Multicore Computing
Lecture02 – Paradigm Shift

SUNG KYUN KWAN UNIVERSITY

남 범 석

bnam@skku.edu

Classifying Parallel Systems - Flynn's Taxonomy

- Distinguishes multi-processor computer architectures along the two independent dimensions
  - ***Instruction*** and ***Data***
  - Each dimension can have one state: ***Single*** or ***Multiple***

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD **Multiple** instruction stream Single data stream	(MIMD) **Multiple** instruction stream Multiple data stream

classic von Neumann

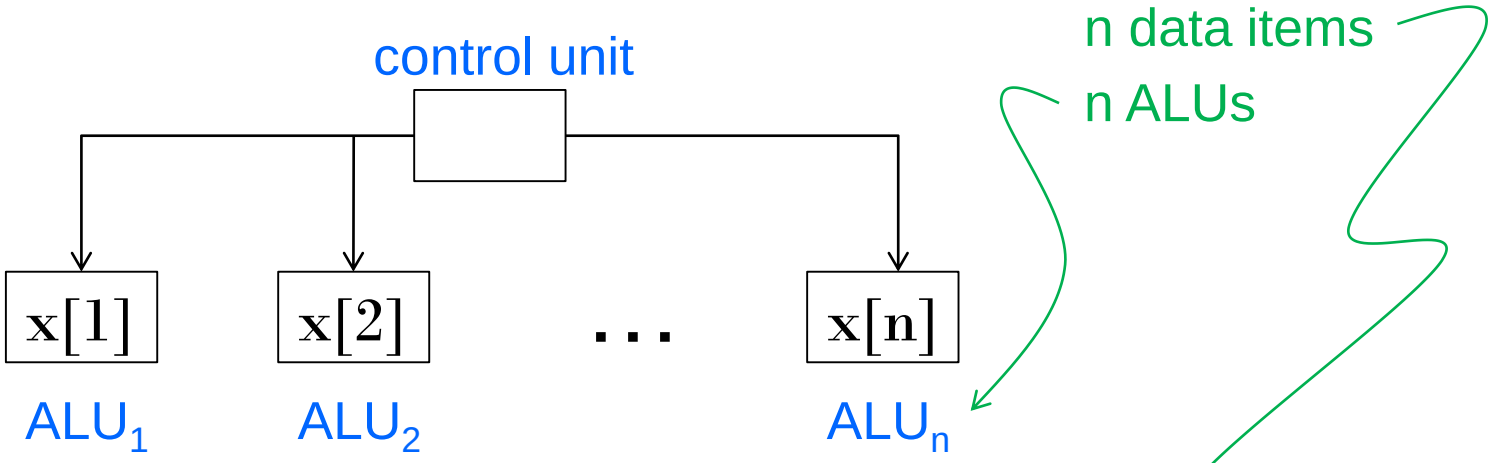
Processor arrays and vector machines

not covered

Most common parallel computer systems

SIMD

- Parallelism achieved by dividing data among the processors.
- Applies the same instruction to multiple data items.
- Called data parallelism.




```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

SIMD

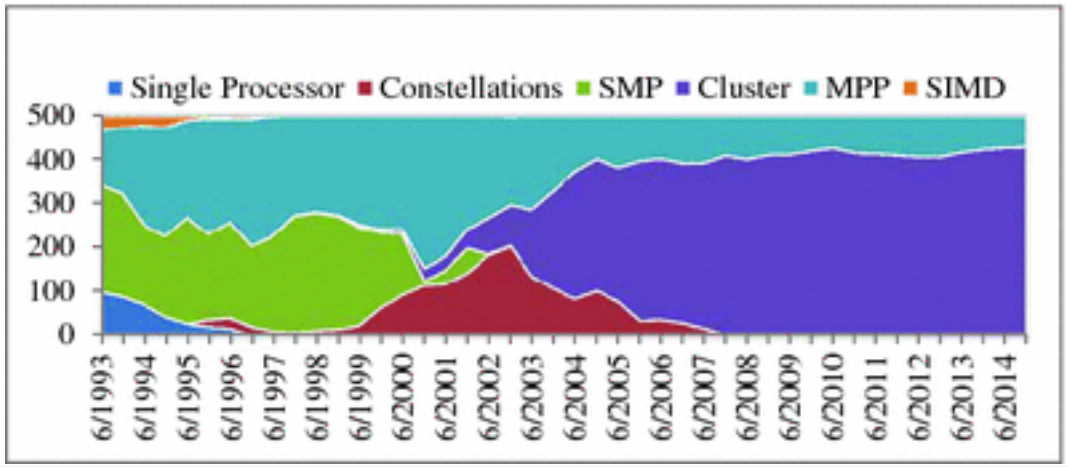
- What if we don't have as many ALUs as data items?
- Divide the work and process iteratively.
- Ex. m = 4 ALUs and n = 15 data items.

Round3	ALU_1	ALU_2	ALU_3	ALU_4
1	X[0]	X[1]	X[2]	X[3]
2	X[4]	X[5]	X[6]	X[7]
3	X[8]	X[9]	X[10]	X[11]
4	X[12]	X[13]	X[14]	

- All ALUs are required to execute the same instruction, or remain idle.
- In classic design, they must also operate synchronously.
- Efficient for large data parallel problems, but not other types of more complex parallel problems.

Parallel Architecture Types

- Instruction-Level Parallelism
  - Parallelism captured in instruction processing
- Vector processors ('70s)
  - Operations on multiple data stored in vector registers
- Shared-memory Multiprocessor (SMP)
  - Multiple processors sharing memory
- Massively Parallel Processor (MPP)
- Cluster
  - Multiple computer connect via network

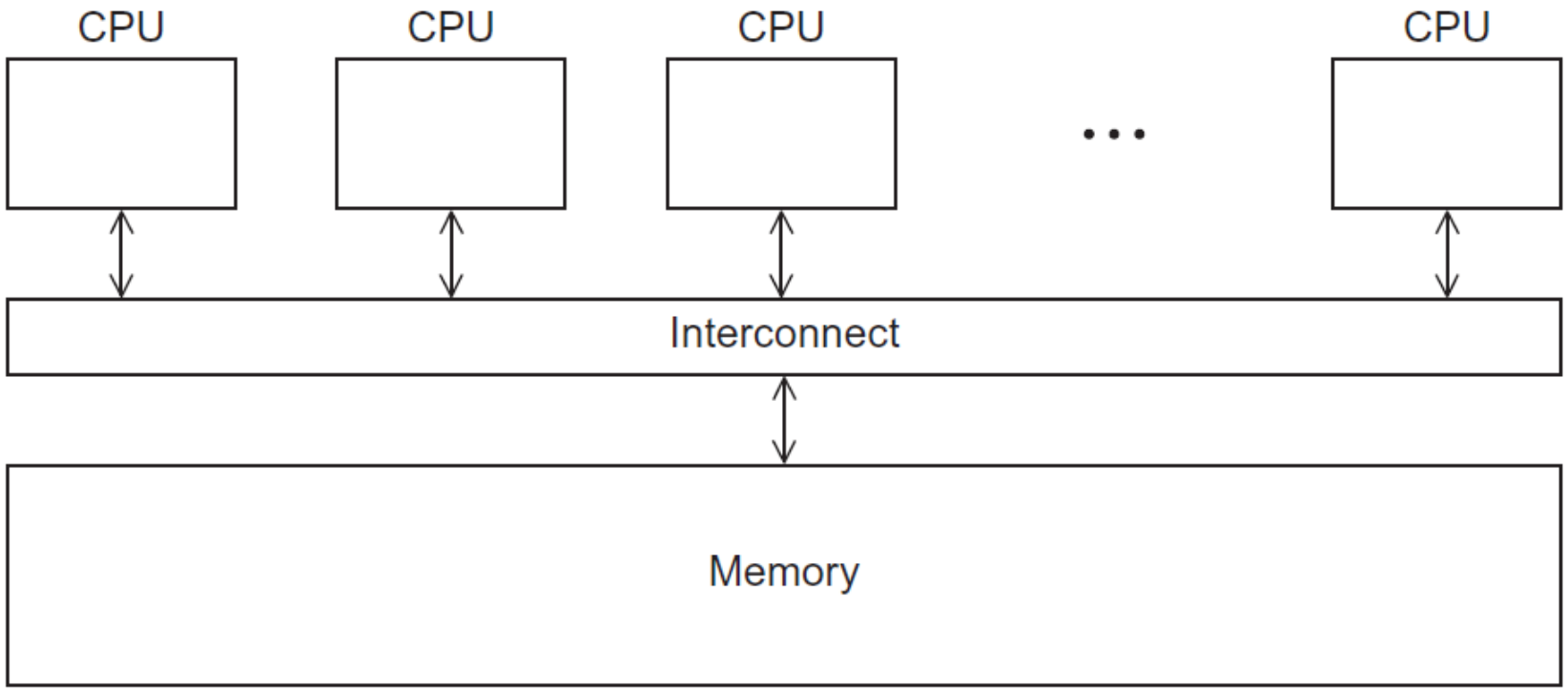


Vector processors

- Vector registers.
  - Capable of storing a vector of operands and operating simultaneously on their contents.
- Vectorized and pipelined functional units.
  - The same operation is applied to each element in the vector
- Vector instructions.
  - Operate on vectors rather than scalars.
- Interleaved memory – High memory bandwidth.
  - Multiple “banks” of memory, which can be accessed independently.
  - Distribute elements of a vector across multiple banks, so reduce or eliminate delay in loading/storing successive elements.

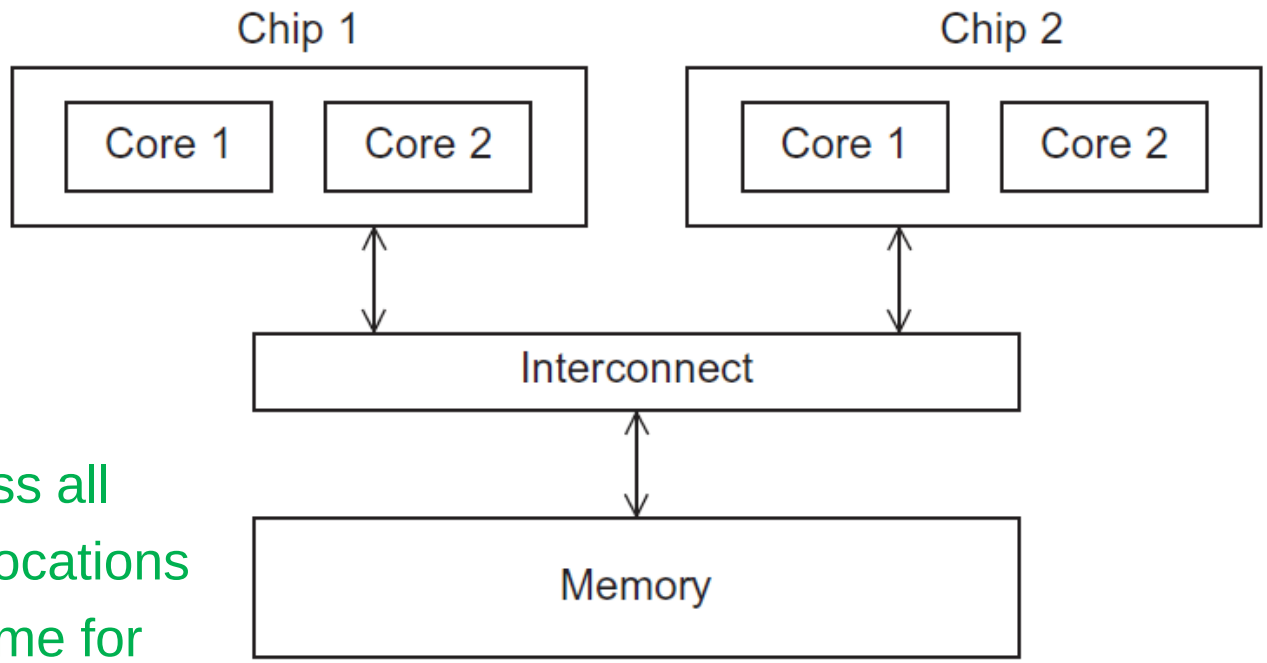
Shared Memory System

- A collection of autonomous processors is connected to a memory system via an interconnection network.
- The processors usually communicate implicitly by accessing shared data structures.



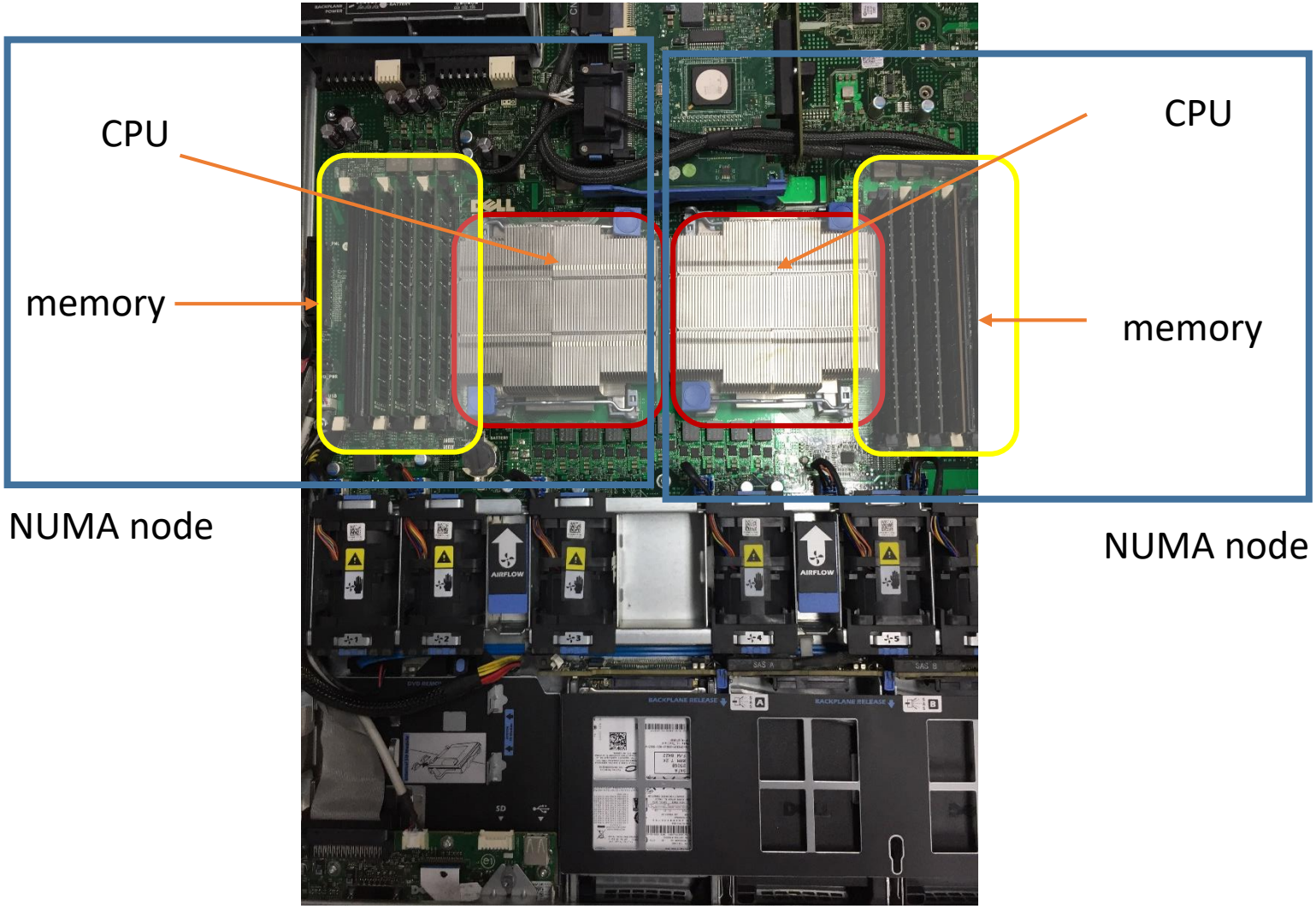
UMA multicore system

- Uniform-Memory-Access



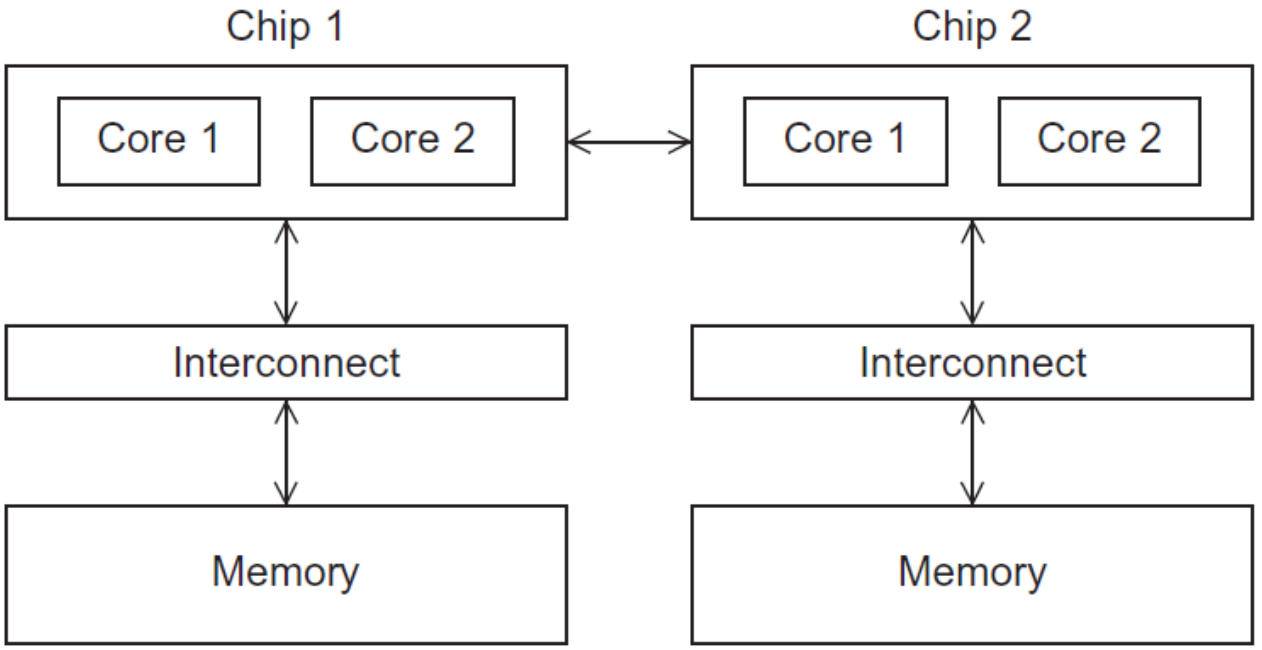
Time to access all the memory locations will be the same for all the cores.

NUMA multicore system



NUMA multicore system

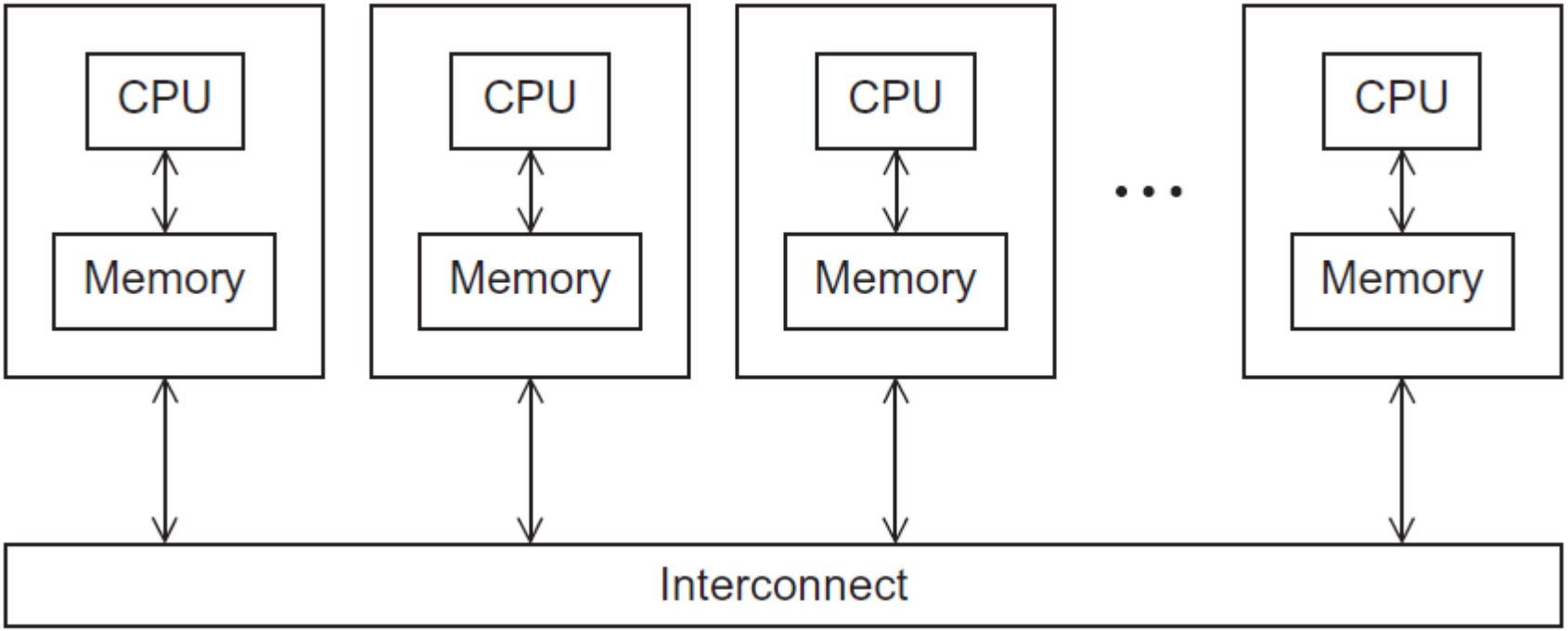
- Non-Uniform Memory Access



A memory location a core is directly connected to can be accessed faster than a memory location that must be accessed through another chip.

Distributed Memory System

- Clusters (most popular)
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.



Time for Paradigm Shift

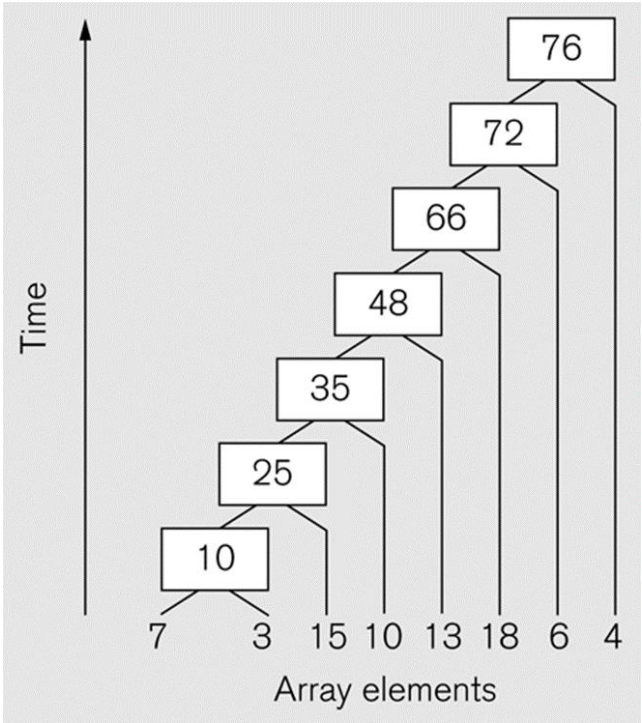
Sum of Numbers: Sequential vs. Parallel Programming

- Simple example: finding the sum of a sequence of numbers
- Assume that the sequence has n data values
  - x[0], x[1], …, x[n-1]

- Iterative sum

