COL380

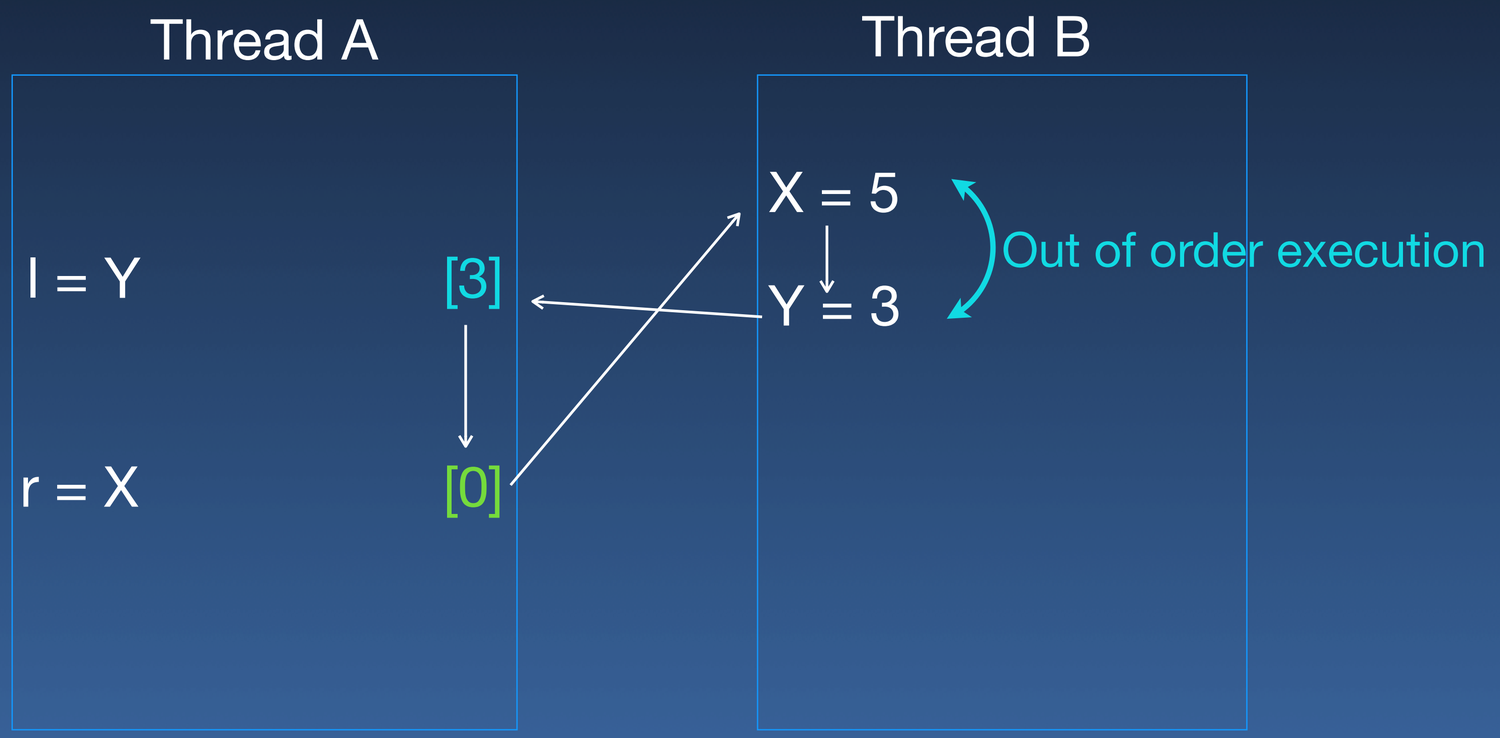
Introduction to
Parallel & Distributed Programming

