KOÇ UNIVERSITY

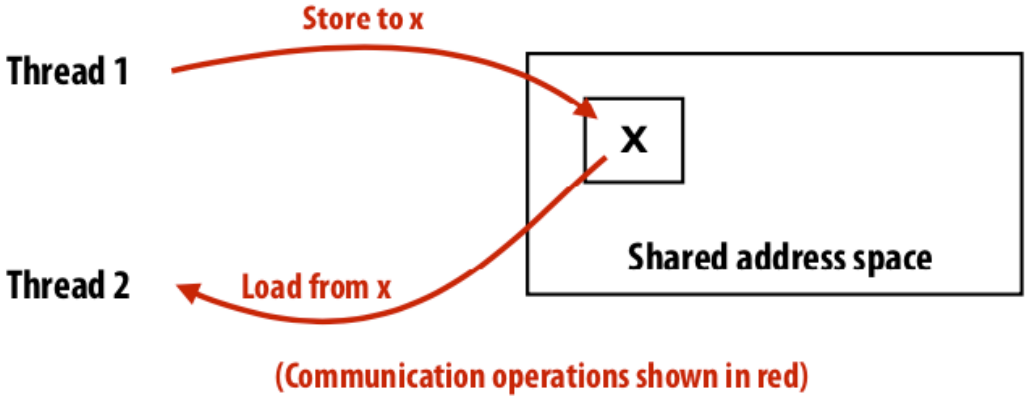
Shared Memory Programming

Didem Unat

COMP 429/529 Parallel Programming

Shared-Memory Programming Model

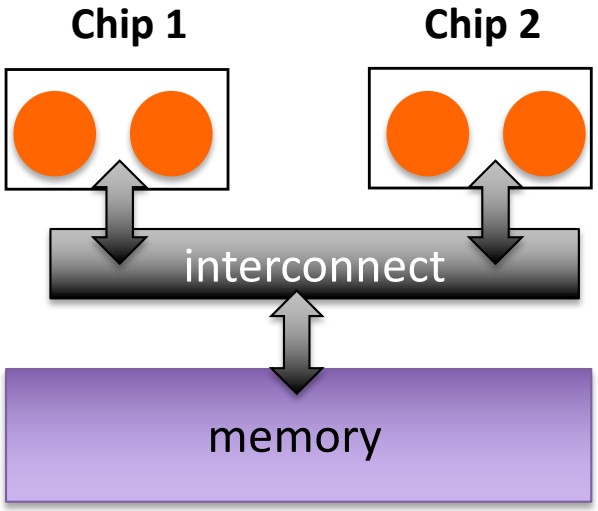
- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board
  - Any thread can read or write to shared variables



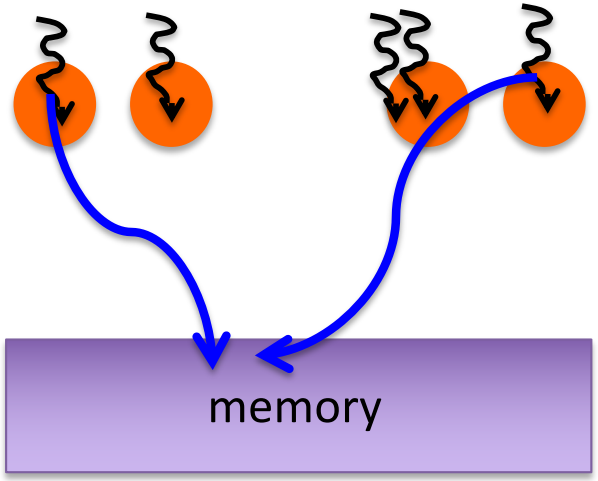
- Inter-thread communication is **implicit** in memory operations
  - Thread 1 stores to X
  - Later, thread 2 reads X (and observes update of value by thread 1)
- Manipulating synchronization primitives
  - e.g., ensuring mutual exclusion via use of locks

Shared-Memory Programming Model

- More correct name: Shared-address space programming
  - Threads communicate through shared memory as opposed to messages
  - Threads coordinate through synchronization (also through shared memory).

