Synchronization

Rupesh Nasre.

IIT Madras
January 2025

Learning Outcomes

- Data Race, Mutual Exclusion, Deadlocks
- Atomics, Locks, Barriers
- Reduction
- Prefix Sum
- Concurrent List Insertion
- CPU-GPU Synchronization

Data Race

- A datarace occurs if ***all*** of the following hold:
    1. Multiple threads
    2. Common memory location
    3. At least one write
    4. Concurrent execution
- Ways to remove datarace:
    1. Execute sequentially
    2. Privatization / Data replication
    3. Separating reads and writes by a barrier
    4. Mutual exclusion

Classwork

- Is there a datarace in this code?
- What does the code ensure?
- Can mutual exclusion be generalized for N threads?

If initially flag == 0, then S2 executes before S1.

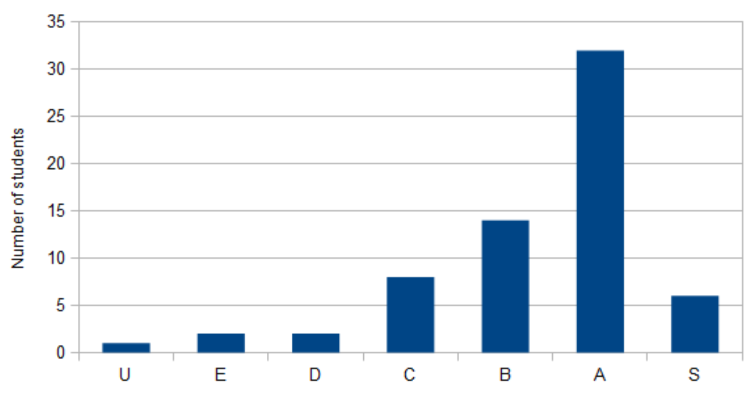
If initially flag == 1, then S2 executes and after that S1 may execute or T1 may hang.

T1	T2

flag = 1;
while (flag)
;
S1;
	while (!flag)
	;
	S2;
	flag = 0;

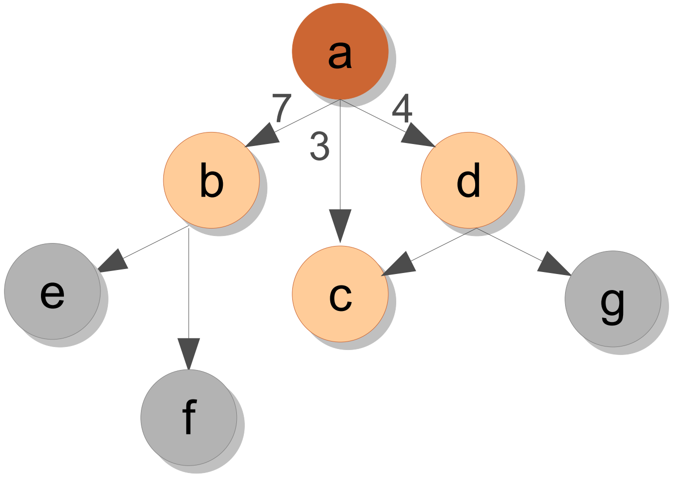
Classwork: Grading

- Given roll numbers and marks of 80 students in GPU Programming, assign grades.
  - S = 90, A = 80, B = 70, …, E = 40, and U.
  - No W grades.
  - Use input arrays and output arrays.
- Compute the histogram.
  - Count the number of students with a grade.



Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.




```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = …
    for each n in g.allneighbors(id) {     // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;
}   }   }
```

What is the error in this code?

Synchronization

- **Control + data flow**
- Atomics
- Barriers
- ...

Classwork: Implement mutual exclusion for two threads.

Classwork: Can we allow either **S1** or **S2** to happen first?

Initially, flag == false.

```
while (!flag) ;
S1;
```

```
S2;
flag = true;
```

Synchronization

- **Control + data flow**
- Atomics
- Barriers
- ...

Classwork: Implement mutual exclusion for two threads.

Classwork: Can we allow either **S1** or **S2** to happen first?

Initially, flag could be true or false.

```
while (!flag) ;
S1;
flag = false;
```

```
while (flag) ;
S2;
flag = true;
```

It helps to abstract this out into an API.

Assumptions:
- Reading of and writing to flag is atomic (seemingly one step).
- Both the threads execute their codes.
- flag is volatile.

Mutual Exclusion: 2 threads

- Let's implement **lock()** and **unlock()** methods.
- The methods should be the same for both the threads (can have threadid == 0, etc.).
- Should use only control + data flow.
- **Desirable Properties**
  - Deadlock-free / Livelock-free
  - Starvation-free (bounded wait)
  - Progress (when alone, thread should enter CS)
  - Fair (threads should acquire locks in the order of their expression of interest to enter CS)

Mutual Exclusion: 2 threads

v1

- Thread ids are 0 and 1.
- flag[] is volatile and atomically written.

```
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    while (flag[other])
        ;
unlock():
    flag[tid] = false;
```

- Mutual exclusion is guaranteed (if volatile).
- May lead to deadlock.
- If one thread runs before the other, all goes well.

Mutual Exclusion: 2 threads

v1

- Thread ids are 0 and 1.
- flag[] is volatile and atomically written.

```
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    while (flag[other])
        ;
unlock():
    flag[tid] = false;
```

- If flag is not volatile, then the update (Line 3) may never reach the other thread (caching).
- This can lead to both threads executing CS.

Mutual Exclusion: 2 threads

v2

- Thread ids are 0 and 1.
- victim needs to be volatile.

```
volatile int victim;
lock:
    me = tid;
    victim = me;
    while (victim == me)
        ;
unlock():
    victim = me;
```

- Mutual exclusion is guaranteed.
- May not guarantee progress.
- If threads repeatedly take locks, all goes well.

Peterson's Lock

v3

```
volatile bool flag[2];
volatile int victim;
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    victim = me;
    while (flag[other] &&
             victim == me)
       ;
unlock():
       flag[tid] = false;
```

- Mutual exclusion is guaranteed.
- Does not lead to deadlock.
- The algorithm is starvation-free.

- flag indicates if a thread is interested.
- victim = me is *pehle aap*.

What about N threads?

Peterson's Lock

```
volatile bool flag[2];
volatile int victim;
lock:
    me = tid;
    other = 1 – me;
    flag[me] = true;
    victim = me;
    while (flag[other] &&
              victim == me)
        ;
unlock():
        flag[tid] = false;
```

```
flag[me] = true;
victim = me;
while (flag[other] &&
    victim == me)
```

✓

```
victim = me;
flag[me] = true;
while (flag[other] &&
    victim == me)
```

```
victim = me;
flag[me] = true;
while (victim == me &&
    flag[other])
```

```
flag[me] = true;
victim = me;
while (victim == me &&
    flag[other])
```

Peterson's Lock

Time

Thread 0	Thread 1
	victim = 1
victim = 0	
flag[0] = true	
while (flag[1] ...	
… enters CS	
	flag[1] = true
	while (flag[0] &&
	victim == 1)
	… enters CS

```
flag[me] = true;
victim = me;
while (flag[other] &&
      victim == me)
```
✓

```
victim = me;
flag[me] = true;
while (flag[other] &&
      victim == me)
```
✗

```
victim = me;
flag[me] = true;
while (victim == me &&
      flag[other])
```
✗

```
flag[me] = true;
victim = me;
while (victim == me &&
      flag[other])
```
✓

Bakery Algorithm

- Devised by Lamport
- Works with N threads.
- Maintains FCFS using ever-increasing numbers.

```
bool flag[N];  // false
int label[N];  // 0
lock:
    me = tid;
    flag[me] = true;
    label[me] = 1 + max(label);
    while (∃k != me: flag[k] &&
            (label[k], k) < (label[me], me))
      ;
```

```
flag[tid] = false;
```

- The code works in absence of caches.
- In presence of caches, mutual exclusion is not guaranteed.
- There are variants to address the issue.

max is not atomic.

Bakery Algorithm: GPU?

- Across warps is similar to CPU.
- What happens within warp-threads?
- Threads get the same label, **<** prioritizes.

```
bool flag[N];  // false
int label[N];  // 0
lock:                                          unlock():
    me = tid;                                      flag[tid] = false;
    flag[me] = true;
    label[me] = 1 + max(label);
    while (∃k != me: flag[k] &&
            (label[k], k) < (label[me], me))
      ;
```

max is not atomic.

Bakery Algorithm: GPU?

- Across warps is similar to CPU.
- What happens within warp-threads?
- Threads get the same label, **<** prioritizes.

- On GPUs, locks are usually prohibited.
- High spinning cost at large scale.
- But locks are feasible!
- Locks can also be implemented using atomics.

Synchronization

- Control + data flow
- Atomics
- Barriers
- ...

atomics

- Atomics are primitive operations whose effects are visible either none or fully (never partially).
- Need hardware support.
- Several variants: atomicCAS, atomicMin, atomicAdd, ...
- Work with both global and shared memory.

atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<2, 1>>>(x);
```

After dkernel completes, what is the value of x[0]?

Classwork: What if the kernel configuration is <<<1, 2>>>?

++x[0] is equivalent to:

```
Load x[0], R1
Increment R1
Store R1, x[0]
```

Time ↓

Thread 1	Thread 2
Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

Final value stored in x[0] could be 1 (rather than 2).
What if x[0] is split into multiple instructions? What if there are more threads?

Atomics in ATMs

Twins at ATMs

Twin withdraws 1000 rupees.

System executes the steps:

- Check if balance is >= 1000.
- If yes, reduce balance by 1000 and give cash to the user.
- Otherwise, issue error.

Twins may be able to get 2000 rupees!
The balance can be negative!

Time

Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

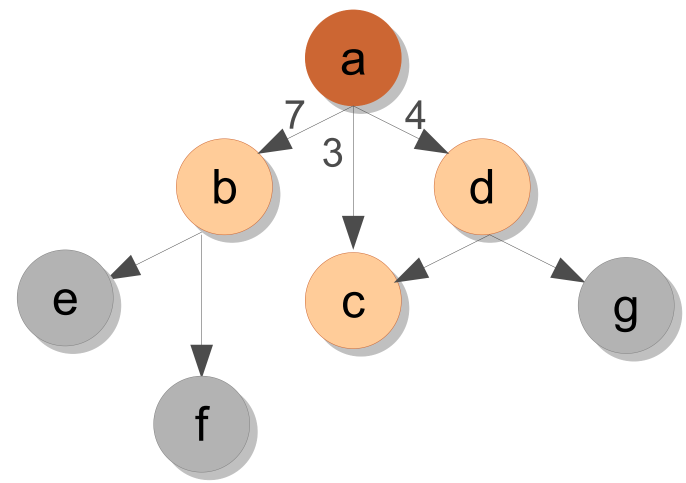
atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<2, 1>>>(x);
```

- Ensure all-or-none behavior.
  - e.g., atomicInc(&x[0], ...);
- **dkernel**<<<K1, K2>>> would ensure x[0] to be incremented by exactly K1*K2 – irrespective of the thread execution order.
  - When would this effect be visible?

Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.




```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = …
    for each n in g.allneighbors(id) {    // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;  atomicMin(&dist[n], altdist);
}   }   }
```

AtomicCAS

- Syntax: oldval = atomicCAS(&var, x, y);

- **Typical usecases:**
  - *Locks*: critical section processing
  - *Single*: Only one arbitrary thread executes the block.
  - Other atomic variants

Classwork: Implement *lock* with *atomicCAS*.

Lock using atomicCAS

Does this work?

```
atomicCAS(&lockvar, 0, 1);
```

Does not ensure mutual exclusion.

Then how about

```
if (atomicCAS(&lockvar, 0, 1) == 0)
    // critical section
```

Does not block other threads.

Make the above code blocking.

```
do {
    old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);
```

Correct code?

Lock using atomicCAS

- The code works on CPU.
- It also works on GPU across warps.
- But it hangs for threads belonging to the same warp.
  - When one warp-thread acquires the lock, it waits for other warp-threads to reach the instruction just after the do-while.
  - Other warp-threads await this successful thread in the do-while.

```
do {
    old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);
```

Correct code?

Lock using atomicCAS

```
do {
    old = atomicCAS(&lockvar, 0, 1);
} while (old != 0);

// critical section

lockvar = 0;     // unlock
```

On CPU

```
do {
    old = atomicCAS(&lockvar, 0, 1);
    if (old == 0) {
        // critical section
        lockvar = 0;     // unlock
    }
} while (old != 0);
```

On GPU

Classwork: Implement *single* with *atomicCAS*.

Single using atomicCAS

```
if (atomicCAS(&lockvar, 0, 1) == 0)
    // single section
```

Important not to set lockvar to 0 at the end of the single section.

What is the output?

```
#include <stdio.h>
#include <cuda.h>

__global__ void k1(int *gg) {
        int old = atomicCAS(gg, 0, threadIdx.x + 1);
        if (old == 0) {
                printf("Thread %d succeeded 1.\n", threadIdx.x);
        }
        old = atomicCAS(gg, 0, threadIdx.x + 1);
        if (old == 0) {
                printf("Thread %d succeeded 2.\n", threadIdx.x);
        }
        old = atomicCAS(gg, threadIdx.x, -1);
        if (old == threadIdx.x) {
                printf("Thread %d succeeded 3.\n", threadIdx.x);
        }
}
int main() {
        int *gg;
        cudaMalloc(&gg, sizeof(int));
        cudaMemset(&gg, 0, sizeof(int));
        k1<<<2, 32>>>(gg);
        cudaDeviceSynchronize();

        return 0;
}
```

- Some thread out of 64 updates gg to its threadid+1.
- Warp threads do not execute atomics together! That is also done sequentially.
- Irrespective of which thread executes the first atomicCAS, no thread would see gg to be 0. Hence second printf is not executed at all.
- If gg was updated by some thread 0..30, then the corresponding thread with id 1..31 from either of the blocks would update gg to -1, and execute the third printf.
- Otherwise, no one would update gg to -1, and no one would execute the third printf.

- On most executions, you would see the output to be that thread 0 would execute the first printf, and thread 1 would execute the third printf.

Classwork

- Each thread adds elements to a worklist.
  - e.g., next set of nodes to be processed in SSSP.
  - worklist is implemented as an array.
- Initially, assume that each thread adds exactly K elements.
- Later, relax the constraint.

atomic-worklist.cu

Convolution Filter

- Each output cell contains weighted sum of input data element and its neighbors. The weights are specified as a filter (array).
- The idea can be applied in multiple dimensions.
- We will work with 1D convolution and odd filter size.

convolution.cu **Implement convolution.**

input	1	2	3	4	5	6	7	8
filter		3	4	5	4	3		
filter output		6	12	20	20	18		
output	1	2	3	76	5	6	7	8

1	2	3	4	5	6	7	8
		3	4	5	4	3	
		9	16	25	24	21	
22	38	57	76	95	114	106	86

Source: Prof. Marco Bertini's slides

Synchronization

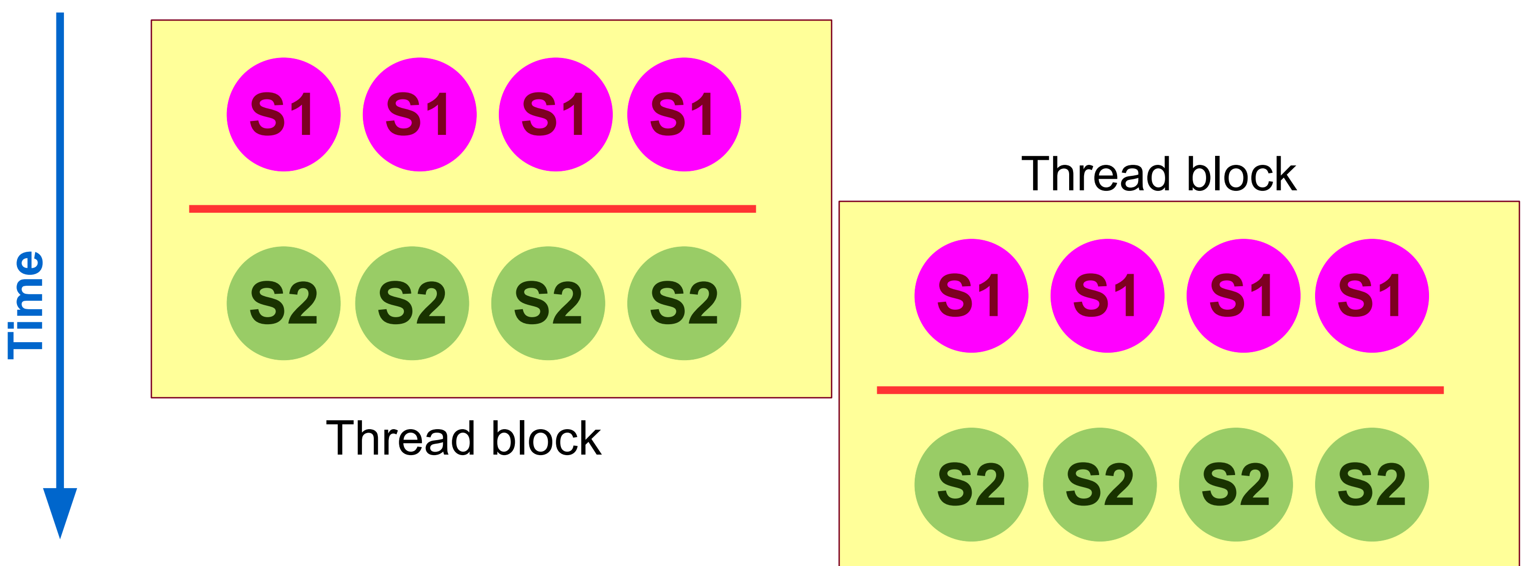
- Control + data flow
- Atomics
- Barriers
- ...

Barriers

- A barrier is a program point where all threads need to reach before any thread can proceed.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- ~~There is no explicit global barrier supported in CUDA~~. *grid*.sync() is now supported (from CUDA 9).
- Threads in a thread-block can synchronize using __syncthreads().
- How about barrier within warp-threads?