Agenda

- Weak Consistency Models

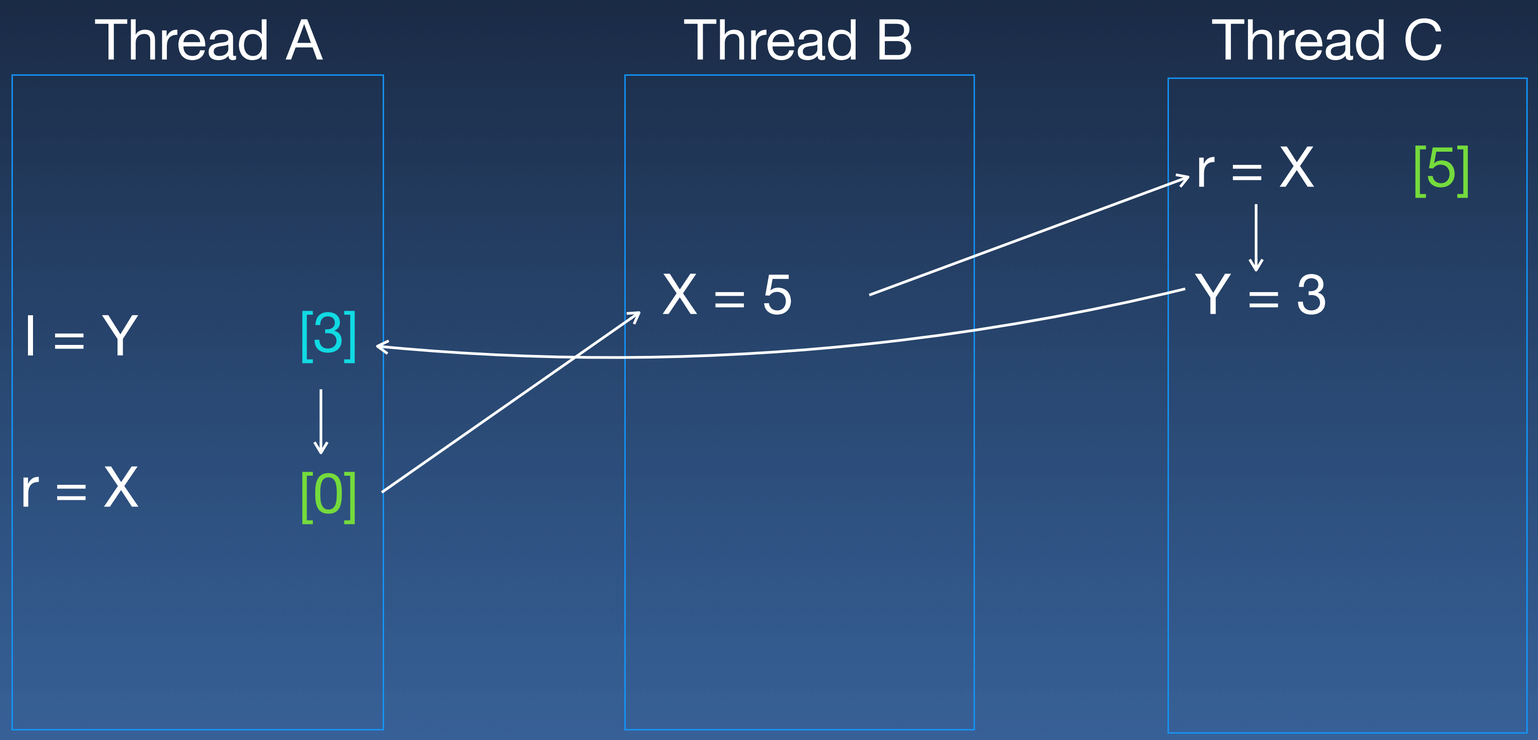
Coherence *vs* Consistency

Initially: X = 0; Y = 0;



Coherence *vs* Consistency

Initially: X = 0; Y = 0;



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 er
  - ➡ Batching of memory transactions
  - ➡ Network can also reorder two mer

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

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

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

★ Solutions: Handle inconsistency, force synchronization

Not SC

thread A	thread B	thread C	thread D
A1 x = **a**	B1 x = **b**	C1 y1 = x (**b**)	D1 z1=x (**a**)
		C2 y2 = x (**a**)	D2 z2=x (**b**)

