



# MVP

## Introduction to Parallel Programming

---

Alexandre David

1.2.05

[adavid@cs.aau.dk](mailto:adavid@cs.aau.dk)



# Notion of Parallelism

---

- Familiar concept
  - ex: building a house, car manufacturing
  - decomposition into tasks
  - task dependency
  - static decomposition
- Less familiar concept
  - dynamic decomposition
  - dynamic load balancing
  - synchronization



# Implicit Parallelism

---

- Instruction-level parallelism
  - independent instructions run in parallel
- Super-scalar CPUs
  - different execution units
- Pipelines
  - cut instructions in small steps executed by different states and keep all the stages busy.
- And other techniques: OO exec, prefetch...

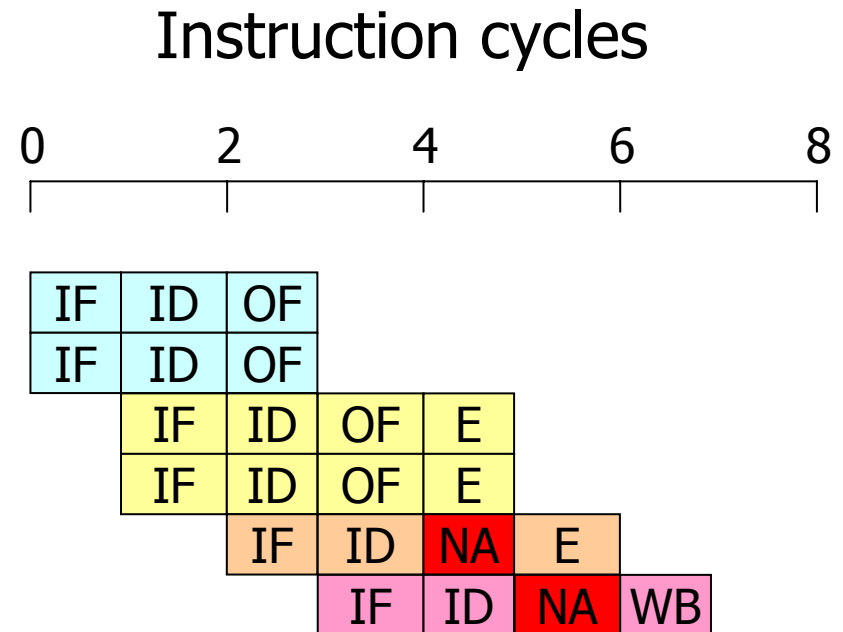
# Pipelining and Superscalar Execution - Example

Compiler

c=a+b+c+d  
as  
c=(a+b)+(c+d)

CPU

1. load R1,@1000
2. load R2,@1008
3. addR1,@1004
4. addR2,@100C
5. add R1,R2
6. store R1,@2000



2x IF, ID, OF, ... in the same cycle:  
**superscalar.**



# The Change in Thinking

---

- So far, programmer got the benefits of implicit parallelism for free.
  - No paradigm changes, profit from Moore's law.
  - Sequential algorithms, sequential reasoning, sequential programming, the hardware **keeps** it sequential.
  - We are TOO used to that.
- Multi-cores
  - No implicit parallelism anymore.
  - We need to change habits.



# Challenges

---

- No existing bullet-proof
  - language for parallelism,
  - methodology & technique for parallelism.
- Existing programs cannot exploit multi-cores.
- Programmers do not know how to write parallel programs in general.
- Algorithms are generally sequential and not fit for parallel programming directly.
- Write parallel & **scalable** programs to use more cores efficiently “for free”.



# GPUs

---

- Special case
  - parallel by design from the start
  - graphical processing pipelined
  - so far application specific
  - try to generalize now but still, SIMD type of computations, computation intensive applications.



# Parallel Hardware

---

- Home PC, stations.
- Supercomputers – history
  - but still powerful shared memory machines  
Vega 3 Series from Azul Systems up to  
864 cores, 768GB RAM, for Java applications.
- Clusters – popular, cheap, scalable.
- Grid computing.





# Parallel vs. Distributed

---

- “Parallel”: “parallel in the small”.
  - shared memory, multi-cores
- “Distributed”: “parallel in the big”.
  - clusters, different machines
- Can have both of course.



# System Level Parallelism

---

- See PSS.
- Not a solution, limited to the number of tasks you have.
- Important to know:
  - bound threads – scheduled by the OS
  - unbound threads – scheduled by a library.



# Paradigm Shift

---

- What compilers do:
  - change ordering,
  - remove redundancy,
  - reallocate resources,
  - but keep the semantics of the sequential program.
  - They preserve the original algorithm.
- We need to design parallel algorithms, compilers are intrinsically limited.
  - Automatic algorithmic transformations are limited.

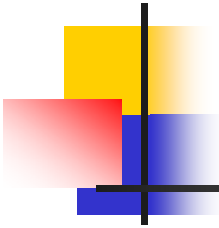


# Example

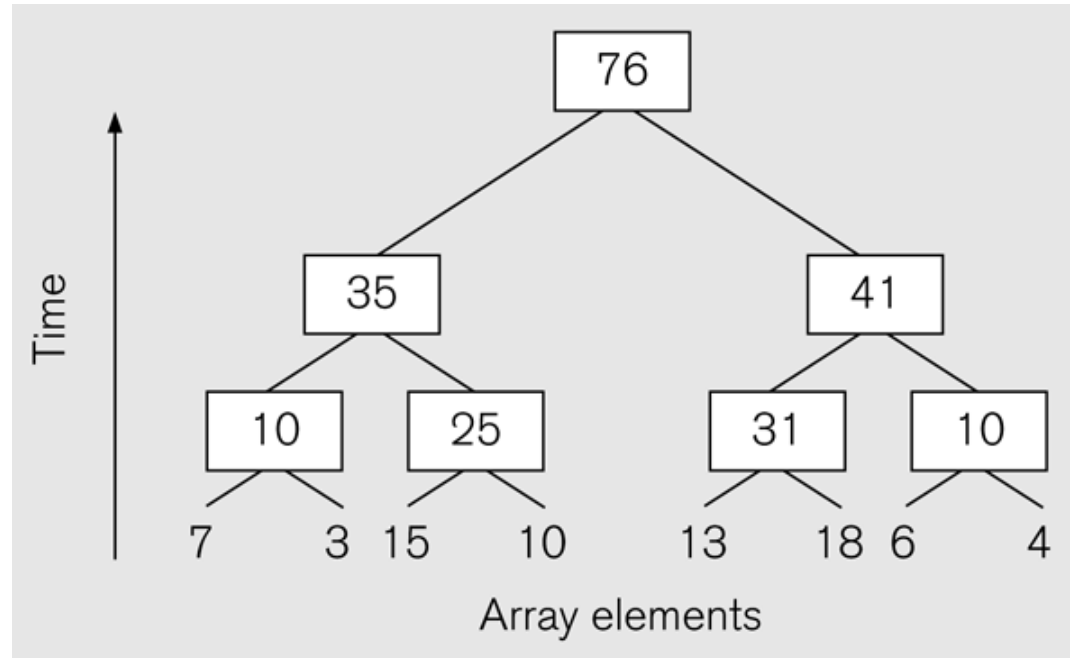
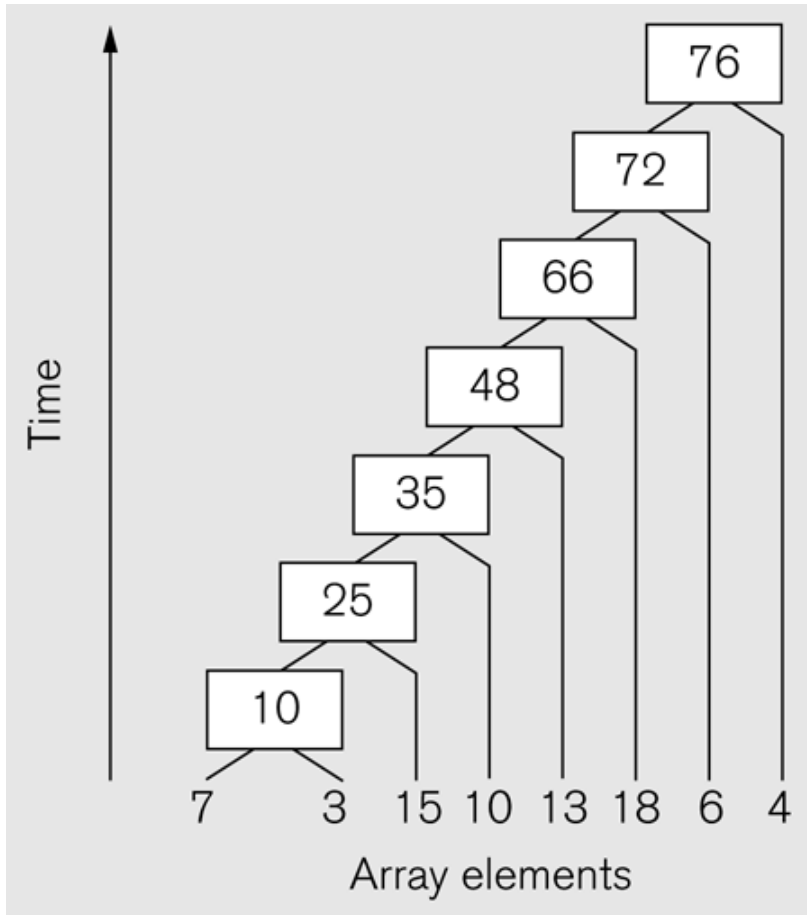
---

