Lecture 4:
Basics of Parallel Programming

Beakal Gizachew Assefa

So far . . .

- Importance of Parallel Programming
- Parallel Architectures
- Scalability, Speedup, and Performance
- Performance Modeling
  - With a pencil-‐and-‐paper
  - Will resume with case studies later
- Today
  - Let’s write our first parallel program
  - Reading Chapter 2.4.1-‐2.4.3

How to Parallelize?

- Assume we are starting with a **sequential algorithm** and trying to modify it to execute in parallel
  - Not always the best strategy, as sometimes the best parallel algorithms are NOTHING like their sequential counterparts
  - But useful since you are accustomed to sequential algorithms

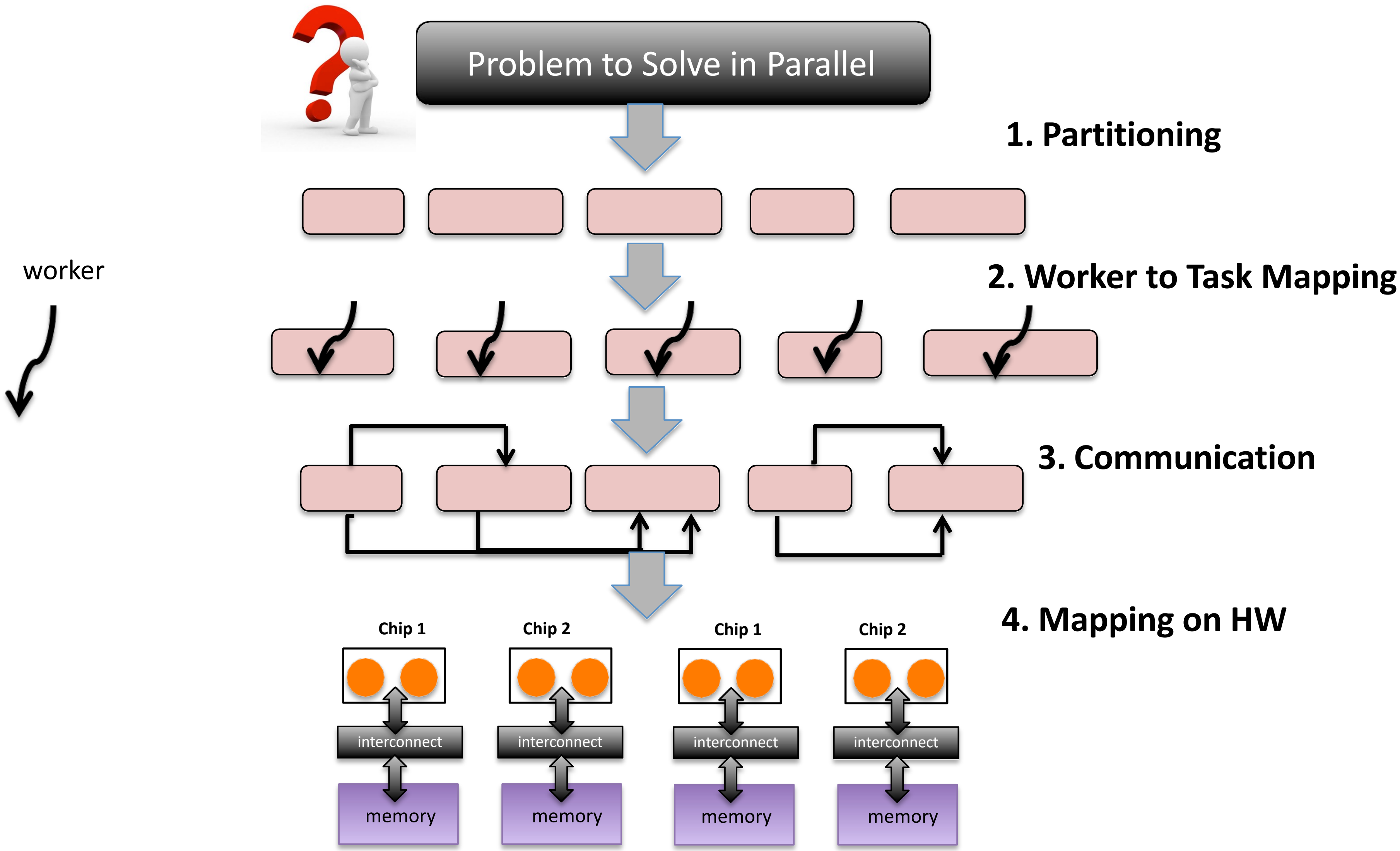
How to Parallelize?

