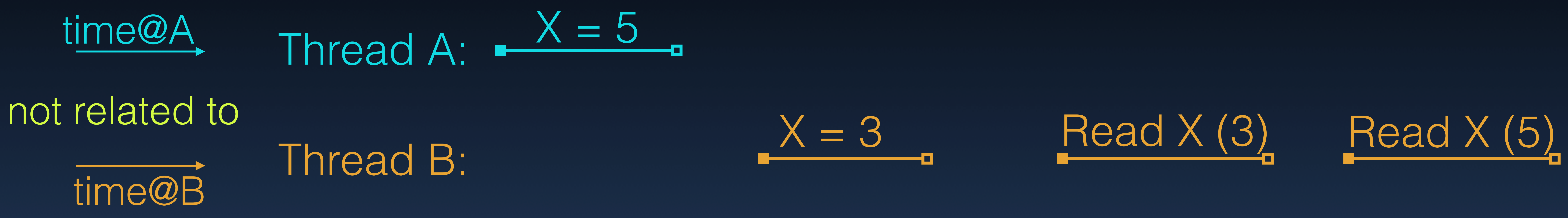
Introduction to
Parallel & Distributed Programming

Sequential Consistency

"A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lamport, 1979]

Weaker than Linearizability
Does not depend on global time
(Still not efficient to guarantee.)

Sequentially Consistent



- No global notion of time
  - ➡ Only consistent Order

Sequentially Consistent

time@A → not related to → time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order

Sequentially Consistent

time@A

not related to

time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order

Thread A: $A_1 \longrightarrow A_2$

Thread B: $B_1 \longrightarrow B_2$

Thread C: $C_1 \longrightarrow C_2 \longrightarrow C_3$

Sequential History

$A_1 \to A_2 \quad C_1 \to C_2 \to C_3 \quad B_1 \to B_2$

$A_1 \quad C_1 \quad A_2 \quad B_1 \to B_2 \quad C_2 \to C_3$

Sequentially Consistent

time@A →

not related to

→ time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order

Thread A: $A_1 \rightarrow A_2$ W

Thread B: R $B_1 \rightarrow B_2$ W

Thread C: $C_1 \rightarrow$ R $C_2 \rightarrow C_3$

Sequential History

$A_1 \rightarrow A_2$ $C_1 \rightarrow C_2 \rightarrow C_3$ $B_1 \rightarrow B_2$