- Sequential description.
- Need to reformulate to get it parallel.
- Still simple because additions are associative.
- However, operations on double are approximate. The ordering matters for the precision.

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



# Compare:



What if it's not +?



# Prefix Sum

- Useful primitive, aka scan.
- Input: sequence of  $x_i$ .
- Output: sequence of  $y_i$ .  
 $y_i = \sum_{j \leq i} x_j$
- How to parallelize?



For  $n > 0$ :

```
y[0] = x[0]
```

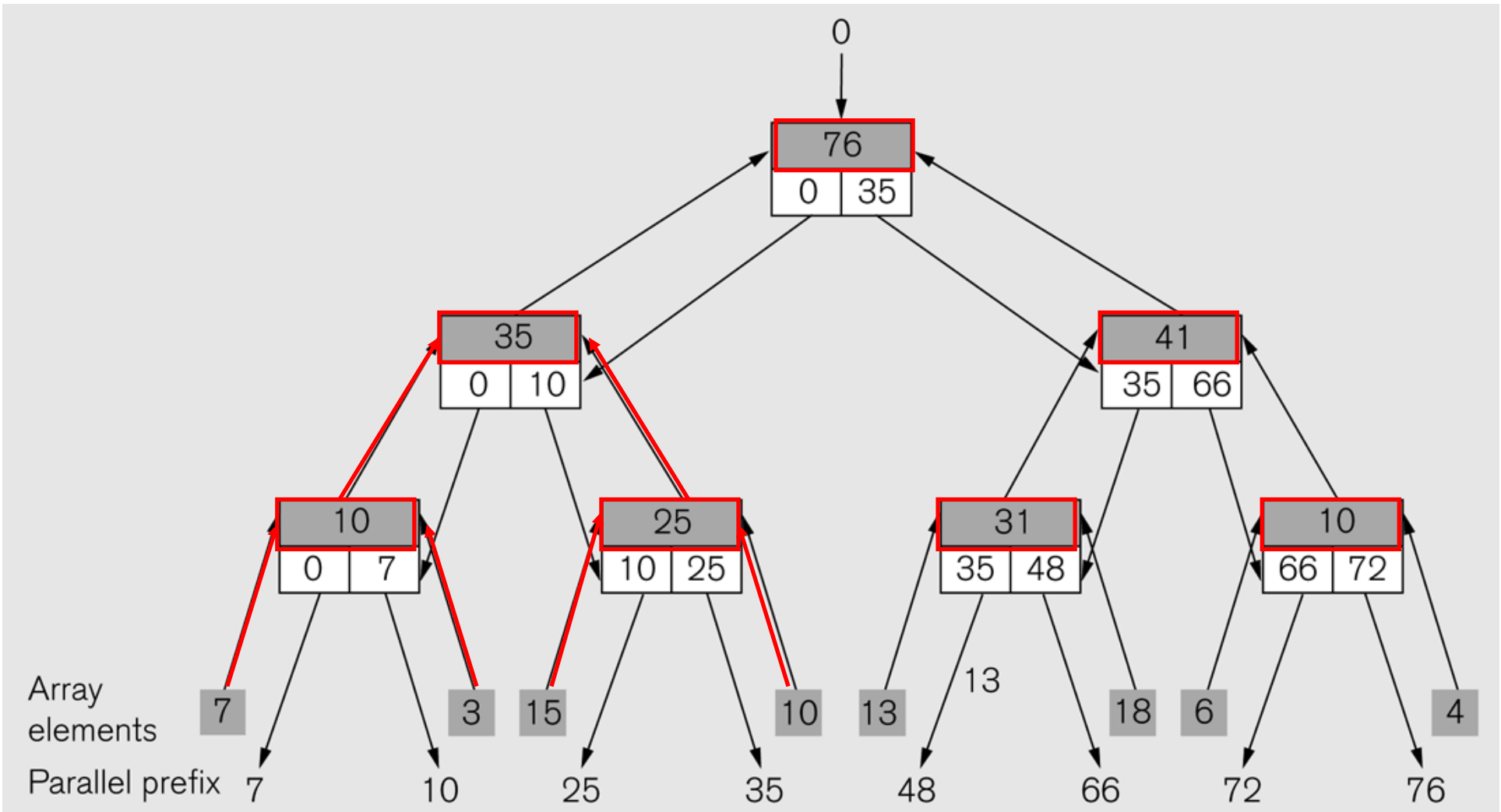
```
for(i=1; i<n; ++i)
```

```
{
```

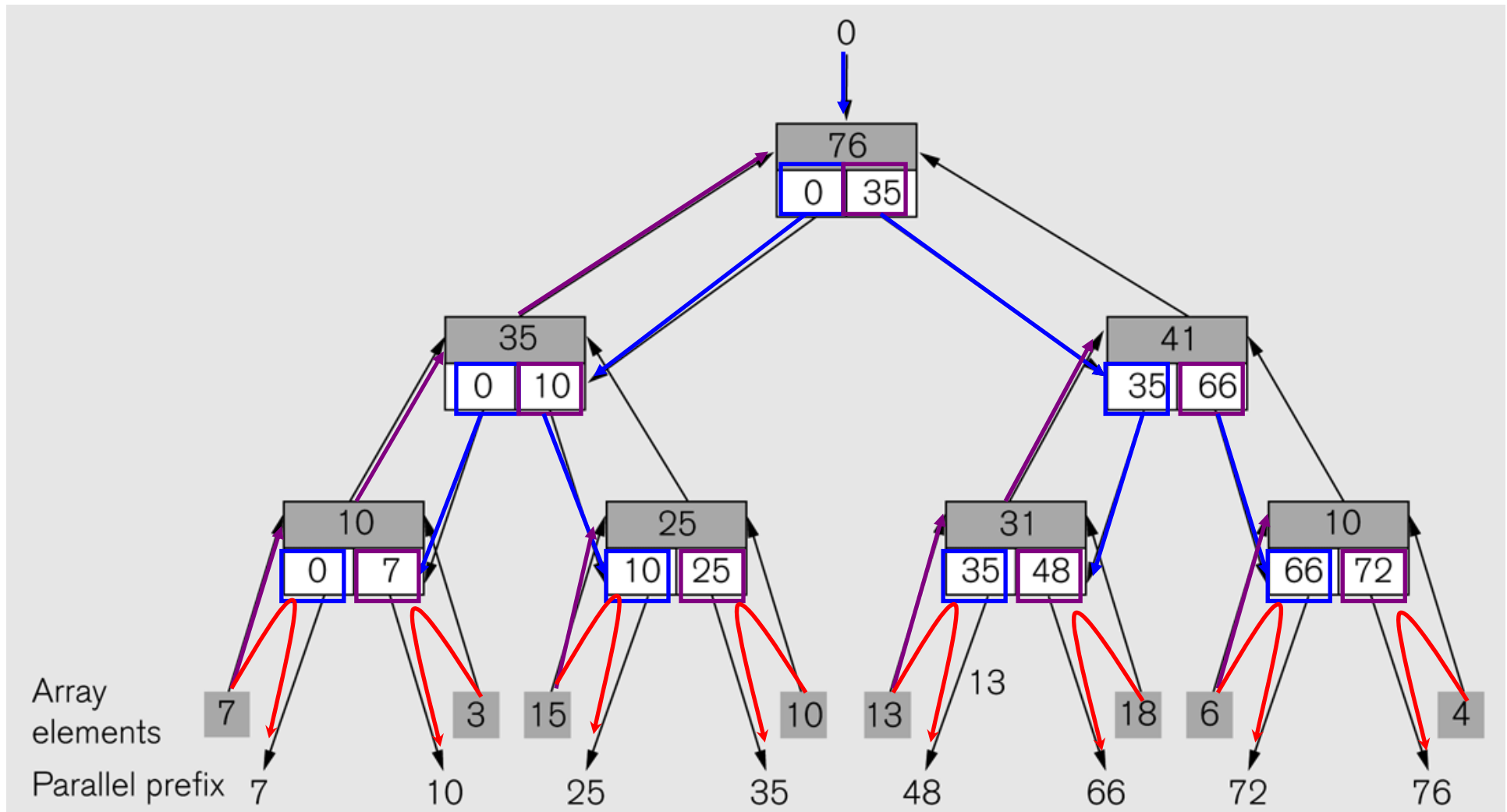
```
    y[i] = y[i-1] + x[i]
```

```
}
```