- Identify work that can be performed in parallel
  - Remember Amdahl's Law: reduce the serial fraction as much as you can for scalability
- Partition work among workers
- Manage data access, communication and coordination of workers

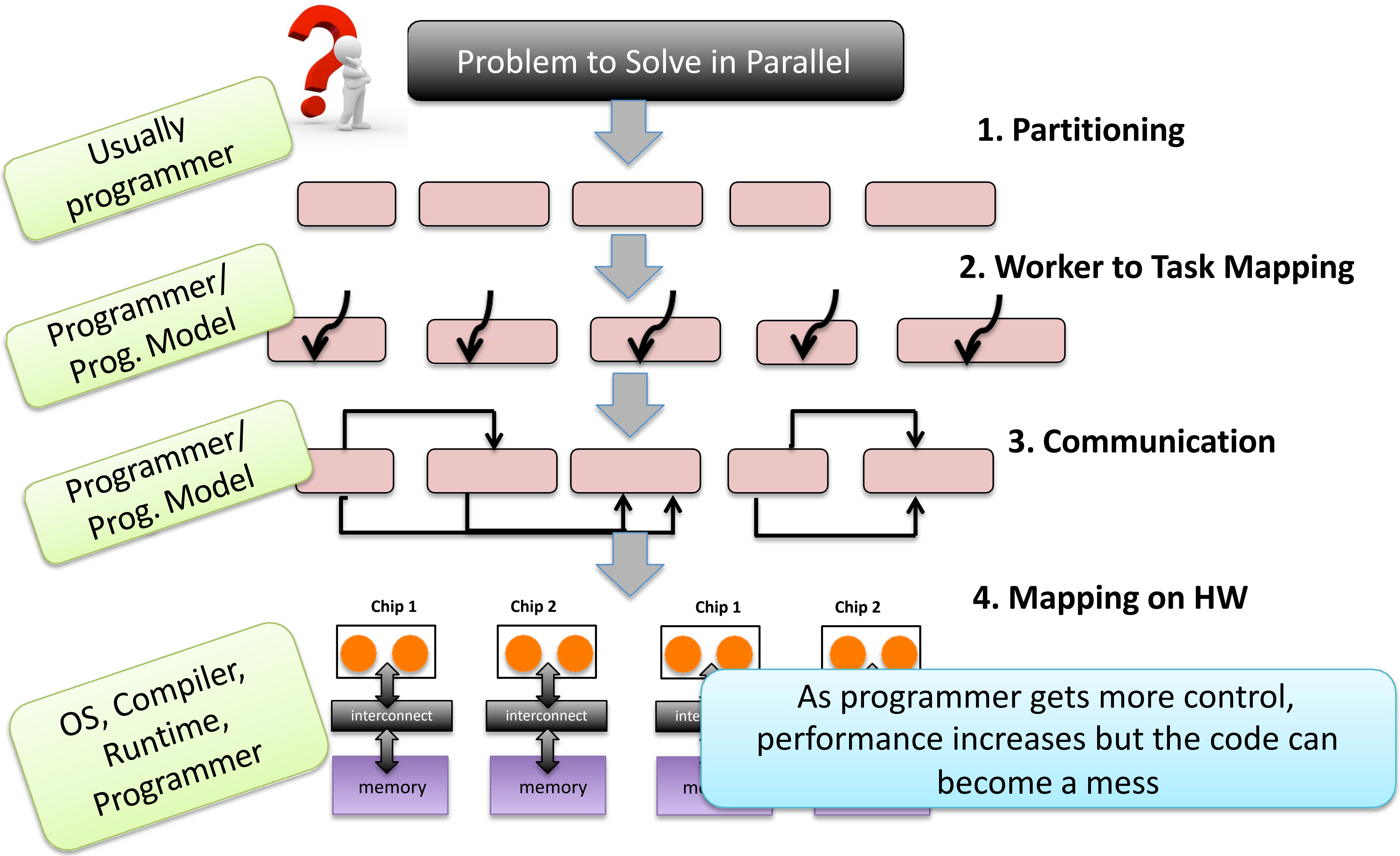
- Remember our ultimate goal is "**Performance**"
  - Ignore architectural details at first
  - Then, optimize for architecture
    - What is the downside of this?

Performance Portable is more desirable

Reasoning about a Parallel Program



Who is Responsible?



