

Multi-core Architectures

Dr. Yingwu Zhu

What is parallel computing?

Using multiple processors in parallel to solve problems more quickly than with a single processor

Examples of parallel computing

- A **cluster computer** that contains multiple PCs combined together with a high speed network
- A **shared memory multiprocessor** (SMP*) by connecting multiple processors to a single memory system
- A **Chip Multi-Processor** (CMP) contains multiple processors (called cores) on a single chip

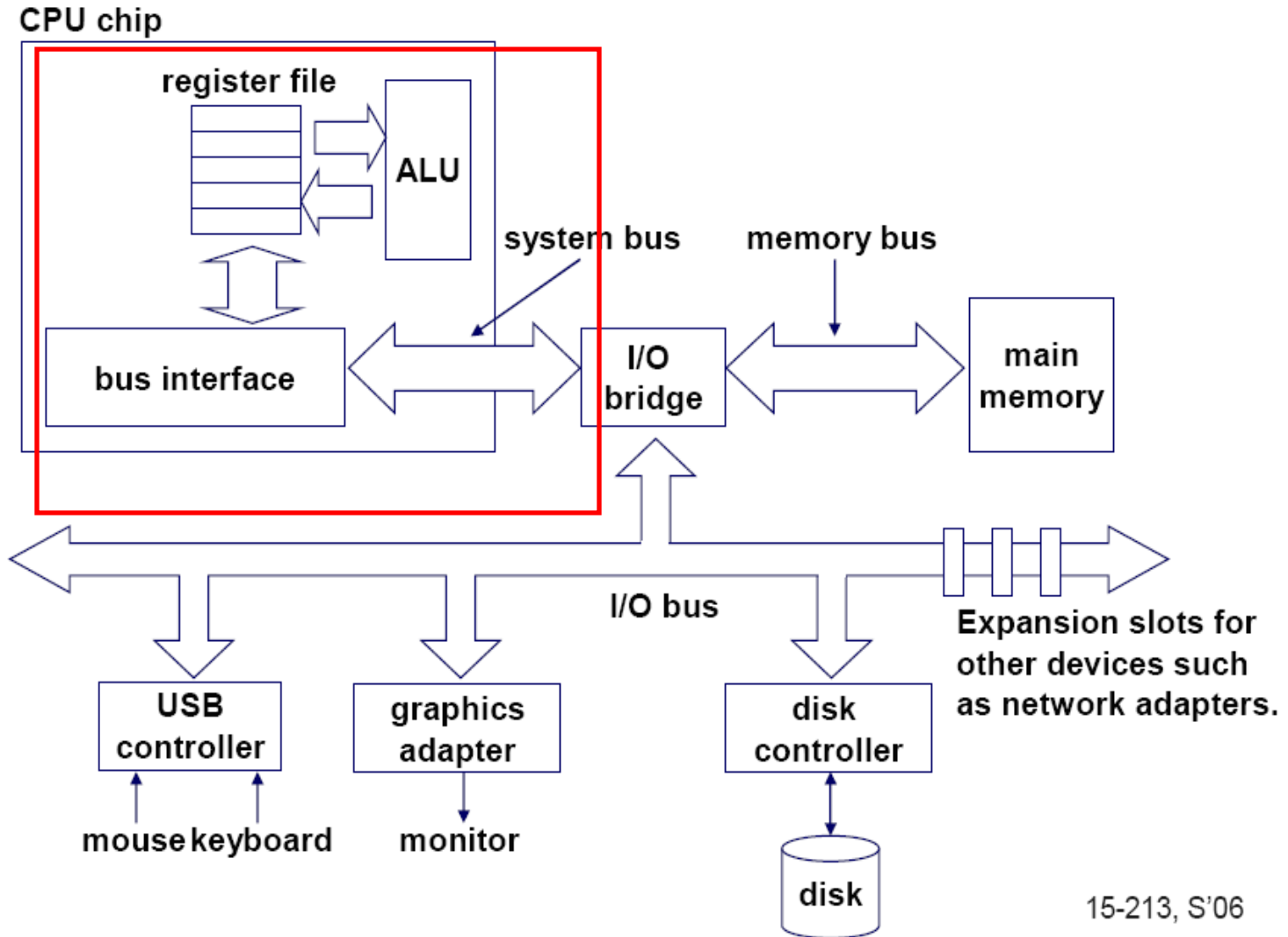
Concurrent execution comes from desire for performance!