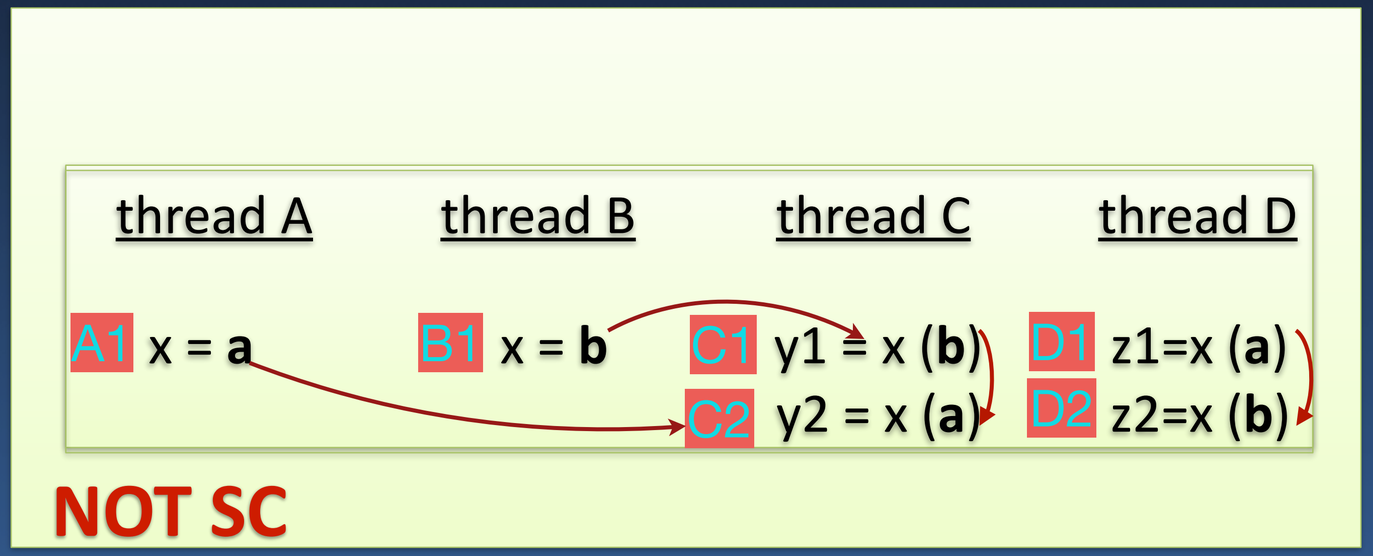
NOT SC

Not SC

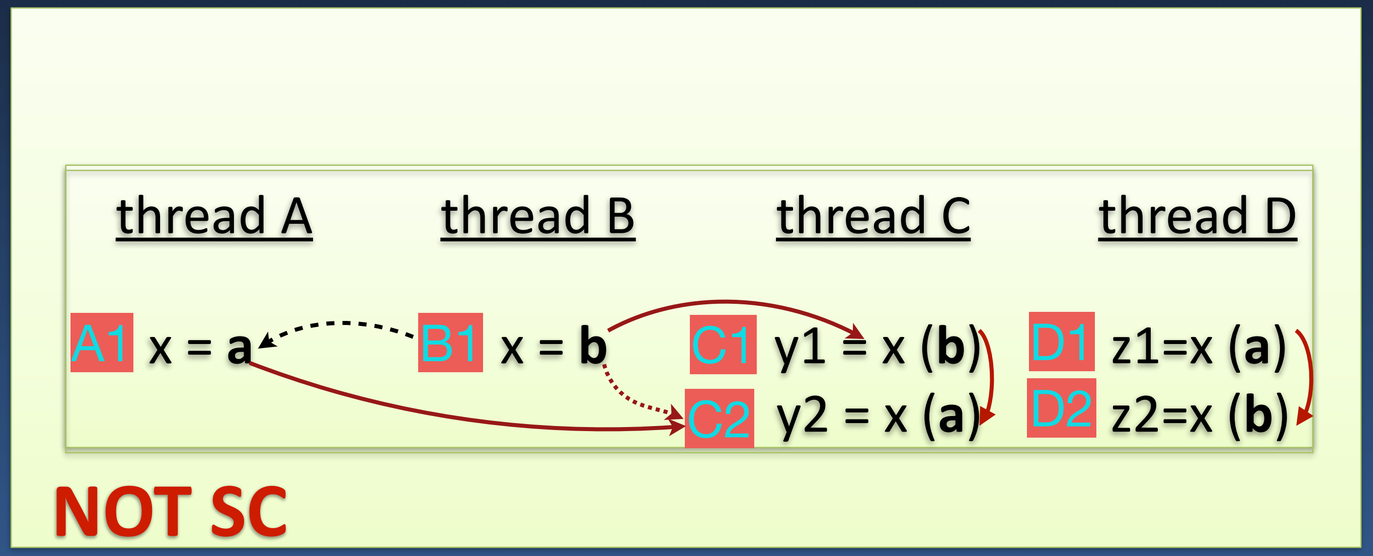
thread A	thread B	thread C	thread D
A1 x = **a**	B1 x = **b**	C1 y1 = x (**b**)	D1 z1=x (**a**)
		C2 y2 = x (**a**)	D2 z2=x (**b**)