Parallelization Steps

- 1. Partitioning: Divide the computation into tasks
  - Will cover throughout the semester
- 2. Worker-to-Task Mapping: Assign threads/workers to tasks to execute
  - Advanced topic
  - This can be very complicated when dealing with irregular applications (e.g. graphs)
- 3. Communication: determine what communication needs to be carried out among the tasks
  - Will cover throughout the semester
- 4. Mapping: Mapping threads to hardware execution units
  - Advanced topic
  - Will cover with respect to NUMA--architectures and GPUs

1.Partitioning

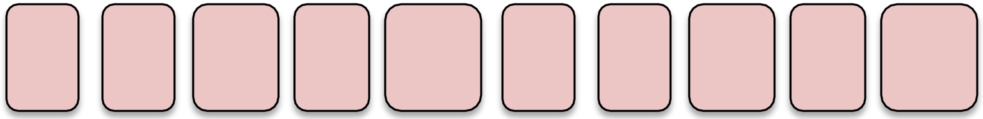
- **Computation Partitioning:**
  - Divide the sequential computation among parallel threads/ processors/computations
  - The focus here should be on identifying **tasks** that can be executed in parallel.
- **Data Partitioning**
  - Also known as data decomposition or domain decomposition
  - Partition the data operated on by the computation

- One might imply the other
  - Think of whichever is easier, then think the corresponding partitioning

When partitioning the computation

- There are two important aspects:
  - Granularity of tasks
    - Size of each task
  - Static vs dynamic partitioning
    - New tasks can be discovered as program runs
  - Preserving data dependencies
    - Keeping the data values consistent with respect to the sequential execution.

Granularity of Tasks

- Fine--grained
  
  - Tasks are small
  - Increases the degree of parallelism ☺
  - Increases the parallelization overhead ☹
- Coarse--grained
  
  - Tasks are coarser
  - Decreases the degree of parallelism ☺
  - Reduces the overhead ☺
- Finding the right granularity is the key
  - Create at least enough tasks to keep all execution units on a machine busy
  - Each task working--set should fit into L1 if not L2 cache

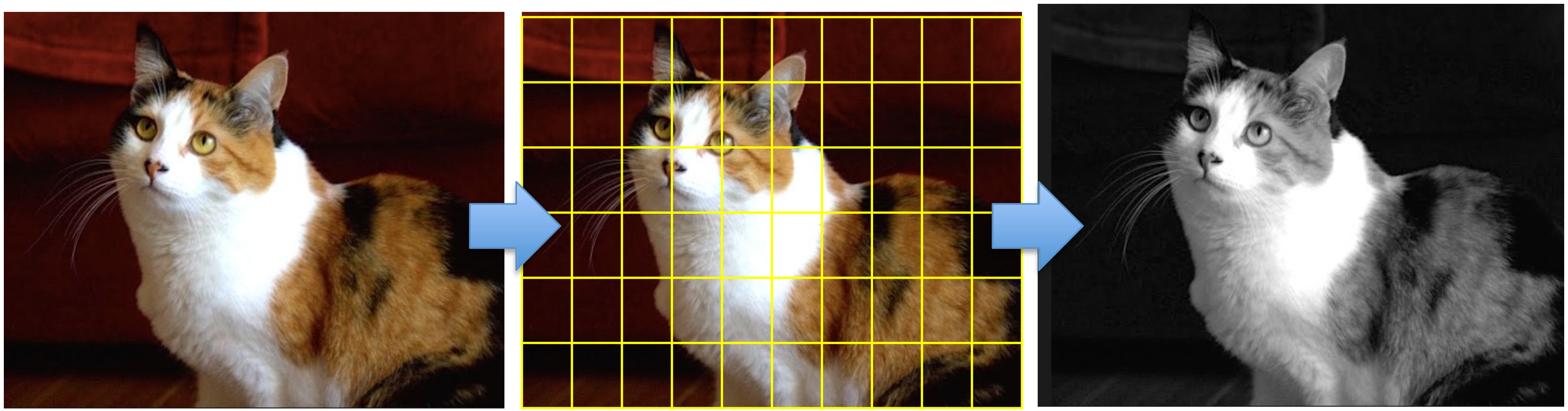
Data Dependencies

- One of the difficulties of parallel programming comes from the data dependencies between tasks
- Parallel execution has to obey the data dependencies otherwise we will end up with an incorrect program

- A formal definition:
  - *A* ***data dependence*** *is an ordering on a pair of memory operations that must be preserved to maintain correctness.*

A Simple Example

- On an N‐by‐N image, consider a computation that converts color to grayscale
  - Each color pixel is described by a triple (R, G, B) of intensities for red, green, and blue
  - Average method simply averages the values: (R + G + B) / 3 on each pixel
- Here, computation on each pixel is **independent**, no data dependencies between tasks
  - These types of parallelization are called **‘embarrassingly parallel’** algorithms



