CS/EE 217

GPU Architecture and Parallel Programming

Lecture 14 Atomic Operations and Histogramming (part 1)

Objective

- To understand atomic operations
  - Read-modify-write in parallel computation
  - Use of atomic operations in CUDA
  - Why atomic operations reduce memory system throughput
  - How to avoid atomic operations in some parallel algorithms

- Histogramming as an example application of atomic operations
  - Basic histogram algorithm
  - Privatization

A Common Collaboration Pattern

- Multiple bank tellers count the total amount of cash in the safe
- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, add the subtotal of the pile to the running total
- A bad outcome
  - Some of the piles were not accounted for

A Common Parallel Coordination Pattern

- Multiple customer service agents serving customers
- Each customer gets a number
- A central display shows the number of the next customer who will be served
- When an agent becomes available, he/she calls the number and he/she adds 1 to the display
- Bad outcomes
  - Multiple customers get the same number
  - Multiple agents serve the same number

A Common Arbitration Pattern

- Multiple customers booking air tickets
- Each
  - Brings up a flight seat map
  - Decides on a seat
  - Update the the seat map, mark the seat as taken

- A bad outcome
  - Multiple passengers ended up booking the same seat

Atomic Operations

thread1: $Old \leftarrow Mem[x]$
$New \leftarrow Old + 1$
$Mem[x] \leftarrow New$

thread2: $Old \leftarrow Mem[x]$
$New \leftarrow Old + 1$
$Mem[x] \leftarrow New$

If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?

– What does each thread get in their Old variable?

The answer may vary due to data races. To avoid data races, you should use atomic operations

Timing Scenario #1

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

Timing Scenario #2

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

- Thread 1 Old = 1
- Thread 2 Old = 0
- Mem[x] = 2 after the sequence

Timing Scenario #3

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Timing Scenario #4

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

Atomic Operations – To Ensure Good Outcomes

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Without Atomic Operations

Mem[x] initialized to 0

thread1: Old $\leftarrow$ Mem[x]

New $\leftarrow$ Old + 1

Mem[x] $\leftarrow$ New

thread2: Old $\leftarrow$ Mem[x]

New $\leftarrow$ Old + 1

Mem[x] $\leftarrow$ New

- Both threads receive 0
- Mem[x] becomes 1

Atomic Operations in General

- Performed by a single ISA instruction on a memory location *address*
  - Read the old value, calculate a new value, and write the new value to the location
- The hardware ensures that no other threads can access the location until the atomic operation is complete
  - Any other threads that access the location will typically be held in a queue until its turn
  - All threads perform the atomic operation **serially**

Atomic Operations in CUDA

- Function calls that are translated into single instructions (a.k.a. *intrinsics*)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide 4.0 for details
- Atomic Add

  int atomicAdd(int **address**, int **val**);*

  reads the 32-bit word **old** pointed to by **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. The function returns **old**.

More Atomic Adds in CUDA

- Unsigned 32-bit integer atomic add

  unsigned int atomicAdd(unsigned int address, unsigned int val);*

- Unsigned 64-bit integer atomic add

  unsigned long long int atomicAdd(unsigned long long int address, unsigned long long int val);*

- Single-precision floating-point atomic add (capability > 2.0)
  - float atomicAdd(float* address, float val);