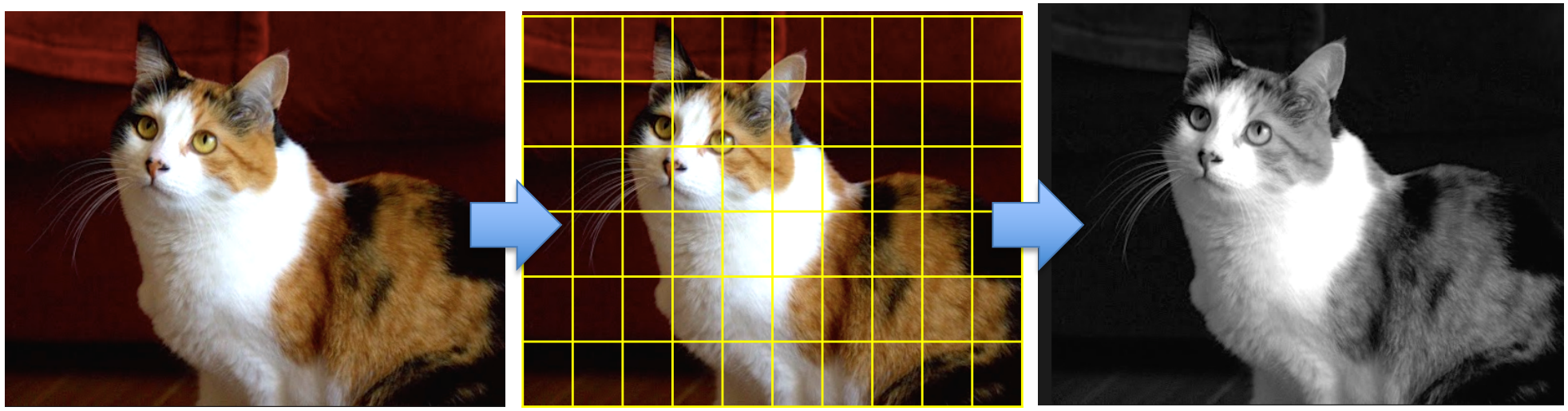


Recall shared memory system
(can be either UMA, NUMA)



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

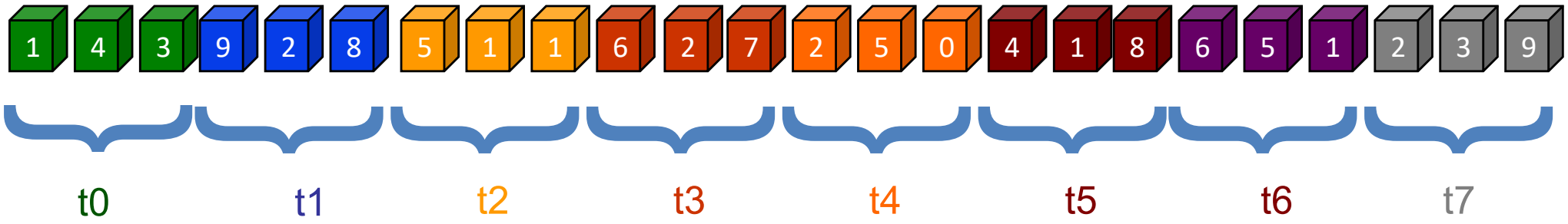
- Compute *n* values and add them together
- Serial formulation:

```
sum = 0;
for (I = 0; I < N ; I++ )
{
      x= compute( I , …);
      sum  = sum + x;

}
```

- An example: Perform gradient decent in parallel on batches, compute the loss function
- Parallel formulation?