Another Example: Parallel Sum

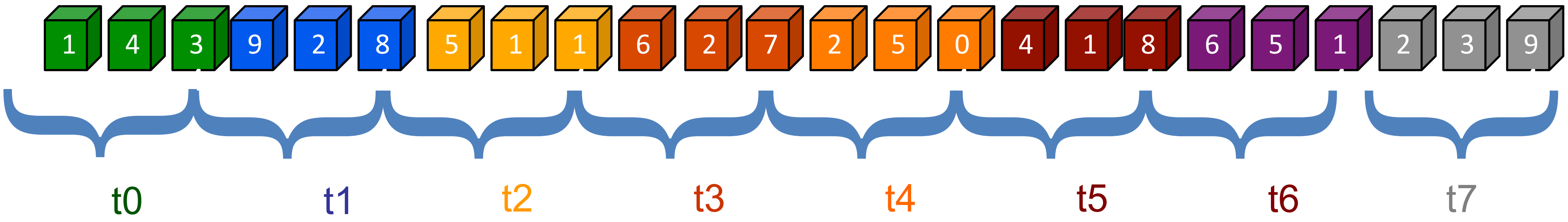
- Compute n values and add them together
- Serial formulation:

```
sum = 0;
for (I = 0; I < N ; I++ )
{
        x= compute_next_value( . . .);
        sum  = sum + x;

}
```

- Parallel formulation?

Version 1: Naïve

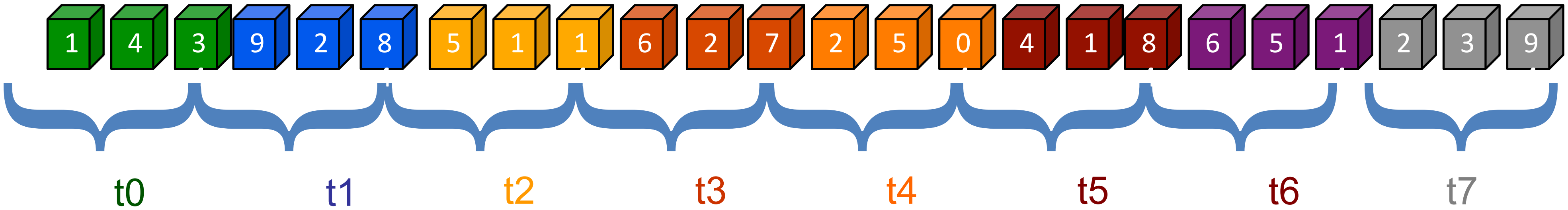
- Computation Partitioning
  - Suppose each task computes a partial sum on n/t consecutive elements (t is the number of tasks)
  - Example: n = 24 and t = 8 tasks



- Workers to Task Mapping
  - Assume we have 8 cores/processors
  - Each worker/thread gets a task
  - Need to calculate the start index for each thread

Version 1: Naïve

- Example: n = 24 and t = 8 tasks (threads)




```
int items_per_task = n/t;
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
      x = Compute_next_value(…);
      sum += x;
}
```

Correct?

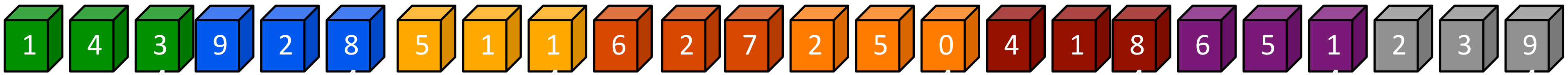
Data Dependencies?

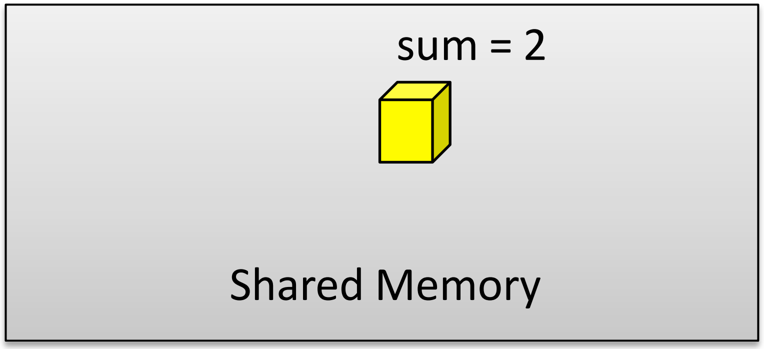
- Load/increment/store must be done *atomically* to preserve sequential meaning
  - More than one thread may update sum at the same time
- A ***race condition*** exists when the result of an execution depends on the ***timing*** of two or more events.
- **Mutual exclusion**: at most one thread can execute the code at any time

```
int items_per_task = n/t;
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
     x = Compute_next_value(…);
     sum += x;
}
```