Histogramming

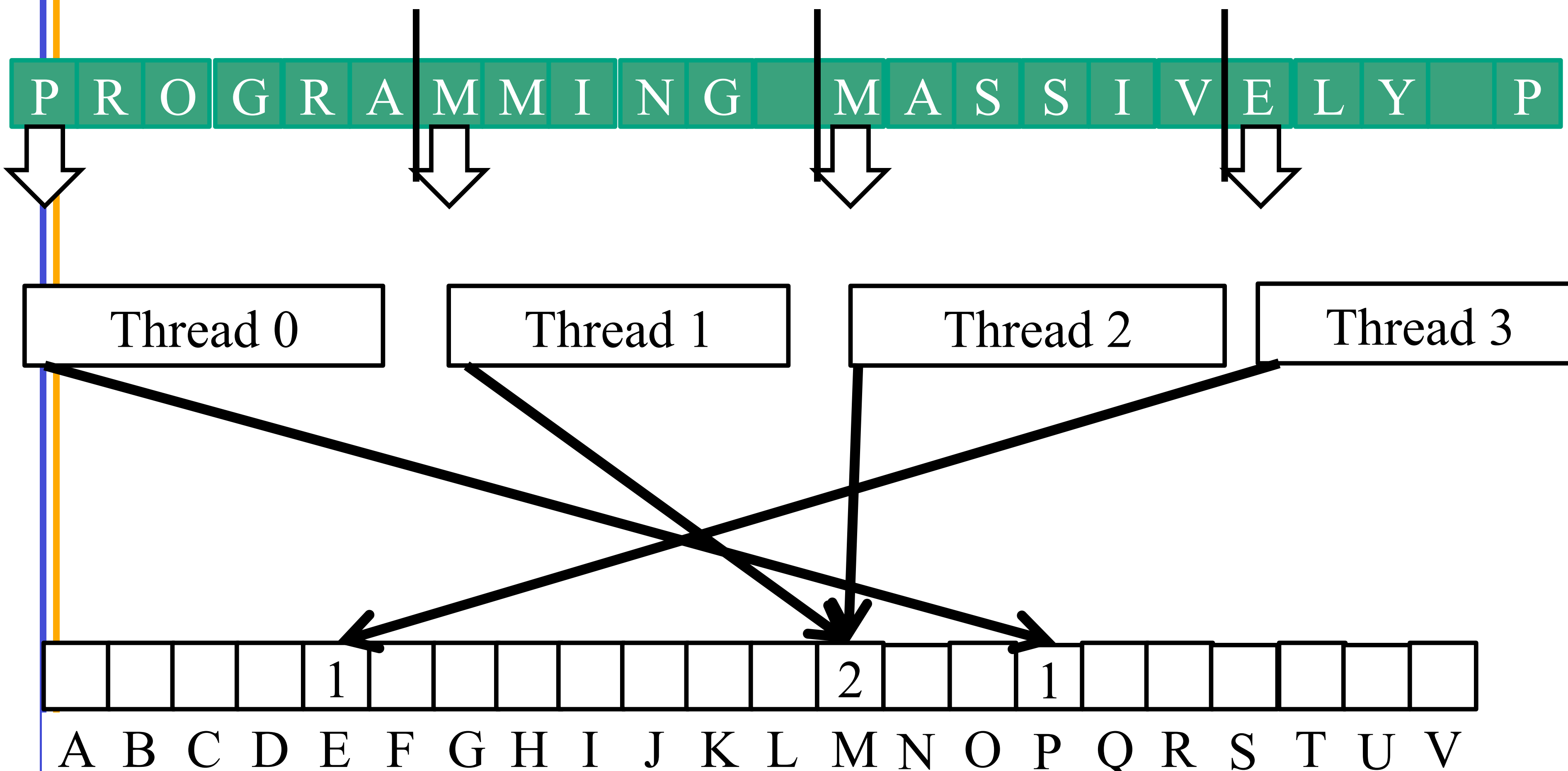
- A method for extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …

- Basic histograms - for each element in the data set, use the value to identify a “bin” to increment