Version 1: Naïve

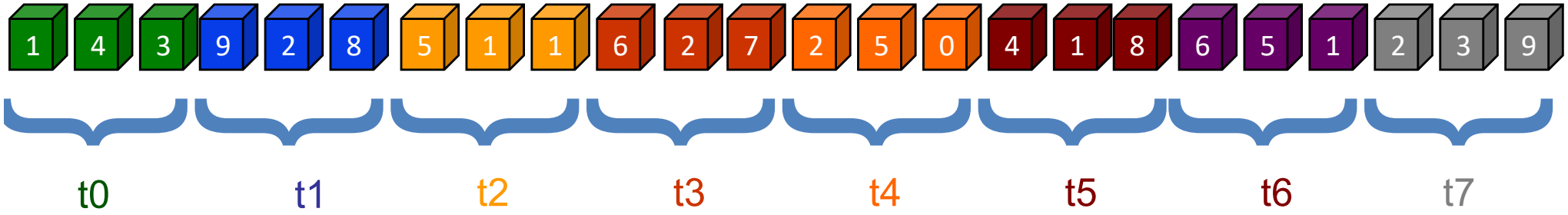
- Partitioning
  - Suppose each task computes a partial sum on *n/t* consecutive elements (*t* is the number of tasks)
  - Example: *n* = 24 and *t* = 8 tasks



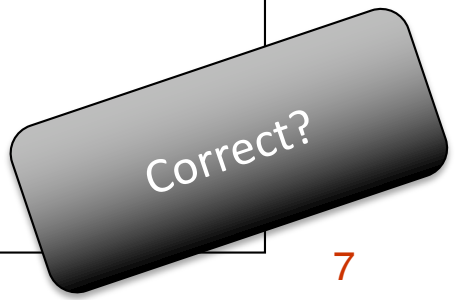
- Workers to Task Mapping
  - Assume we have 8 cores
  - Each thread gets a task
  - Need to calculate the start index for each thread

Version 1: Naïve

- Example: n = 24 and t = 8 tasks (threads)
- `sum` is a global shared variable




```
//Assumes threads are created already,
//Threads running below code in parallel
private int items_per_task, start, i; //thread local
shared int x, sum=0;
. . .
items_per_task = n/t;
start = thread_id * items_per_task;
i = 0;
for (i=start; i<start + items_per_task; i++)  {
      x = compute (i, ….);
      sum += x;
}
```



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.*

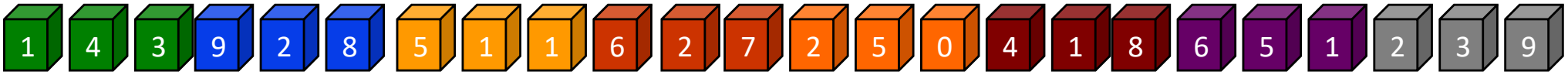
Data Dependencies?

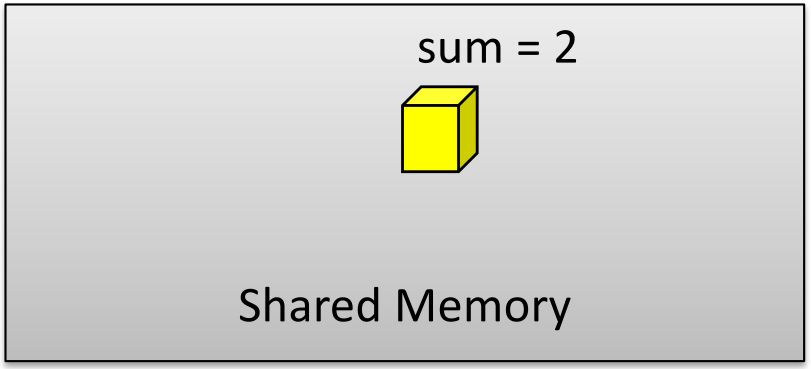
- Load/increment/store must be done *atomically* to preserve sequential meaning
  - More than one thread may update **sum** at the same time
- A ***race condition*** exists when the result of an execution depends on the ***timing*** of two or more events.
- **Mutual exclusion**: at most one thread can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

```
. . .
items_per_task = n/t;
start = thread_id * items_per_task;   //thread local
i = 0; //thread local
for (i=start; i<start + items_per_task; i++)   {
      x = Compute_next_value(…);
      sum += x;
}
```