Race Condition

- The value of sum is non--deterministic

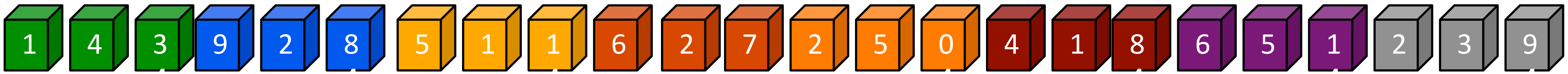


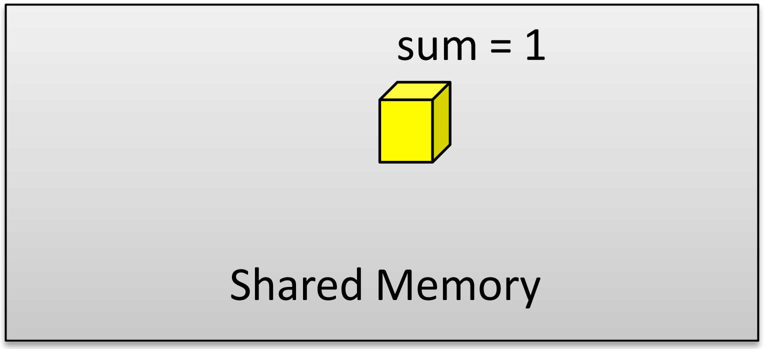


Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Race Condition

- The value of sum is non--deterministic





Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Version 2: Add Locks

- Insert mutual exclusion (mutex) so that only one thread at a time is loading/incrementing/storing sum **atomically**
  - Atomicity: a set of operations is atomic if either they all execute or none executes. Thus, there is no way to see the results of a partial execution.

```
int items_per_task = n/t;
mutex m;
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
      my_x = Compute_next_value(…);
      mutex_lock(m);
      sum += my_x;
      mutex_unlock(m);

}
```

Now, it is correct!

Version 3: Reduce the use of Locks

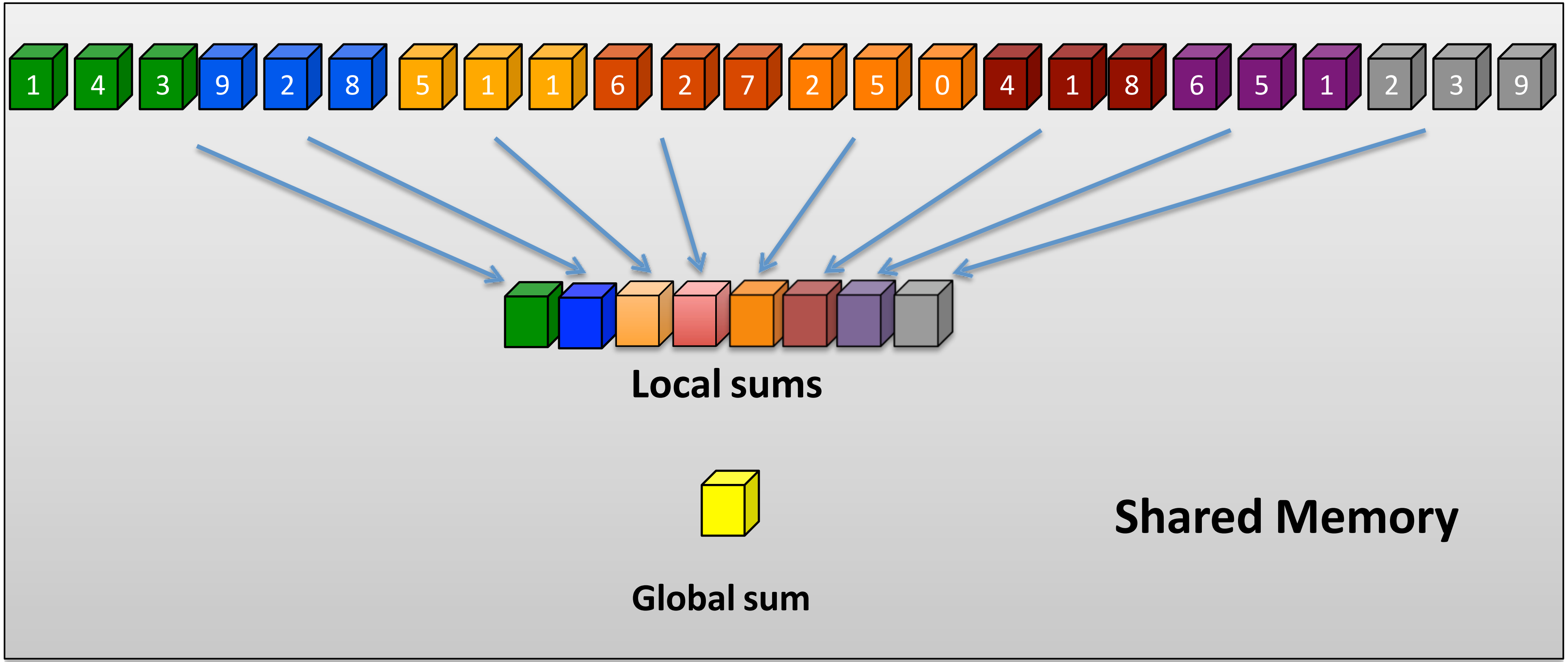
- Acquiring lock brings overhead because it serializes parallel execution
- Lock only to update final sum from thread--private copy

```
int items_per_task = n/t;
mutex m;
int my_sum;
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
mutex_lock(m);
sum+= my_sum;
mutex_unlock(m);
```