Barriers

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    vector[id] = id;        S1
    __syncthreads();
    if (id < vectorsize - 1 && vector[id + 1] != id + 1)   S2
        printf("syncthreads does not work.\n");
}
```



Barriers

- __syncthreads() is not only about control synchronization, it also has data synchronization mechanism.
- It performs a **memory fence** operation.
  - A memory fence ensures that the writes from a thread are made visible to other threads.
  - __syncthreads() executes a fence for all the block-threads.
- There is a separate __threadfence_block() instruction also. Then, there is __threadfence().
- *[In general]* A fence does not ensure that other thread will read the updated value.
  - This can happen due to caching.
  - The other thread needs to use volatile data.
- *[In CUDA]* a fence applies to both read and write.

Classwork

- Write a CUDA kernel to find maximum over a set of elements, and then let thread 0 print the value in the same kernel.
- Each thread is given work[id] amount of work. Find average work per thread and if a thread's work is above average + K, push extra work to a worklist.
  - This is useful for load-balancing.
  - Also called work-donation.

Taxonomy of Synchronization Primitives

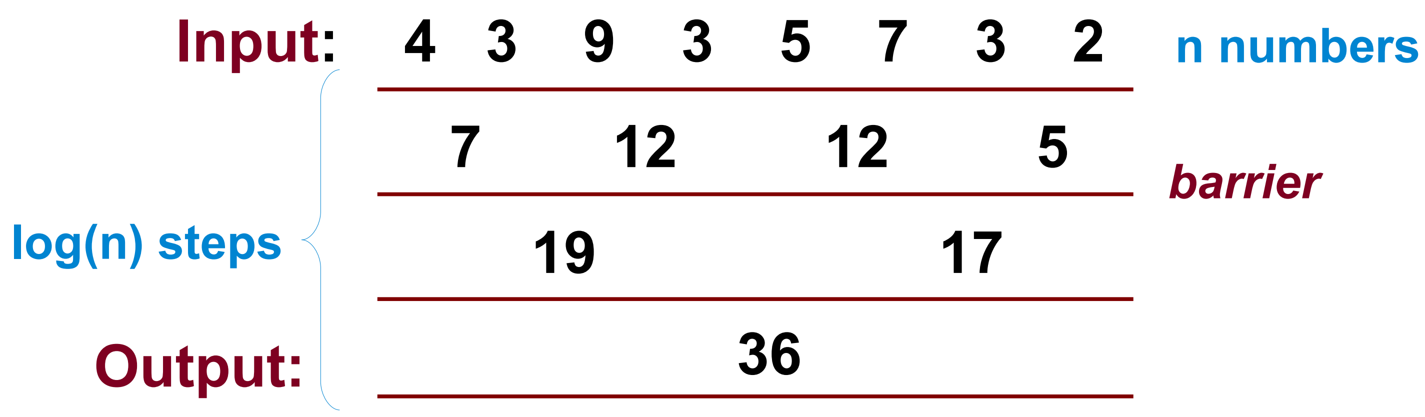
Primitive	Control-sync	Data-sync
__syncthreads __syncwarp	Block Warp	Block Warp
atomic	--	Block for shared All for global
__threadfence_block	--	block
__threadfence	--	All
Global barrier	All	All
while loop	Customizable	– (but not useful without data-synchronization)
volatile	--	All

Reductions

- Converting a set of values to few values (typically 1)
- Computation must be *reducible*.
  - Must satisfy associativity property (a.(b.c) = (a.b).c).
  - Min, Max, Sum, XOR, ...
- Can be often implemented using atomics
  - atomicAdd(&sum, a[i]);
  - atomicMin(&min, a[i]);
  - But adds sequentiality.
- Reductions allow improving parallelism.
  - Different from reductions in OpenMP and MPI.

Reductions

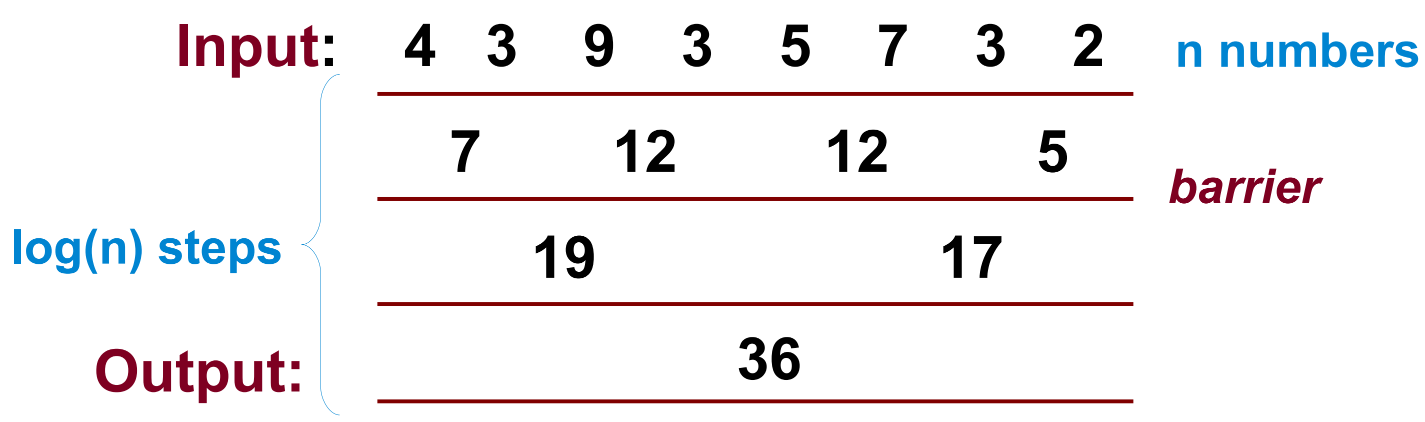
- Converting a set of values to few values (typically 1)
- Computation must be *reducible*.
  - Must satisfy associativity property (a.(b.c) = (a.b).c).
  - Min, Max, Sum, XOR, ...
- Complexity measures



Classwork: Write the reduction code.

Reductions

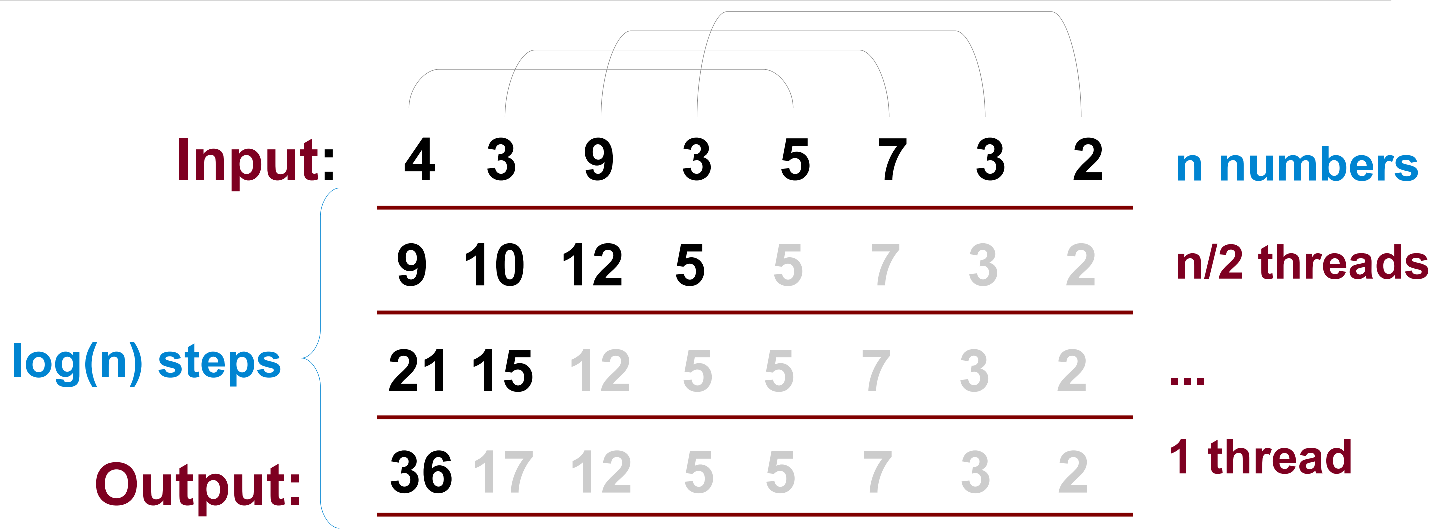
```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```



Reductions

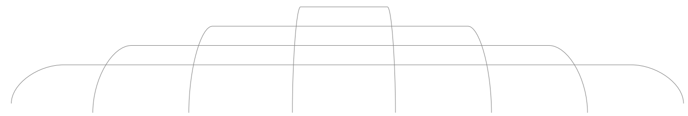
n must be a power of 2

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```



Reductions

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

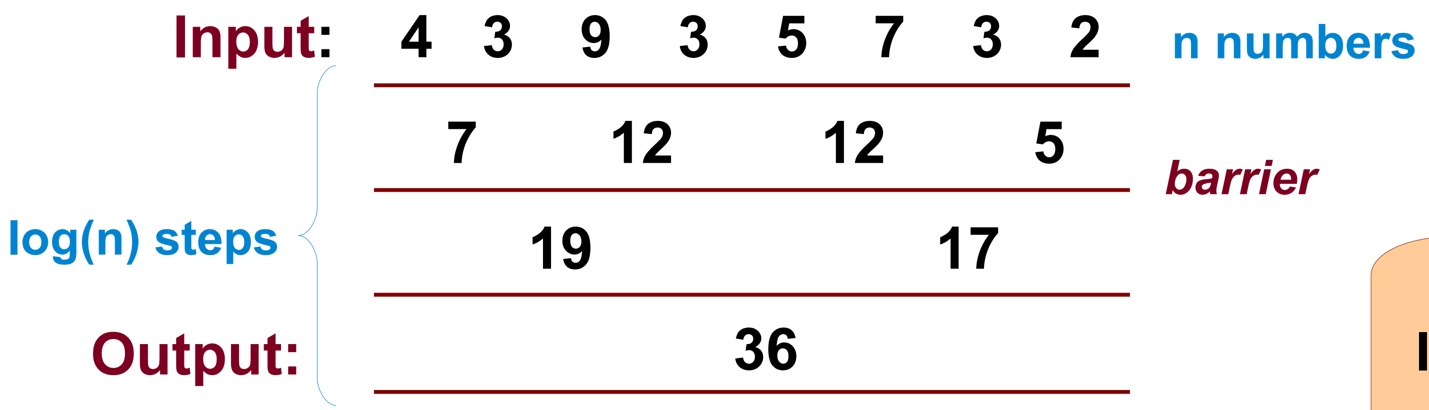


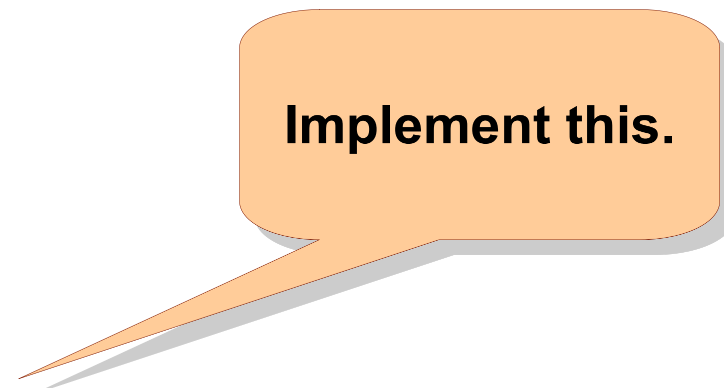
Write the reduction as: **4 3 9 3 5 7 3 2**

```
for (int off = n/2; off; off /= 2) {
    if (threadIdx.x < off) {
        a[threadIdx.x] += a[2 * off - threadIdx.x - 1];
    }
    __syncthreads();
}
```

Reductions

- Let's go back to our first diagram.