NOT SC

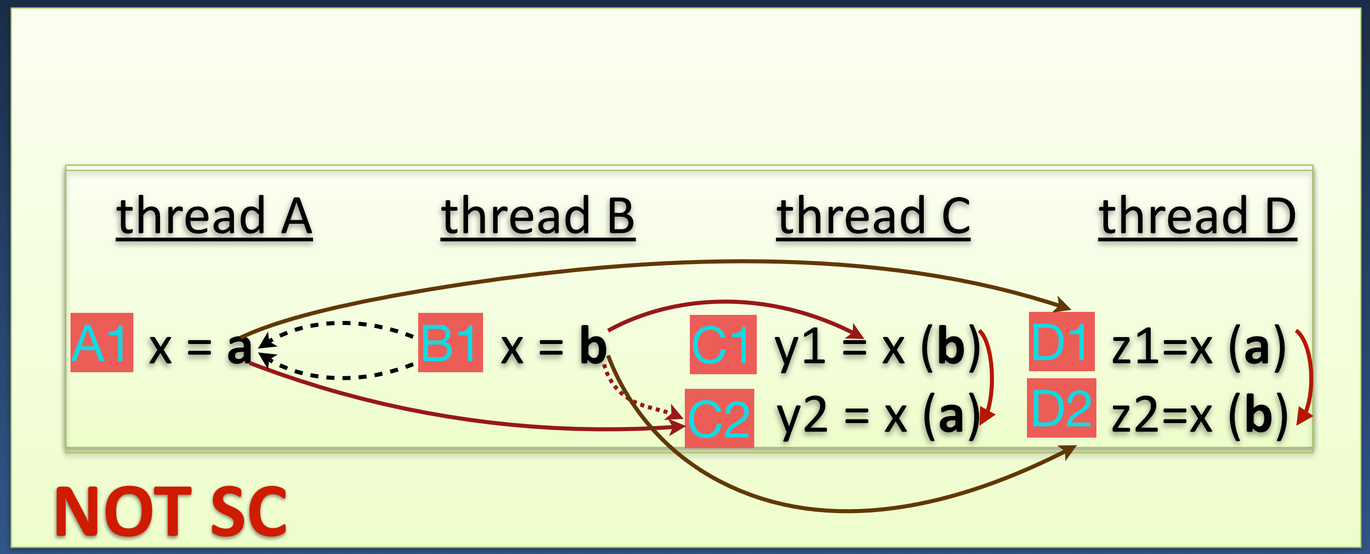
Not SC


thread A
thread B
thread C
thread D
A1 x = a
B1 x = b
C1 y1 = x (b)
C2 y2 = x (a)
D1 z1=x (a)
D2 z2=x (b)
NOT SC

Not SC


thread A
thread B
thread C
thread D
A1 x = a
B1 x = b
C1 y1 = x (b)
C2 y2 = x (a)
D1 z1=x (a)
D2 z2=x (b)
NOT SC

