www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Atomic operations

Marc Jordà, Antonio J. Peña

Montevideo, 21-25 October 2019

Objective

- Understand atomic operations
  - Why we need them? Read-modify-write in parallel computation
  - How are atomic operations used in CUDA
  - Why atomic operations reduce memory system throughput
  - How to avoid atomic operations in some parallel algorithms

Atomic Operations

- Example: 2 threads sharing a counter (Mem[x]), each thread incrementing the counter once
- If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?
  - Also, what does each thread get in their *Old* variable?

Thread 1:	Thread 2:
Old ← Mem[x]	Old ← Mem[x]
New ← Old + 1	New ← Old + 1
Mem[x] ← New	Mem[x] ← New

Atomic Operations

- Example: 2 threads sharing a counter (Mem[x]), each thread incrementing the counter once
- If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?
  – Also, what does each thread get in their *Old* variable?

Thread 1:	Thread 2:
Old ← Mem[x]	Old ← Mem[x]
New ← Old + 1	New ← Old + 1
Mem[x] ← New	Mem[x] ← New

- The answer depends on the interleaving of the operations performed by threads 1 and 2
  - Operations from one thread are (usually) guaranteed to be in program order
  - There is no guarantee on the interleaving of operations from different threads

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) Old ← 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	(0) Old ← Mem[x]	
3		(1) New ← Old + 1
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

- Only timing scenarios 1 and 2 give a correct result

Thread 1	Thread 2
Old $\leftarrow$ Mem[x]	
New $\leftarrow$ Old + 1	
Mem[x] $\leftarrow$ New	
	Old $\leftarrow$ Mem[x]
	New $\leftarrow$ Old + 1
	Mem[x] $\leftarrow$ New

Thread 1	Thread 2
	Old $\leftarrow$ Mem[x]
	New $\leftarrow$ Old + 1
	Mem[x] $\leftarrow$ New
Old $\leftarrow$ Mem[x]	
New $\leftarrow$ Old + 1	
Mem[x] $\leftarrow$ New	

- We have a race condition
  - Depending on the interleaving of operations, executions of the same program will give different results
- To ensure we get the correct result always, we have to use atomic operations
  - An operation that performs several operations (read, modify, write) as if they were a single (atomic) one

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 same location will typically be held in a queue until its turn
  - All threads perform the atomic operation **serially**

Atomic Operations in CUDA

- Functions named atomic<*Operation*>(…)
  - Add, sub, inc, dec, min, max, and, or, xor, exch (exchange), CAS (compare and swap)
  - Check the *CUDA C programming Guide* for details

- Atomic Add

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

  *"Equivalent" to: *address += val;*

  Reads the integer pointed to by **address (old)** in global or shared memory, computes **old + val**, and stores the result back to 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

  float atomicAdd(float address, float val);*

- Double-precision floating-point atomic add (since CUDA 8, Pascal GPUs)

  double atomicAdd(double address, double val);*

- Half-precision floating-point atomic add (since Volta GPUs)

  __half atomicAdd(__half address, __half val);*

Other Atomic Operations in CUDA

- Atomic Exchange (or Swap)

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

  – Sets ****address*** = ***val*** and returns the previous value of ****address***

  – The read of the previous value and the write are performed atomically

- Atomic Compare and Swap

  int atomicCAS(int **address**, int **compare**, int **val**);*

  – Similar to the previous one but only updates ****address*** if its value is equal to ***compare***

    • Read *address (old)

    • If old == compare

      – *address = val

    • Else

      – *address is not changed

Implementing atomic operations with atomicCAS()

- For example, double-precision atomicAdd() for devices with compute capability < 6.0 can be implemented as follows:

```
__device__ double atomicAdd(double* address, double val)
{
    // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
    unsigned long long* address_as_ull = (unsigned long long int*)address;
    unsigned long long old = *address_as_ull;
    unsigned long long assumed, new;
    do {
        assumed = old;
        new = _d_as_ull(val + _ull_as_d(assumed))
        old = atomicCAS(address_as_ull,  assumed,  new);
    } while (assumed != old);

    return _ull_as_d(old);
}
```