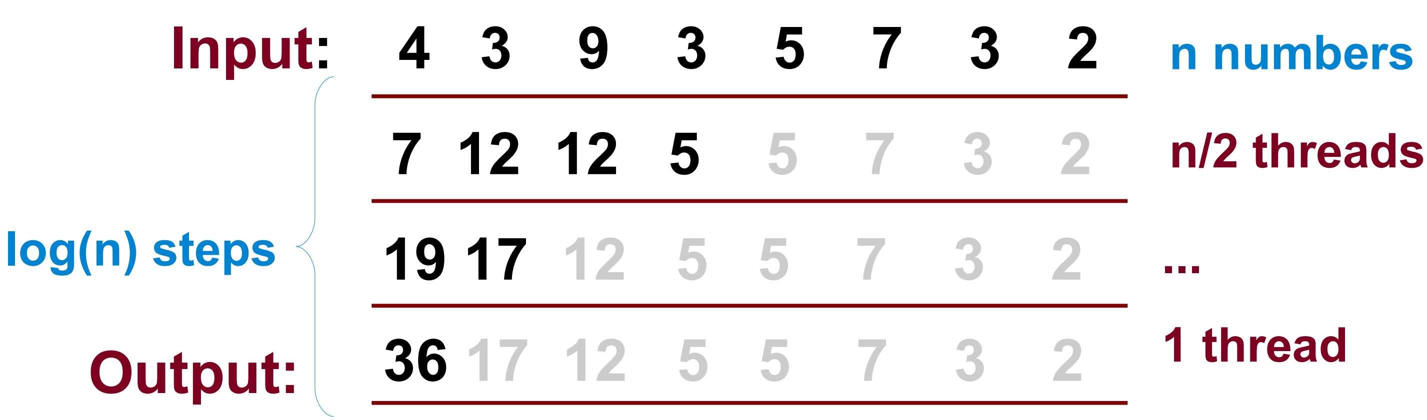


- This can be implemented as

Reductions

- A challenge in the implementation is:
  - a[1] is read by thread 0 and written by thread 1.
  - This is a data-race.
  - Can be resolved by separating R and W.
  - This requires another barrier and a temporary.

Homework: Try this out.



Classwork

- Assuming each a[i] is a character, find a concatenated string using reduction.
- String concatenation cannot be done using a[i] and a[i + n/2], but computing sum was possible; why?
- What other operations can be cast as reductions?

Prefix Sum

- Imagine threads wanting to push work-items to a central worklist.
- Each thread pushes different number of work-items.
- This can be computed using atomics or prefix sum (also called as *scan*).

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**4**	**7**	**16**	**19**	**24**	**31**	**34**	**36**
OR								
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**34**

Classwork: Write the prefix-sum code.

Prefix Sum

v1

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

This is reduction.

Number of threads should be initially O(n).

v2

```
for (int off = n; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

Array index is incorrect.

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 34 36
OR
Output: 0 4 7 16 19 24 31 34

Prefix Sum

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
            a[threadIdx.x] += a[threadIdx.x + (n - off)];
        }
        __syncthreads();
}
```

v3

Smaller indices are written to more frequently.

```
for (int off = 0; off < n; off *= 2) {
        if (threadIdx.x > off) {
            a[threadIdx.x] += a[threadIdx.x - off];
        }
        __syncthreads();
}
```

v4

Infinite loop?

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 34 36
OR
Output: 0 4 7 16 19 24 31 34

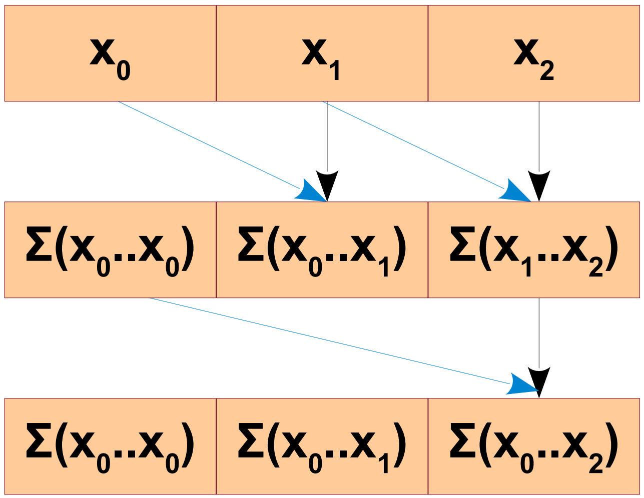
Prefix Sum

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$	$\Sigma(x_0..x_7)$

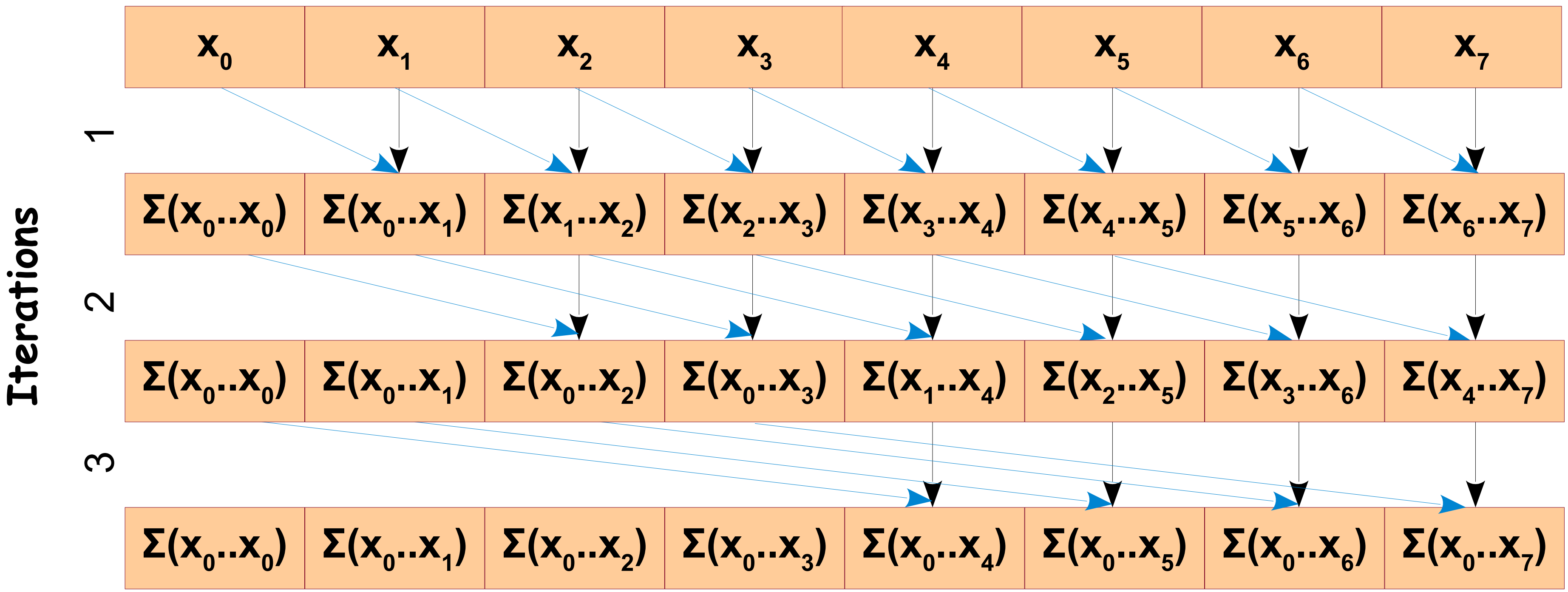
Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 34 36
OR
Output: 0 4 7 16 19 24 31 34

Prefix Sum



Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 34 36
OR
Output: 0 4 7 16 19 24 31 34

Prefix Sum



Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 34 36
OR
Output: 0 4 7 16 19 24 31 34

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
    if (threadIdx.x > off) {
        a[threadIdx.x] += a[threadIdx.x - off];
    }
    __syncthreads();
}
```

Datarace

v5

```
for (int off = 1; off < n; off *= 2) {
    if (threadIdx.x > off) {
        tmp = a[threadIdx.x – off];
        __syncthreads();
        a[threadIdx.x] += tmp;
    }
    __syncthreads();
}
```

Separating R and W in time

v6

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
        if (threadIdx.x >= off) {
                tmp = a[threadIdx.x - off];
        }
        __syncthreads();

        if (threadIdx.x >= off) {
                a[threadIdx.x] += tmp;
        }
        __syncthreads();
}
```

v7

Can this be done with single syncthreads()?

Prefix Sum with One Barrier

```
for (int off = 1; off < n; off *= 2) {
	if (tid >= off) {
		int val = tid % (2 * off);
		if (val >= off)
			a[tid] += a[tid – val + off - 1];
	}
	__syncthreads();
}
```

by Surya

```
__global__ void prefixSum(int *a){
	int tid = threadIdx.x;	int tmpId = tid;	int idx = 0;
	if (tid < N) {
		for (int off = 1; off < N; off *= 2) {
			int bit = tmpId & 1;
			tmpId = tmpId >> 1;
			if (bit){
			   int step = tid / off;
			   a[tid] += a[step * off + ((1<<idx) & (1<<idx + 1)) - 1];
			}
			idx += 1;
			__syncthreads();
 }	}	}
```

by Prasanna

Application of Prefix Sum

- Assuming that you have the prefix sum kernel, insert elements into the worklist.
  - Each thread inserts nelem[tid] many elements.
  - The order of elements is not important.
  - You are forbidden to use atomics.
- Computing cumulative sum
  - Histogramming
  - Area under the curve
  - Fenwick Tree (Binary Indexed Tree)

nelem

Input:	**4**	**3**	**9**	**3**	**5**	**7**	**3**	**2**
Output:	**0**	**4**	**7**	**16**	**19**	**24**	**31**	**33**

Start offset

Global Barrier

- Barrier across all the GPU threads.
- Useful to store transient data, partial computations, shared memory usage, etc.
- Can be readily implemented using atomics.

```
atomicInc(&counter, ∞);
while (counter != blockDim.x * gridDim.x)
        ;
```

Classwork: Implement global barrier with atomics.