# Prefix Sum - sum



# Prefix Sum - prefix



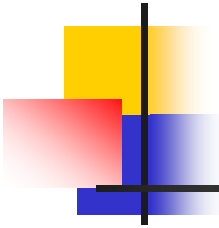


# Prefix Computation 2

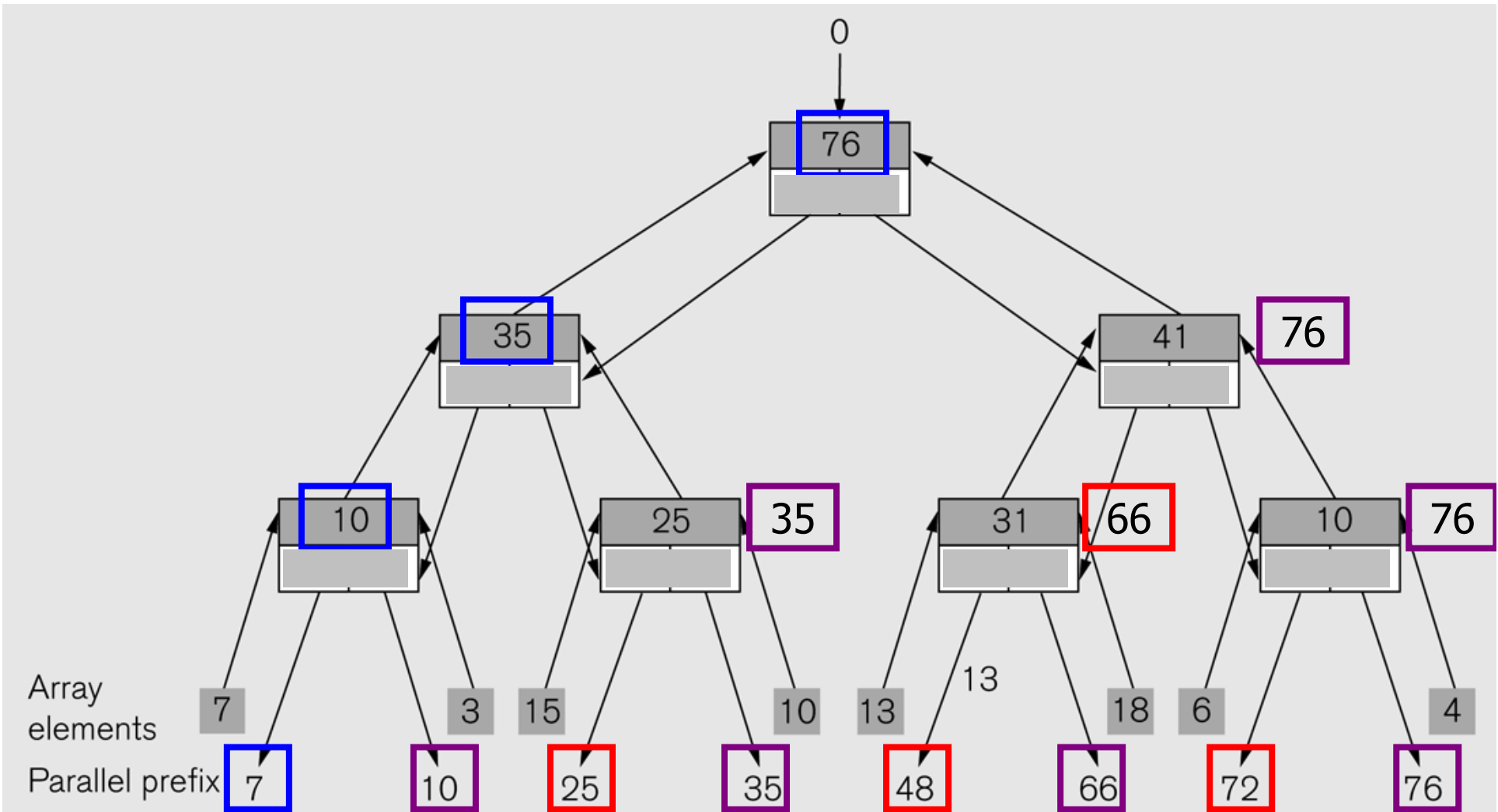
```
function prefix+(A,n)
  B[1] := A[1]
  if n > 1 then
    for i = 1 to n/2 pardo
      C[i] := A[2i-1] + A[2i]
    od
    D := prefix+(C, n/2)
    for i = 1 to n/2 pardo
      B[2i] := D[i]
    od
    for i = 2 to n/2 pardo
      B[2i-1] := D[i-1] + A[2i-1]
    od
  fi
  prefix+ := B
end
```

# Parallel Prefix Computation 2

```
function prefix+(A,n)[p1,...,pn]  
  p1: B[1] := A[1]  
  if n > 1 then  
    for i = 1 to n/2 pardo  
      pi: C[i] := A[2i-1] + A[2i]  
    od  
    D := prefix+(C,n/2)[p1,...,pn/2]  
    for i = 1 to n/2 pardo  
      pi: B[2i] := D[i]  
    od  
    for i = 2 to n/2 pardo  
      pi: B[2i-1] := D[i-1] + A[2i-1]  
    od  
  fi  
  prefix+ := B  
end
```



# Prefix Computation 2





# Pause

---

- What have we done here?
- Is it scalable?
  - What does it mean?
- Is it efficient?
- Is it optimal?
  - What does it mean?
- What about correctness?
- It is cache friendly?
- ...