Histogramming: Objective

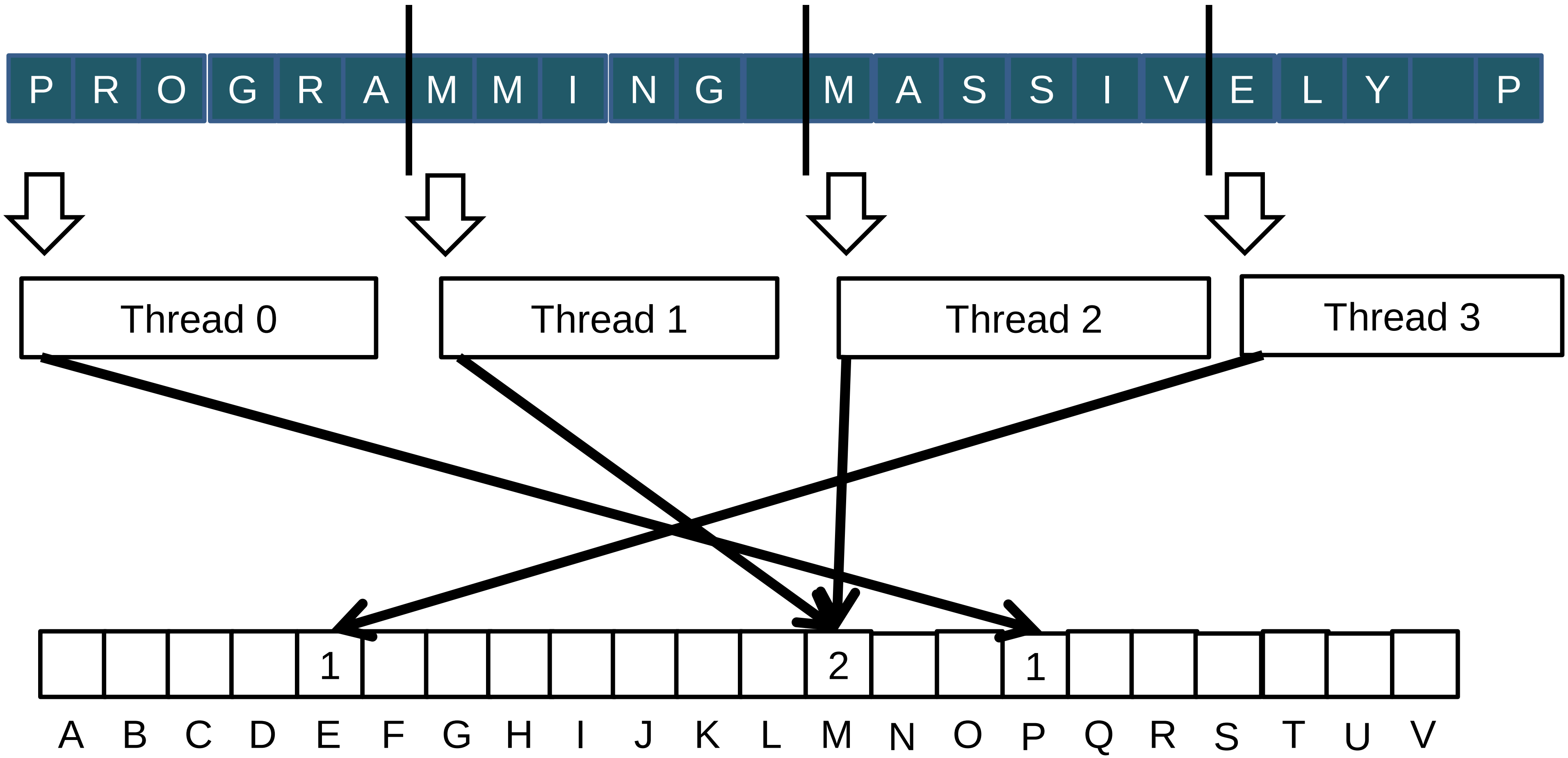
- To learn practical histogram programming techniques
  - Basic histogram algorithm using atomic operations
  - Privatization
  - Alternative histogram algorithm without atomic operations

A Histogram Example

- Build a histogram of the frequency of each letter in the sentence “Programming Massively Parallel Processors”
- A(3), C(1), E(1), G(1), …

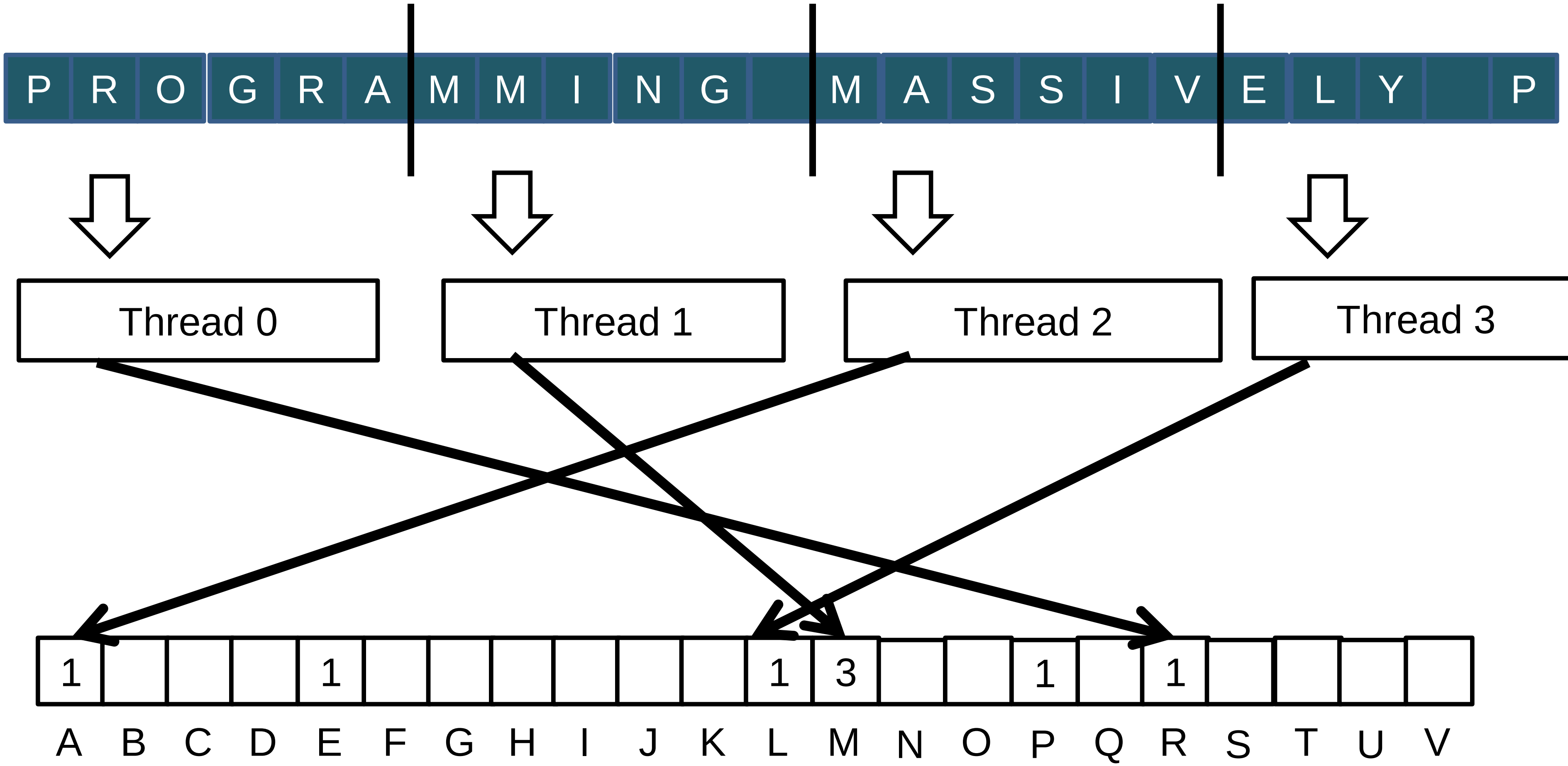
- How do you do this in parallel?