```
sum = 0;
for (i=0; i<n; i++)
        sum += x[i];
```

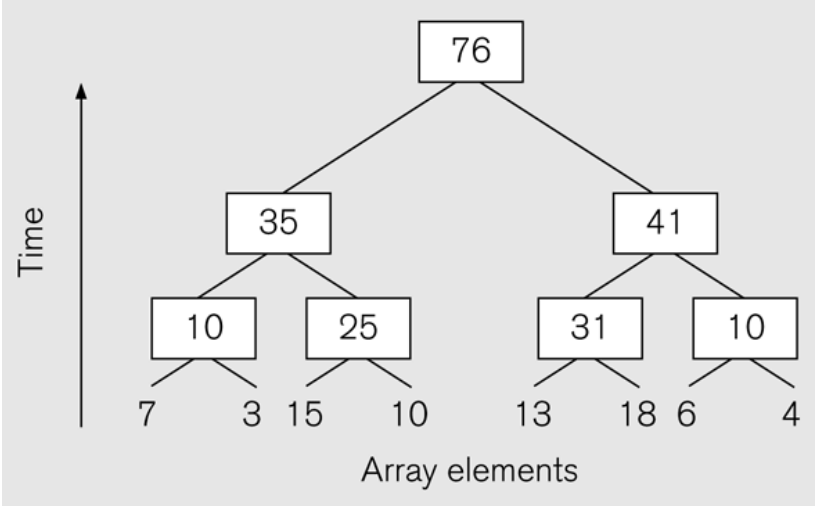
- The “iterative sum” is a sequential algorithm
  - Cannot benefit from parallel systems



Pair-Wise Summation

- Add even/odd pairs of data values, yielding the intermediate sums
  - (x[0]+x[1]), (x[2]+x[3]), …
  - ((x[0]+x[1])+(x[2]+x[3])), …

- The two implementations require the same number of operations
  - No time advantage on a single processor
  - With n/2 processors, all of the additions at the same level of the tree can be computed simultaneously
    - Time complexity is proportional to log n



Parallel Prefix Sum

- For the sequence of n values (x_0, x_1, x_2, …, x_{n-1}), the prefix sum is defined as follows
  - $y_i = x_0 + x_1 + \ldots + x_i$
  - Sequence of the prefix sums: y_0, y_1, …, y_{n-1}
  - Example:

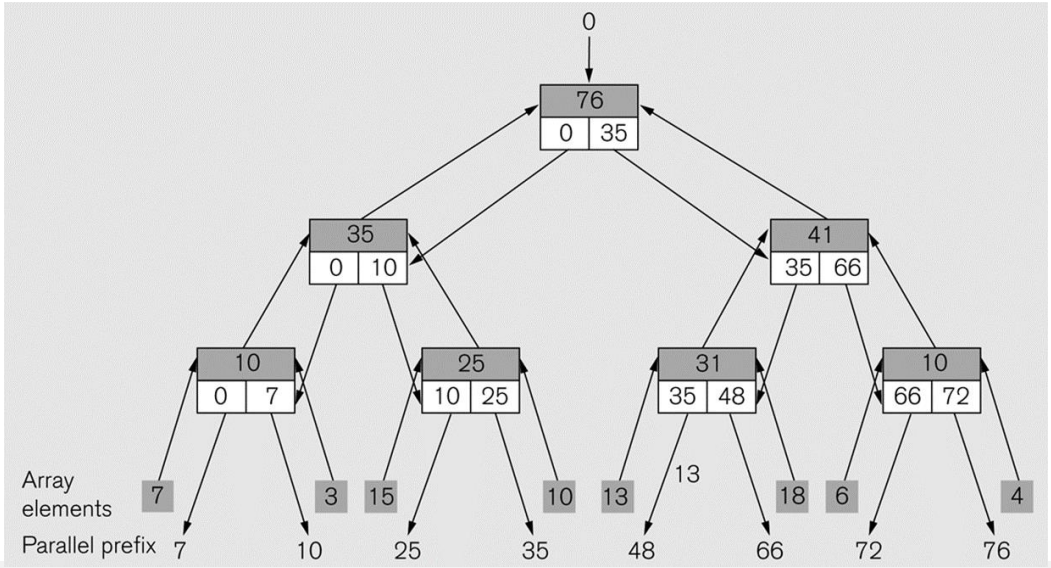
    {2, 7, 9, 4} → {2, 9, 18, 22}

- Can we find a parallel algorithm that computes the sequence of the prefix sums?
  - It seems less obvious than summation
    - All of the intermediate values of the sequential solution are needed

- The pair-wise summation can be modified to compute the prefix values