Race Condition

- The value of sum is non-deterministic

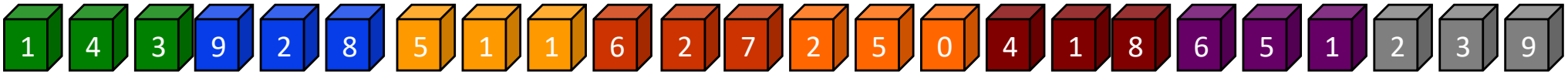


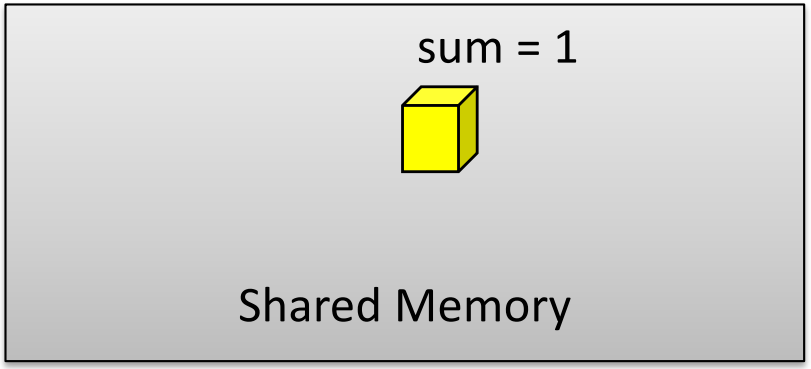


Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Race Condition

- The value of sum is non-deterministic



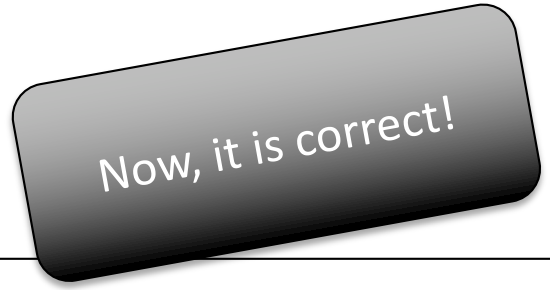


Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

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.

```
mutex m; //shared lock
…
items_per_task = n/t;
start = thread_id * items_per_task;
private int my_x; //thread local
for (i=start; i<start + items_per_task; i++)  {
     my_x = Compute (i, …);
     mutex_lock(m);
     sum += my_x;
     mutex_unlock(m);
}
```