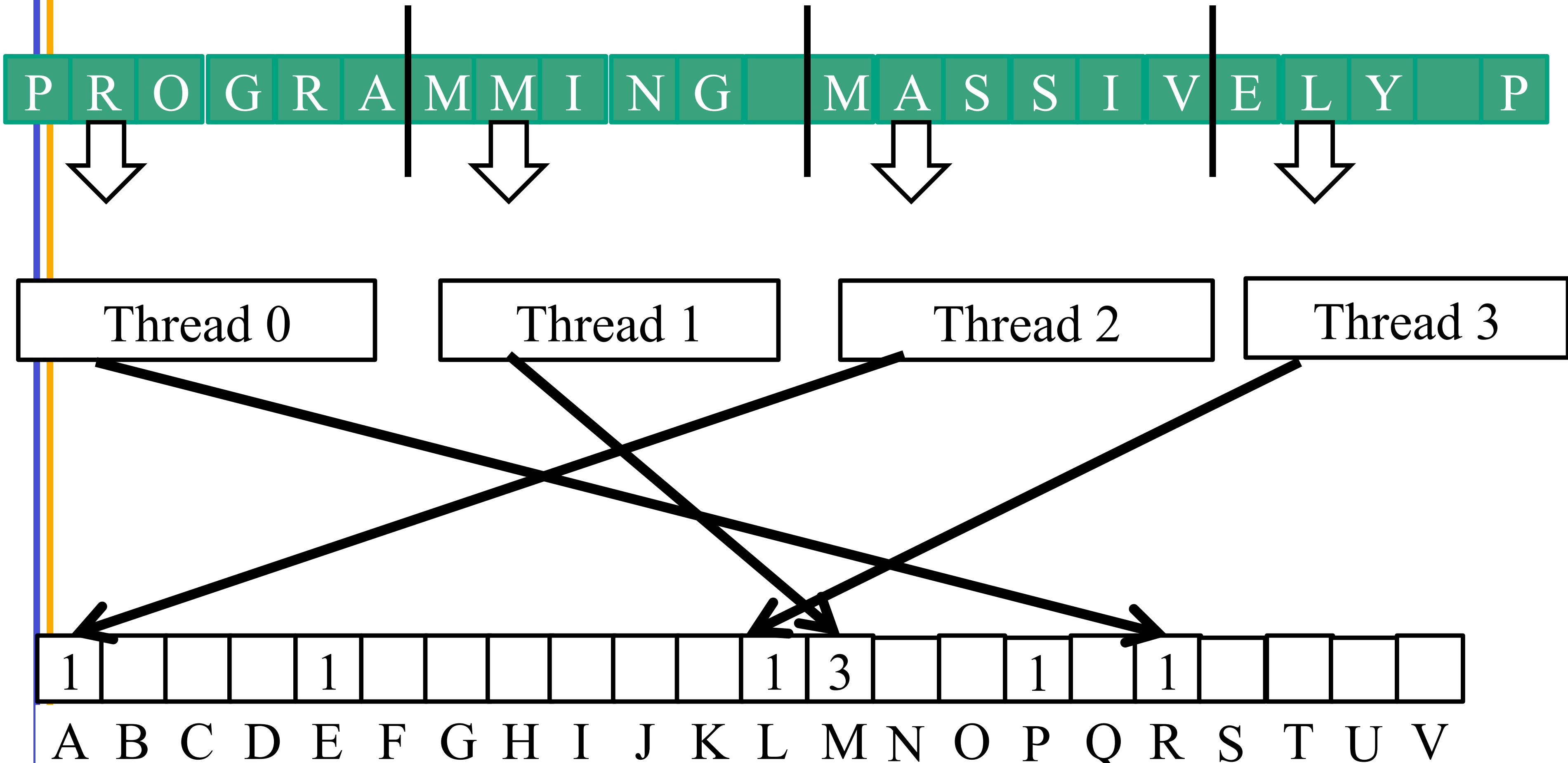
A Histogram Example

- In sentence “Programming Massively Parallel Processors” build a histogram of frequencies of each letter
- A(4), C(1), E(1), G(1), …
- How do you do this in parallel?