Global Barrier

- Can use hierarchical synchronization for efficiency.
  - __syncthreads() within each thread block.
  - Representative from each block then synchronizes using atomics.

```
__syncthreads();
if (threadIdx.x == 0) {
        atomicInc(&counter, ∞);
        while (counter != gridDim.x)
                ;
}
__syncthreads();
```

Show that removing either of the barriers breaks the global barrier.

Classwork: Implement global barrier with atomics and syncthreads.

Global Barrier

Without	Warp1 in Block1	Warp2 in Block1	Warp3 in Block2	Warp4 in Block2
Second syncthreads	atomicInc	S2	S1	S1
First syncthreads	atomicInc	syncthreads	atomicInc	Slow
	S2	S2	syncthreads	S1

S1
Barrier
S2

```
__syncthreads();
if (threadIdx.x == 0) {
    atomicInc(&counter, ∞);
    while (counter != gridDim.x)
        ;
}
__syncthreads();
```

Show that removing either of the barriers breaks the global barrier.

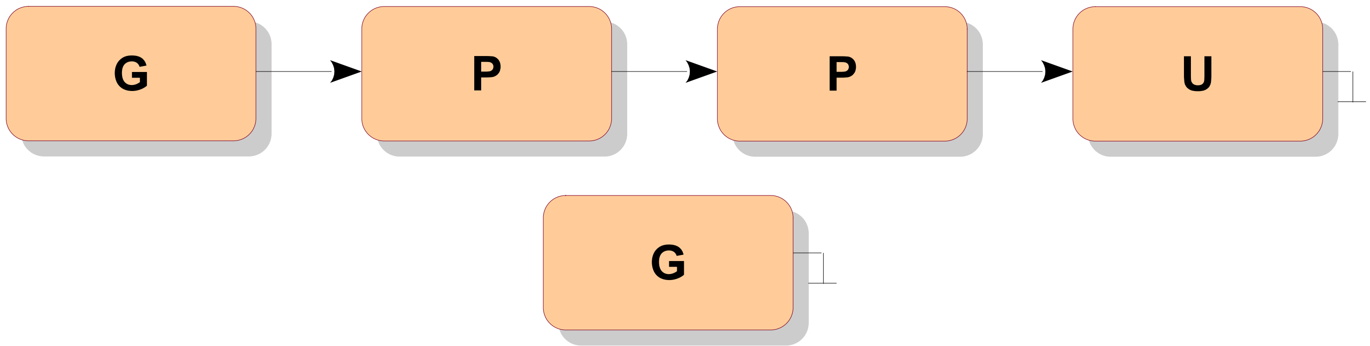
Classwork: Implement global barrier with atomics and syncthreads.

Global Barrier

- For a single global barrier, the previous method works.
- When barrier is called multiple times, we need to be careful:
  - Counter may interfere across barriers; look for the fastest thread moving to the second barrier.
  - Using two different counters for odd / even would help.
  - Guarantees that i^{th} barrier does not interfere with $(i+2)^{th}$ barrier.

Concurrent Data Structures

- Array
  - atomics for index update
  - prefix sum for coarse insertion
- Singly linked list
  - insertion
  - deletion [marking, actual removal]



Concurrent Data Structures

Type definition

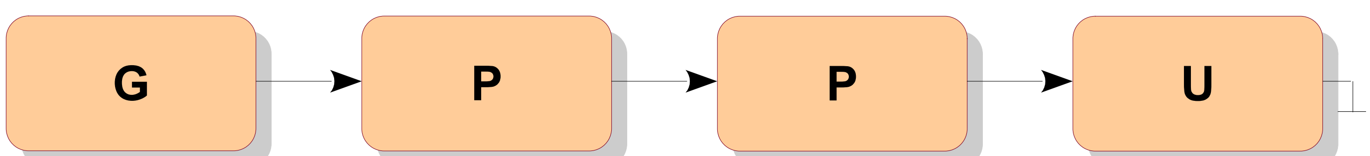
```
struct node {
    char item;
    struct node *next;
};
```

Sequential insert

```
G2->next = P2;
P1->next = G2;
```

- In the concurrent setting, the exact order of insertions is not expected.
- Elements can be inserted in any order.
- So, w.l.o.g. we assume elements being added at the head.

head



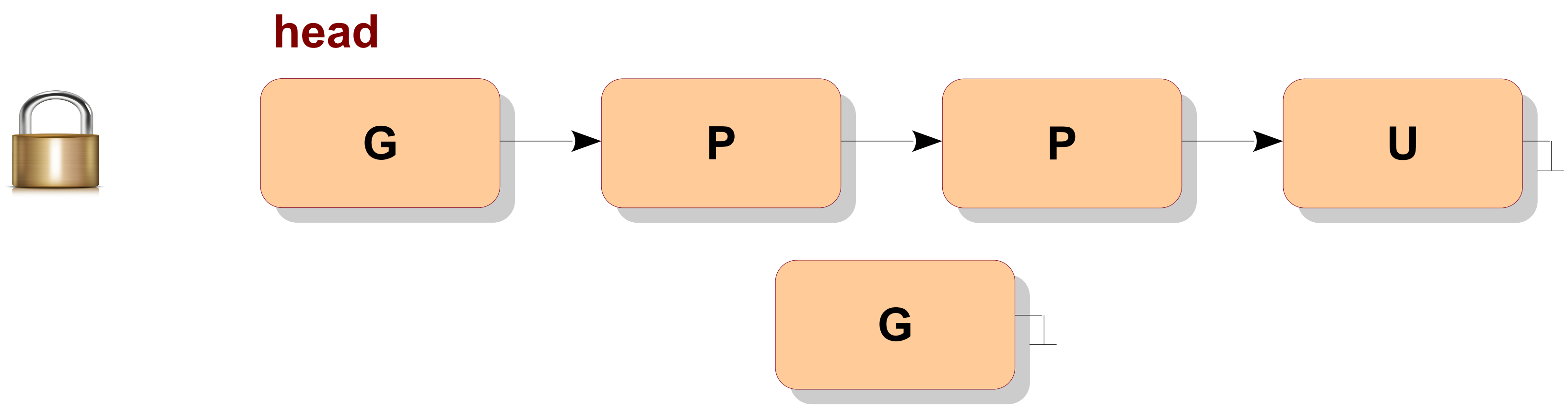
Classwork: Write the code to insert G2 at head.


```
G2->next = head;
head = G2;
```

Concurrent Linked List

Solution 1: Keep a lock with the list.

- Coarse-grained synchronization
- Low concurrency / sequential access
- Easy to implement
- Easy to argue about correctness



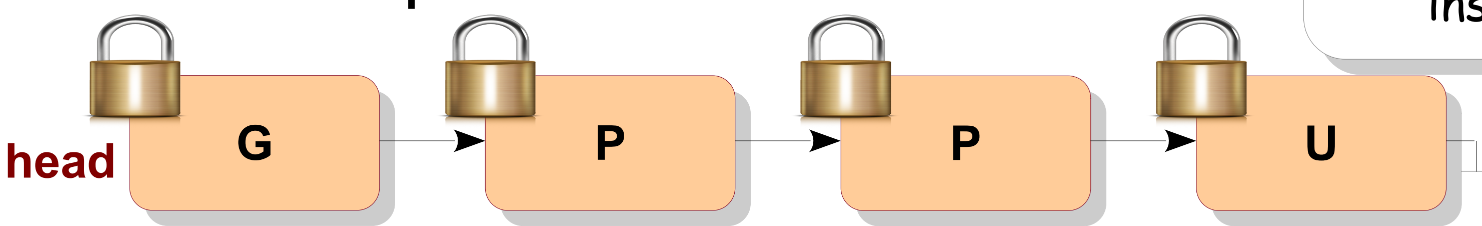
Concurrent Linked List

Solution 2: Keep a lock with each node.

- Fine-grained synchronization
- Better concurrency
- Moderately difficult to implement, need to finalize the supported operations
- Difficult to argue about correctness when multiple nodes are involved

Classwork: Implement insert().

Classwork: Check if two concurrent inserts work.



Concurrent Linked List

```
void insert(Node *naya) {
    head→lock();
    naya→next = head;
    head = naya;
    head→unlock();
}
```

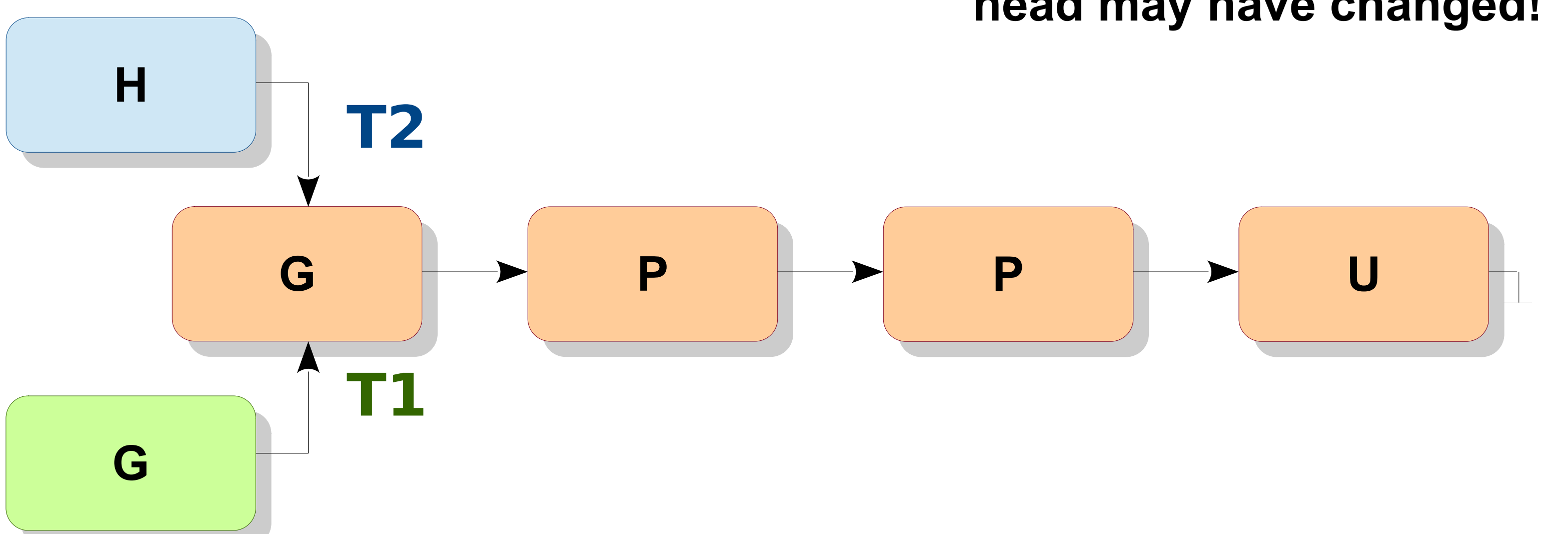
Danger

```
void insert(Node *naya) {
    ptr = head;
    ptr→lock();
    naya→next = head;
    head = naya;
    ptr→unlock();
}
```

Danger

head changes.