Iteration #1



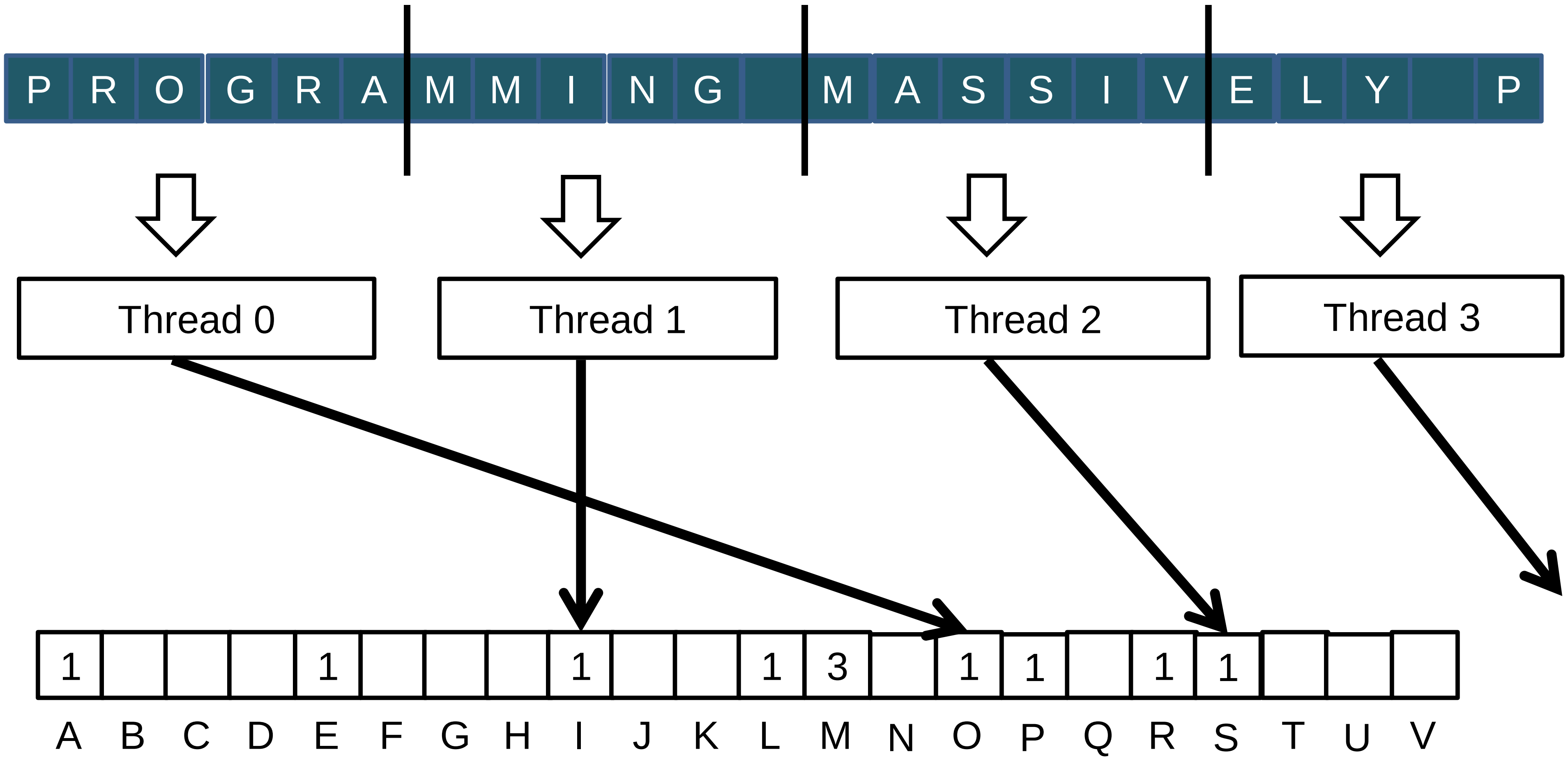
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1
2
1
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #2