  - Key idea: each leaf processor storing x_i could compute the value y_i if it knew the sum of all elements to its left (i.e., prefix)
    - If we save that information, we can determine the prefixes without directly summing them

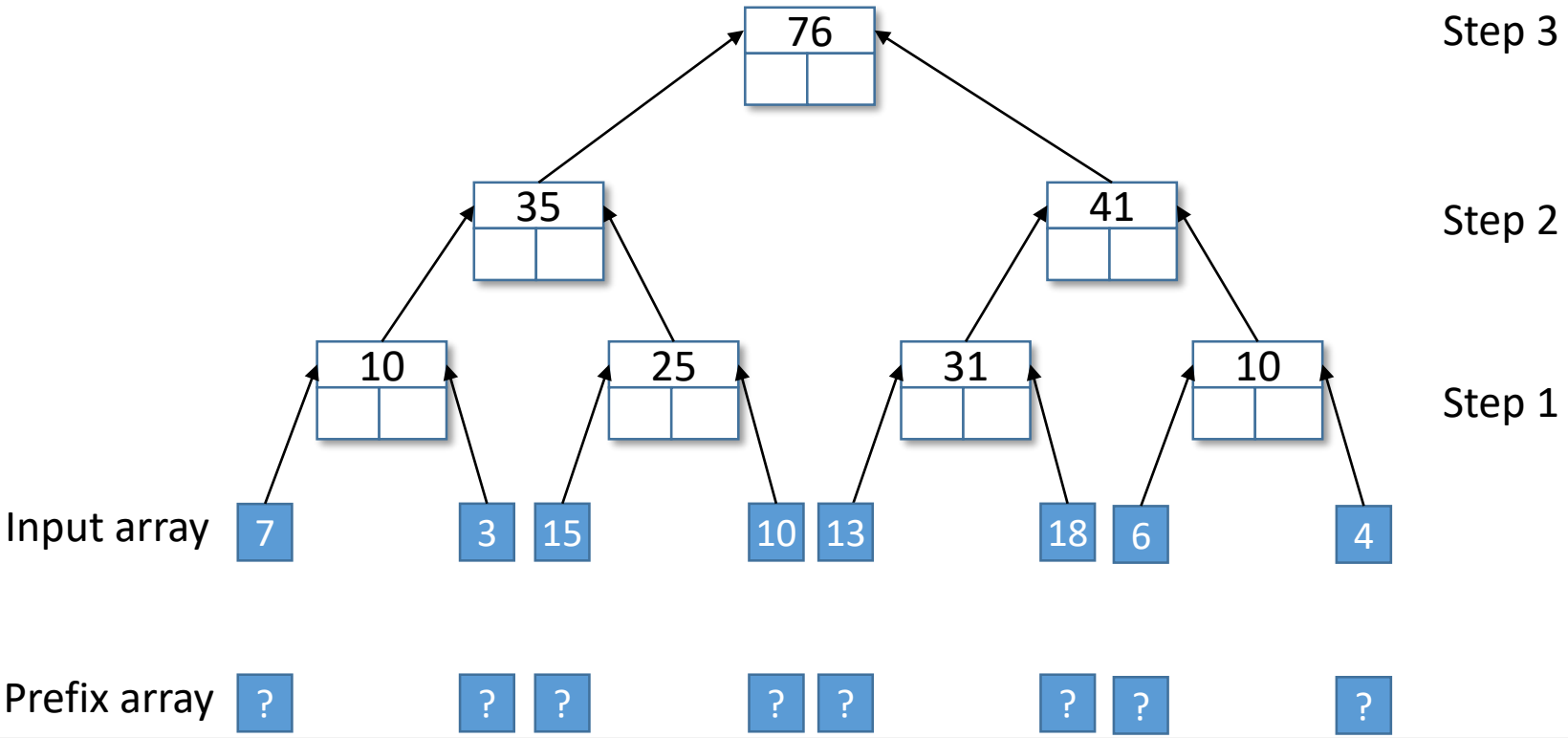
Parallel Prefix Sum

- Compute the grand total at the root by pair-wise sum (as before)
- On completion, imagine the root receiving a 0 (from its imaginary parent)
- All non-leaf nodes
  - Receive a value from their parent
  - Relay that value to their left child
  - Send their right child the sum of the parent's value and their left child's value
    - These are the prefixes of their child nodes
- Leaf nodes add the prefix value above and the saved input



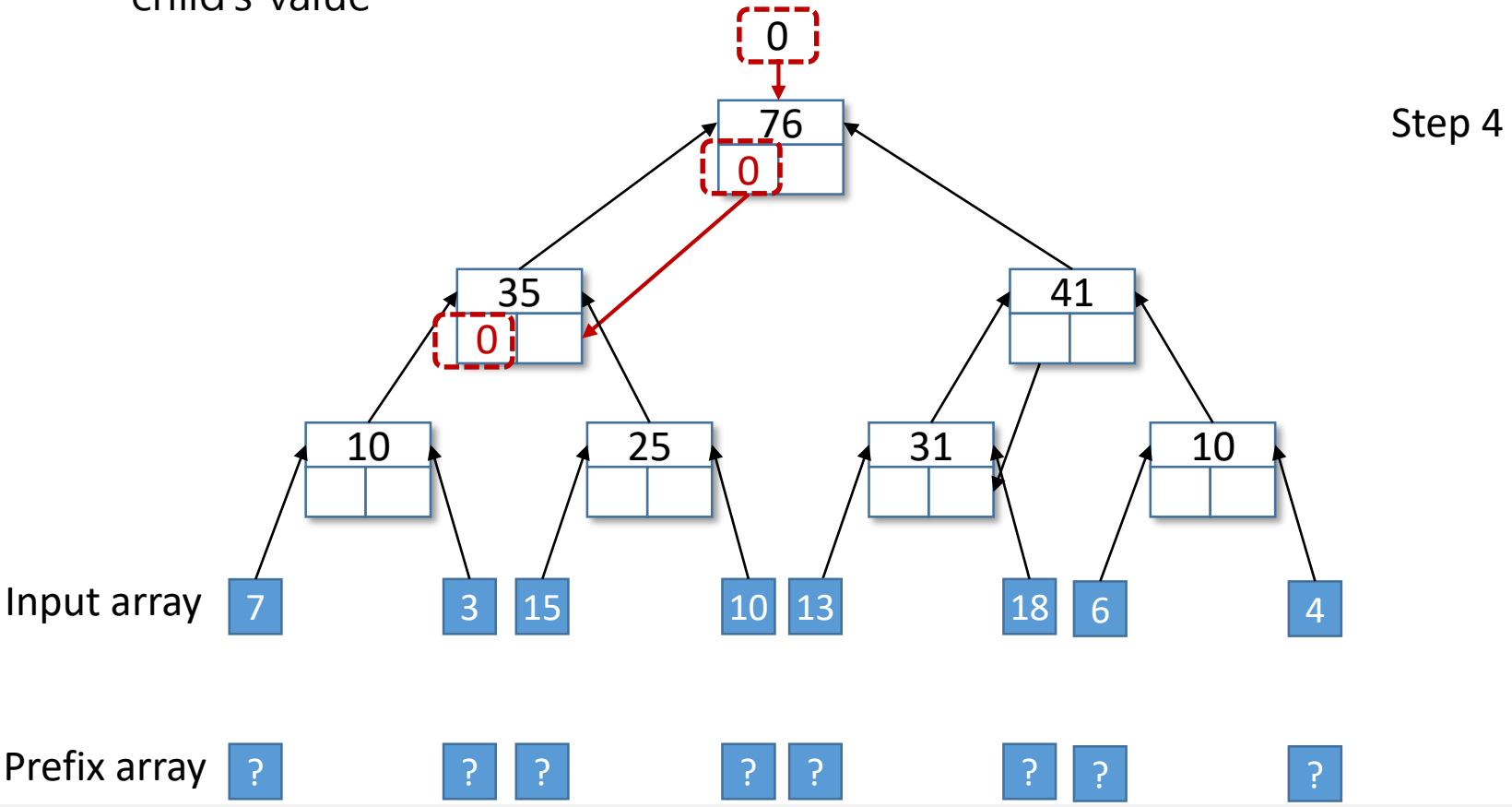
Parallel Prefix Sum

- Phase 1: Up Sweep
  - Pair-wise Sum



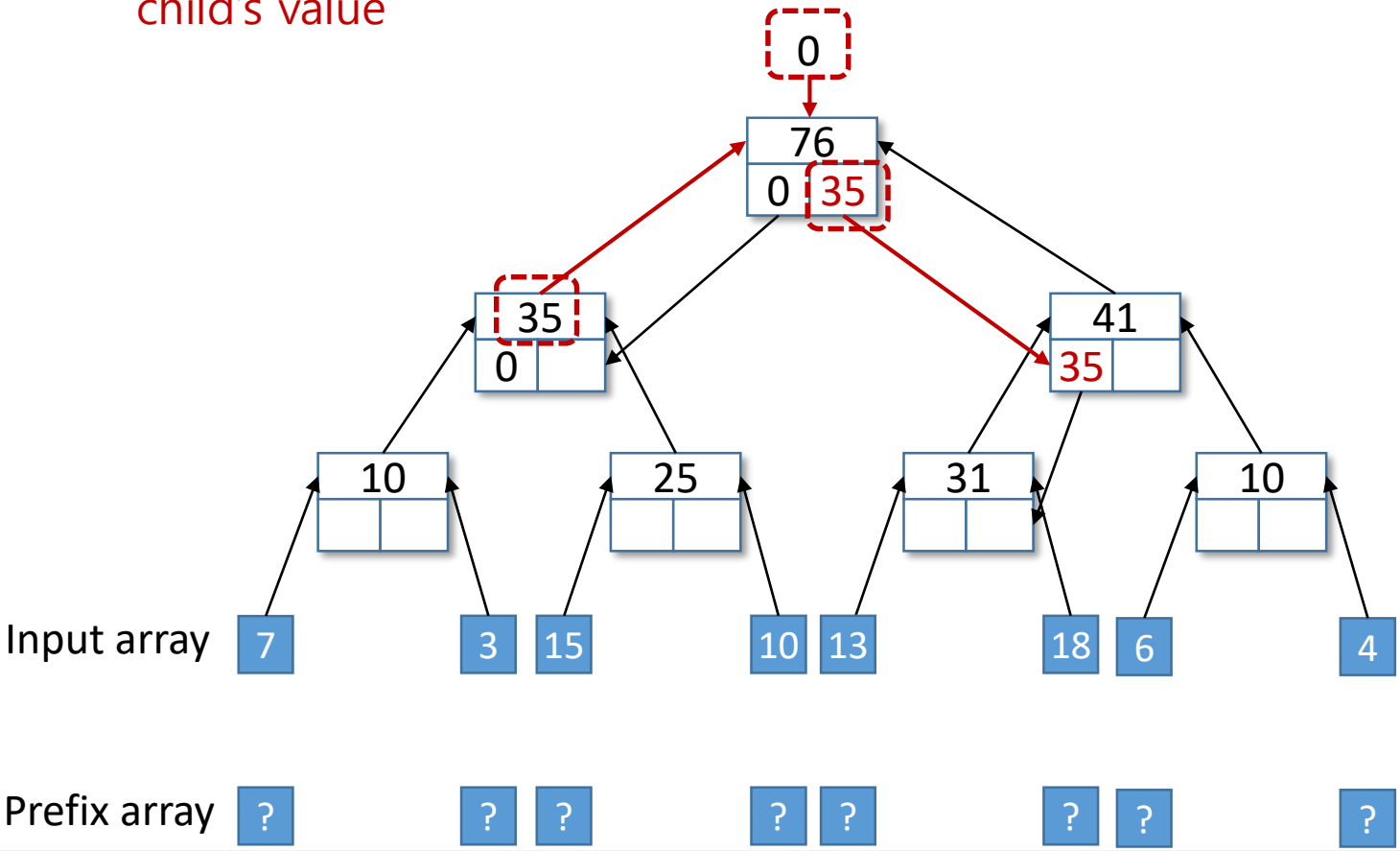
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



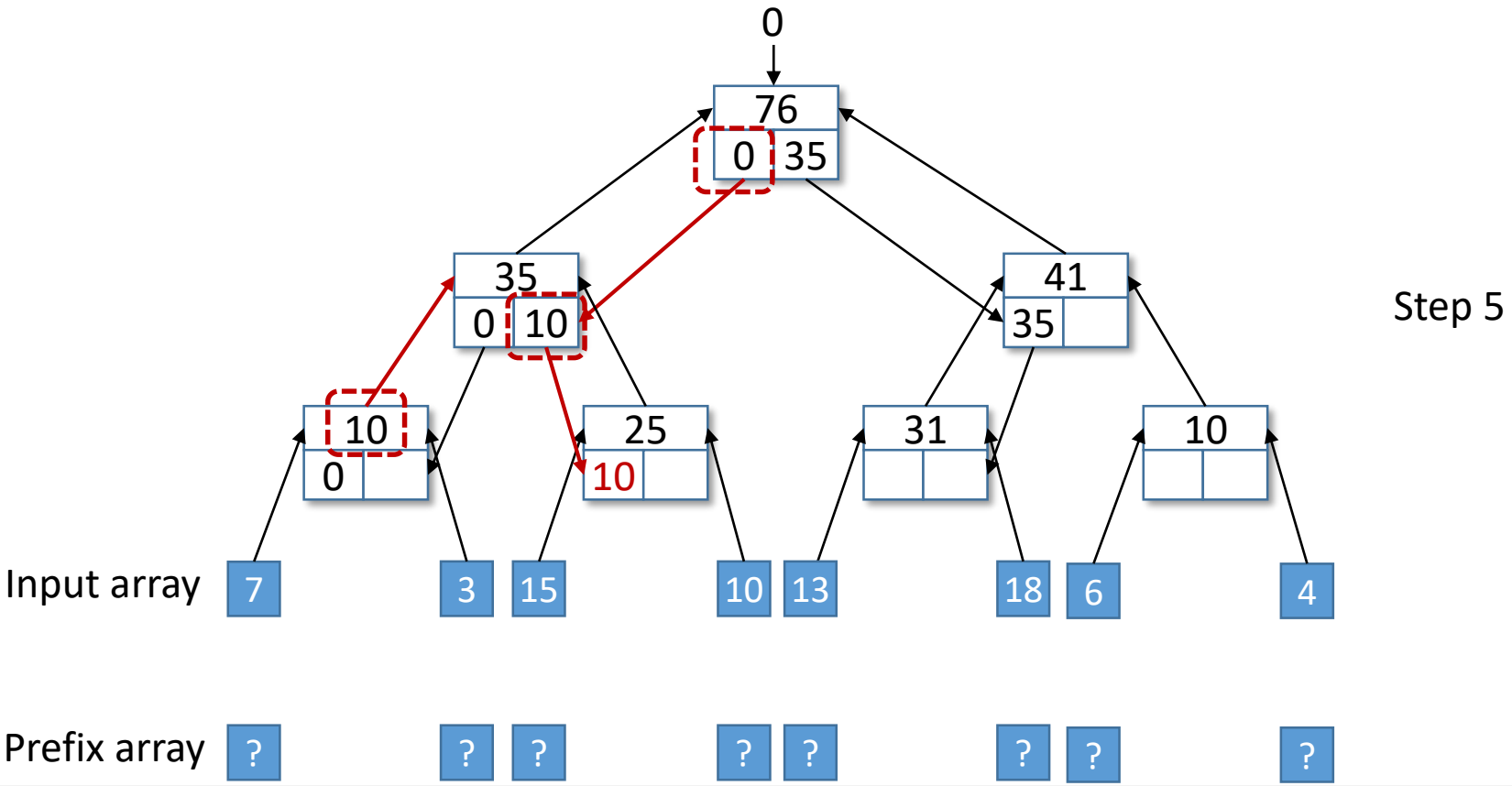
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



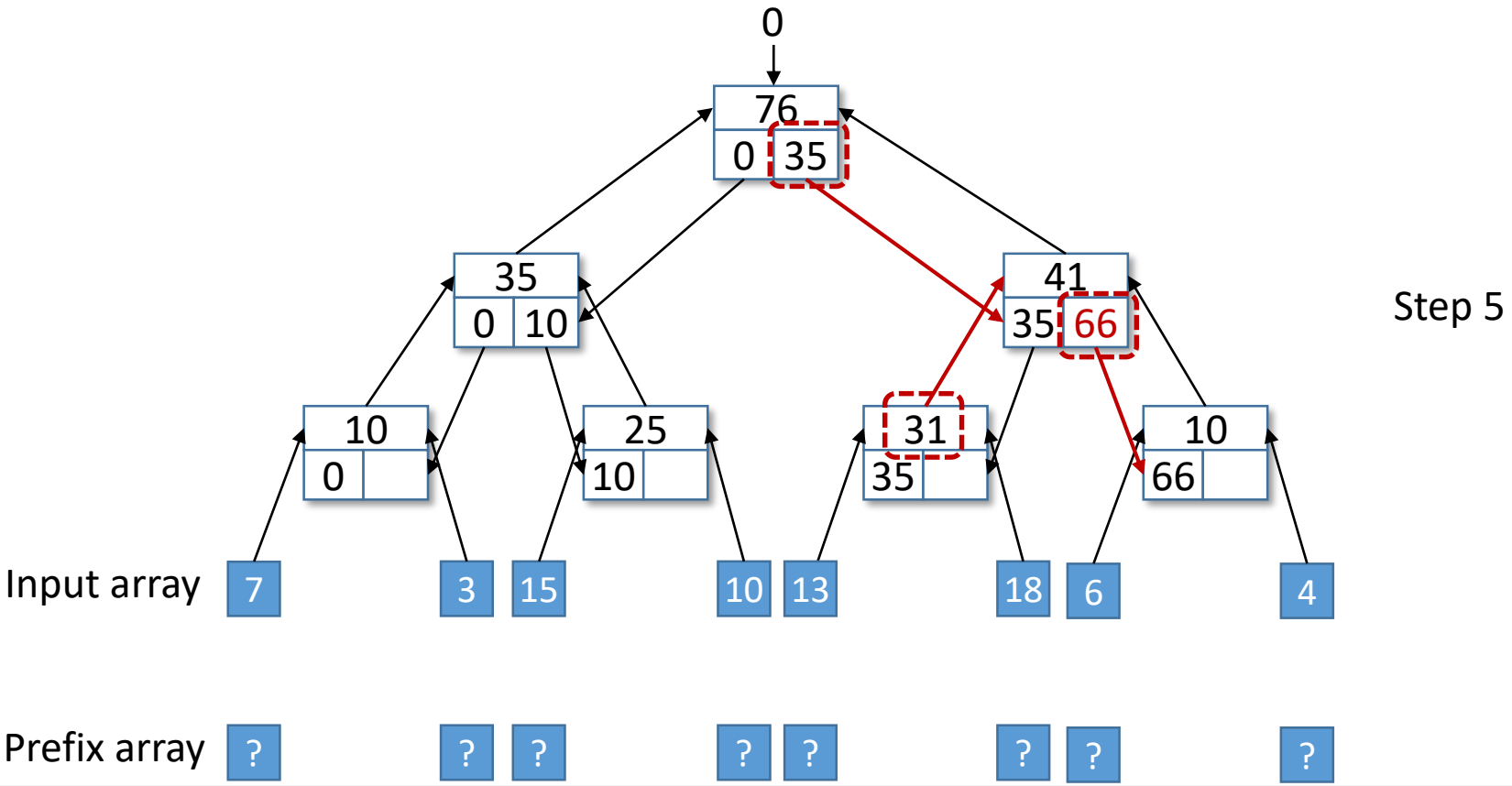
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