A_1 C_1 $A_2 \rightarrow B_1 \rightarrow B_2 \rightarrow C_2 \rightarrow C_3$

Sequentially Consistent

time@A →

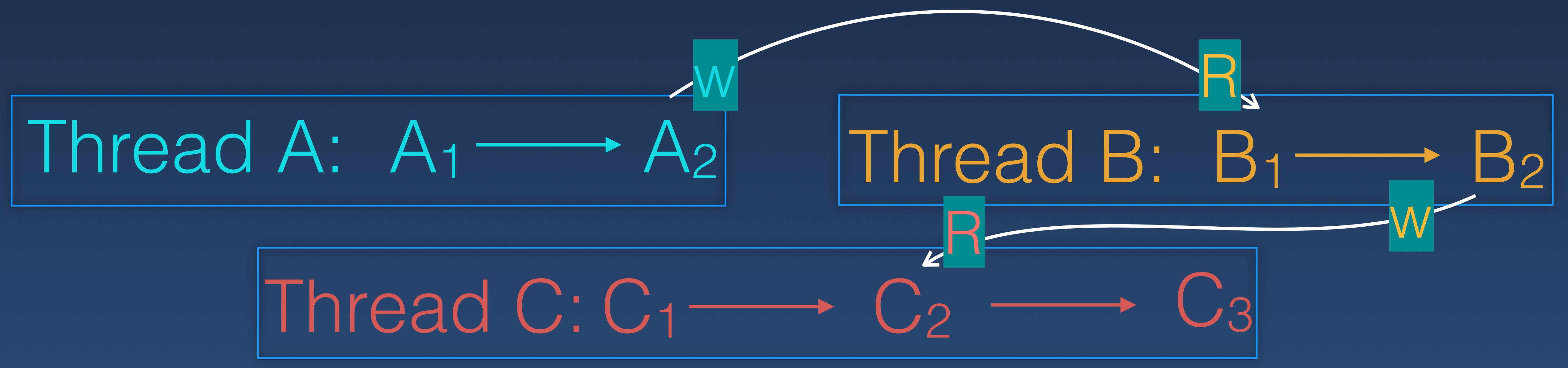
not related to

→ time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order

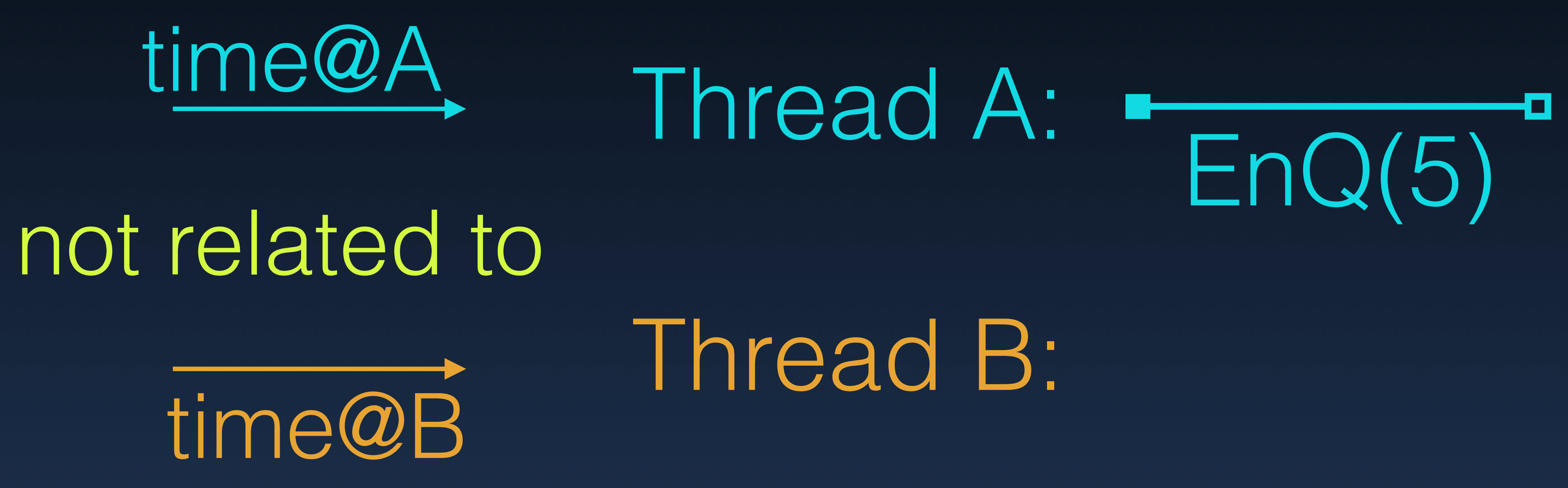


Sequential History

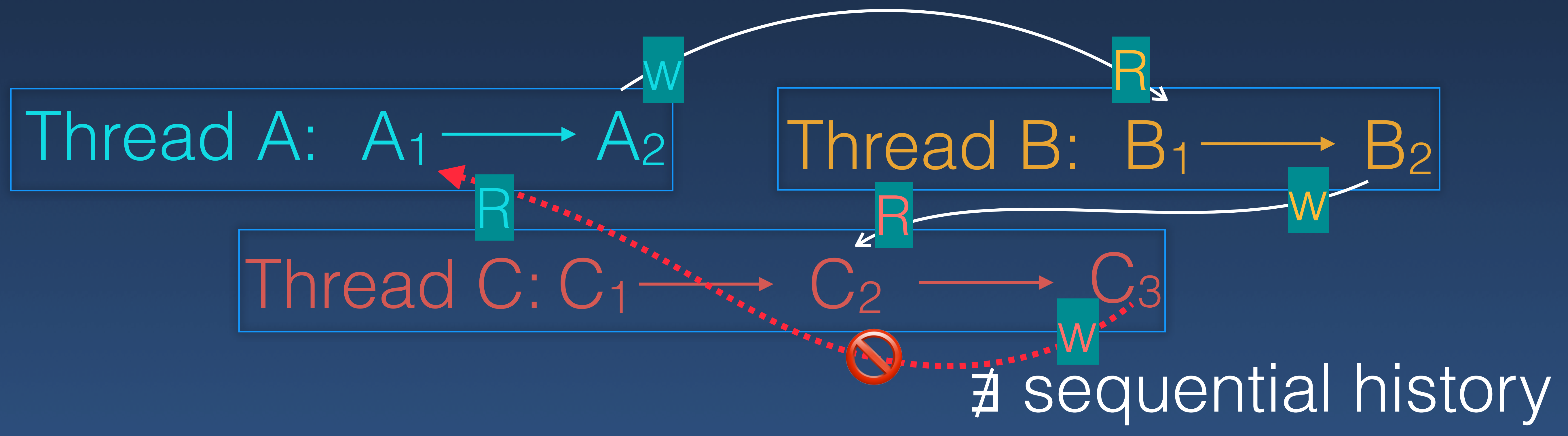
~~$A_1 \rightarrow A_2\ C_1 \rightarrow C_2 \rightarrow C_3\ B_1 \rightarrow B_2$~~