Version 3: Reduce the use of Locks

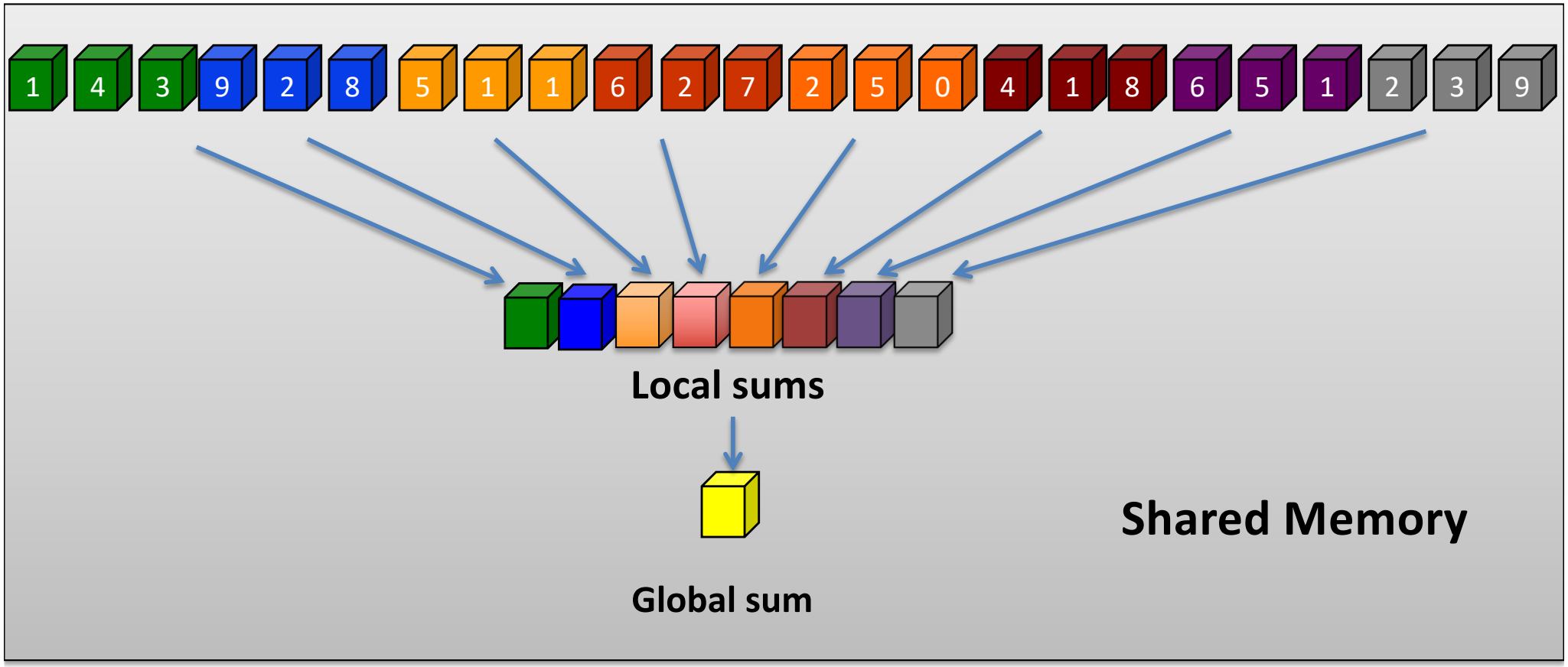
- Excessive use of locks brings overhead because it serializes parallel execution
- Lock only to update final sum from thread-private copy

```
mutex m; //shared lock
. . .
items_per_task = n/t;
start = thread_id * items_per_task; //thread local
private int my_sum=0, my_x; //thread local

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

Can we have a lock-free implementation?

Version 4: Local-Sum Array

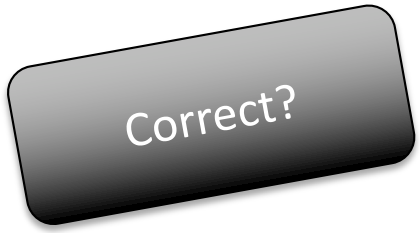
- Have local sum array, one of the threads can accumulate result
- Local sum is indexed by thread ID



Version 4: Local-Sum Array

- One of the threads can accumulate the local results

```
. . .
shared int my_sum[t]; //size is number of threads
//initialize my_sum to zero ...
. . .

start = thread_id * items_per_task;