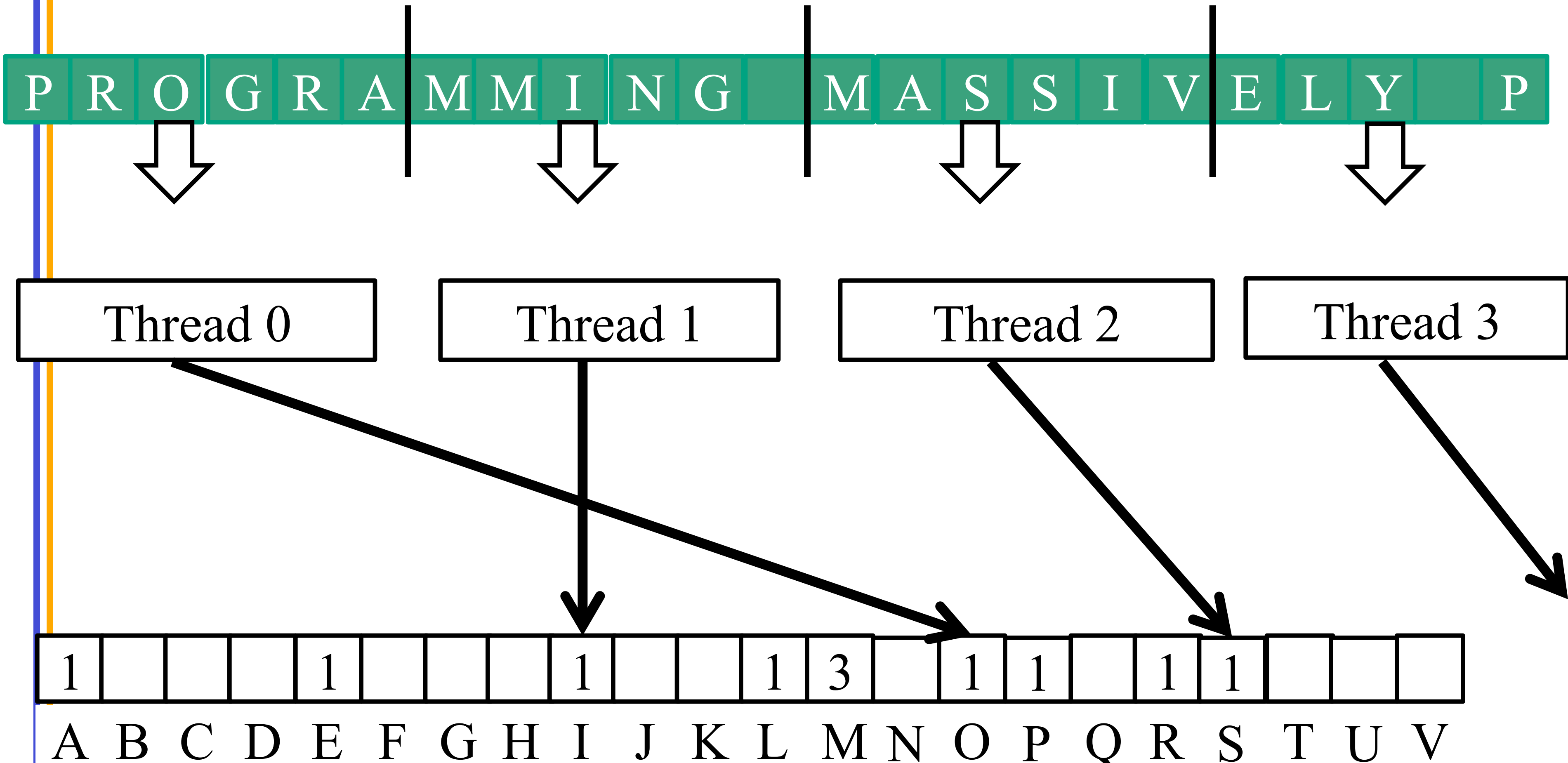
Iteration #1 – 1st letter in each section