Not SC


thread A
thread B
thread C
thread D
A1 x = a
B1 x = b
C1 y1 = x (b)
C2 y2 = x (a)
D1 z1=x (a)
D2 z2=x (b)
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

thread1	thread 2
data1 = 1	SYNC
data2 = 1	sav1 = data1
SYNC	sav2 = data2

Relaxing SC

initially: ready=0, data=0

thread1	thread 2
data1 = 1	SYNC
data2 = 1	sav1 = data1
SYNC	sav2 = data2

Causal Consistency

- Write is causally ordered after all earlier reads/writes in its thread
  - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
  - ➡ Non-related writes may be seen in different order by different threads

Causal Consistency

- Write is causally ordered after all earlier reads/writes in its thread
  - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
  - ➡ Non-related writes may be seen in different order by different threads

Causally Consistent

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

Causal Consistency

- Write is causally ordered after all earlier reads/writes in its thread
  - ➡ write may depends on the current complete ‘state’
- Read is causally ordered after its causative write
- Causality is transitive
- ∃ sequential order of causally related operations consistent with every thread’s view
  - ➡ Non-related writes may be seen in different order by different threads

~~Causally Consistent~~

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

Processor Consistency

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread in the same order
  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

Processor Consistency

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread in

FIFO consistency relaxes this constraint

  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

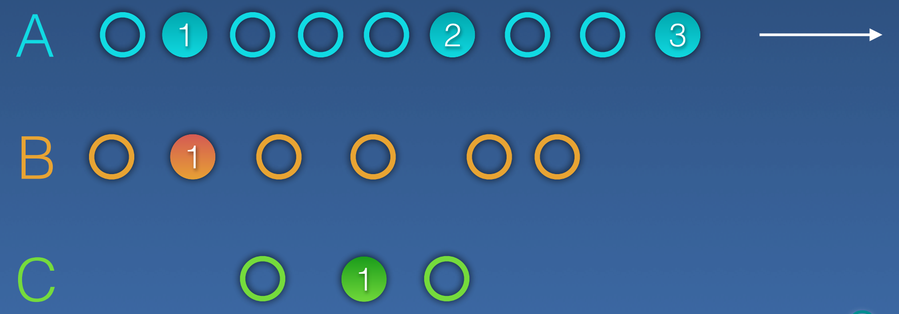
Processor Consistency

- All threads see all writes by each thread in the order of that thread
  - ➡ all instances of write(**x**) are seen by each thread in the same order
  - ➡ No need to consistently order writes to different variables by different threads
- Easy to implement
  - ➡ Two or more writes from a single source must remain in order, as in a pipeline
  - ➡ All writes are through to the memory