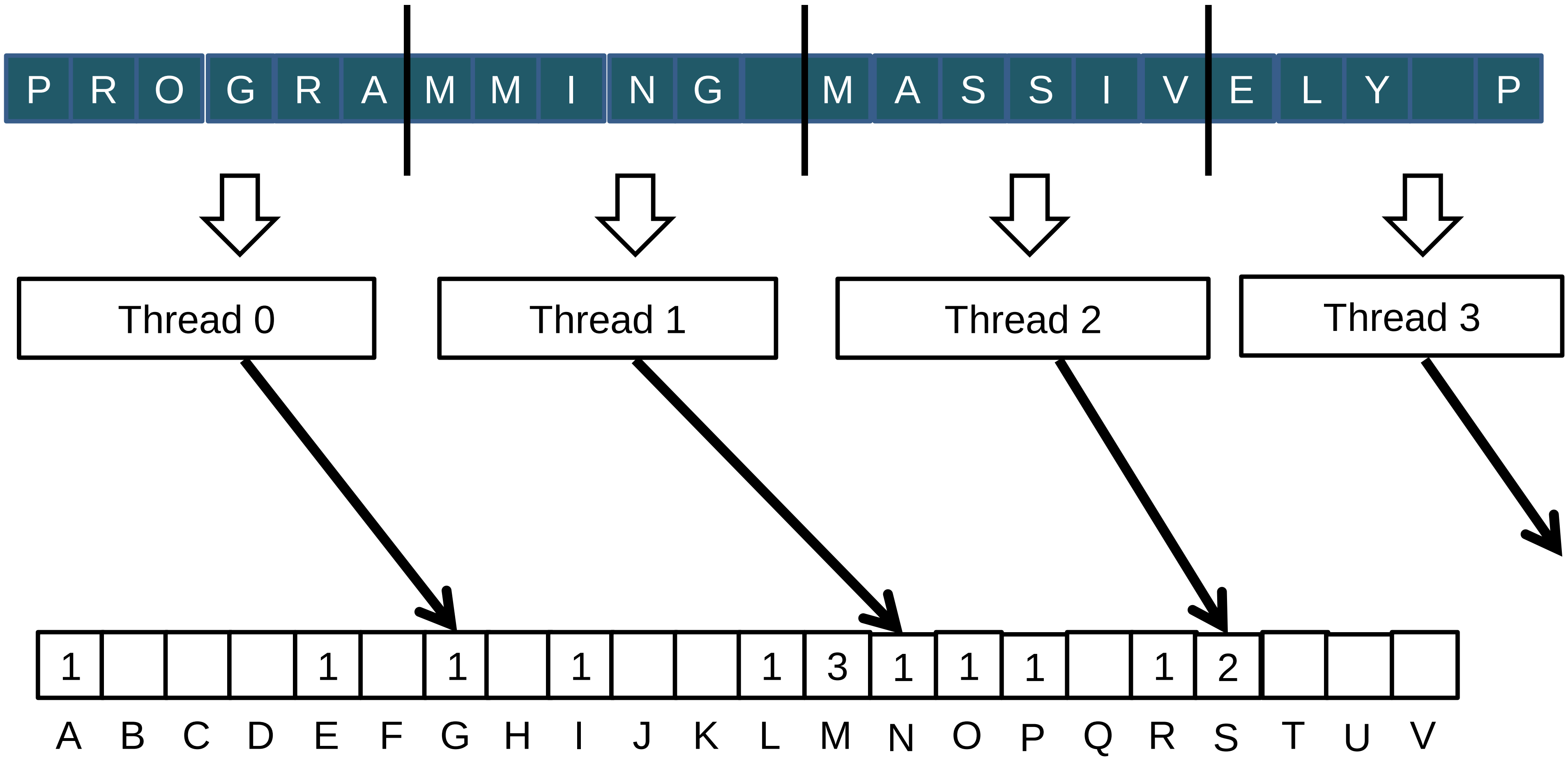
P R O G R A M M I N G M A S S I V E L Y P
Thread 0
Thread 1
Thread 2
Thread 3
1 1 1 3 1 1
A B C D E F G H I J K L M N O P Q R S T U V