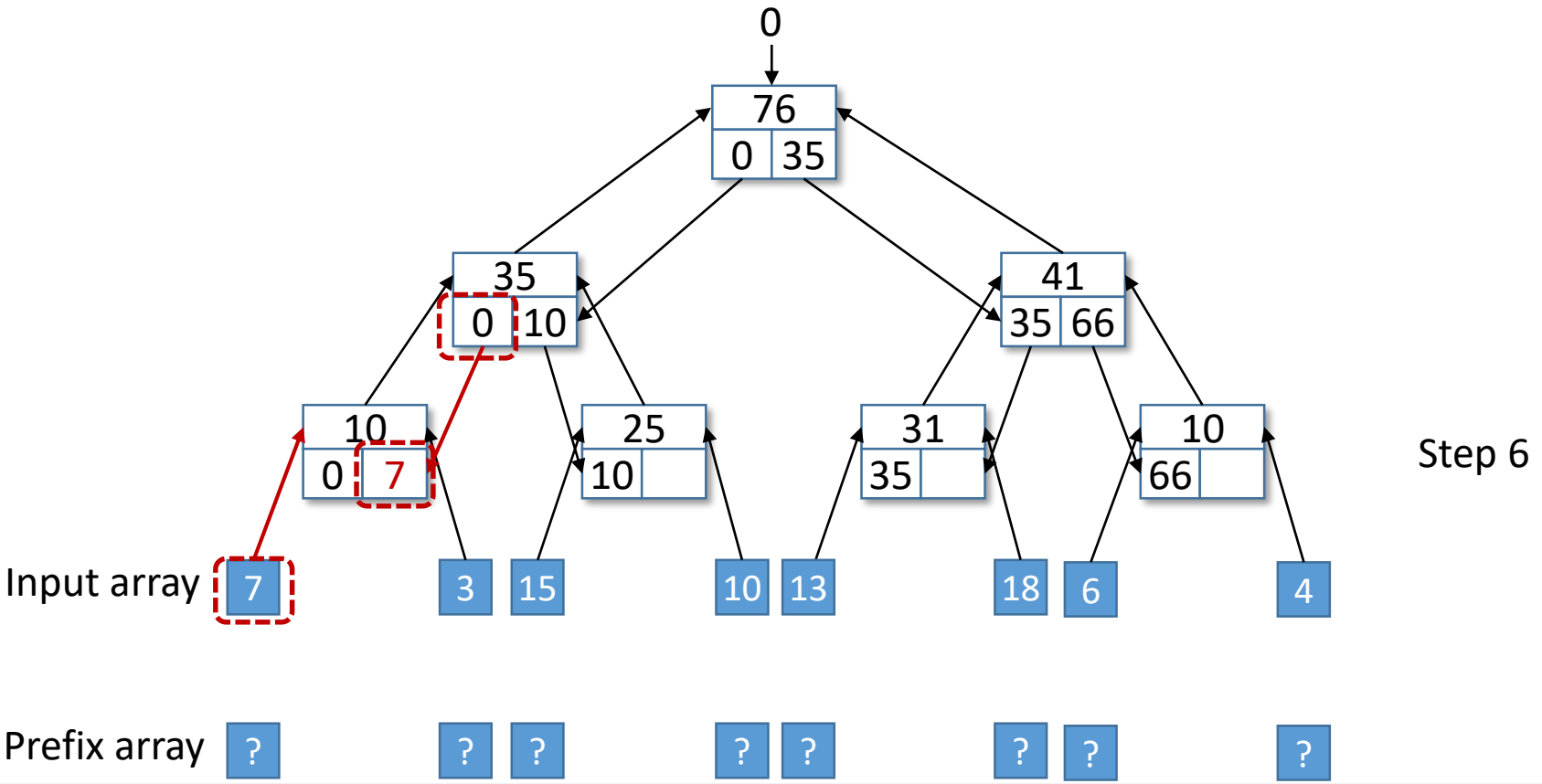
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



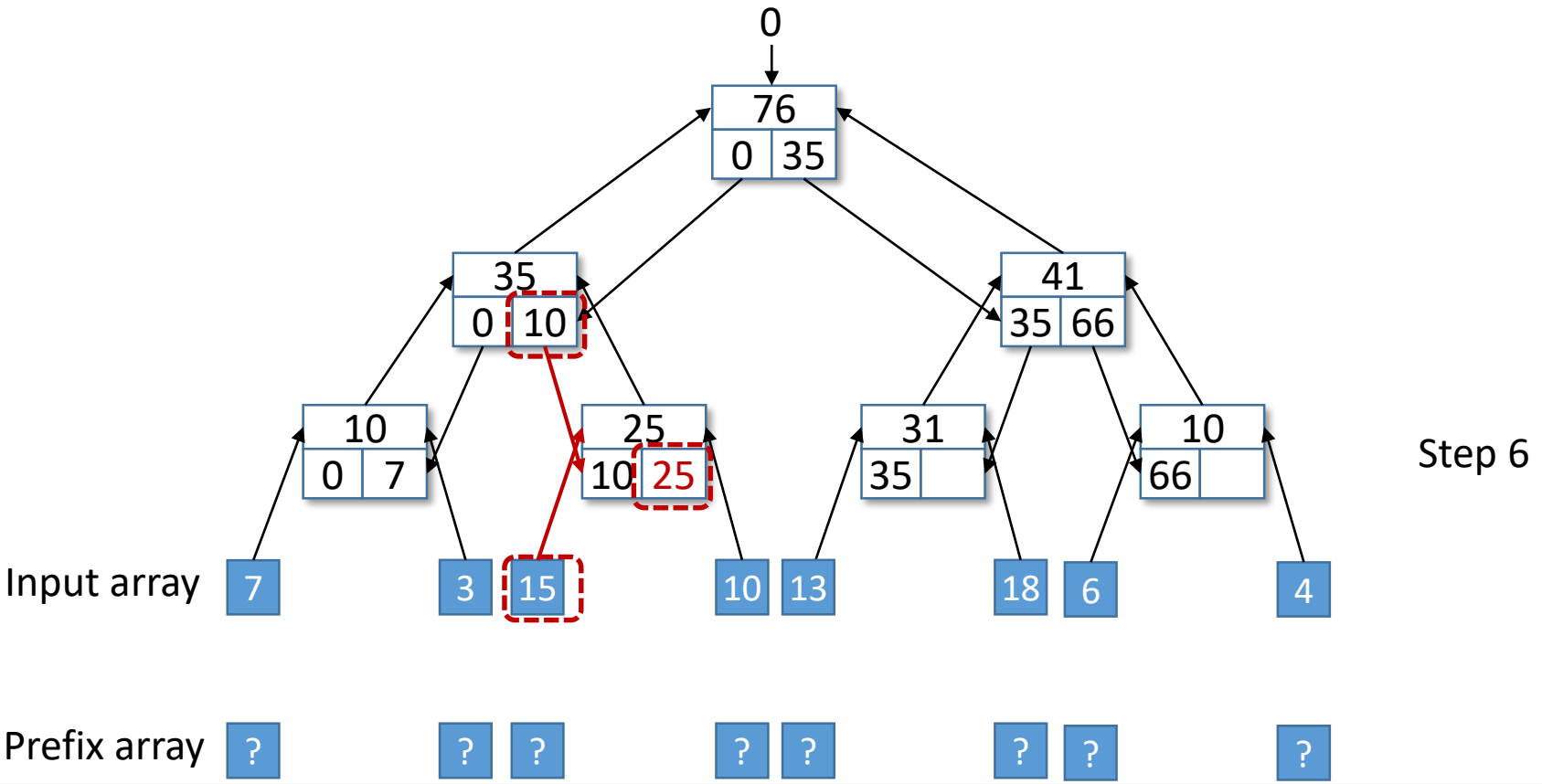
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



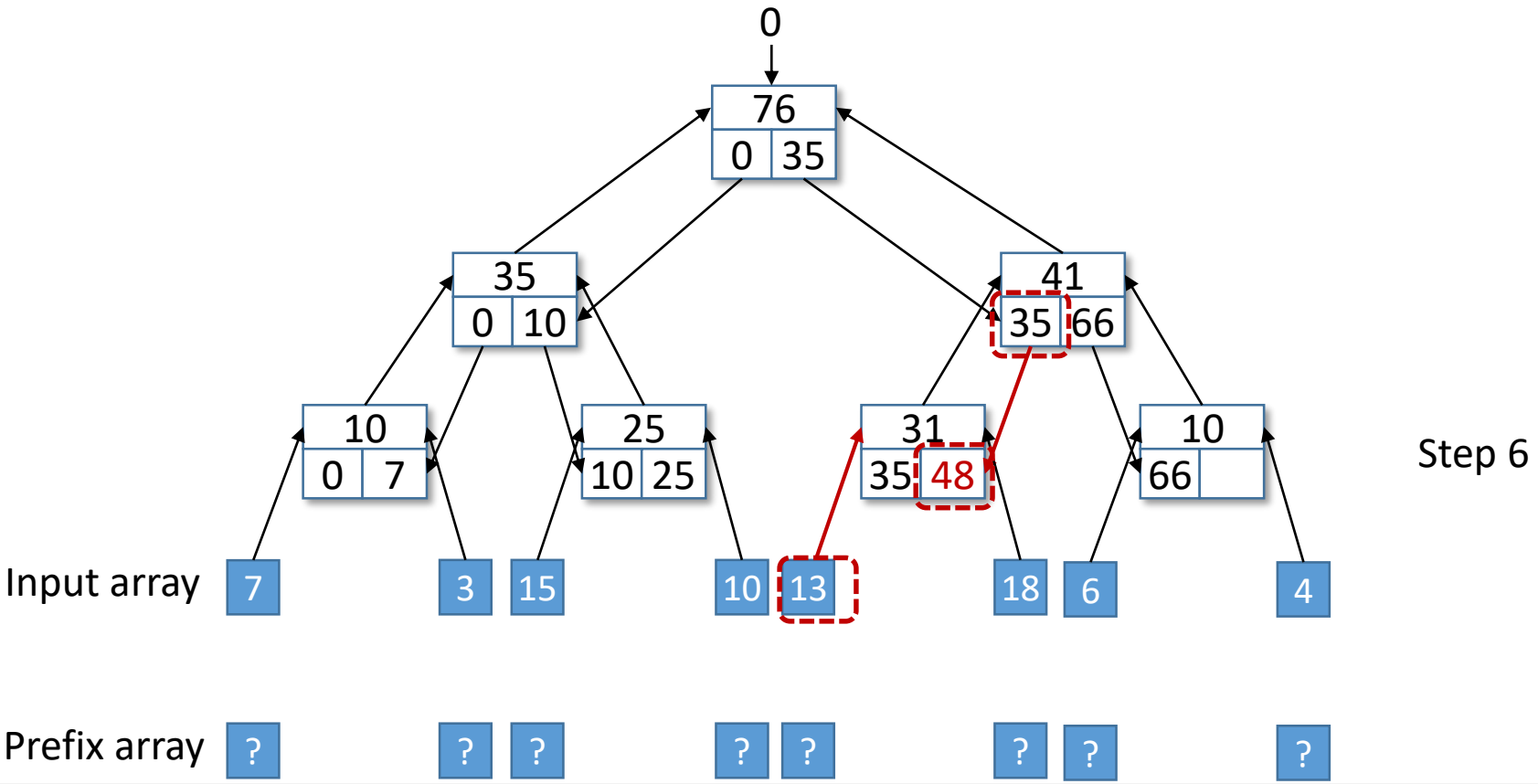
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



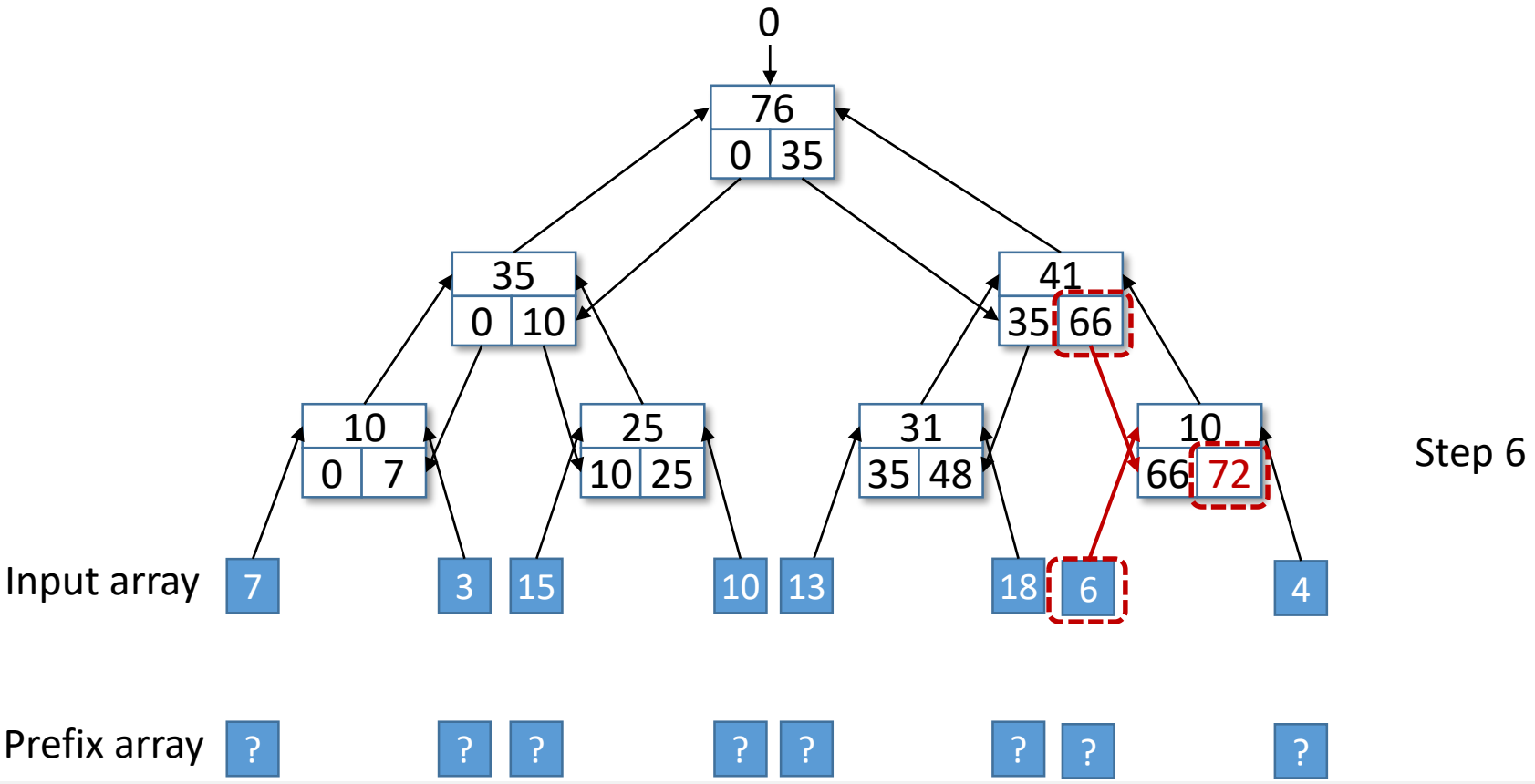
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



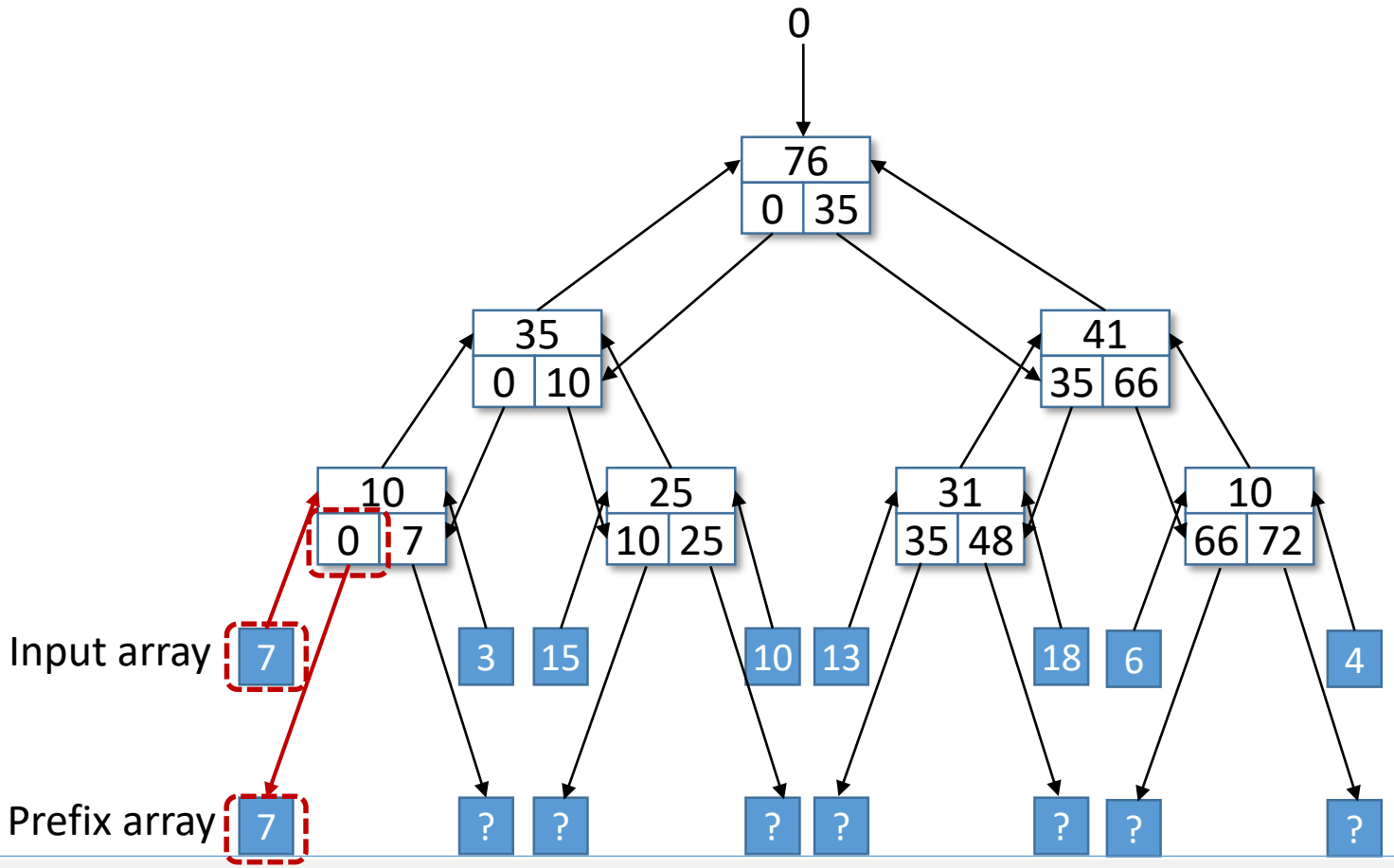
Parallel Prefix Sum

- Second Phase:
  - Receive a value from parent
  - Relay it to their left child
  - Send their right child the sum of the parent's value and their left child's value



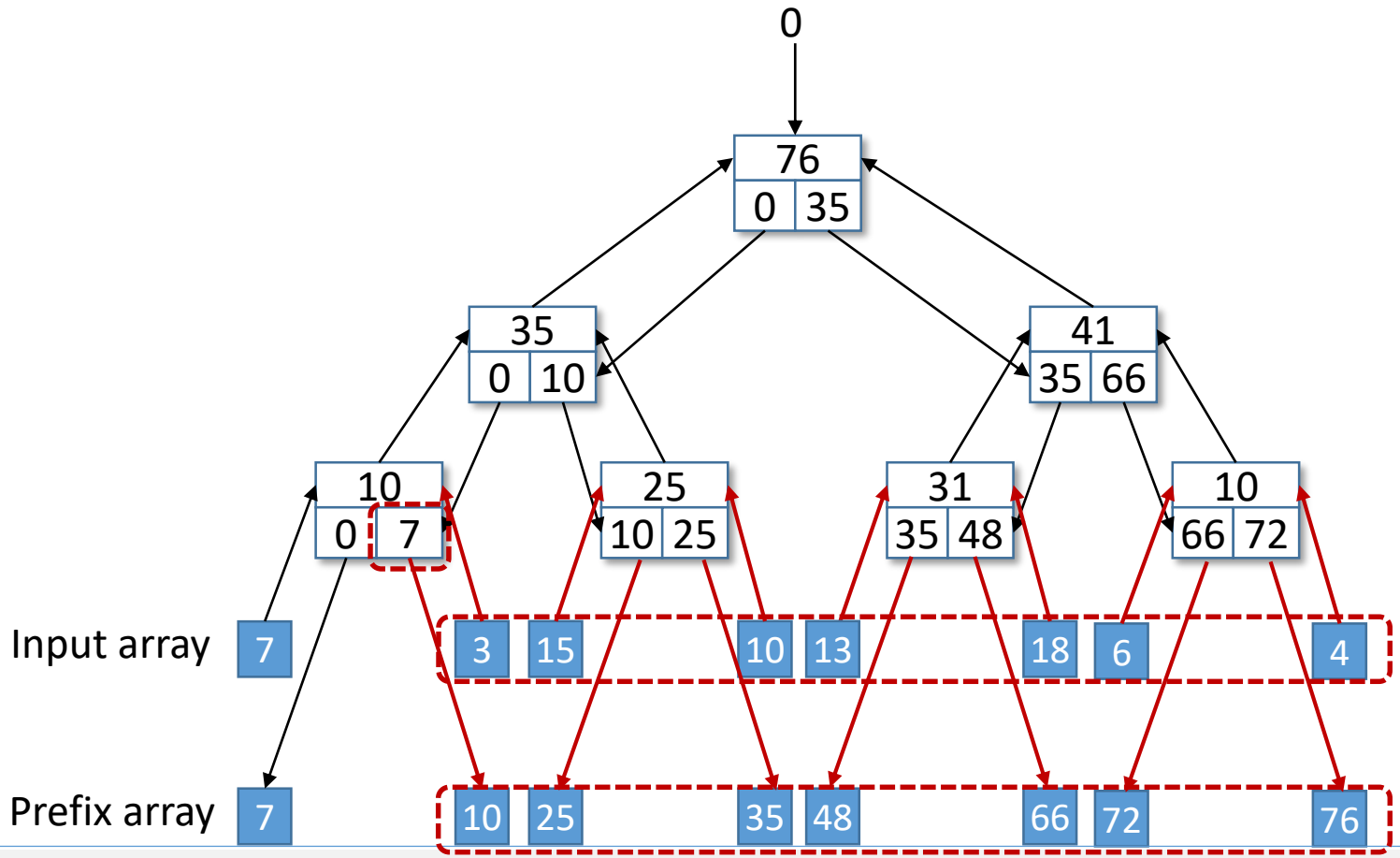
Parallel Prefix Sum

- Second phase
  - Leaf nodes add the prefix value above and the saved input



Parallel Prefix Sum

- Second phase
  - Leaf nodes add the prefix value above and the saved input



Step 7

Parallel Prefix Sum

- Requires an up sweep and a down sweep in the tree
  - But all operations at each level in a sweep can be performed concurrently
  - At most two add operations are required at each node
    - One going up and one going down, plus the routing logic