Can we eliminate the lock completely?

Version 4: Eliminate lock

- One of the threads can accumulate result
- Local sum is indexed by thread ID



Version 4: Eliminate lock

- One of the threads can accumulate result

```
int items_per_task = n/t;

shared int my_sum[t]; //number of threads
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
      my_x = Compute_next_value(…);
      my_sum[thread_id] += my_x;
}

if (thread_id == 0 ) //master thread
{
    sum = my_sum[0];
     for(i=1; i< t; i++) sum+ = my_sum[i];
}
```

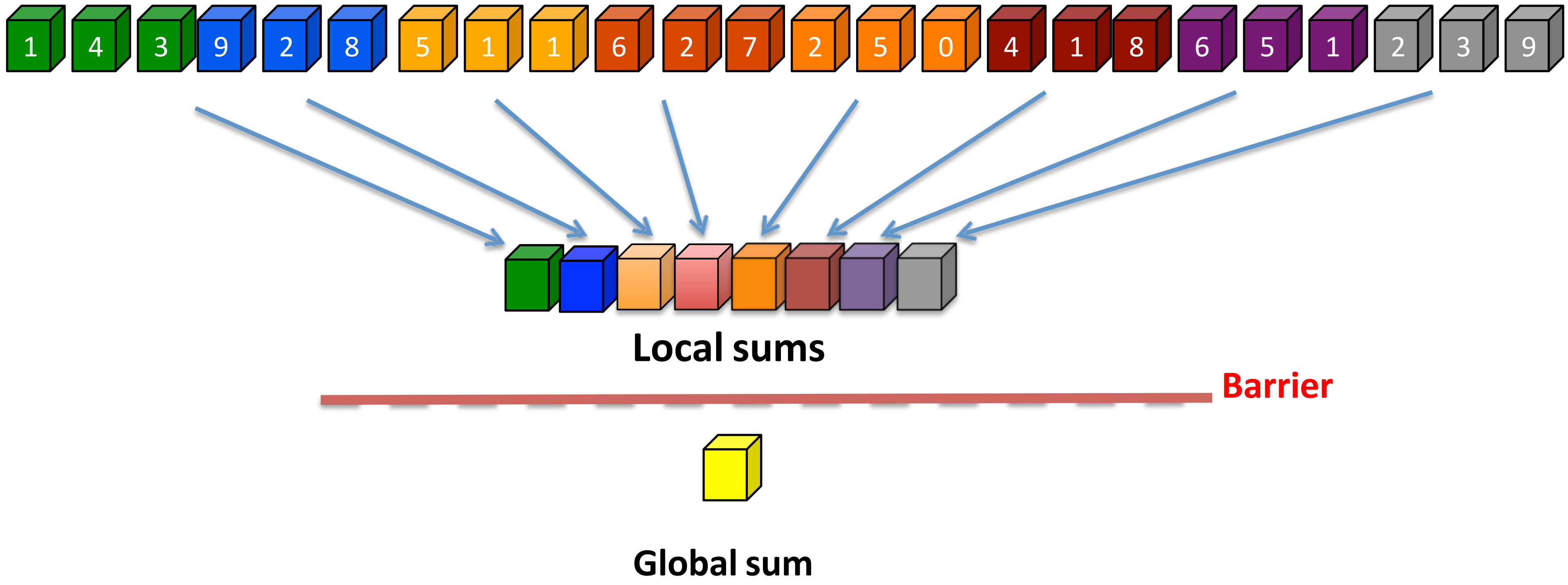
Correct?

Synchronization: Barriers

- Sum is incorrect if `master’ thread begins accumulating final result before other threads are done
- ***Synchronization*** is used to sequence control among threads or to sequence accesses to data in parallel code.
- How can we force the master to wait until the threads are ready?
  - A ***barrier*** is used to block threads from proceeding beyond a program point until all of the participating threads has reached the barrier.