Iteration #3



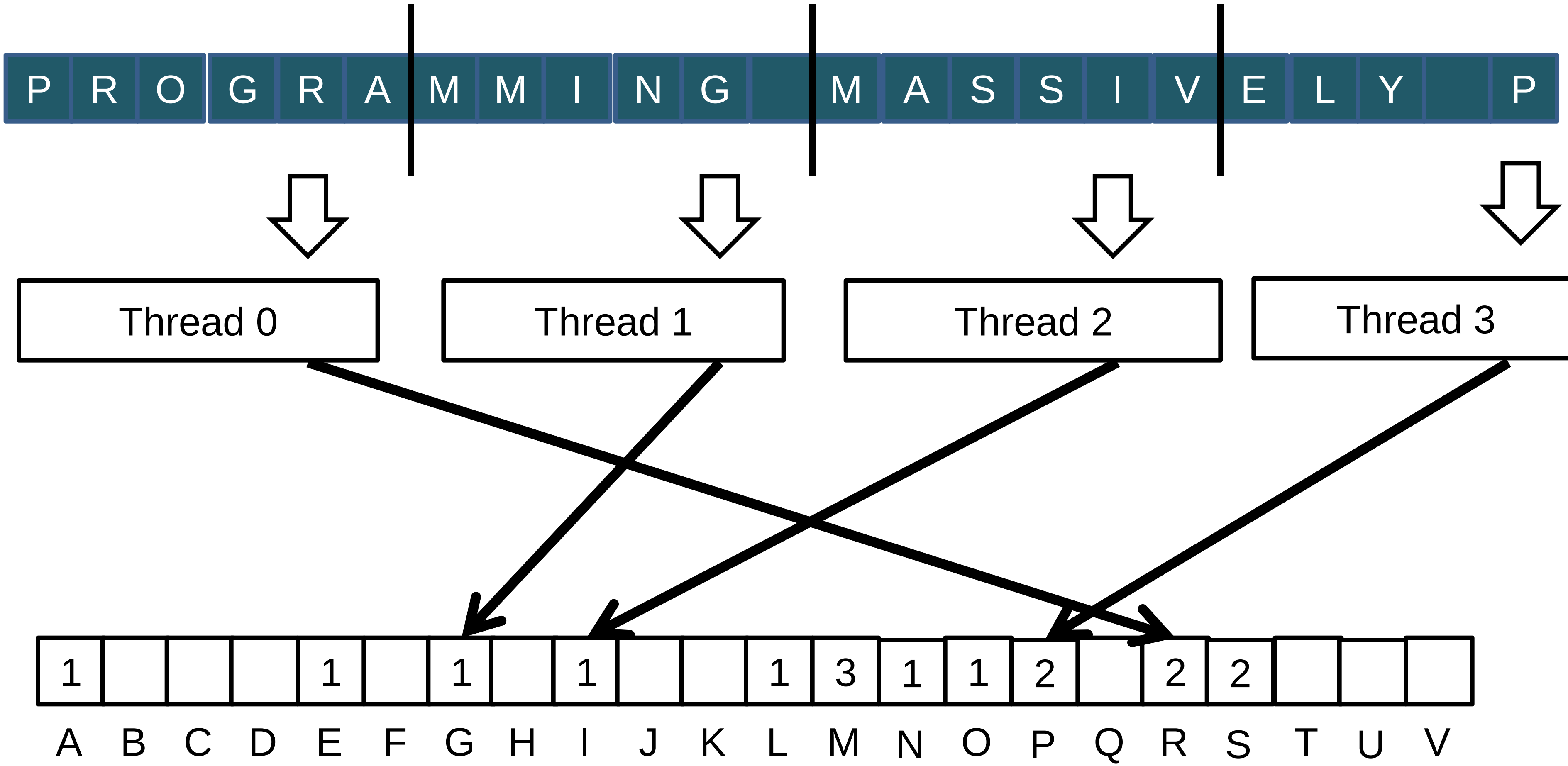
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Iteration #4