  - Thus, the algorithm has logarithmic time complexity

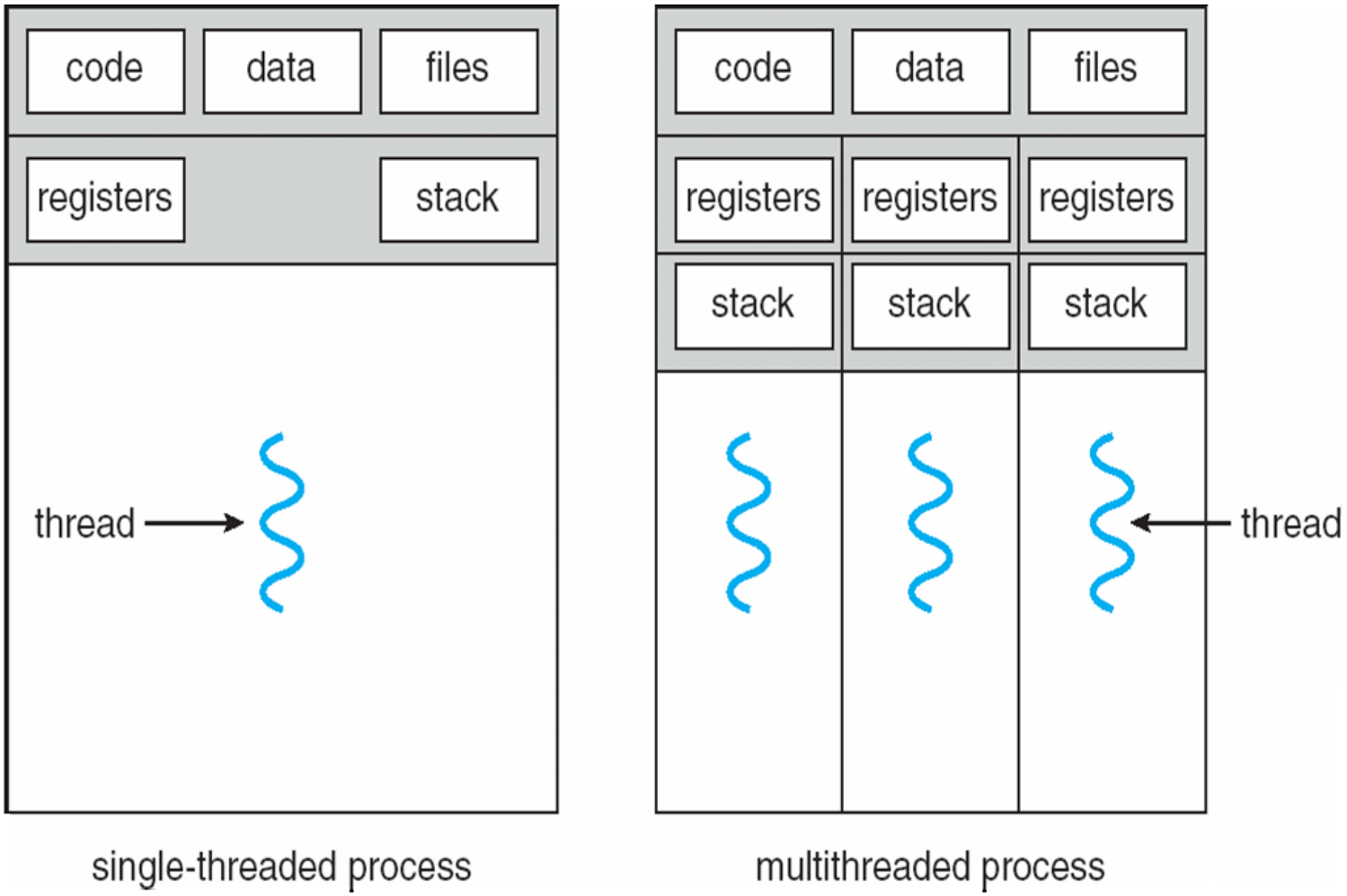
- An essential difference between the sequential and parallel algorithms
  - We organized the parallel algorithms to change the order of the computation without affecting correctness
    - Based on the associativity and commutativity of addition


Performance!
OS/Hardware knowledge is a must
for high-performance code!
Hardware knowledge

Parallelism using Multiple Threads

- The concept of a thread
  - A thread is a unit of parallelism (or execution)
  - A thread has everything needed to execute a stream of instructions
    - Program text, call stack, program counter, etc.
  - Threads share memory with other threads
    - Can cooperate to compute on global data

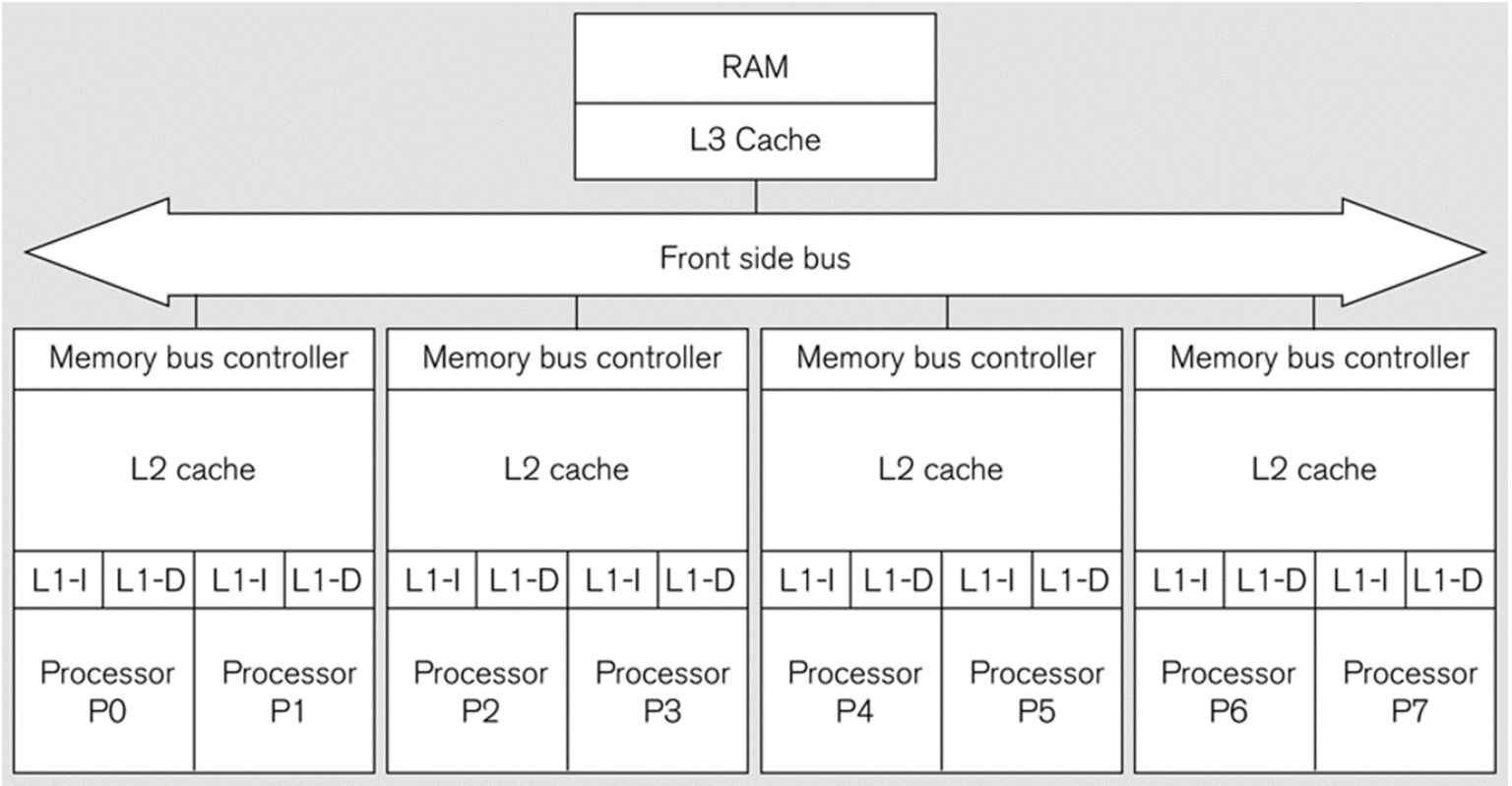
Threads



Source: A. Silberschatz et al., Operating System Concepts, 8th ed.

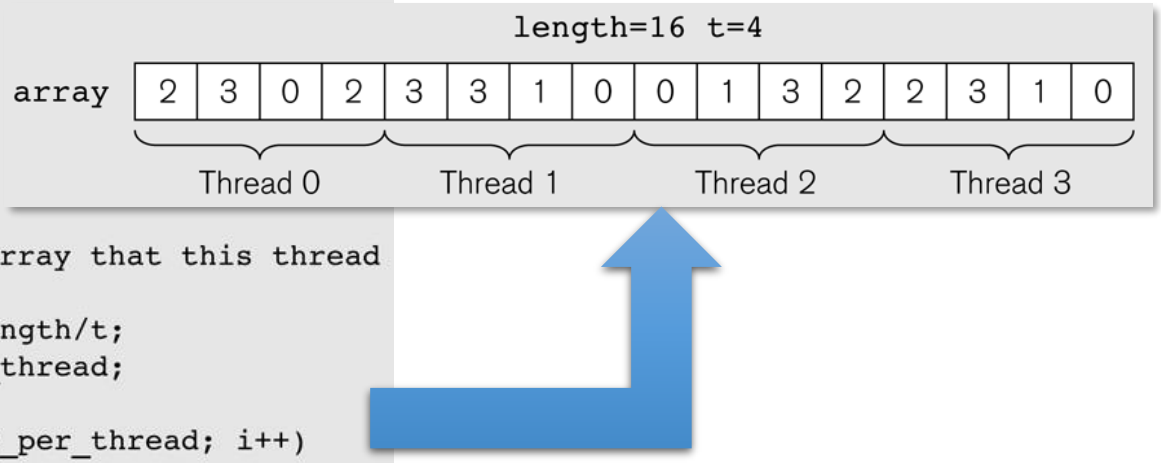
A Multithreaded Solution to Count 3s

- Consider the problem of counting the number of 3s in an array
- Our parallel program will run on a parallel computer shown below



First Solution: Try 1