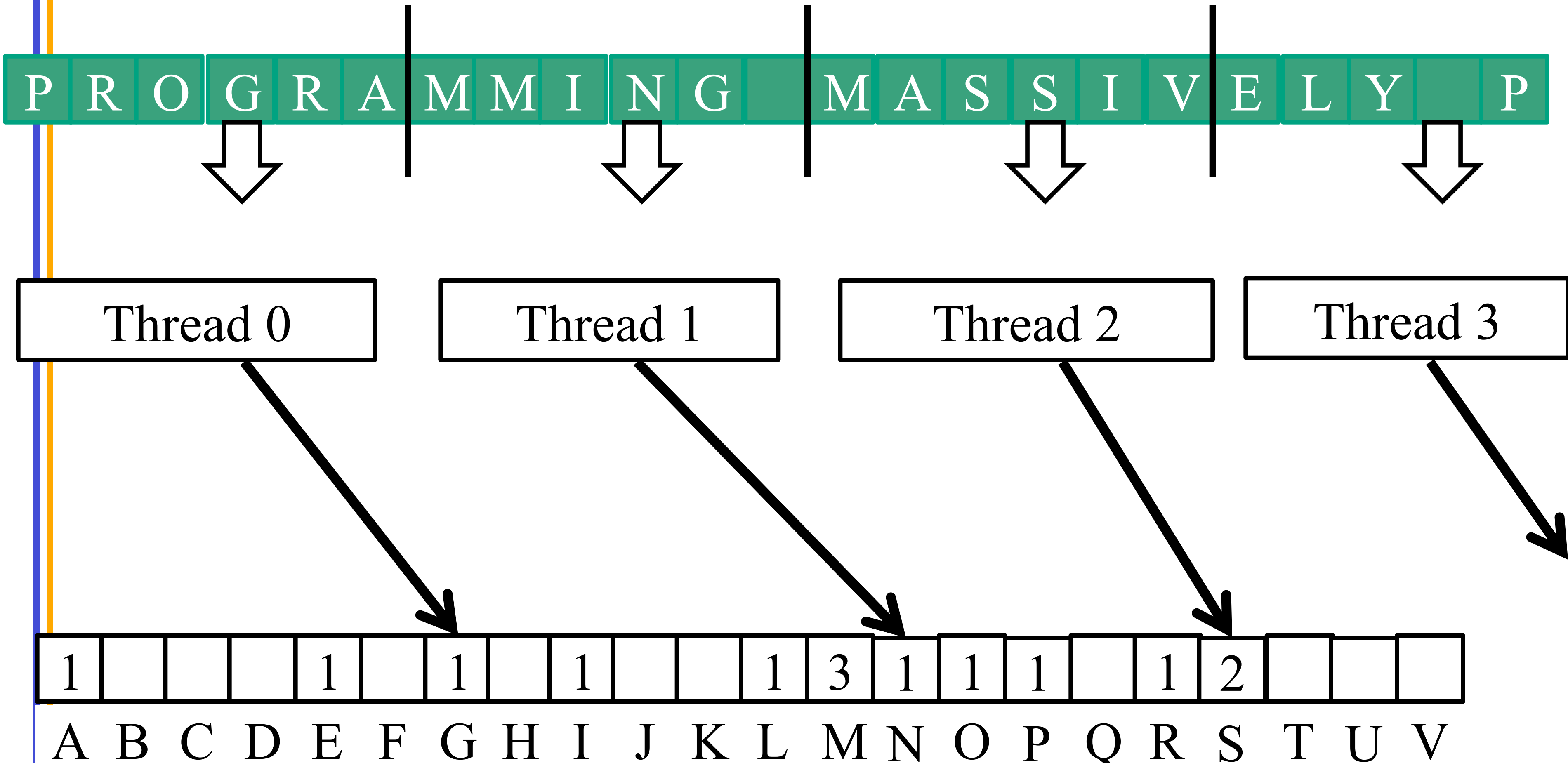
Iteration #2 – 2nd letter in each section