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

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

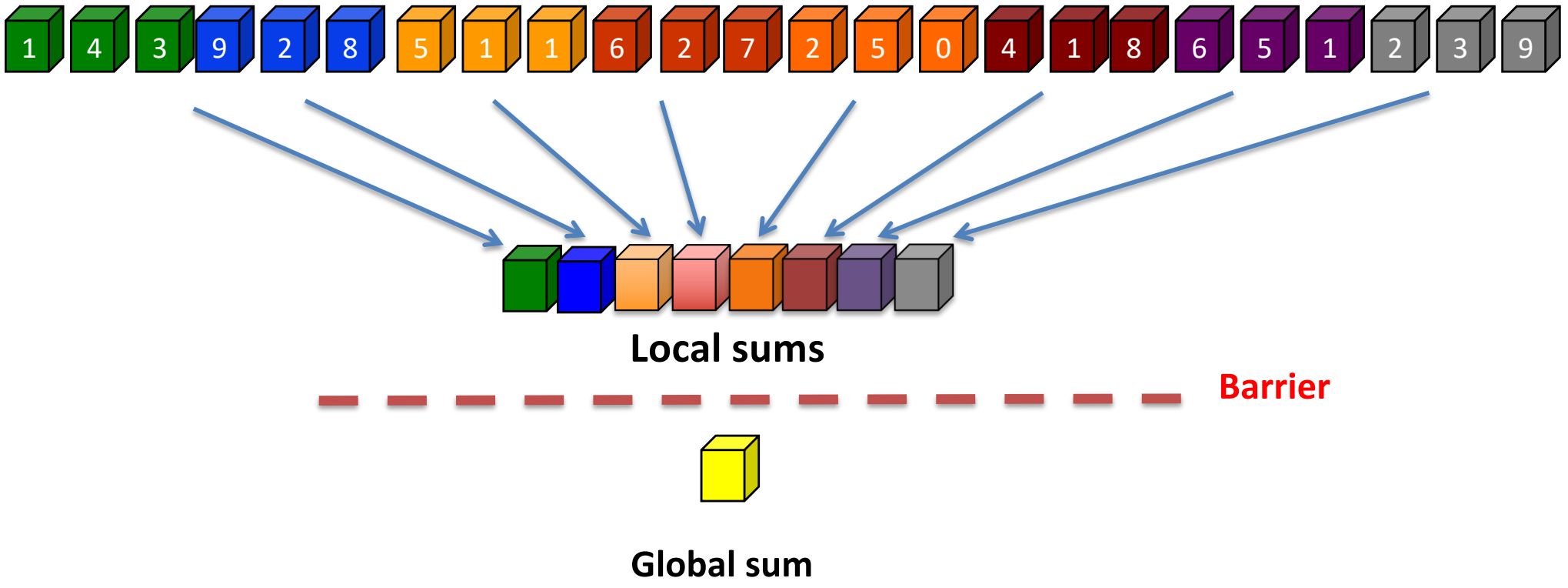


Synchronization: Barriers

- Sum is incorrect if `master' thread begins accumulating final result before other threads are done computing local sum
- ***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)