```
 1  int t;                  /* number of threads */
 2  int *array;
 3  int length;
 4  int count;
 5
 6  void count3s()
 7  {
 8    int i;
 9    count = 0;
10    /* Create t threads */
11    for(i=0; i<t; i++)
12    {
13      thread_create(count3s_thread, i);
14    }
15
16    return count;
17  }
18
19  void count3s_thread(int id)
20  {
21    /* Compute portion of the array that this thread
         should work on */
22       int length_per_thread=length/t;
23       int start=id*length_per_thread;
24
25    for(i=start; i<start+length_per_thread; i++)
26    {
27      if(array[i]==3)
28      {
29        count++;
30      }
31    }
32  }
```

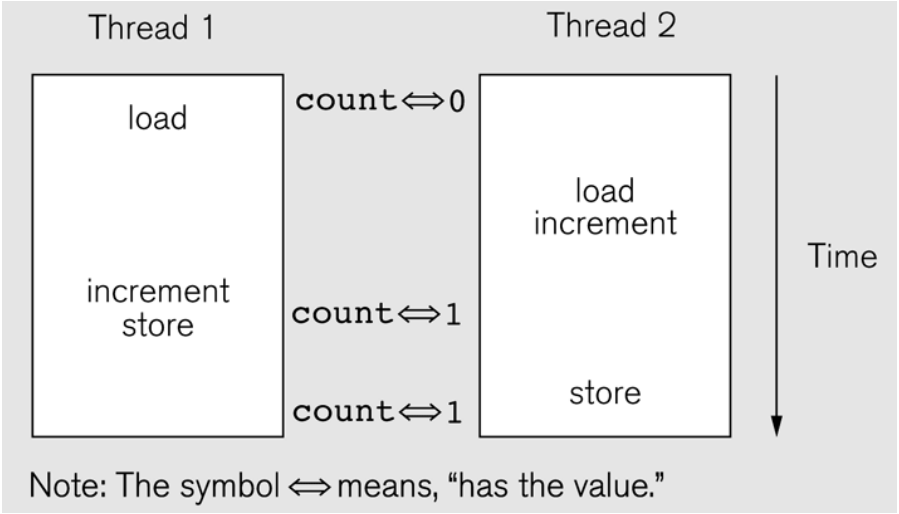


First Solution: Try 1

- The first solution will not produce the correct answer
  - Because of the race condition in line 29

- Incrementing the count variable is typically implemented on modern machines as follows
  - Load count into a register
  - Increment count
  - Store count back into memory

- When two threads execute the count3s_thread() code, these instructions might be interleaved as in the fig.
  - The final value of count could be 1 rather than 2



Second Solution: Try 2

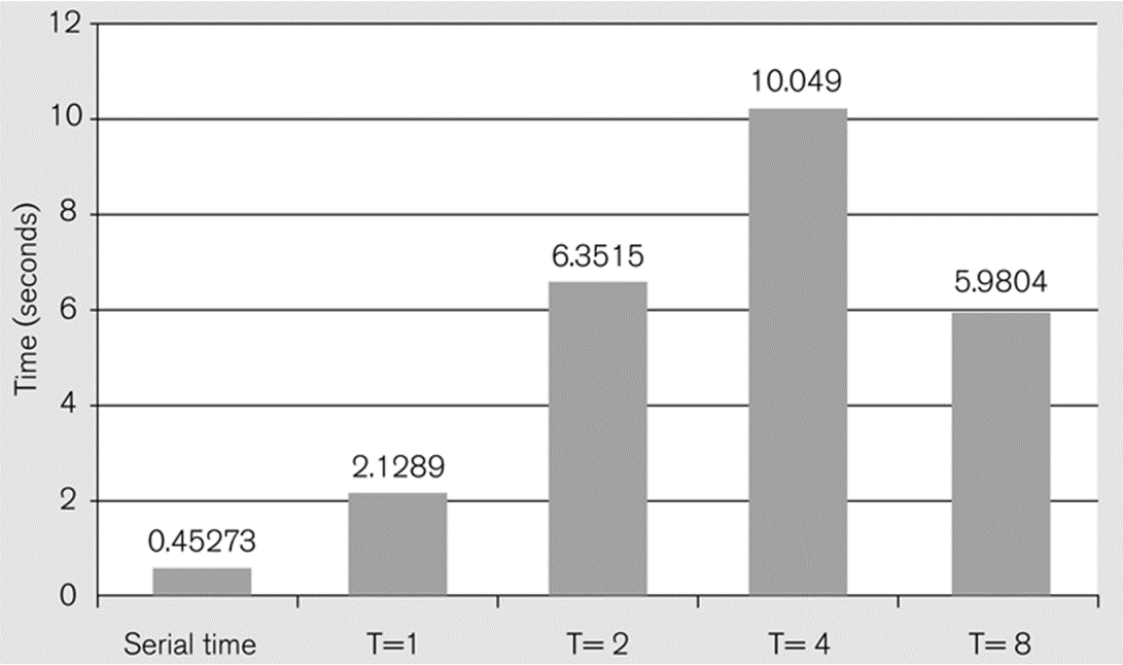
- Solve the previous correctness issue using a mutex
  - A mutex has two states – locked and unlocked
  - When a thread attempts to lock a mutex, it checks if it is locked or unlocked
  - If locked, it waits until the mutex is in an unlocked state
- Use of the mutex creates a critical section
  - At most one thread can execute the code