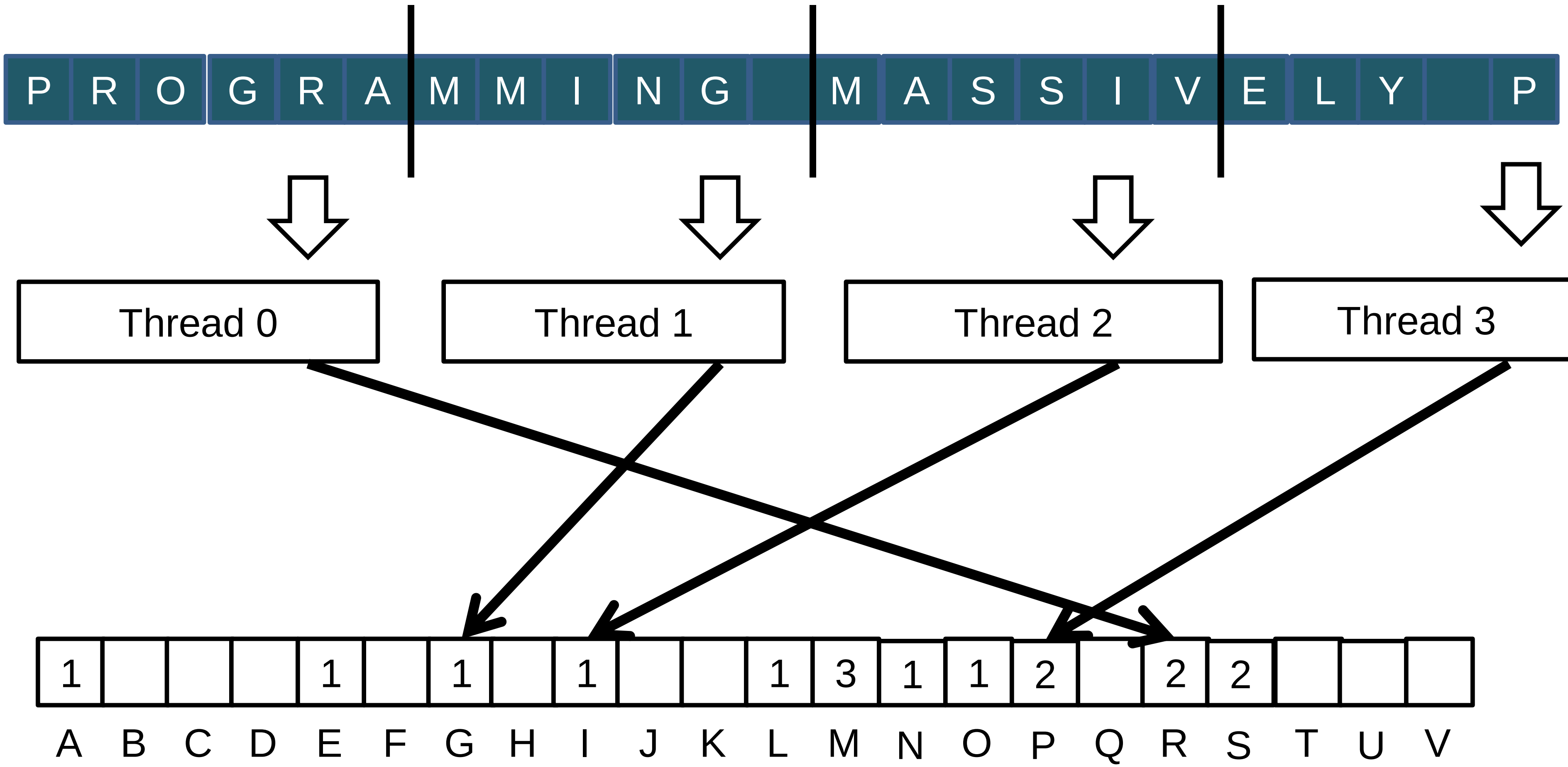


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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