By the time, ptr→lock happens, head may have changed!



Concurrent Linked List

```
void insert(Node *naya) {
    head→lock();
    ptr = head;
    naya→next = head;
    head = naya;
    ptr→unlock();

}
```

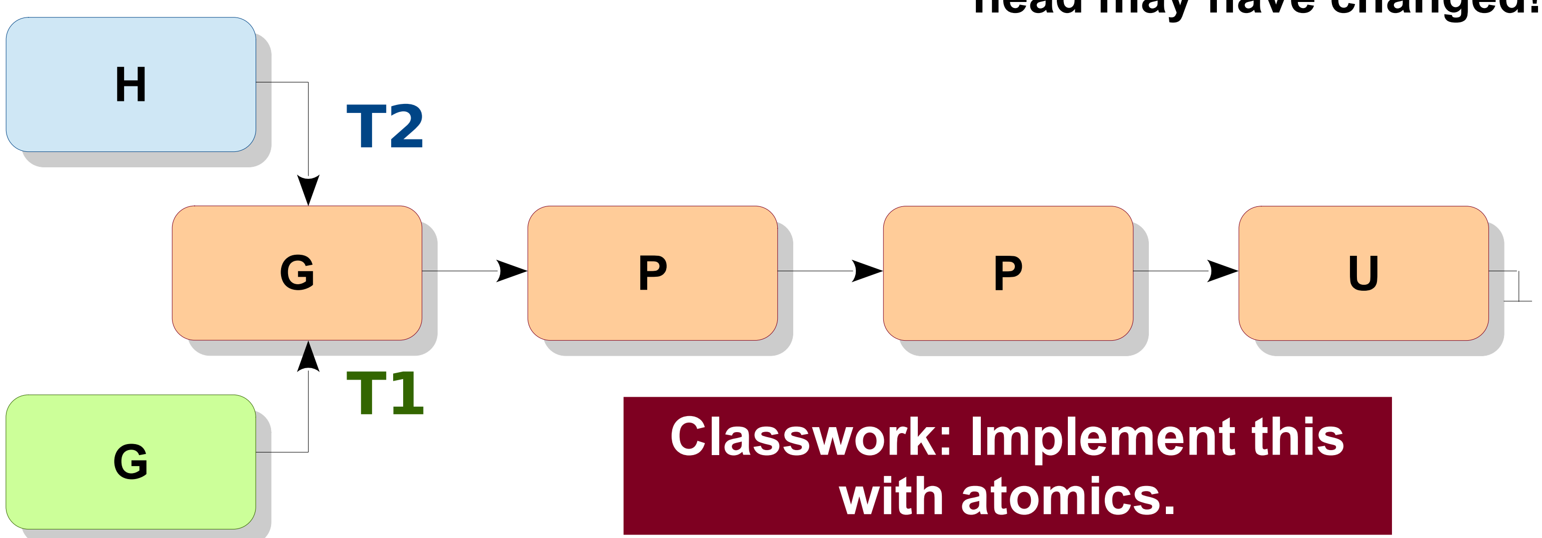
✓

Lock head first, then copy.

```
void insert(Node *naya) {
    ptr = head;
    ptr→lock();
    naya→next = head;
    head = naya;
    ptr→unlock();
}
```

Danger

By the time, ptr→lock happens, head may have changed!



Classwork: Implement this with atomics.

Concurrent Linked List

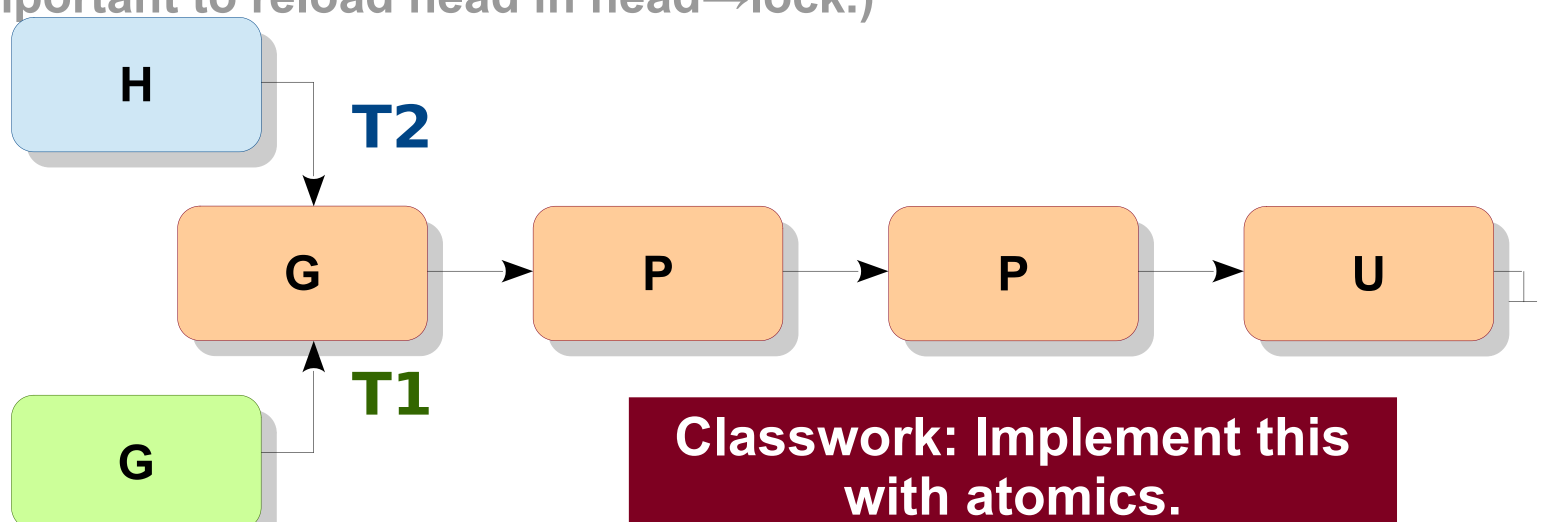
```
void insert(Node *naya) {
        head→lock();
        ptr = head;
        naya→next = head;
        head = naya;
        ptr→unlock();

}
```

Lock head first, then copy.

(It is important to reload head in head→lock.)

```
void insert(Node *naya) {
        head→lock();
        naya→next = head→next;
        head→next = naya;
        head→unlock();
}
```

Insert naya as the second node.



Classwork: Implement this with atomics.

Source: linkedlist-add.cu

CPU-GPU Synchronization

- While GPU is busy doing work, CPU may perform useful work.
- If CPU-GPU collaborate, they require synchronization.

Classwork: Implement
a functionality to print sequence 0..10.
CPU prints even numbers,
GPU prints odd.

CPU-GPU Synchronization

```
#include <cuda.h>
#include <stdio.h>

__global__ void printk(int *counter) {
	++*counter;			// in general, this can be arbitrary processing
	printf("\t%d\n", *counter);
}
int main() {
	int hcounter = 0, *counter;

	cudaMalloc(&counter, sizeof(int));

	do {
		printf("%d\n", hcounter);
		cudaMemcpy(counter, &hcounter, sizeof(int), cudaMemcpyHostToDevice);
		printk <<<1, 1>>>(counter);
		cudaMemcpy(&hcounter, counter, sizeof(int), cudaMemcpyDeviceToHost);
	} while (++hcounter < 10); // in general, this can be arbitrary processing

	return 0;
}
```

Pinned Memory

- Typically, memories are pageable (swappable).
- CUDA allows to make host memory pinned.
- CUDA allows direct access to pinned host memory from device.
- cudaHostAlloc(&pointer, size, 0);

Last parameter is flag. Memory is initialized to zero by default.

Classwork: Implement
the same functionality to print sequence 0..10.
CPU prints even numbers,
GPU prints odd.

Pinned Memory

```
#include <cuda.h>
#include <stdio.h>

__global__ void printk(int *counter) {
	++*counter;
	printf("\t%d\n", *counter);
}
int main() {
	int *counter;

	cudaHostAlloc(&counter, sizeof(int), 0);

	do {
		printf("%d\n", *counter);
		printk <<<1, 1>>>(counter);
		cudaDeviceSynchronize();
		++*counter;
	} while (*counter < 10);

	cudaFreeHost(counter);
	return 0;
}
```

No cudaMempcy!

Classwork: Can we avoid repeated kernel calls?