```
 1  mutex m;
 2
 3  void count3s_thread(int id)
 4  {
 5    /* Compute portion of the array that this thread
        should work on */
 6    int length_per_thread=length/t;
 7    int start=id*length_per_thread;
 8
 9    for(i=start; i<start+length_per_thread; i++)
10    {
11      if(array[i]==3)
12      {
13        mutex_lock(m);
14        count++;
15        mutex_unlock(m);
16      }
17    }
18  }
```

Second Solution: Try 2

- The second solution is a correct parallel program

- But, it is much slower than the original sequential code
  - Performance degradation due to the overhead of the mutex



Third Solution: Try 3

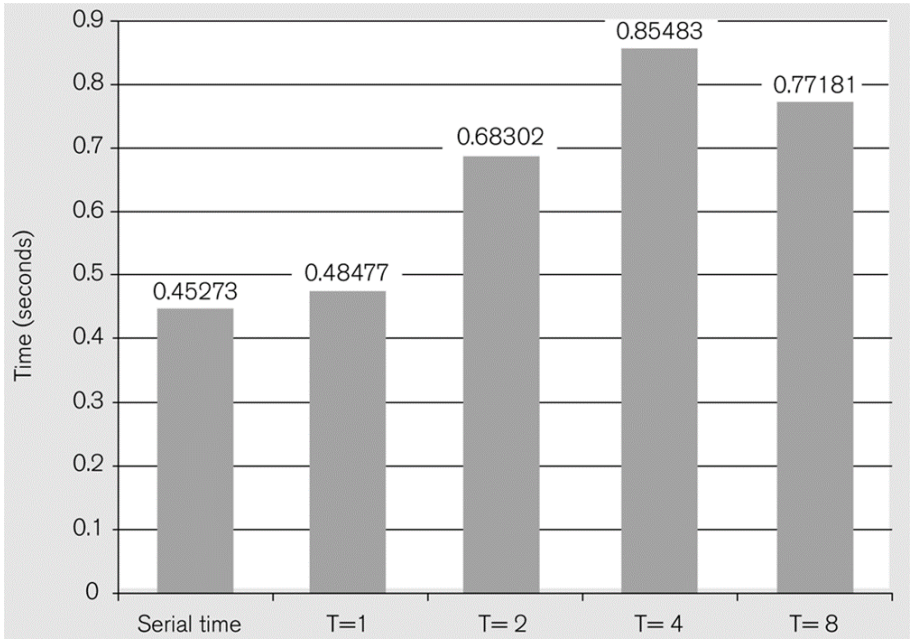
```
 1  private_count[MaxThreads];
 2  mutex m;
 3
 4  void count3s_thread(int id)
 5  {
 6    /* Compute portion of array for this thread to
        work on */
 7    int length_per_thread=length/t;
 8    int start=id*length_per_thread;
 9
10    for(i=start; i<start+length_per_thread; i++)
11    {
12      if(array[i] == 3)
13      {
14        private_count[id]++;
15      }
16    }
17    mutex_lock(m);
18    count+=private_count[id];
19    mutex_unlock(m);
20  }
```

- Address the lock contention problem using a private variable
  - private_count
- First, accumulate all the local contribution in private_count
- Only access the critical section for updating “count” once per thread
- Significantly reducing the frequency of acquiring/releasing locks

Third Solution: Try 3

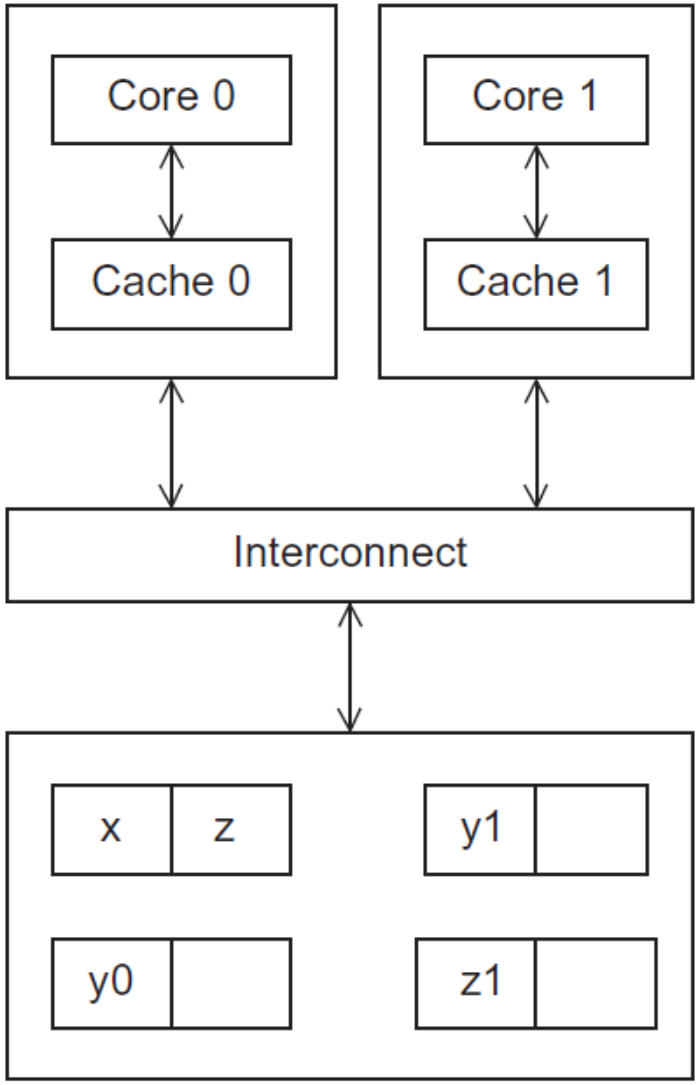
- The performance of the third solution is still not good
  - Still slower than the sequential version
  - Much better than the second try though

- This performance degradation is caused by the underlying HW
  - Known as “false sharing” (more on the next slide)



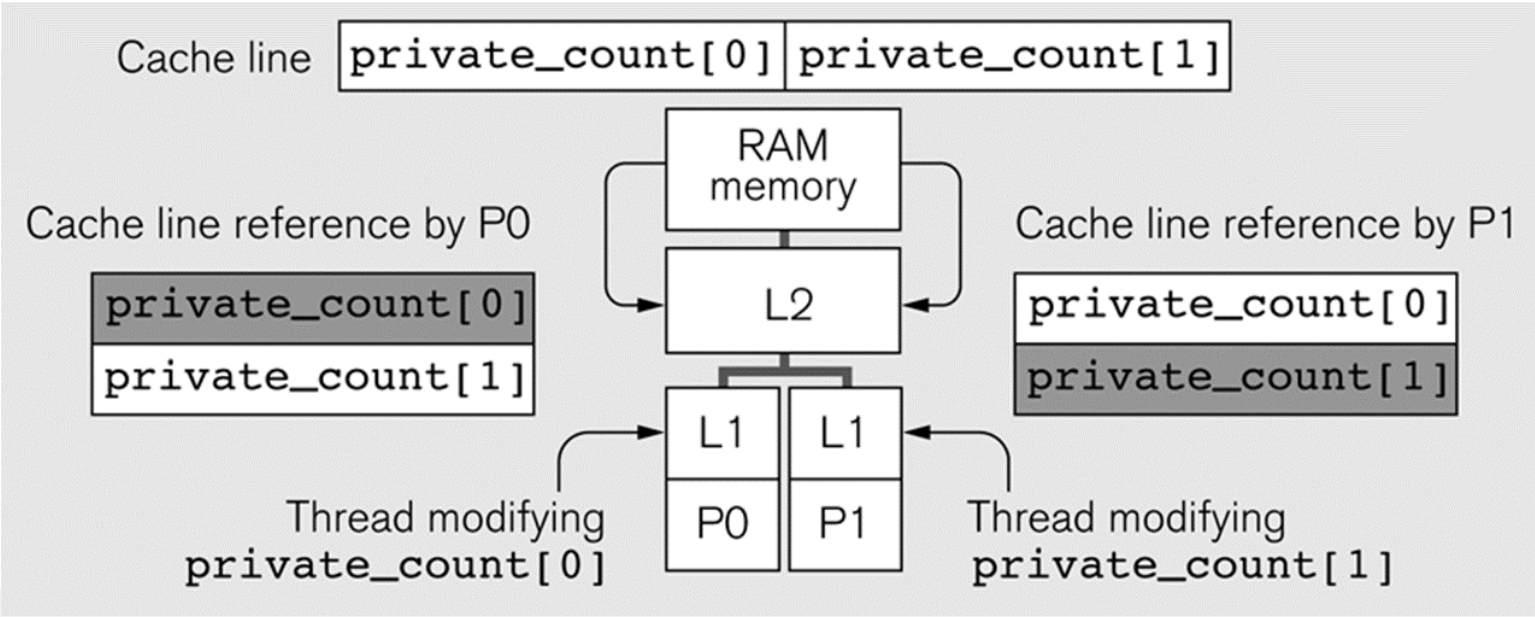
Review: Cache coherence

- At HW level, data is managed at the granularity of a cache line (e.g., 64B)
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
  - If Core 1 has the copy of x, it will be invalidated
- Programmers have no control over caches and when they get updated.



Third Solution: Try 3

- The false sharing problem
  - When a thread (on P0) references its private_count, the corresponding cache line moves into P0's L1 cache
  - When the other thread (on P1) modifies its private_count, the copy in P0's L1 cache is invalidated first
    - To enforce the coherence between caches
  - When P0 references its private_count again later, there will be a cache miss
  - This repeats again & again ➔ essentially almost no cache hits!



Snooping Cache Coherence

- The cores share a bus .
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of x stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that x has been updated and it can mark its copy of x as invalid.

Directory Based Cache Coherence

- Uses a data structure called a directory that stores the status of each cache line.

- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches are invalidated.

Fourth Solution: Try 4

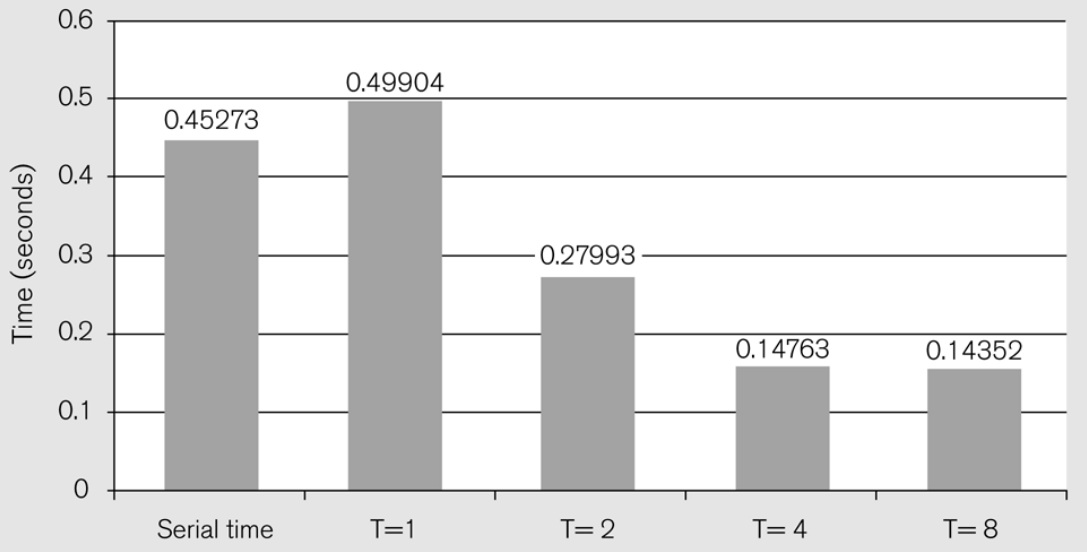
- To eliminate false sharing, the array of private_count is padded
  - Cache line size: 64 bytes
  - int value: 4 bytes
  - char padding[60]: 60 bytes
- With padding, each private_count resides in a separate cache line
  - No more false sharing!