Version 5: Add a barrier

- Ensure all the local sums are ready (all the threads are done calculating their local sums)



Version 5: Add a barrier

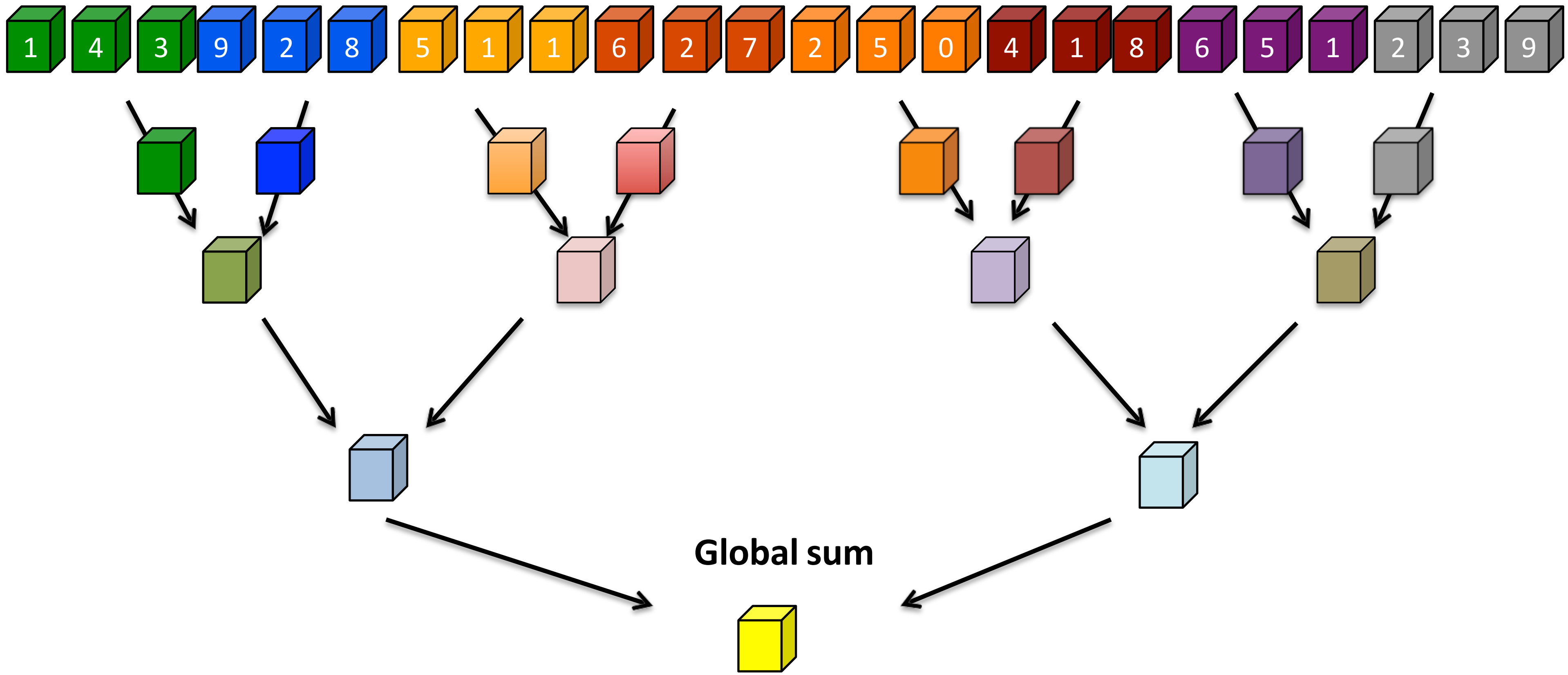
- Master waits for others to finish

Now it is correct!

```
int items_per_task = n/t;

shared int my_sum[t]; //number of threads
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
      my_x = Compute_next_value(…);
      my_sum[thread_id] += my_x;
}

synchronize_threads(); // barrier for all participating threads

if (thread_id == 0 ) //master thread
{
    sum = my_sum[0];
    for(i=1; i< t; i++) sum+ = my_sum[i];
}
```