$A_1\ C_1\ A_2 \rightarrow B_1 \rightarrow B_2 \rightarrow C_2 \rightarrow C_3$

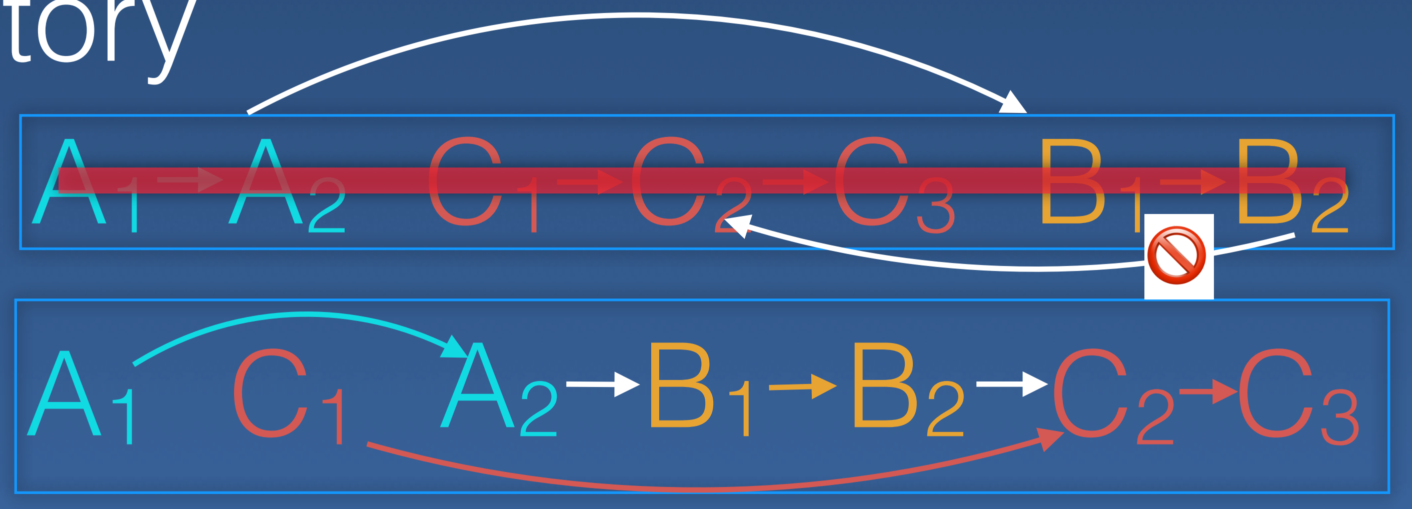
Sequentially Consistent



- No global notion of time
  - ➡ Only consistent Order



Sequential History



Subodh Kumar

Sequentially Consistent

time@A

not related to

time@B

Thread A: EnQ(5)

Thread B: EnQ(3) DeQ is 3 DeQ is 5

- No global notion of time
  - ➡ Only consistent Order

Thread A: $A_1 \rightarrow A_2$

Thread B: $B_1 \rightarrow B_2$

Thread C: $C_1 \rightarrow C_2 \rightarrow C_3$

W R R W R W

∄ sequential history

Sequential History

~~$A_1 \rightarrow A_2$ $C_1 \rightarrow C_2 \rightarrow C_3$ $B_1 \rightarrow B_2$~~

A_1 C_1 $A_2 \rightarrow B_1 \rightarrow B_2 \rightarrow C_2 \rightarrow C_3$

A: (x=3) ⟶ (x=5)

B: [read x]3

before later writes

after causative write

Applying Sequential Consistency

- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread

```
initially:    ready=0, data=0

 thread P                thread C

data = 10;            while(!ready);
ready = 1;            pvt = data;
```

Applying Sequential Consistency

- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread

```
initially:    ready=0, data=0

 thread P                 thread C

data = 10;             while(!ready);
ready = 1;             pvt = data;
```

$P.W_{data}$, $P.W_{ready}$, $C.R_{ready}$, $C.R_{data}$

Show: If C sees the updated ready (=1),
C must also see the updated data (=10)

Applying Sequential Consistency

- Threads always see values written by some thread
  - ➡ No garbage (update is atomic)
- The value seen is constrained by thread-order
  - ➡ for every thread

```
initially:    ready=0, data=0

 thread P              thread C

data = 10;             while(!ready);
ready = 1;             pvt = data;
```

$P.W_{data}$

↓

$P.W_{ready}$

↓

$C.R_{ready}$

↓

$C.R_{data}$

If C sees ready =	then C sees data =
0	0 or 10
1	10

Show: If C sees the updated ready (=1), C must also see the updated data (=10)