```
 1  struct padded_int
 2  {
 3    int value;
 4    char padding[60];
 5  } private_count[MaxThreads];
 6
 7  void count3s_thread(int id)
 8  {
 9    /* Compute portion of the array this thread should
        work on */
10    int length_per_thread=length/t;
11    int start=id*length_per_thread;
12
13    for(i=start; i<start+length_per_thread; i++)
14    {
15      if(array[i] == 3)
16      {
17        private_count[id]++;
18      }
19    }
20    mutex_lock(m);
21    count+=private_count[id].value;
22    mutex_unlock(m);
23  }
```

Fourth Solution: Try 4

- The 4th solution significantly outperforms previous solutions
  - By removing the overhead of mutexes and false sharing
  - With a single thread, it is as fast as the serial execution
  - With two threads, it is almost twice as fast as the single thread run

- Performance is not scaling beyond 4 threads
  - This is due to the limitation of hardware (e.g., L2 memory bandwidth)



The Goals: Scalability and Performance Portability

- Lessons from the Count 3s program
  - Performance can be achieved through parallelism ☺
  - Achieving it can be complicated (definitely not a free lunch) ☹
    - Race condition, issues of granularity, false sharing, …

- Ideally, we want the following characteristics from parallel programs
  - They are correct
  - They achieve good performance
  - They are scalable to large numbers of processors
  - They are portable across a wide variety of parallel platforms

Scalability

- Can our Count 3s program scale to a large number of cores?
  - Maybe not…
  - The scan of the array has been parallelized
  - But, the combining of intermediate results is not
    - We could use the pair-wise sum algorithm to parallelize this!
- In general, scalability requires scalable programming practices