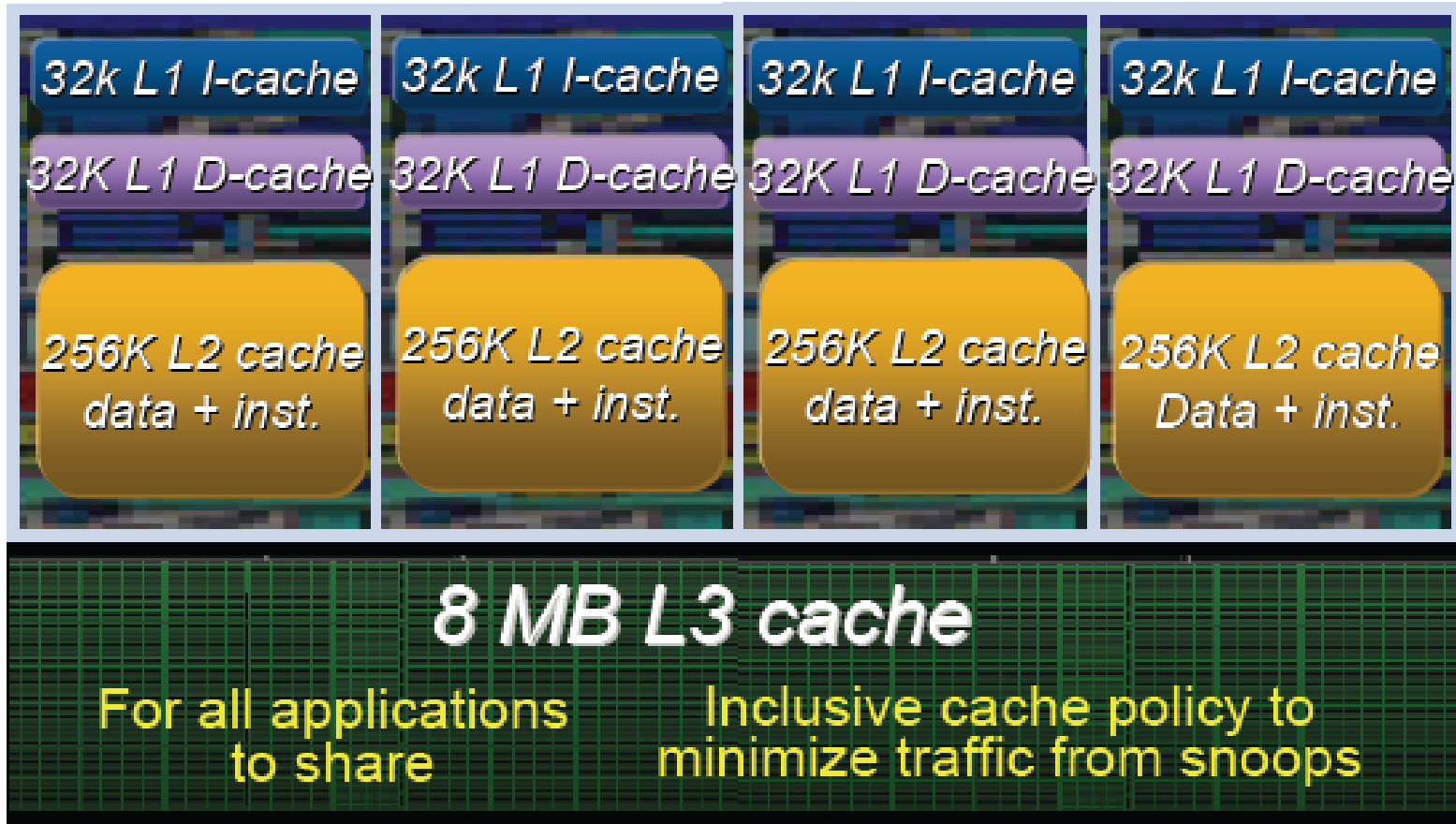


# Concept of a Thread (PSS)

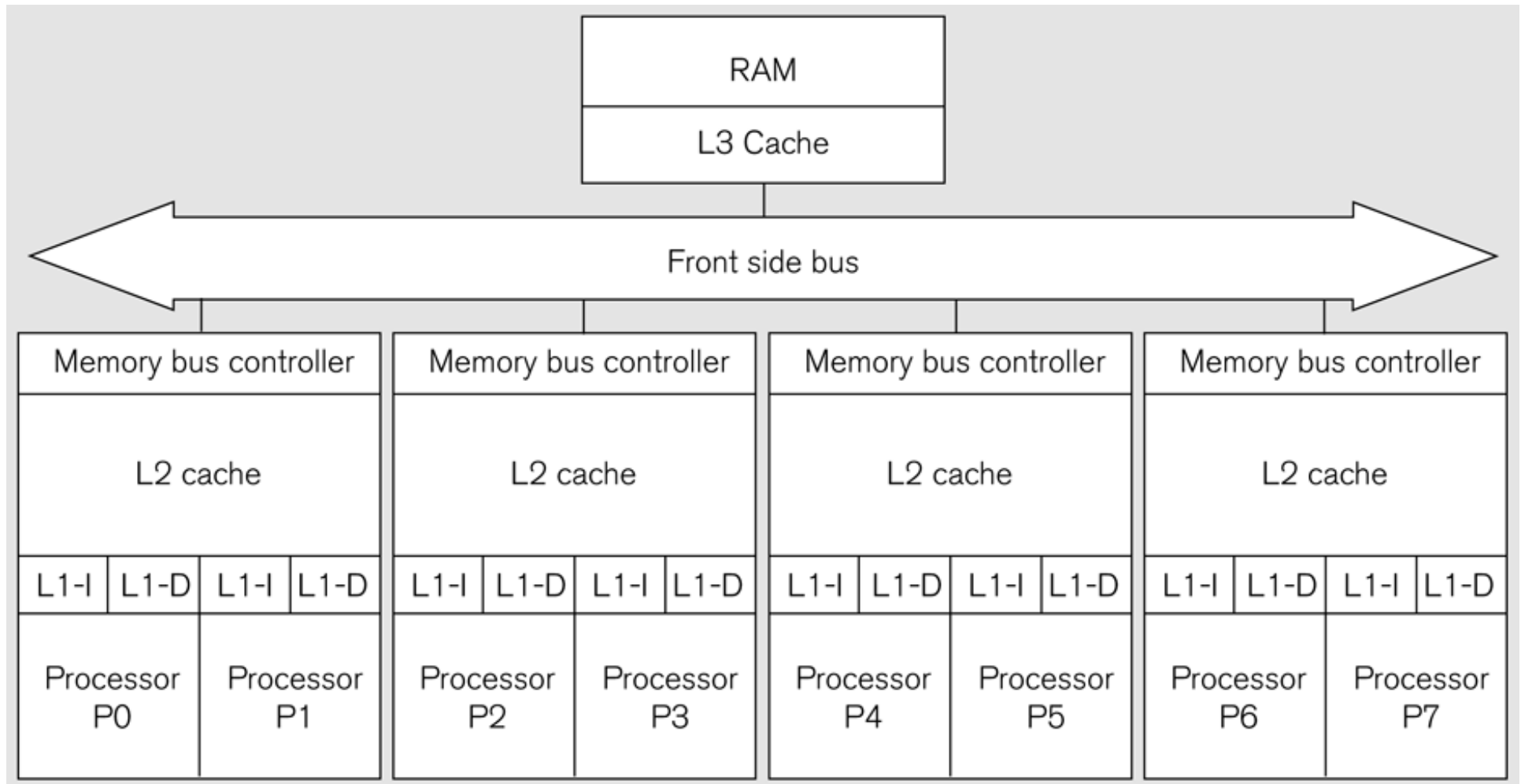
---

- Thread *of execution*
- Private
  - program
  - stack
  - program counter
- Shared
  - memory
  - I/O

# Example of Execution Platform



# Execution Platform of the Book





# Limitations of Memory System Performance

---

- The **memory system** is most often the **bottleneck**.
- Performance captured by
  - latency and
  - bandwidth.
- Remark: In practice latency is complicated to define: CL2, CL3, 2-2-2-5,...






# Effect on Performance: Example

- Processor @1GHz (1ns cycle) capable of executing 4 IPC + DRAM with 100ns latency.
- 4 IPC @1GHz -> 4GFLOPS *peak rating*.
- Processor must wait 100 cycles for every request.
  - Vector operations (dot product) @10MFLOPs.



# Improving with Cache

---

- Note: Often “\$\$” on pictures (cash).
- Hierarchical memory architecture with several levels of cache (2 common).
- Instruction and data separate for L1.
- Low latency, high bandwidth, but small.
-  Why does it improve performance???



# Why is \$\$ good?

---

- Temporal locality
  - Repeated access to the **same** data in a small window of time.
- Spatial locality
  - **Consecutive** data accessed by successive instructions.
- Vital assumptions, almost always hold.
- Very important for parallel computing.