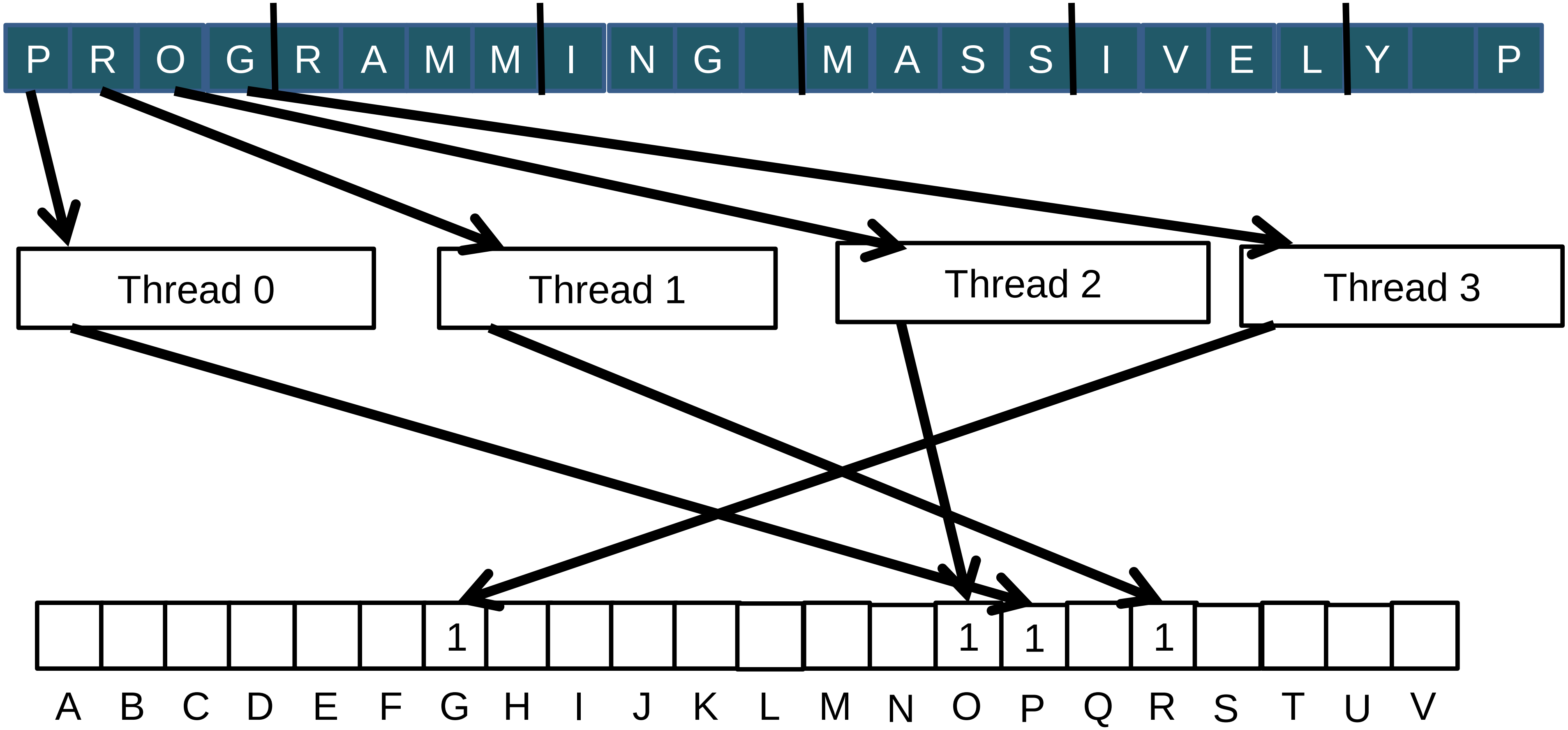
Iteration #5



- It works, but reads from the input array are not coalesced → bad performance

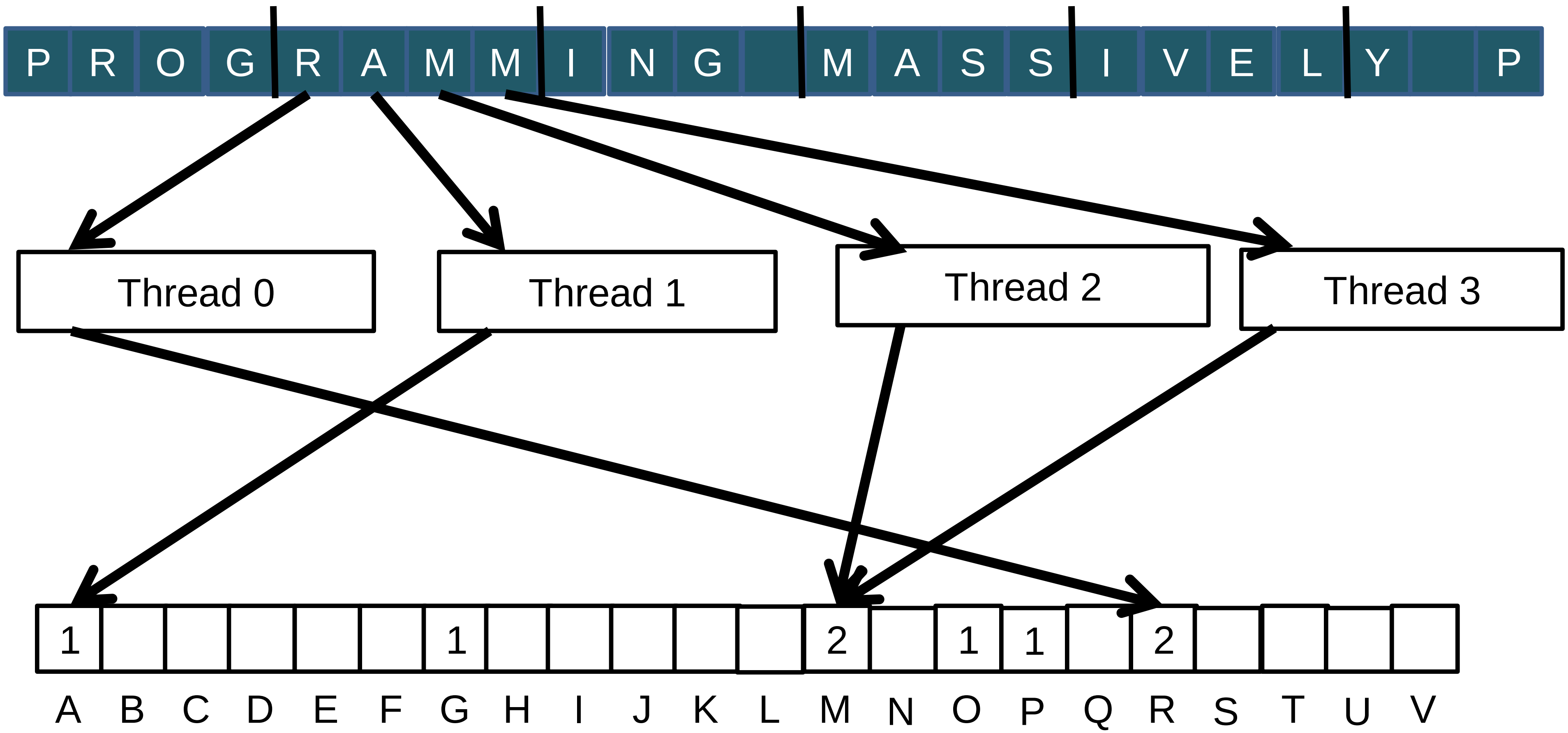
A better approach

- Assign contiguous inputs to the threads, and iterate over the input if its larger than the grid of threads



A better approach

- Assign contiguous inputs to the threads, and iterate over the input if its larger than the grid of threads



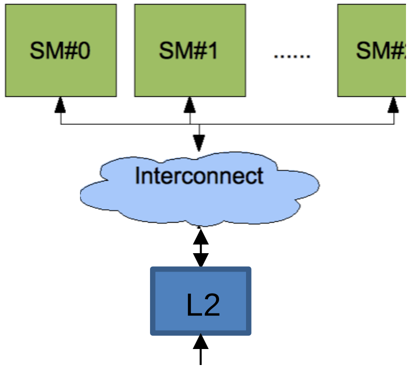
A Histogram Kernel

- The kernel receives a pointer to the input buffer
- Each thread process part of the input in a strided pattern