Version 6: Improve Performance

- Now, our implementation is correct, we can try to improve its performance

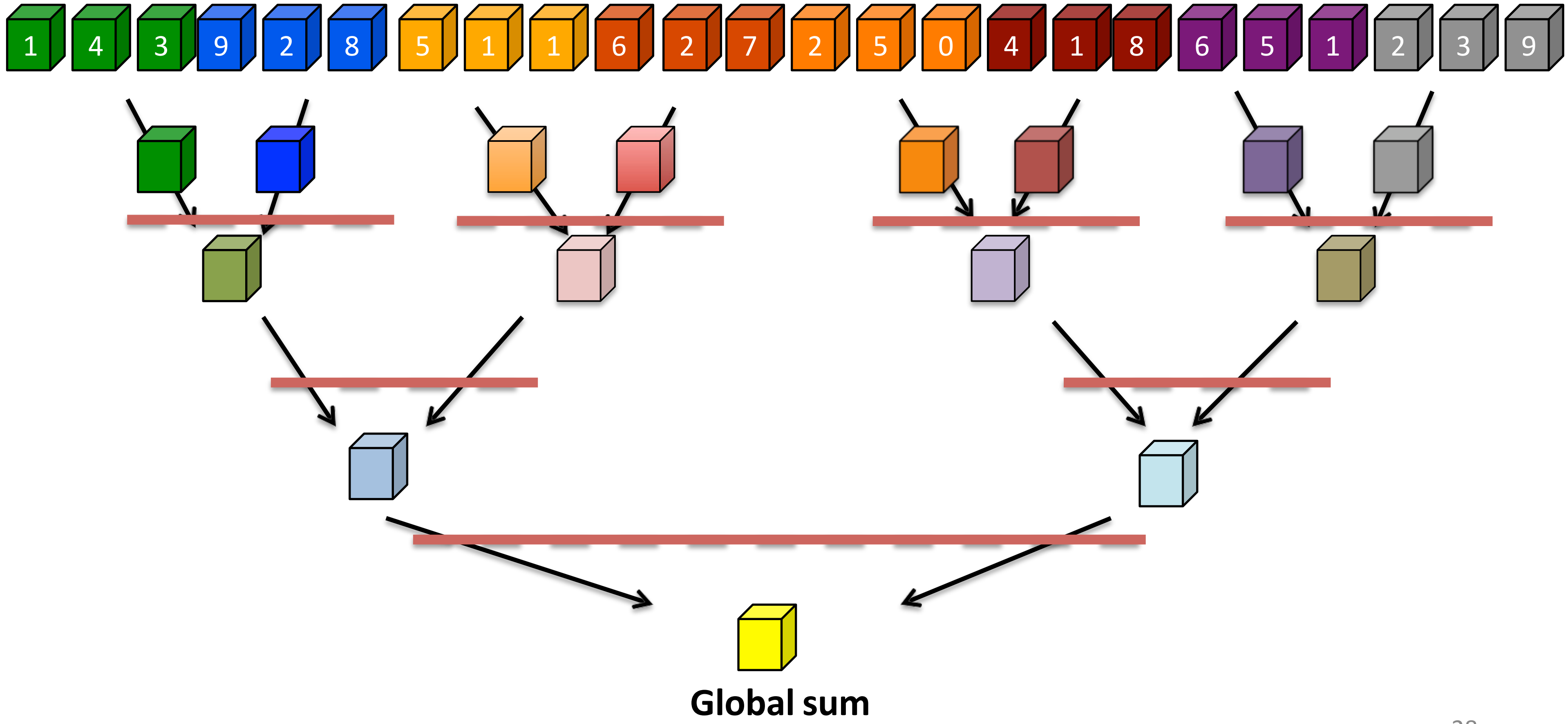


Version 6: Tree Sum

- Threads form a tree to accumulate sum
- Sum is calculated in log (t) steps, where t is number threads/ processors
- For large t, it makes a performance difference
- For example, N=1M, t = 1000
  - Each worker computes N/t elements: 1M/1000 = 1000 elements, then we have 1000 partial sums
  - If only master computes,
    - We have 1000 adds by master (serialization)
    - Total time= Time(partial sum) + MasterTime(global sum)
  - In tree sum
    - Total time= Time(partial sum) + log (t)
    - We have only 10 adds for partial sums

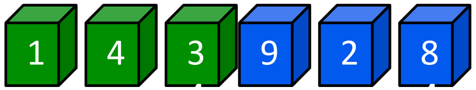
Version 6: Tree Sum

- Need to add synchronization points (not necessarily a global barrier)



Data Dependencies?

- Dependence on sum across iterations/threads?
  – Reordering ok since operations on sum are associative
- Calculating
  - (((((1+4)+3)+9)+2)+8) is the same as
  - (1 + 4 + 3) + (9 + 2 + 8)



- May get slightly different results on floating point operations
  – because of rounding in hardware
  – Real numbers are approximated in hardware

Lessons Learnt from Parallel Sum

- The sum computation had a race condition or data dependence.
- We used mutex and barrier synchronization to guarantee correct execution.
- We performed mostly local computation to increase parallelism granularity across threads.
- What were the overheads we saw with this example?
  - Extra code to determine portion of computation
  - Locking overhead: inherent cost plus contention
  - Load imbalance: use tree sum

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)