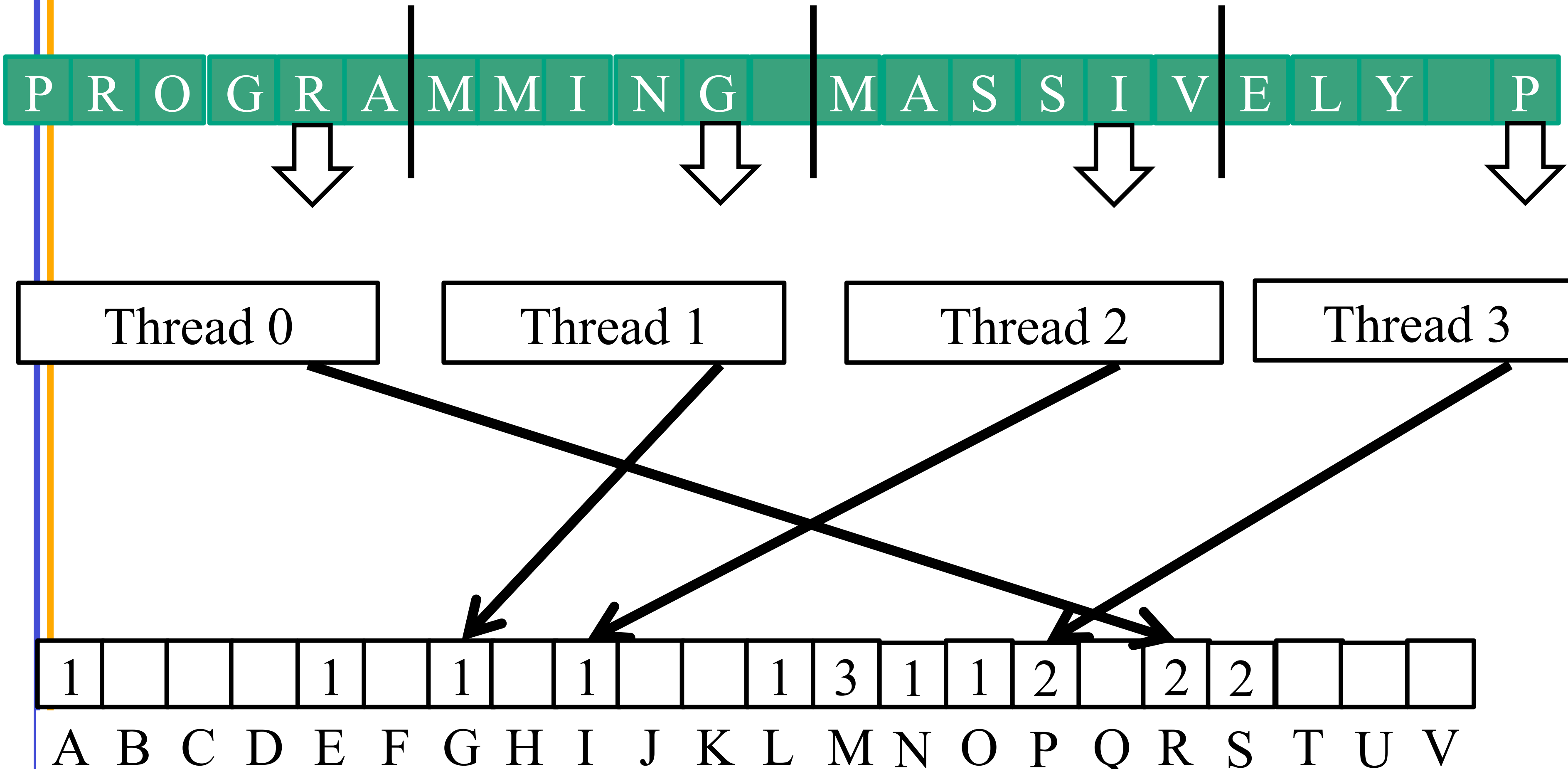
Iteration #3


P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 3 1 1 1 1
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #4


P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 3 1 1 1 1 2
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #5



P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 1 1 3 1 1 2 2 2
A B C D E F G H I J K L M N O P Q R S T U V

What is wrong with the algorithm?

ANY MORE QUESTIONS