# Case Study: Count 3s

---

```
int count3s(int *array, int length)
{
    int count = 0;
    for(i = 0; i < length; ++i)
    {
        if (array[i] == 3) count++;
    }
    return count;
}
```

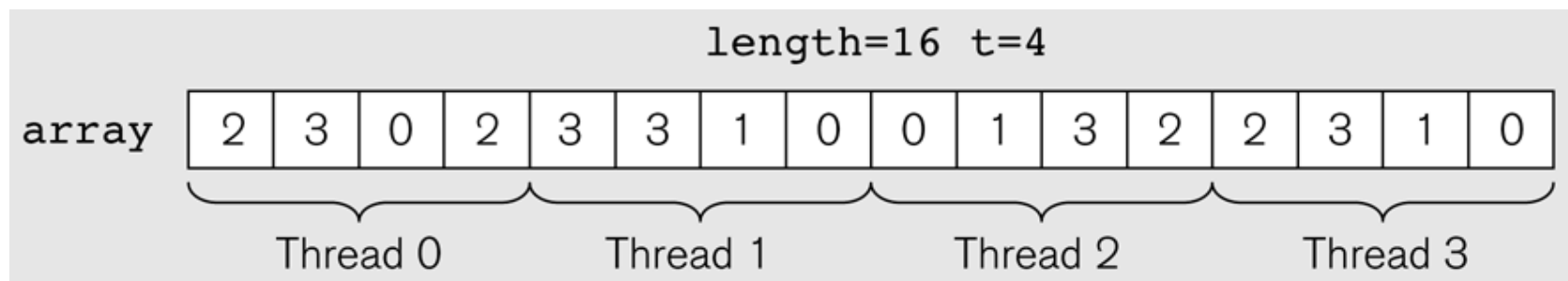
C pointer

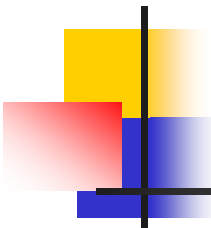
dereference

Serial code → parallel code?

# Try 1

- Partition the input
  - static data partitioning
- Shared variable counter



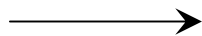


# Try 1

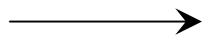
Thread creation



Partitioning



Count



```
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     return count;
16 }
17
18
19 void count3s_thread(int id)
20 {
21     /* Compute portion of the array that this thread
22        should work on */
23     int length_per_thread=length/t;
24     int start=id*length_per_thread;
25
26     for(i=start; i<start+length_per_thread; i++)
27     {
28         if(array[i]==3)
29         {
30             count++;
31         }
32     }
```

(wait for the threads missing)

Not atomic → race



# What can happen?

---

read count

count: 0

inc local value

read count

write count

count: 1

inc local value

count: 1

write count

Expected: 2

Race: The result depends on the interleaving of the threads. It is unpredictable.



# Try 2: Make It Atomic

---

- Use a mutex
  - locked
  - unlocked
- Mutexes are used to define critical sections

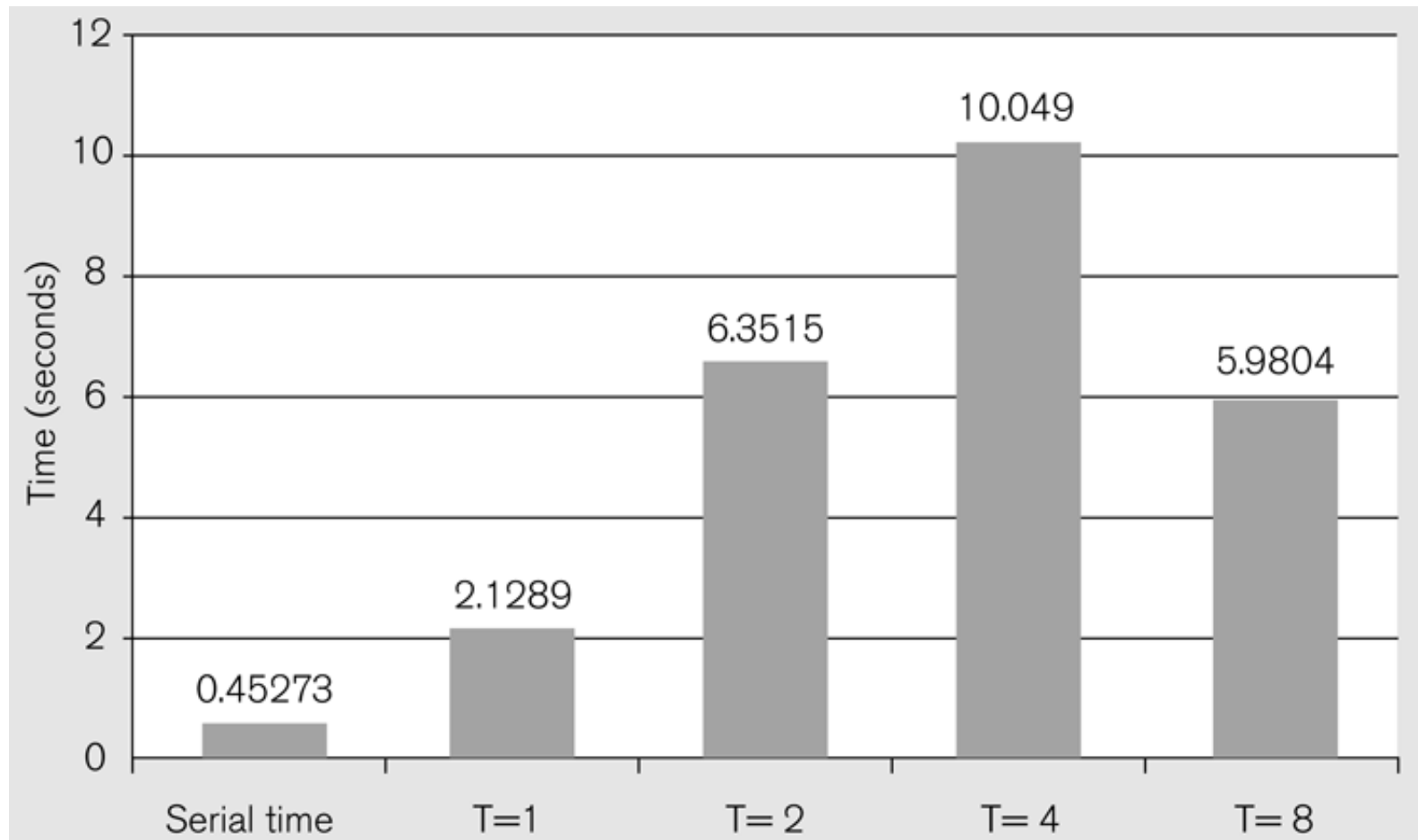




# Try 2

```
1  mutex m;
2
3  void count3s_thread(int id)
4  {
5      /* Compute portion of the array that this thread
6         should 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             mutex_lock(m);
15             count++;
16             mutex_unlock(m);
17         }
18     }
```