FIFO consistency is also known as **PRAM consistency**

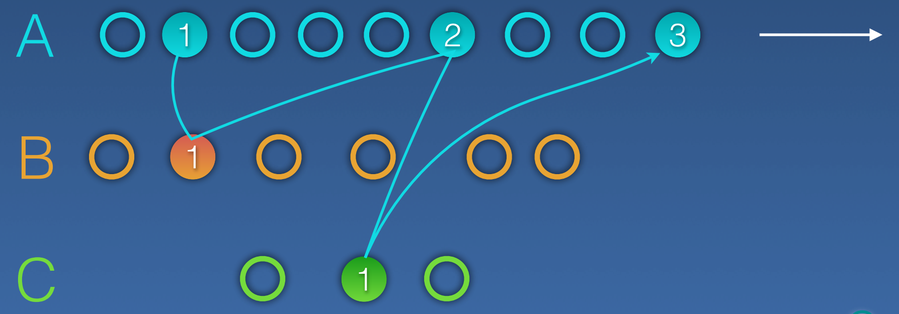
Weak Consistency

- 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



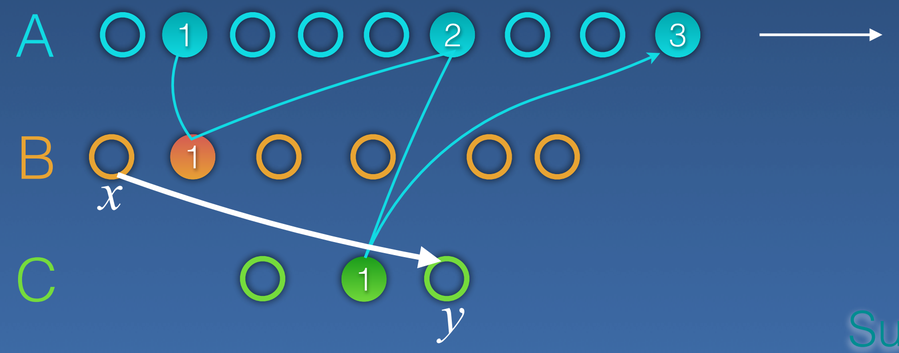
Weak Consistency

- 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



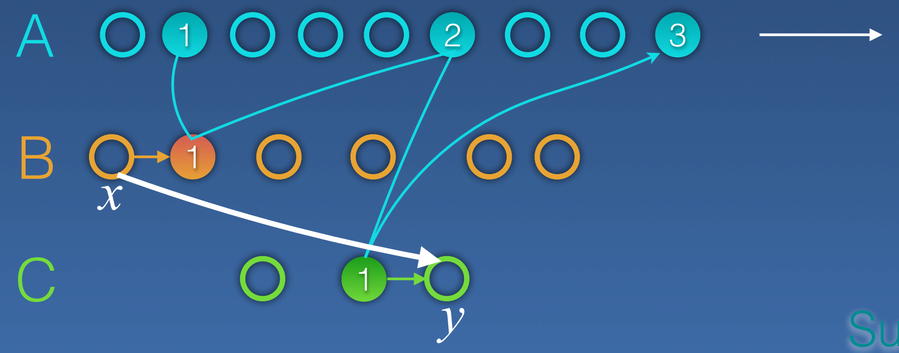
Weak Consistency

- 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



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



OpenMP Flush

flagA = flagB = 0

Thread A

```
flagA = 1;

if (flagB == 0) {
    shared ++;
}
```

Thread B

```
flagB = 1;

if (flagA == 0) {
    shared++;
}
```

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++;
}
```

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++;
}
```

flush-A

flush-B

or

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)

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
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

Thread A

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

Thread B

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

	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

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
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)

OpenMP Flush

flagA = flagB = 0

Thread A

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

Thread B

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

	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)

OpenMP Flush

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

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

flagA = flagB = 0

Implicit Flush for all synchronization operations

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++;
}
```

```
#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

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++;
}
```

	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

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++;
}
```

	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

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++;
}
```

	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)

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

Thread 0

```
// Produce data
data = 42;

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

Thread 1

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

while (flag != 1) {

}

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

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

Thread 0

```
// Produce data
data = 42;

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

Thread 1

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

while (flag != 1) {

}

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

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

Thread 0

```
// Produce data
data = 42;

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

Thread 1

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

while (flag != 1) {

}

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

Producer-Consumer

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

Thread 0

```
// Produce data
data = 42;

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

Thread 1

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

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

Producer-Consumer

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

Thread 0

```
// Produce data
data = 42;

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

Thread 1

```
// 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;
```

Thread 0

```
// Produce data
data = 42;

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

produce

Thread 1

```
// 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;
```

Thread 0

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

Thread 1

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

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

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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.'

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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' ⇏ F2 → F4)

Assume F2 must eventually finish.

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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' ⇏ F2 → F4)

Assume F2 must eventually finish.

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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

Producer-Consumer

int data, flag = 0;

Thread 0

```
// 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 1

```
// 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

Consistency Summary

Model	Description
Strict	Global time based atomic ordering of *all* shared accesses
Sequential	*All* threads see all shared accesses in the same order consistent with program order -- no centralized ordering
Causal	All threads see causally-related shared accesses in the same order
Processor	All threads see writes from each other in the order they were made. Writes to a variable must be seen in the same order by all threads
Weak	Special synchronization based reordering -- shared data consistent only after synchronization

Review

- Causal, Processor, FIFO consistency
- Weak consistency
- OpenMP Flush