Initially: X = 0; Y = 0;

Thread A		Thread B
		X = 5
		Y = 3
l = Y	[3]	
r = X	[0]	

Initially: X = 0; Y = 0;

Thread A		Thread B	
l = Y	[3]	X = 5	Out of order execution (completion)
r = X	[0]	Y = 3	

Coherence *vs* Consistency

Initially: X = 0; Y = 0;

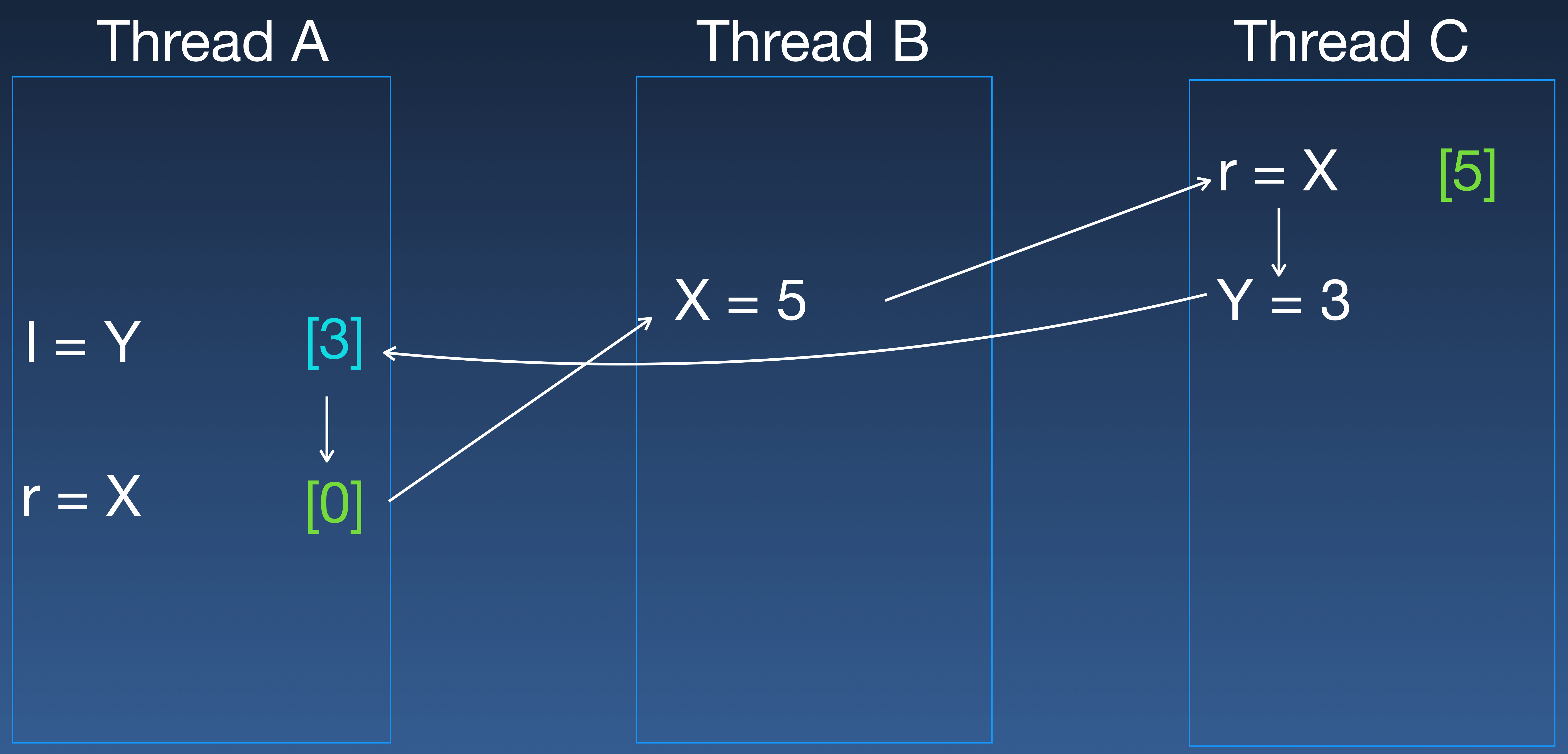
Thread A		Thread B
l = Y	[3]	X = 5
r = X	[0]	Y = 3

Out of order execution
(completion)

Write:
stage in store buffer
read cache-line
for update
update entry

Coherence *vs* Consistency

Initially: X = 0; Y = 0;



Subodh Kumar

Memory inconsistency

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack entry for some shared variable
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two memory messages

Memory inconsistency

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack en
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two mem