# Correct But Abysmal Performance

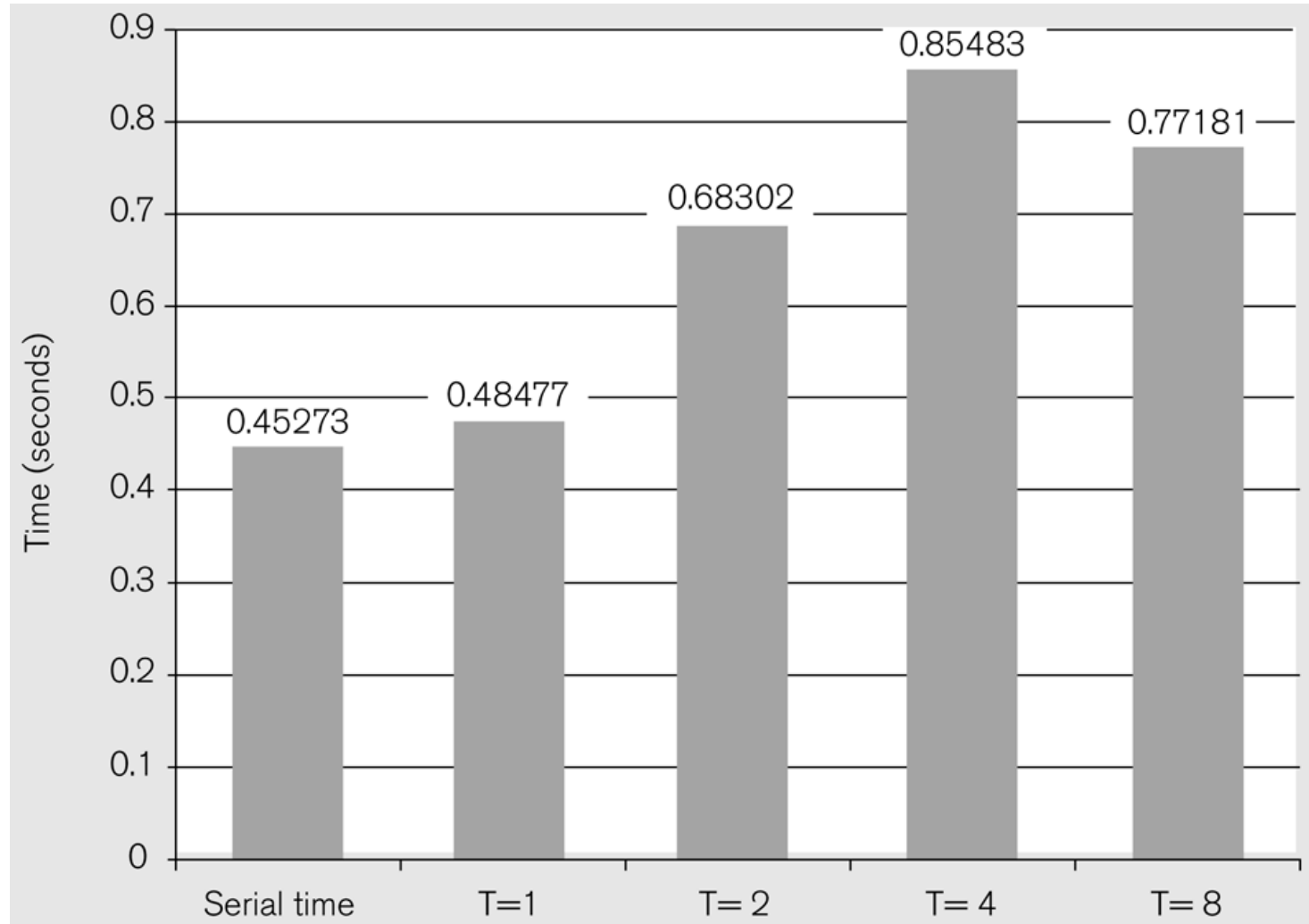




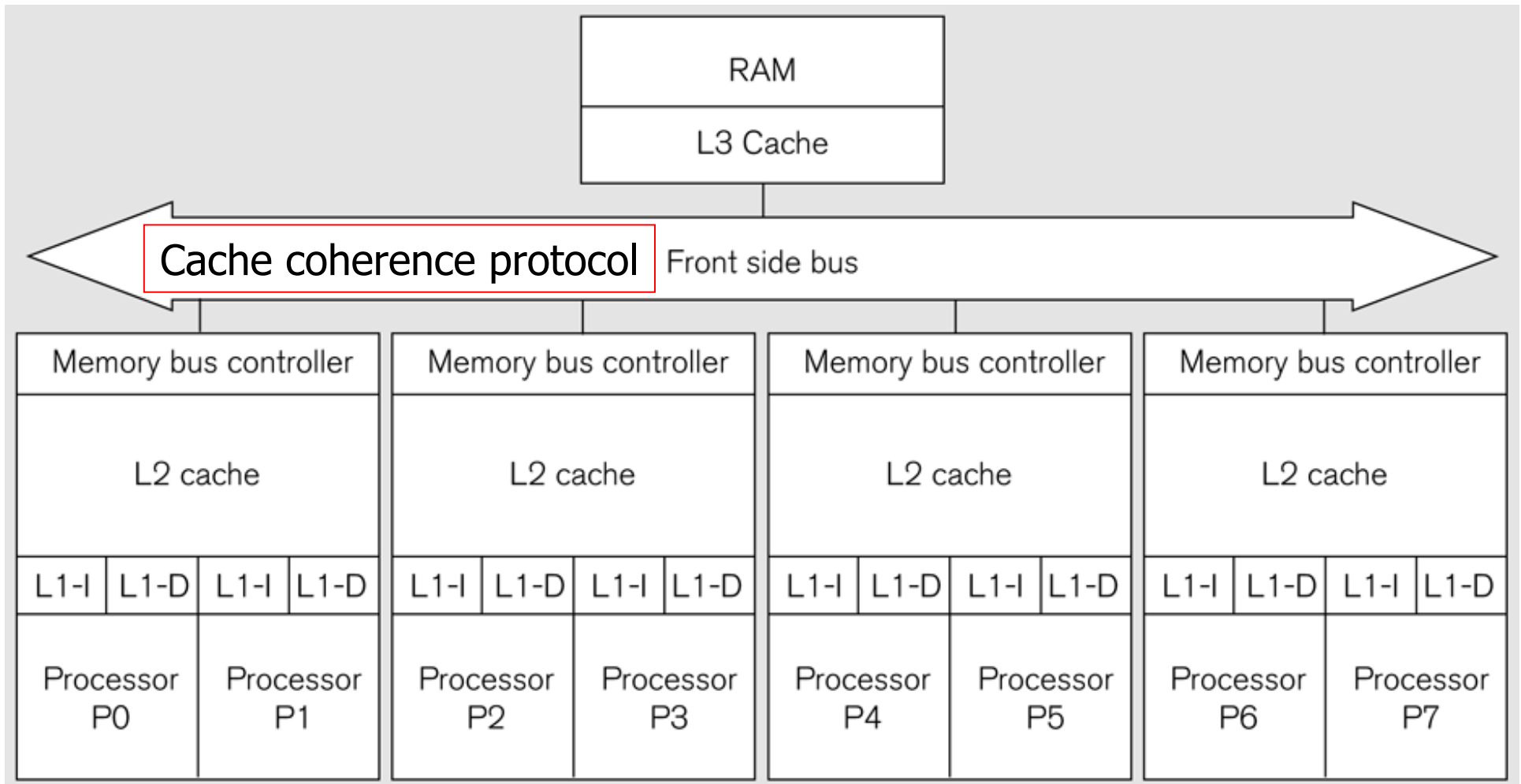
# Try 3

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

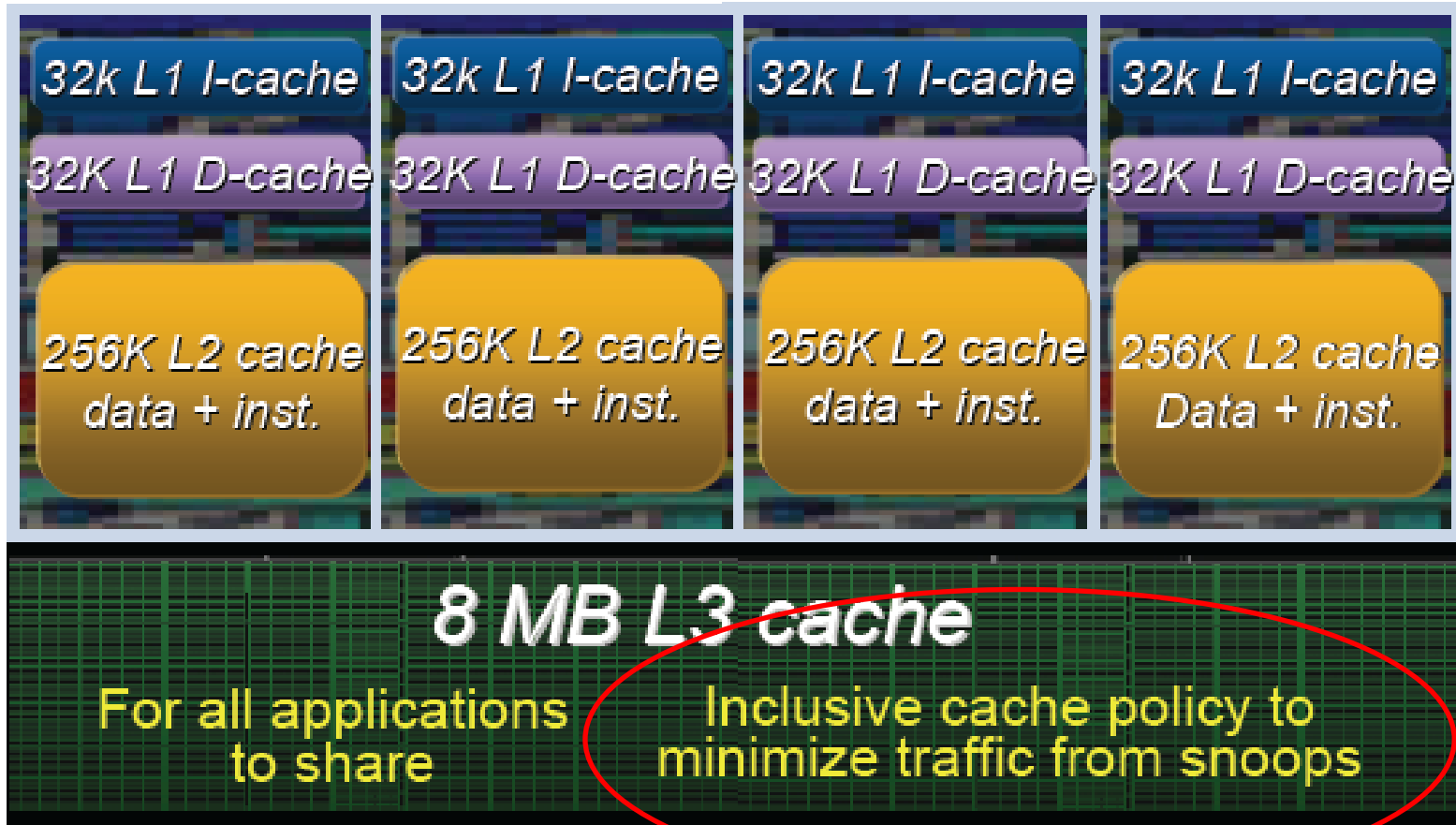
# Better But Still Not Good



# Recall the Architecture



# Core i7





# False Sharing

---

- Caches have some granularity = cache line.
- Usually on several words, 2-4 words.
- Consecutive counters are not **logically** shared but they are **physically** shared on the same cache line.