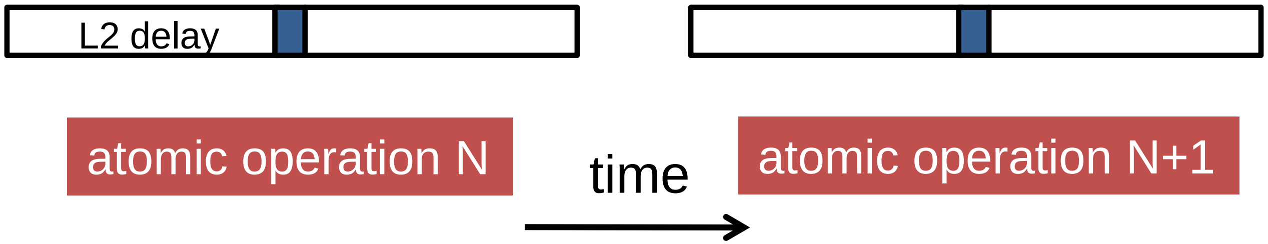
```
__global__ void
histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

// stride is total number of threads
    int stride = blockDim.x * gridDim.x;
// All threads handle blockDim.x * gridDim.x consecutive elements
   while (i < size) {
       atomicAdd( &(histo[buffer[i]]), 1);
       i += stride;
   }
}
```

Cost of Atomic Operations

- Global memory atomics are managed at the L2 cache
  - Must work across SMs



- Atomic operations **on the same address** are serialized



- If **many threads** attempt to do atomic operations on **the same location** (high contention), the performance penalty can be high
  - All atomics are started in parallel, but the HW does them one at a time
  - Possible mitigation → privatization

Privatization in Shared Memory

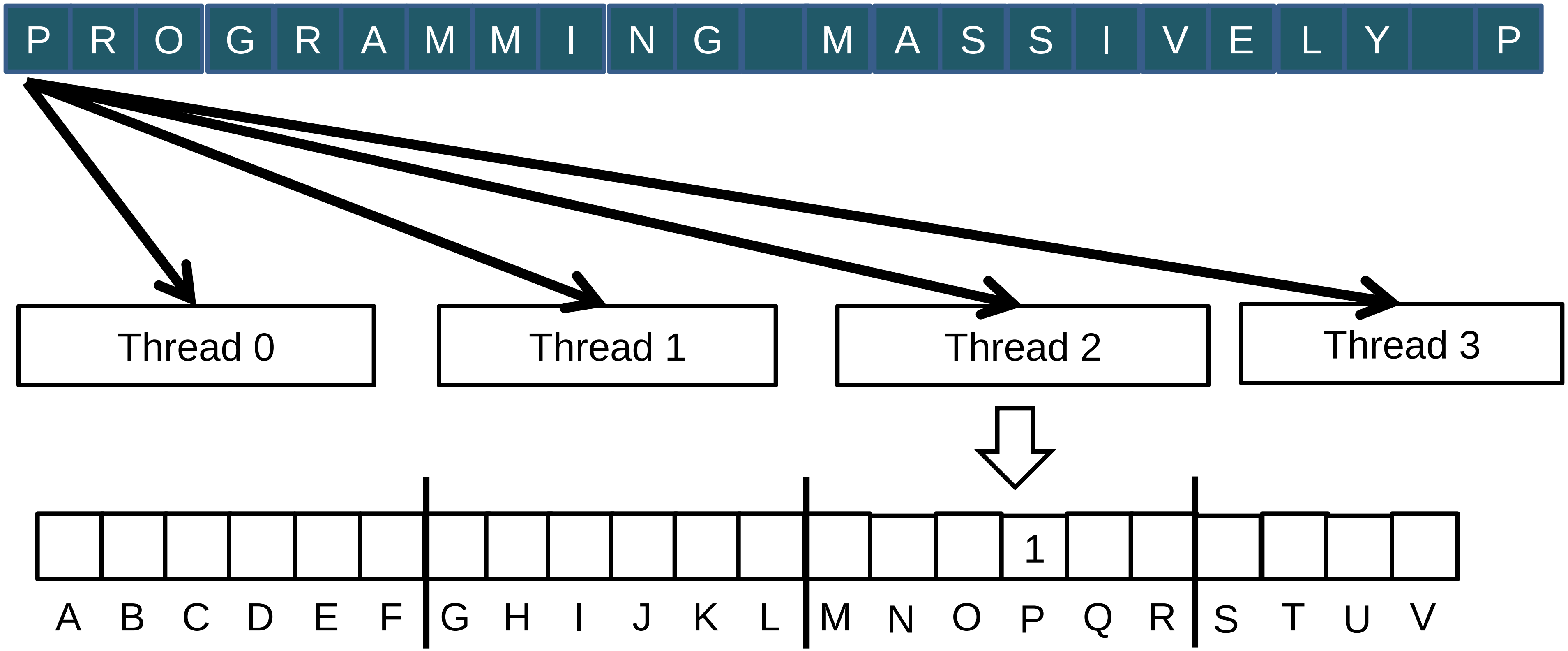
- Each thread block has its private array of bins in shared memory
  - Less threads potentially contending on the bins
  - Atomics on shared memory are faster than on global memory (L2)

- After the whole input is processed, the threads of each thread block have to update the global bins with their partial counts

- The histogram size (number of bins) needs to be small
  - To fit into shared memory

Alternative Histogramming approach

- Split the bins across the threads
- All threads iterate the input array looking for the letters of their bins
- Why don't we need atomic operations in this case?



Alternative Histogramming approach

- Known as Gather design
  - The one using atomic ops is known as Scatter design

- Pros
  - No need for atomic operations → no contention
    - Each bin is only accessed by one thread

- Cons
  - All threads have to iterate over the whole input vector

- Which is better depends on:
  - Input size
  - Number of bins
  - Number of conflicting updates to the same bin
    - E.g. if a large part of the input elements have the same value
  - Latency of atomic operations

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
marc.jorda@bsc.es, antonio.pena@bsc.es