Persistent Kernels

```
__global__ void printk(int *counter) {
        do {
                while (*counter % 2 == 0)  ;      // Line 2
                printf("\t%d\n", *counter);       // Line 3
                ++*counter;                       // Line 4
        } while (*counter < 10);                  // Line 5
}
int main() {
        int *counter;

        cudaHostAlloc(&counter, sizeof(int), 0);
        printk <<<1, 1>>>(counter);

        do {
                while (*counter % 2 == 1)  ;
                printf("%d\n", *counter);
                ++*counter;                   // Line 1
        } while (*counter < 10);              // Line 6

        cudaFreeHost(counter);
        return 0;
}
```

Is it possible that this program does not print 10?

Consider Line 1 (increments to 9) then Lines 2, 3, 4, 5, then Line 6.

Hierarchy of Barriers

- Warp: SIMD / __syncwarp
- Block: __syncthreads
- Grid: Global Barrier
- CPU-GPU: cudaDeviceSynchronize

Who will use CPU-GPU for printing odd-even numbers?

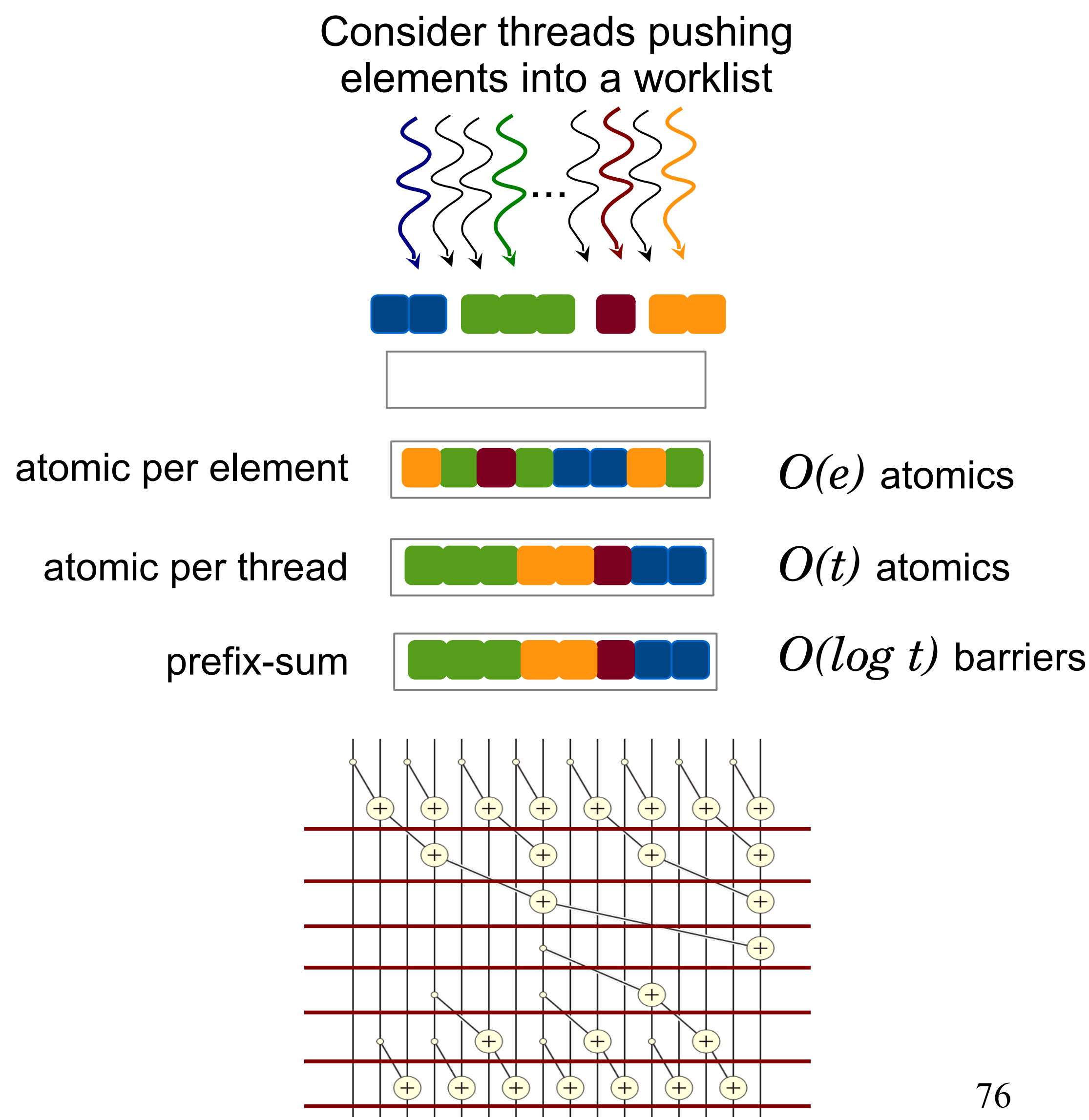
- Increment is replaceable by arbitrary computation.
  - A matrix needs three computation steps. Each step can be parallelized on CPU and GPU. The matrix can be divided accordingly.
  - A graph can be partitioned. CPU and GPU compute shortest paths on different partitions. Their results are merged. Then iterate similarly.
  - …
- Very useful when data does not fit in GPU memory (e.g., billions of data items, twitter graph, …)
- Useful when CPU prepares data for the next GPU iteration.

Synchronization Patterns

- Common situations that demand the same way of synchronizing
- Useful in applications from various domains
- Can be optimized, and applied to all
- Can be further optimized by customizing to an application

Barrier-based Synchronization

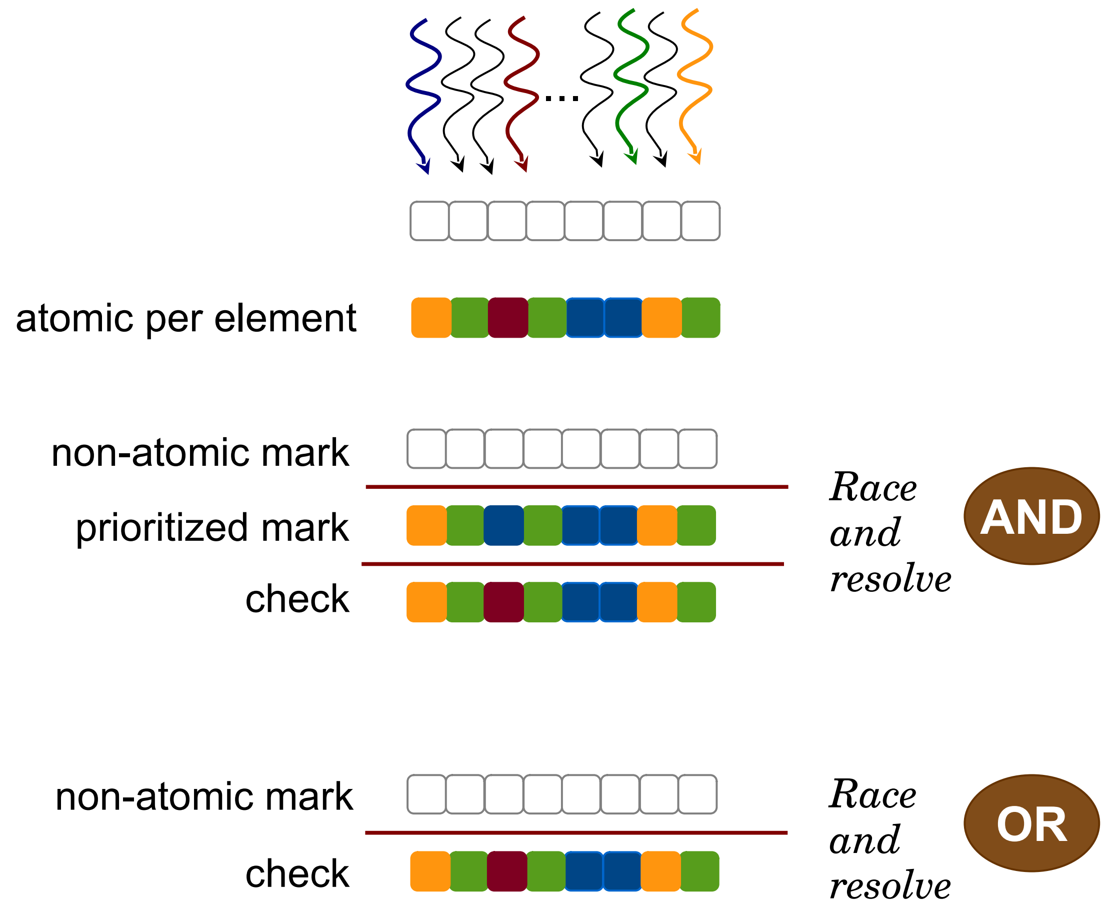
- **Disjoint accesses**
- Overlapping accesses
- Benign overlaps



Barrier-based Synchronization

- Disjoint accesses
- → **Overlapping accesses**
- Benign overlaps

Consider threads trying to own a set of elements



e.g., for owning cavities in Delaunay mesh refinement

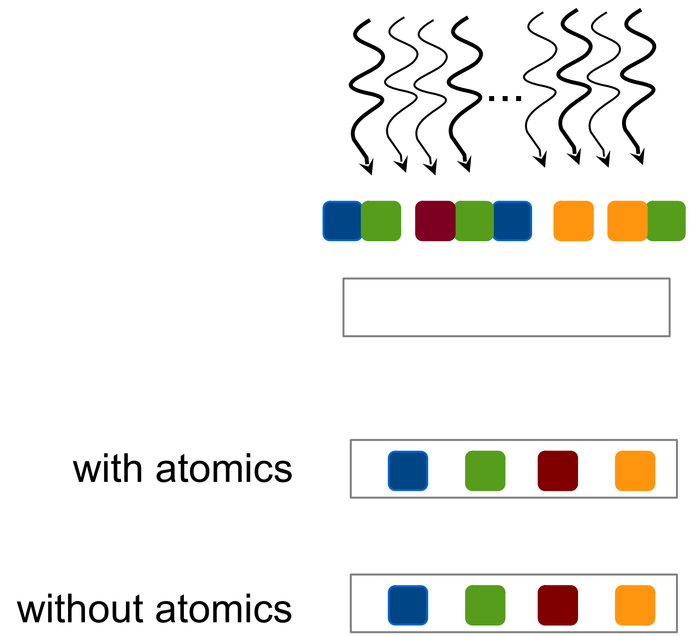
e.g., for inserting unique elements into a worklist