- The cache coherence protocol kicks in and kills performance because the **line is constantly moving**.

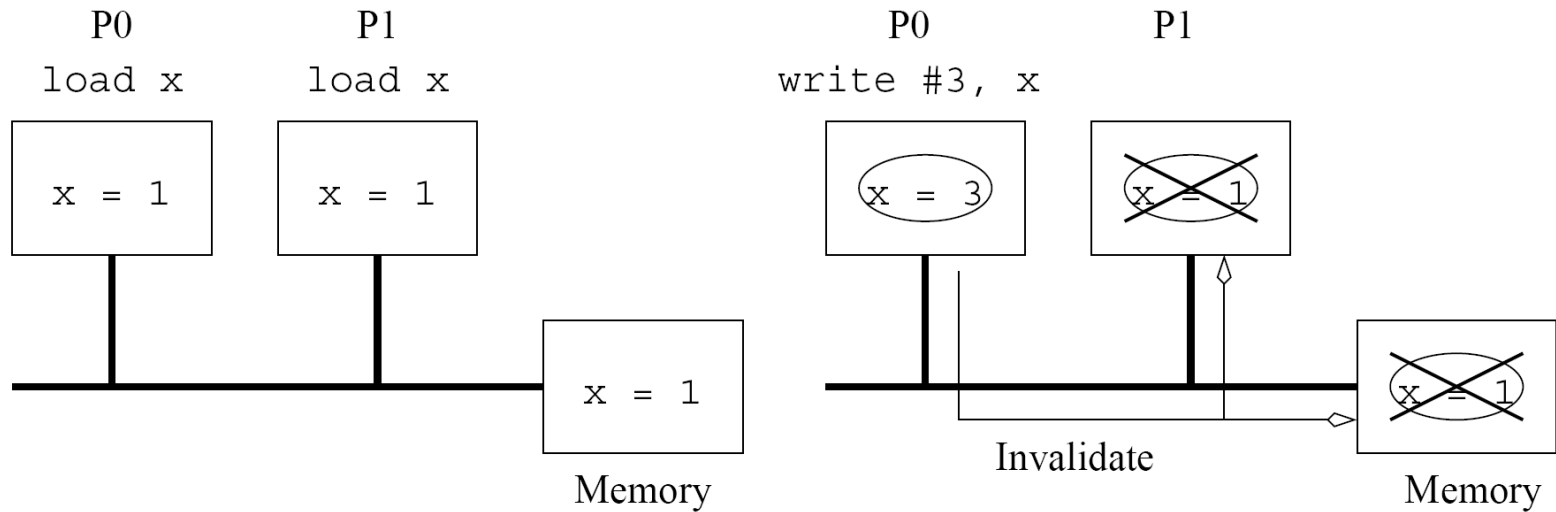


# Cache Coherence Protocols

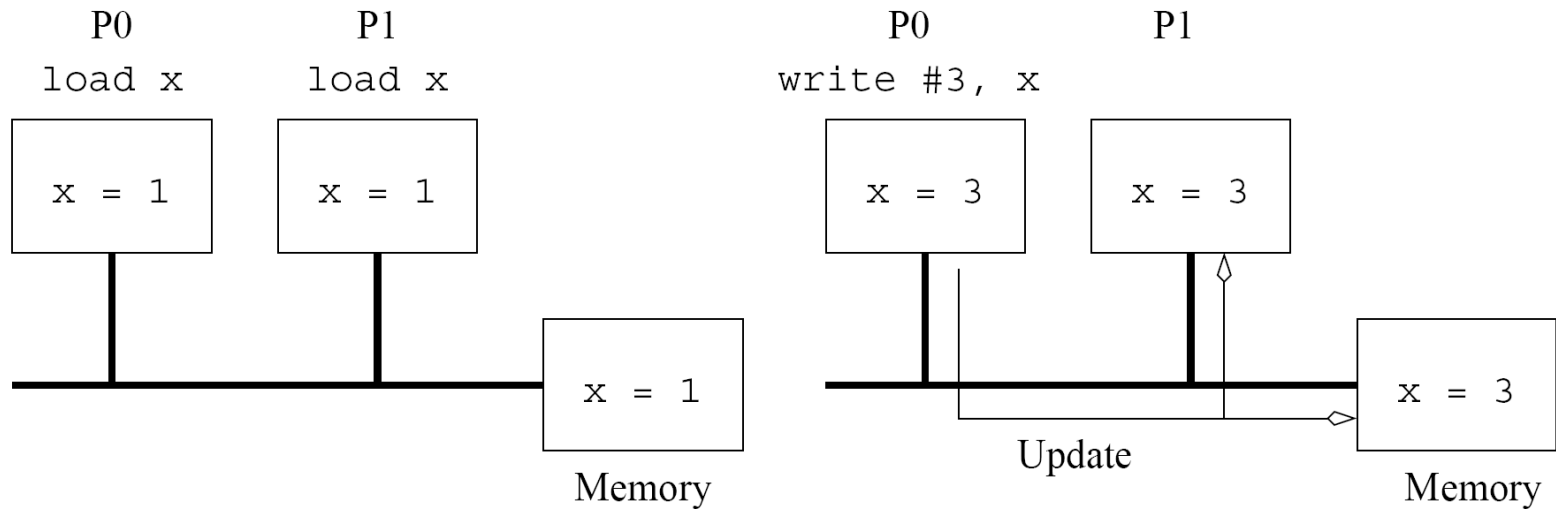
---

- We need additional hardware to keep *multiple copies* of the same memory bank *consistent* with each other.
- We have seen that \$\$ is good but it does not come for free.
- Mechanism known as cache coherence protocol, usually described as state machines.