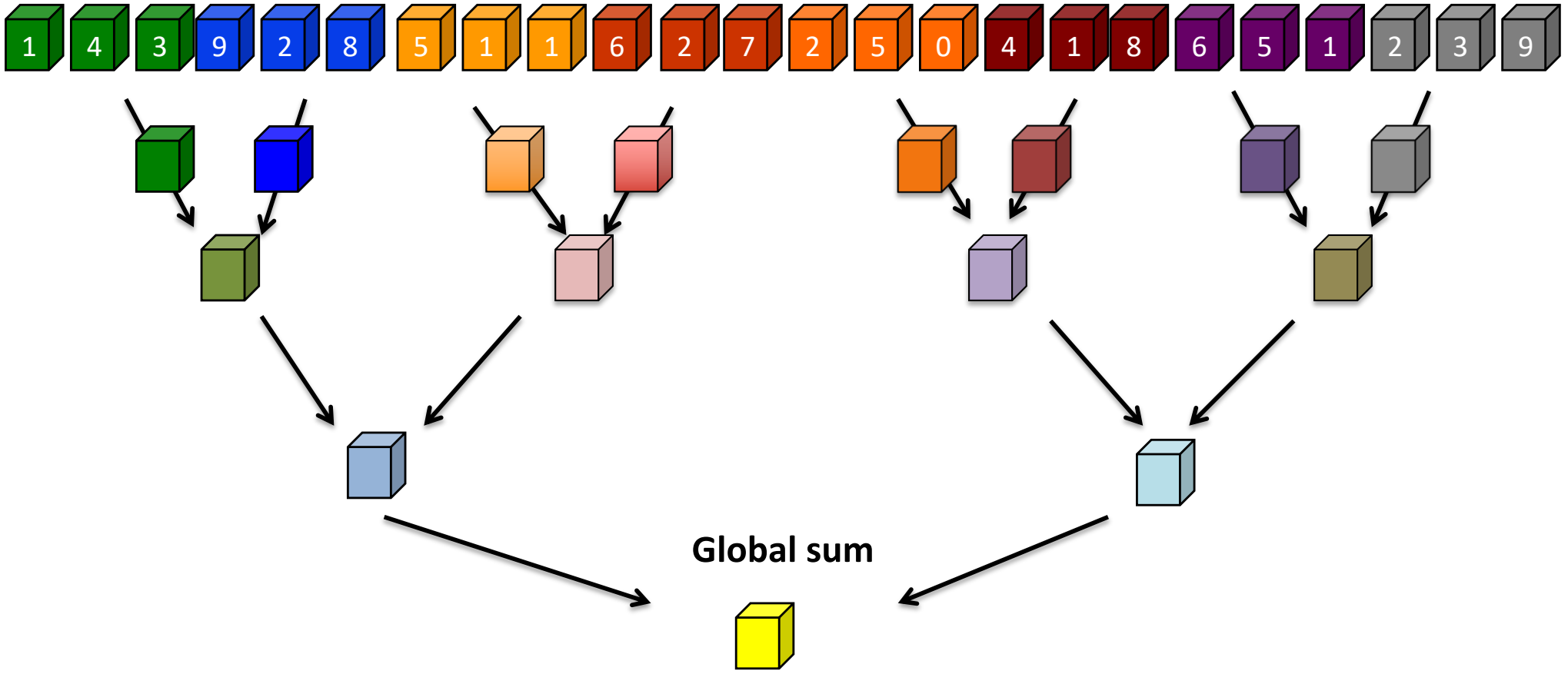
Version 5: Add a barrier

- Master waits for others to finish

```
items_per_task = n/t;
. . .
shared int my_sum[t]; //number of threads
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: Alternative Implementation

- Tree-based implementation



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 may make a performance difference
  - For small *t*, do not bother
- For example, *N=1Million, 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) + 1 sync overhead

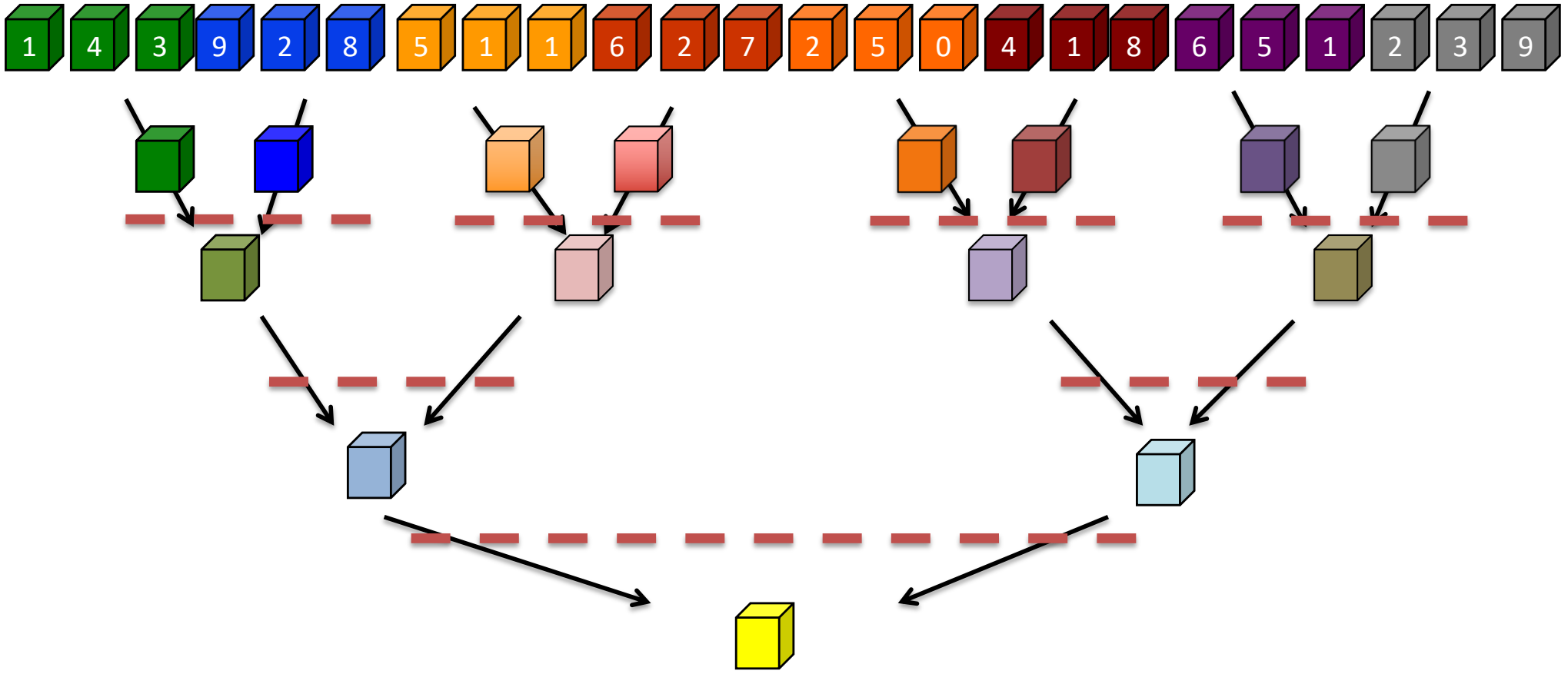
  - In tree sum

Total time= Time(partial sum) + Time(log (t) sums) + log(t) sync overhead

    - We have fewer partial sums, however, we have more synchronization calls (log(t) many) which may offset the performance gain

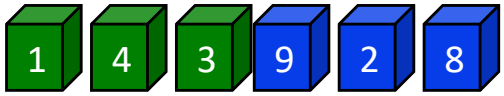
Version 6: Tree Sum

- Need to add point-to-point synchronization points (not necessarily a global barrier)
  - Do not need to synchronize with all



Data Dependencies?

- Dependence on sum across iterations/threads?
  - Reordering is 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
  - Large numbers are added to small numbers
    - Add small numbers first so that they don't disappear!

Lessons Learnt from Parallel Sum

- The sum computation had a race condition.
  - For correction execution, we need to pay attention to accesses to the shared data
- We used mutex or barrier synchronization to guarantee correct execution.
  - Order of execution matters, we need to synchronize to enforce the correct ordering of memory accesses