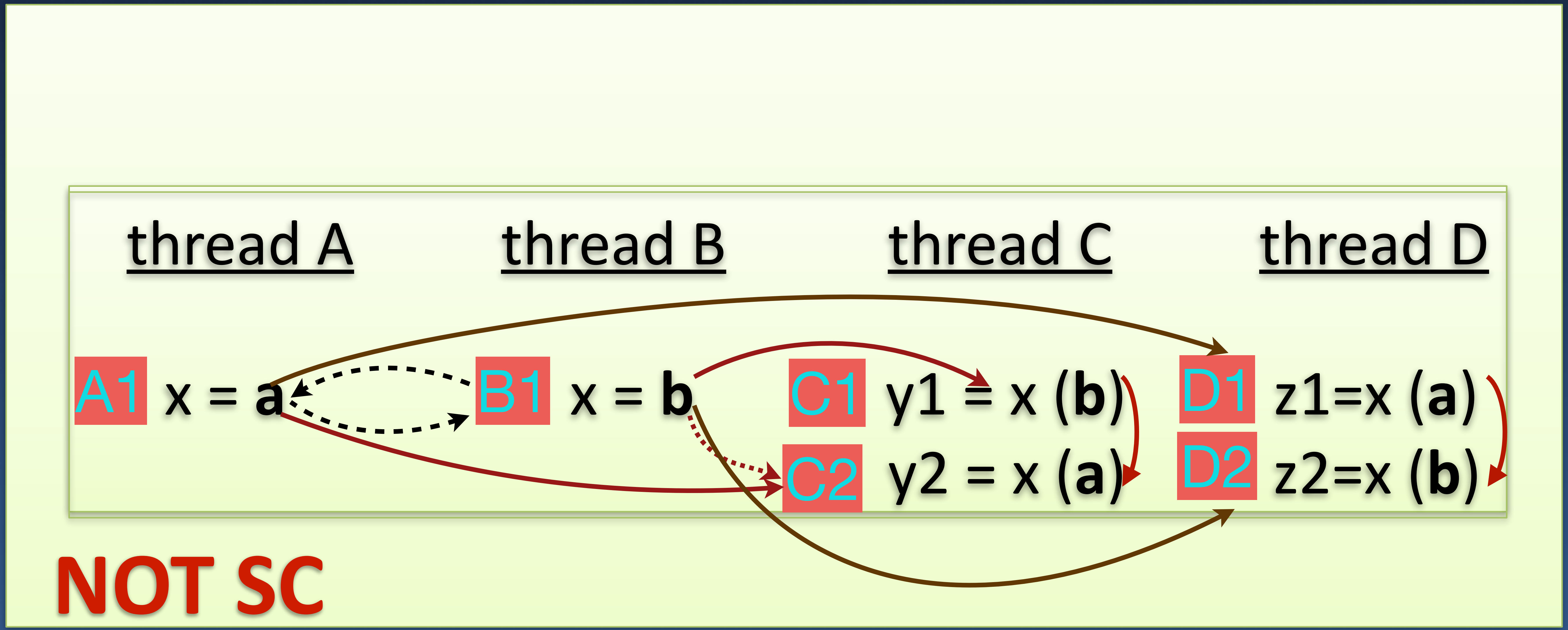
X=1; Y=1; X=2;

- ➡ Second write to X may happen before Y's
- ➡ 1st write may never happen

- Consistency is about global state (not per-variable)
  - ➡ Program must not assume higher consistency than available
- Only local memory dependencies visible to compiler/architecture
  - ➡ Can allocate a register or stack entry for some shared variable
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two memory messages

★ Solutions: Handle inconsistency, force synchronization

Example: Not SC



SC Inefficiency

- Hard to implement efficiently
  - ➡ Need to enforce serialization of operations
  - ➡ No re-ordering of instructions allowed

thread A	thread B	thread C	thread D
x = **a**	x = **b**	y1 = x (**b**) y2 = x (**a**)	z1=x (**a**) z2=x (**b**)

- Some solutions:
  - ➡ Allow out-of-order execution, Detect and recover from SC violation
  - ➡ Only enforce ordering when required
    - programmer enforced