(a)



(b)

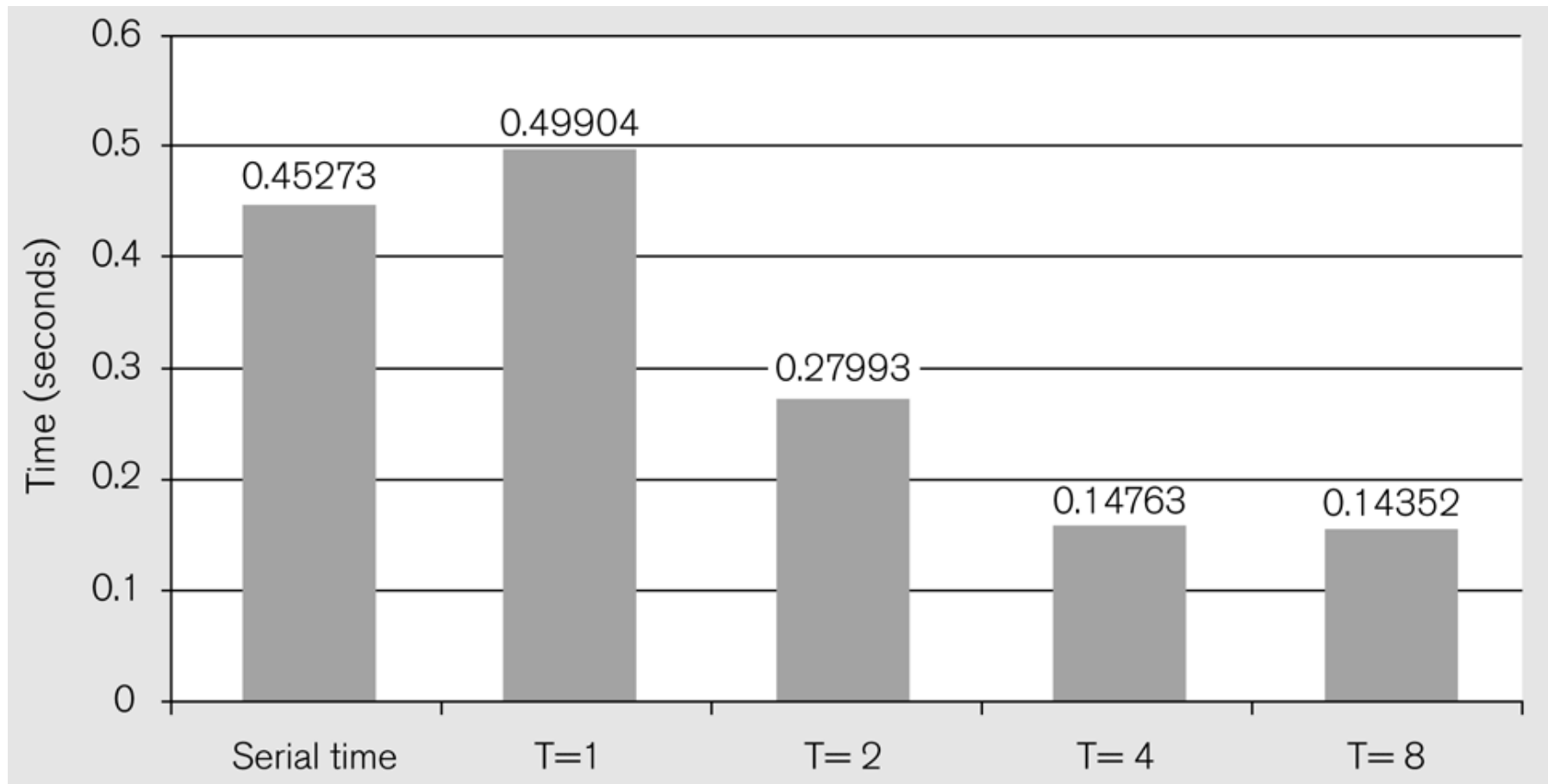
**Figure 2.21** Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.



# Try 4: Pad the Counters

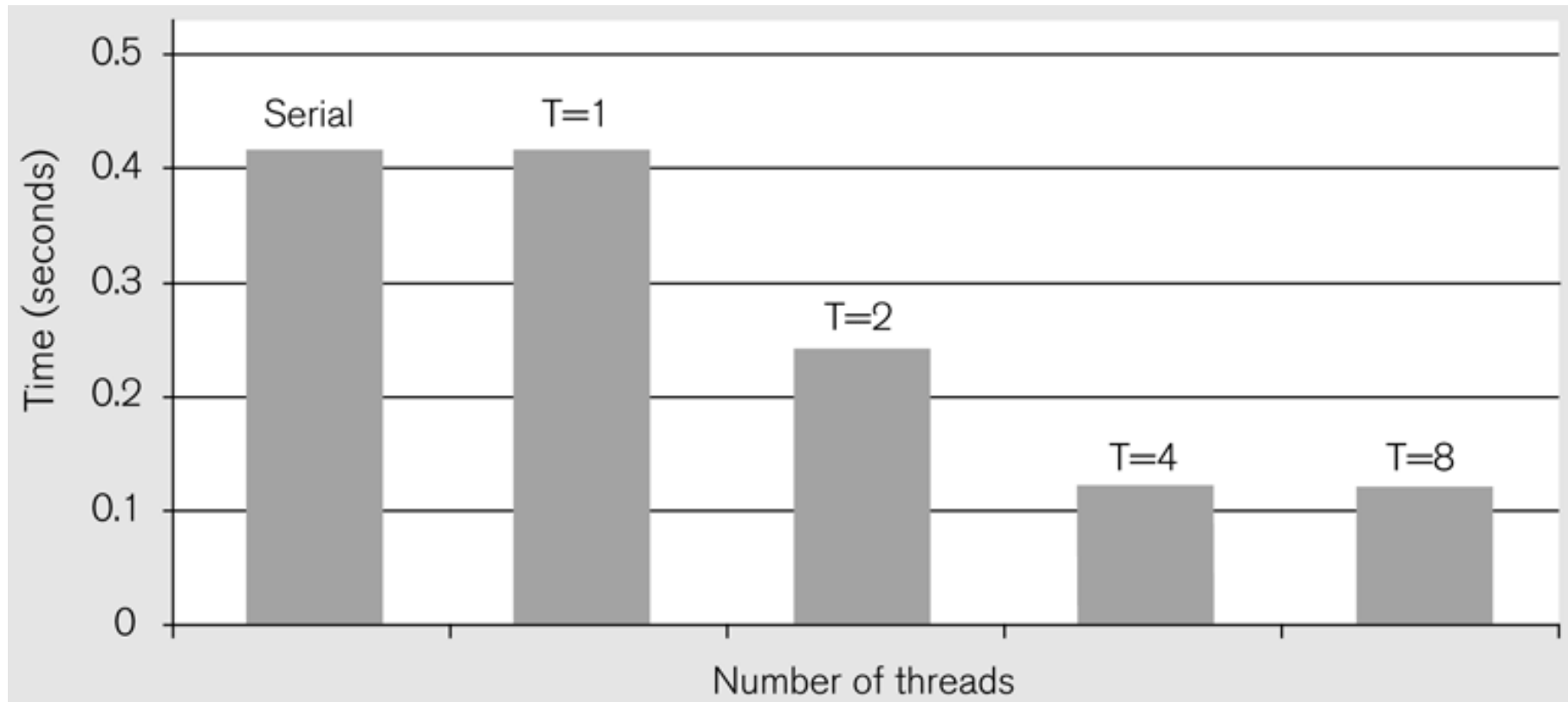
```
1 struct padded_int
2 {
3     int value;
4     char padding[60];
5 } private_count[MaxThreads];
6
7 void count3s_thread(int id)
8 {
9     /* Compute portion of the array this thread should
10        work on */
11     int length_per_thread=length/t;
12     int start=id*length_per_thread;
13
14     for(i=start; i<start+length_per_thread; i++)
15     {
16         if(array[i] == 3)
17         {
18             private_count[id]++;
19         }
20     }
21     mutex_lock(m);
22     count+=private_count[id].value;
23     mutex_unlock(m);
24 }
```

# Correct and Good



Limitation of the hardware

# Confirmation of the Memory Bandwidth Limit



no 3 in the array



# Lessons

---

- Correctness
- Performance
- Scalability
- Portability