- We performed **mostly local computation** to increase parallelism granularity across threads.
  - Compute locally as much as possible, access to shared data only it is absolutely required

Parallelization Overheads

- What are the overheads in this example?
  - Extra code to determine portion of computation
    - Calculating each thread's start index, end index, per thread workload
  - Locking or synchronization overhead: inherent cost plus contention
    - Use of mutex or barriers
  - Extra data structure
    - Local sum array to keep the local sum results
  - Load imbalance
    - Master thread (thread 0) did more work than others

Shared Memory Programming with Threads

- Several thread libraries out there
  - Pthreads, OpenMP, TBB, Cilk, Qthreads, C++11
- Pthreads is the POSIX (Portable Operating System Interface for Unix) Thread Library
  - Very low level of multi-threaded programming
  - Most widely used for systems-oriented code
- OpenMP is a standard
  - High level support for parallel programming on shared memory

OpenMP

- Chapter 5 from the textbook
- Tutorial: https://computing.llnl.gov/tutorials/openMP/
- Standardization Committee: http://www.openmp.org

- Model for shared-memory parallel programming
  - Prevailing approach in scientific computing community
  - A simplified alternative to Pthreads

- OpenMP
  - MP= multiprocessing
  - Open= open specification, developed by community

- Extensions to existing programming languages (Fortran, C and C++)
  - Consists of compiler directives,
  - Runtime routines and environment variables

Acknowledgments

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