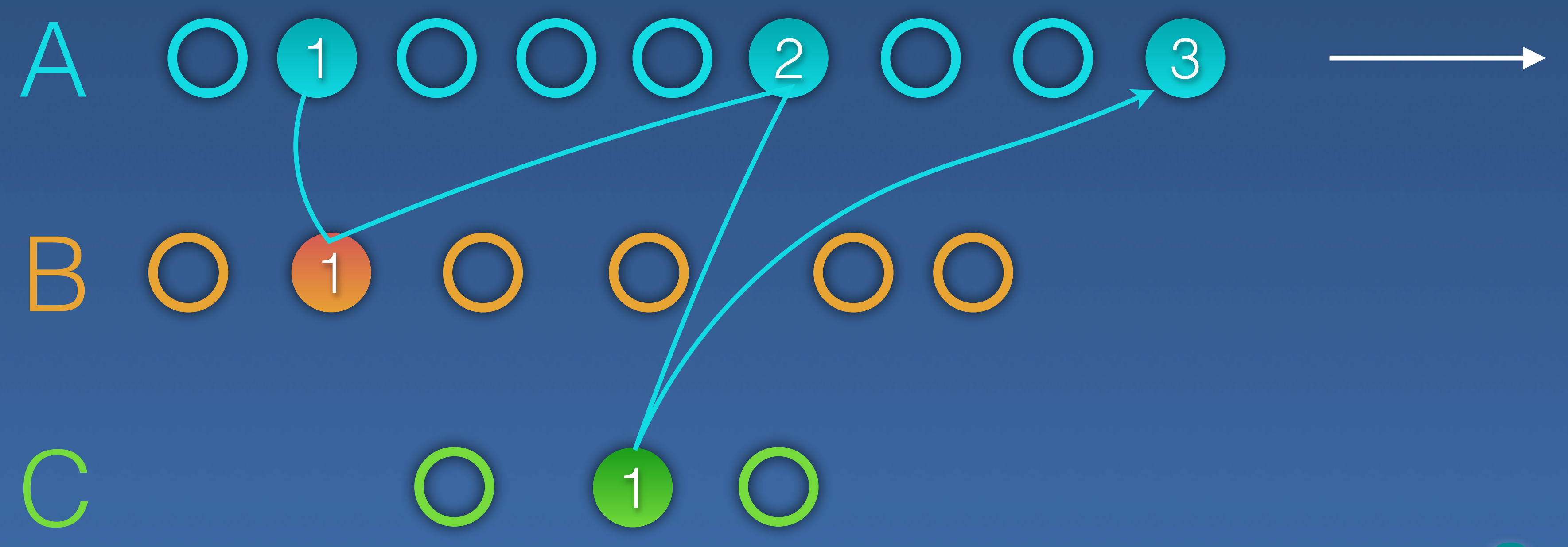
Relaxing SC

initially: ready=0, data=0

thread P	thread C
data1 = 1	Wait for OK
data2 = 1	sav1 = data1
Say OK	sav2 = data2

Weak Consistency

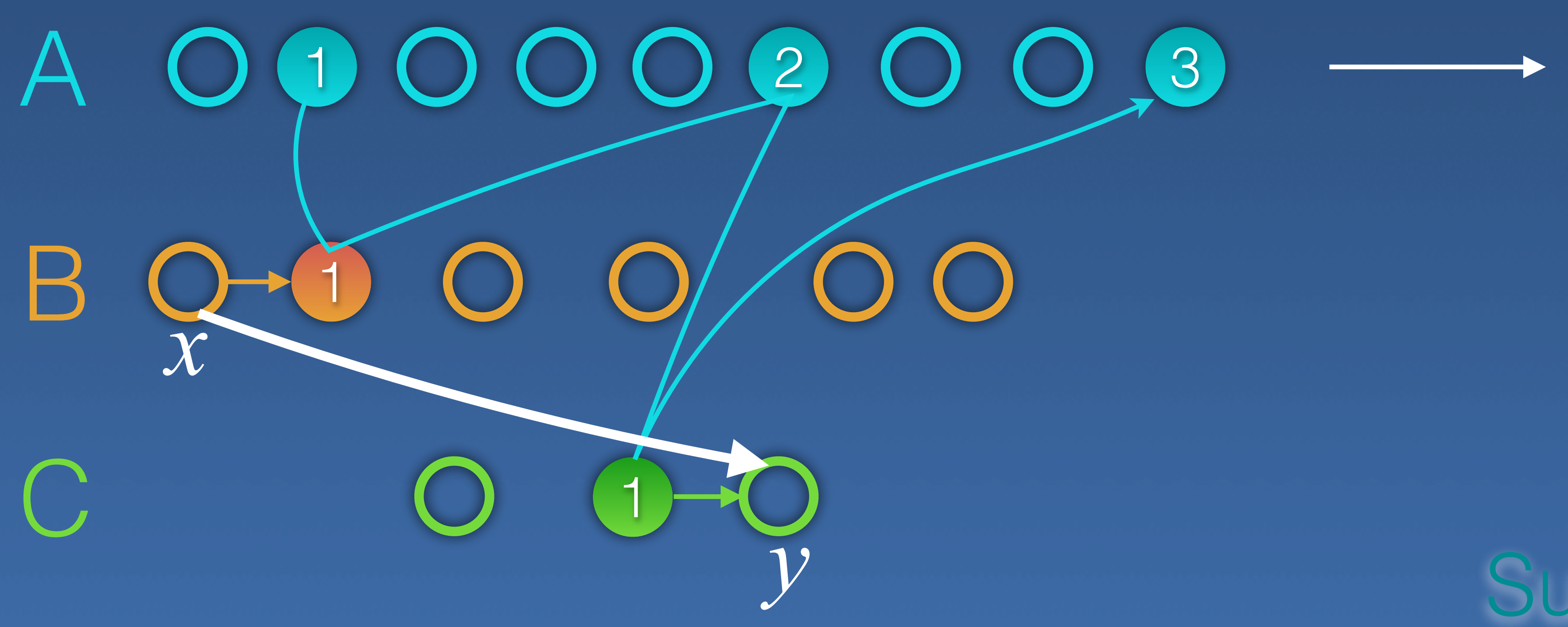
- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
    - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
    - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations



Subodh Kumar

- Special synchronization accesses are sequentially consistent
- Regular accesses ordered only with respect to synchronization accesses
  - Before any regular read/write is allowed to be visible to any other thread, all previous synchronization accesses must become visible
  - Before a synchronization access is allowed to complete, all previous ordinary read/write accesses must be completed
- Suitable for many optimizations

$\Rightarrow x$ before y
(wrt every thread)



OpenMP Flush

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
```

Thread B

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
```

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
```

Thread B

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
```

	or
flush-A	flush-B
flush-B	flush-A

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

Subodh Kumar

OpenMP Flush

flagA = flagB = 0

Thread A	Thread B
flagA = 1;	flagB = 1;
#pragma omp flush	#pragma omp flush
if (flagB == 0) {	if (flagA == 0) {
shared ++;	shared++;
}	}

or

flagA = 1	flagB = 1
flush-A	flush-B
flush-B	flush-A
flagA == 0?	flagB == 0?

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush

flagA = flagB = 0

<u>Thread A</u>

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
```

<u>Thread B</u>

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
```

or

flagB = 1
flagA = 1
flush-A
flush-B
flagA == 0?
flagB == 0?

flagB = 1
flush-B
flush-A
flagB == 0?

Flush (Memory fence) are sequentially consistent
- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush

flagA = flagB = 0

<u>Thread A</u>

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
```

<u>Thread B</u>

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
```

	or
flagB = 1	
flagA = 1	flagB = 1
flush-A	flush-B
flagB == 0?	
flush-B	flush-A
flagA == 0?	flagB == 0?

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

Subodh Kumar

OpenMP Flush

flagA = flagB = 0

<u>Thread A</u>

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
```

<u>Thread B</u>

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
```

	or
flagB = 1	
flagA = 1	flagB = 1
flush-A	flush-B
flagB == 0?	
flush-B	flush-A
flagA == 0?	flagB == 0?

⇓

∀ admissible orders

flagA=1 ➜ flagA==0?

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

Subodh Kumar

OpenMP Flush

flagA = flagB = 0

<u>Thread A</u>

```
flagA = 1;
#pragma omp flush
if (flagB == 0) {
    shared ++;
}
flagA = 0;
#pragma omp flush
```

(allow other)

<u>Thread B</u>

```
flagB = 1;
#pragma omp flush
if (flagA == 0) {
    shared++;
}
flagB = 0;
#pragma omp flush
```

or

flagB = 1 flagA = 1	flagB = 1
flush-A flagB == 0?	flush-B
flush-B	flush-A
flagA == 0?	flagB == 0?

$\Downarrow$

∀ admissible orders

flagA=1 ➜ flagA==0?

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush

flagA = flagB = 0

Thread A	Thread B
flagA = 1;	flagB = 1;
#pragma omp flush	#pragma omp flush
if (flagB == 0) {	if (flagA == 0) {
shared ++; ← mutual exclusion →	shared++;
}	}
flagA = 0;	flagB = 0;
#pragma omp flush	#pragma omp flush

	or
flagB = 1	
flagA = 1	flagB = 1
flush-A	flush-B
flush-B	flush-A
flagA == 0?	flagB == 0?
flagB == 0?	