Barrier-based Synchronization

- Disjoint accesses
- Overlapping accesses

➔ **Benign overlaps**

Consider threads updating shared variables to the same value

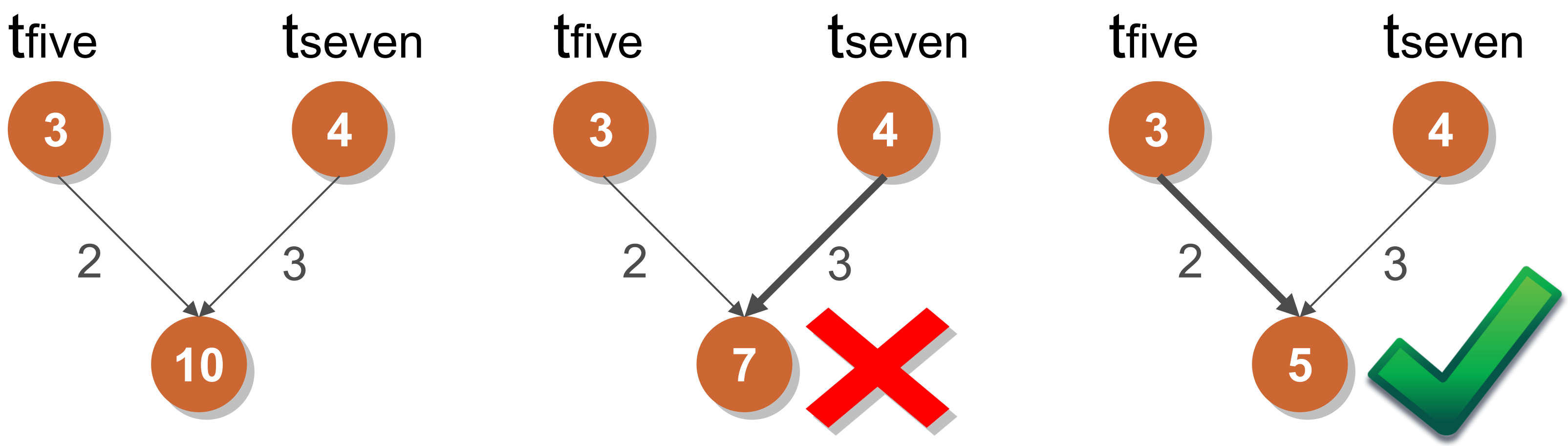


e.g., level-by-level breadth-first search

Exploiting Algebraic Properties

- **Monotonicity**
- Idempotency
- Associativity

Consider threads updating distances in shortest paths computation



Atomic-free update

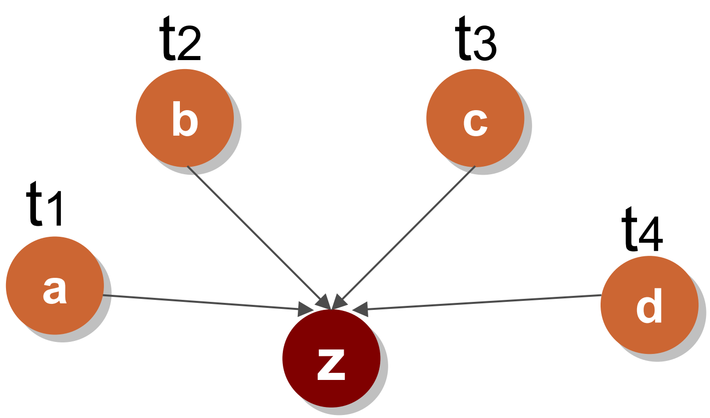
Lost-update problem

Correction by topology-driven processing, exploiting monotonicity

Exploiting Algebraic Properties

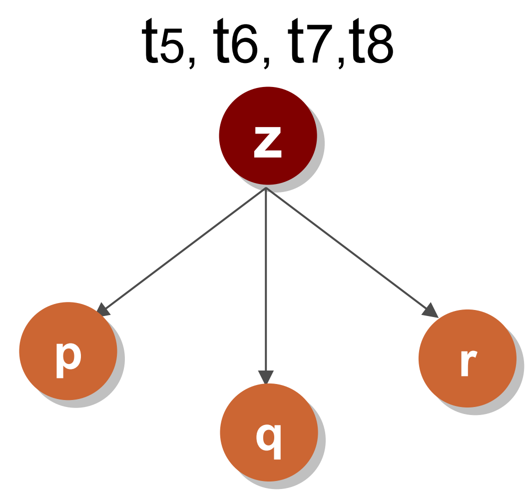
- Monotonicity
- → **Idempotency**
- Associativity

Consider threads updating distances in shortest paths computation



Update by multiple threads

Multiple instances of a node in the worklist

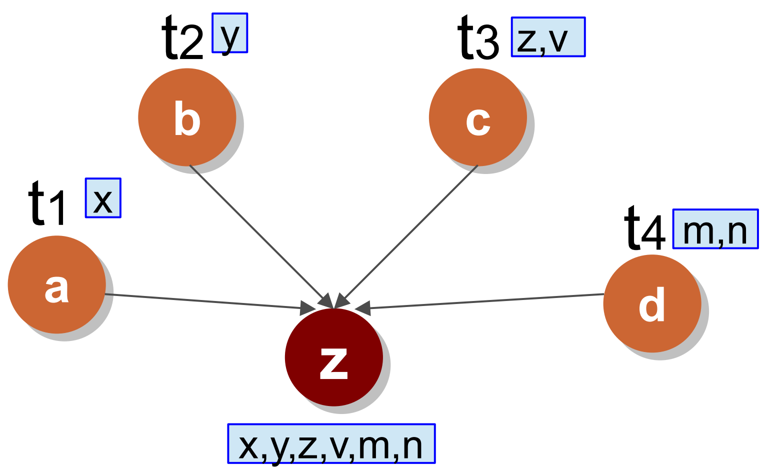


Same node processed by multiple threads

Exploiting Algebraic Properties

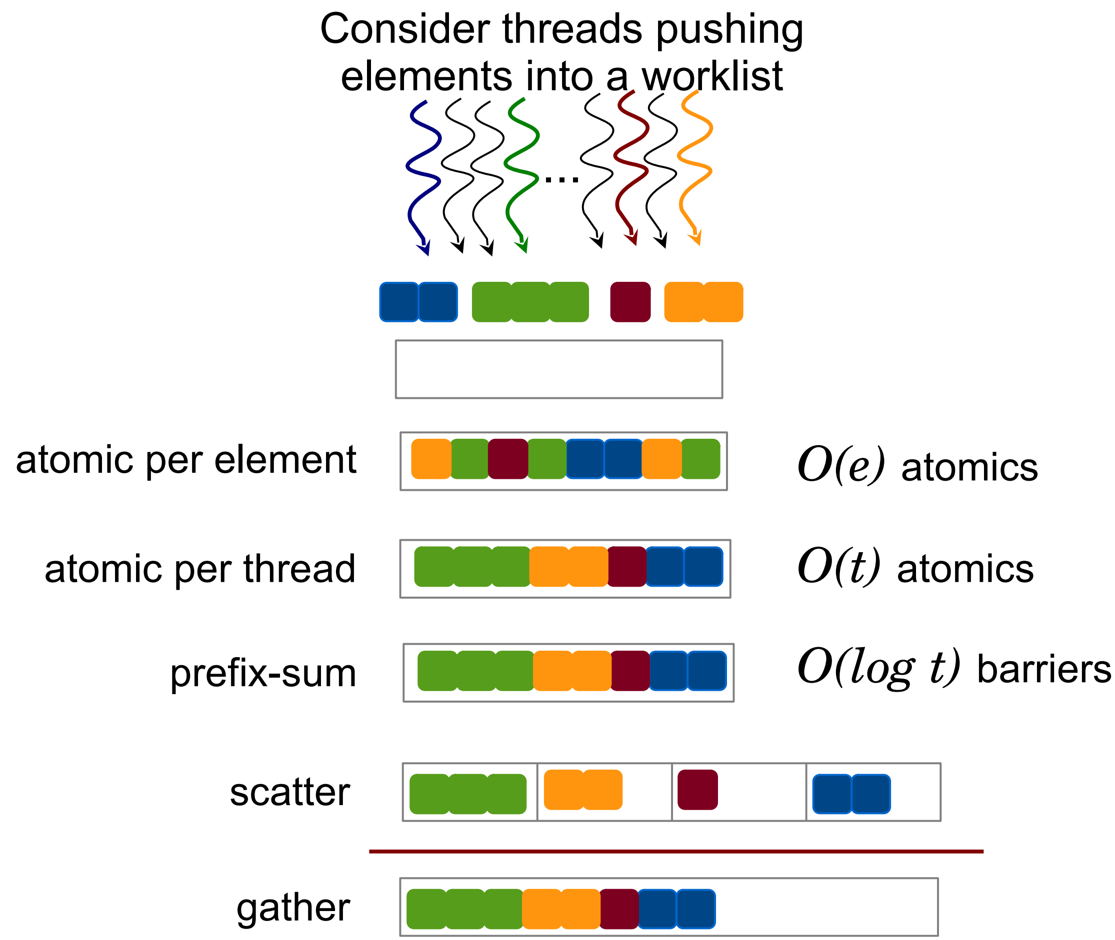
- Monotonicity
- Idempotency
- → **Associativity**

Consider threads pushing information to a node



Associativity helps push information using prefix-sum

Scatter-Gather



Learning Outcomes

- ✓ Data Race, Mutual Exclusion, Deadlocks
- ✓ Atomics, Locks, Barriers
- ✓ Reduction
- ✓ Prefix Sum
- ✓ Concurrent List Insertion
- ✓ CPU-GPU Synchronization