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush vars

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush (flagA, flagB)
if (flagB == 0) {
    shared ++;
}
```

Access other variables

Thread B

```
flagB = 1;
#pragma omp flush (flagA, flagB)
if (flagA == 0) {
    shared++;
}
```

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush vars

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush (flagA, flagB)
if (flagB == 0) {
    shared ++;
}
```

Access other variables

Thread B

```
flagB = 1;
#pragma omp flush (flagA, flagB)
if (flagA == 0) {
    shared++;
}
```

Flush (Memory fence) are sequentially consistent

- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush vars

flagA = flagB = 0

<u>Thread A</u>

```
flagA = 1;
#pragma omp flush (flagA, flagB)
if (flagB == 0) {
    shared ++;
}
```

Access other variables

<u>Thread B</u>

```
flagB = 1;
#pragma omp flush (flagA, flagB)
if (flagA == 0) {
    shared++;
}
```

```
#pragma omp atomic read
    val = var;
#pragma omp atomic write
    var = expr();
```

Flush (Memory fence) are sequentially consistent
- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush vars

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush (flagA, flagB)
if (flagB == 0) {
    shared ++;
}
```

Access other variables

Thread B

```
flagB = 1;
#pragma omp flush (flagA, flagB)
if (flagA == 0) {
    shared++;
}
```

Implicit Flush for all synchronization operations

```
#pragma omp atomic read
    val = var;
#pragma omp atomic write
    var = expr();
```

Flush (Memory fence) are sequentially consistent
- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

OpenMP Flush
one-sided

Operations before Release flush must appear before (Completes)
Operations after Acquire flush must appear after (Initiates)

flagA = flagB = 0

Thread A

```
flagA = 1;
#pragma omp flush release
if (flagB == 0) {
    shared ++;
}
```

Thread B

```
flagB = 1;
#pragma omp flush acquire
if (flagA == 0) {
    shared++;
}
```

flagA = 1
flush-A

flush-B

flagA == 0?

(only this order would work)

or

~~flagB = 1~~
~~flush-B~~

~~flush-A~~

~~flagB == 0?~~

Flush (Memory fence) are sequentially consistent
- Wait for ongoing memory operations to finish
- Discard Local view (Subsequent reads actually load from memory)

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;

// Set flag to signal Thread 1
flag = 1;
```

Thread C

```
// Busy-wait until flag is signalled

while (flag != 1) {

}

// Consume data
printf(data=%d\n", data);
```

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;

// Set flag to signal Thread 1
flag = 1;
```

Thread C

```
// Busy-wait until flag is signalled

while (flag != 1) {

}

// Consume data
printf(data=%d\n", data);
```

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;

// Set flag to signal Thread 1
flag = 1;
// Flush
#pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)
while (flag != 1) {
    #pragma omp flush(flag)
}

// Consume data
printf(data=%d\n", data);
```

Subodh Kumar

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;

// Set flag to signal Thread 1
flag = 1;
// Flush
#pragma omp flush(flag)
```

produce

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)
while (flag != 1) {
    #pragma omp flush(flag)
}

// Consume data
printf(data=%d\n", data);
```

consume

Producer-Consumer

int data, flag = 0;

<u>Thread P</u>

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

<u>Thread C</u>

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)      F4
}

// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees 'flag 1.' (aside: 'flag 1' $\not\Rightarrow$ F2 $\rightarrow$ F4)

Subodh Kumar

Producer-Consumer

int data, flag = 0;

Thread P

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                 F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)       F4
}

// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees ‘flag 1.’ (aside: ’flag 1’ $\not\Rightarrow$ F2 → F4)

F2 must eventually finish.

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)      F4
}

// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees 'flag 1.'

'flag 1' in Th.1 ⇒ W has started and hence F1 has happened ⇒ F1 R

Subodh Kumar

Producer-Consumer

int data, flag = 0;

Thread P

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                 F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)       F4
}

// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees 'flag 1.'

'flag 1' in Th.1 ⇒ W has started and hence F1 has happened ⇒ F1 R

Subodh Kumar

Producer-Consumer

int data, flag = 0;

Thread P

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                      F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)             F4
}
#pragma omp flush(flag, data)                F5
// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees 'flag 1.'

'flag 1' in Th.1 ⇒ W has started and hence F1 has happened ⇒ F1 R F5

Subodh Kumar

Producer-Consumer

```
int data, flag = 0;
```

Thread P

```
// Produce data
data = 42;
// Flush
F1 #pragma omp flush(flag, data)
// Set flag to signal Thread 1
W flag = 1;
// Flush
F2 #pragma omp flush(flag)
```

Thread C

```
// Busy-wait until flag is signalled
#pragma omp flush(flag)                  F3
while (flag != 1) {  R
    #pragma omp flush(flag, data)        F4
}
#pragma omp flush(flag, data)            F5
// Consume data
printf(data=%d\n", data);
```

W F2 F4 guarantees that Th.1 sees 'flag 1.'

'flag 1' in Th.1 ⇒ W has started and hence F1 has happened ⇒ F1 R F5

Subodh Kumar