Cost & Challenges of Parallel Computing/Execution

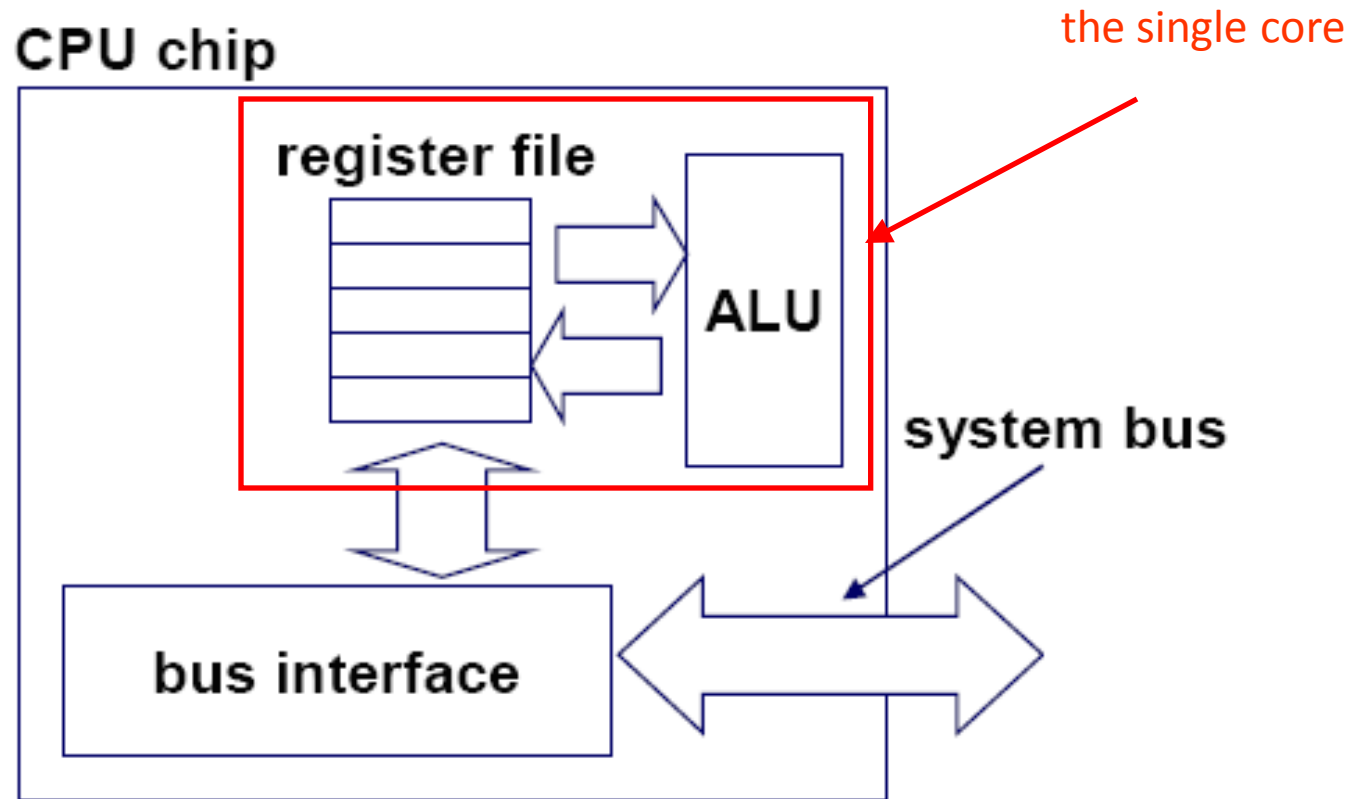
- Communication cost
- Synchronization cost
- Not all problems are amenable to parallelization
- Hard to think in parallel
- Hard to debug

Single Core vs. Multicore

Single-core computer



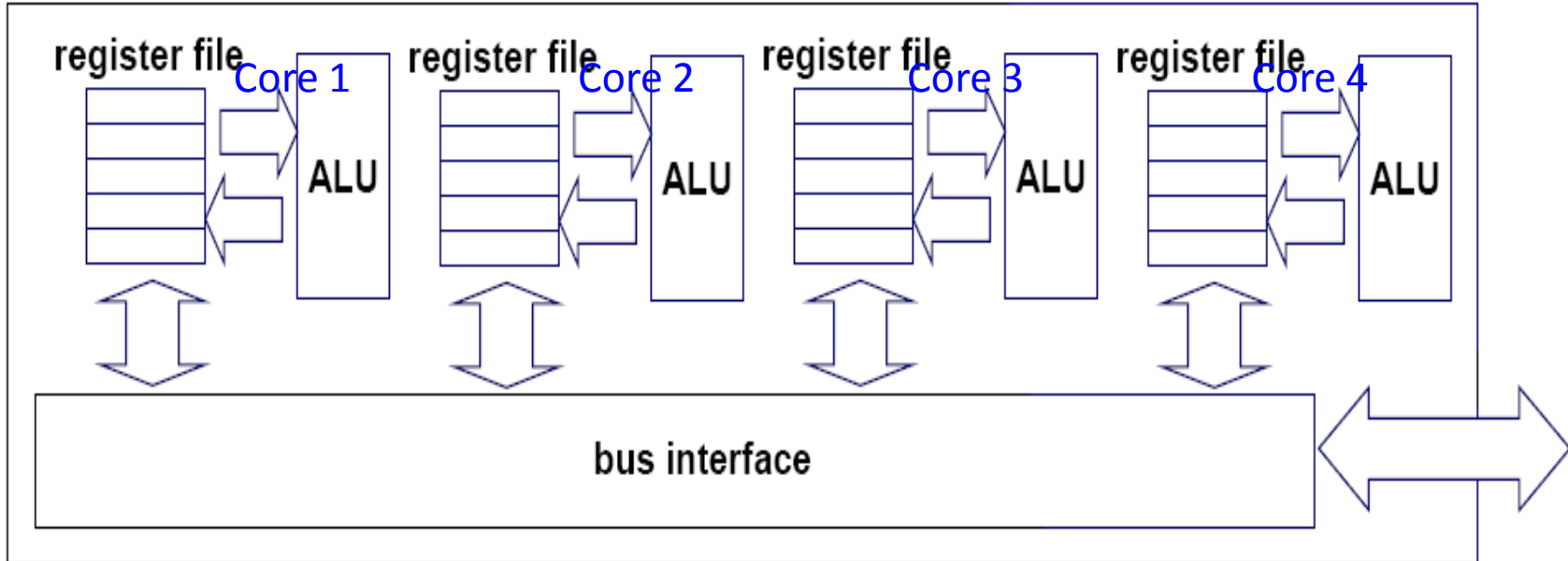
Single-core CPU chip



A **register file** is an array of [processor registers](#) in a [central processing unit \(CPU\)](#)

Multi-core architectures

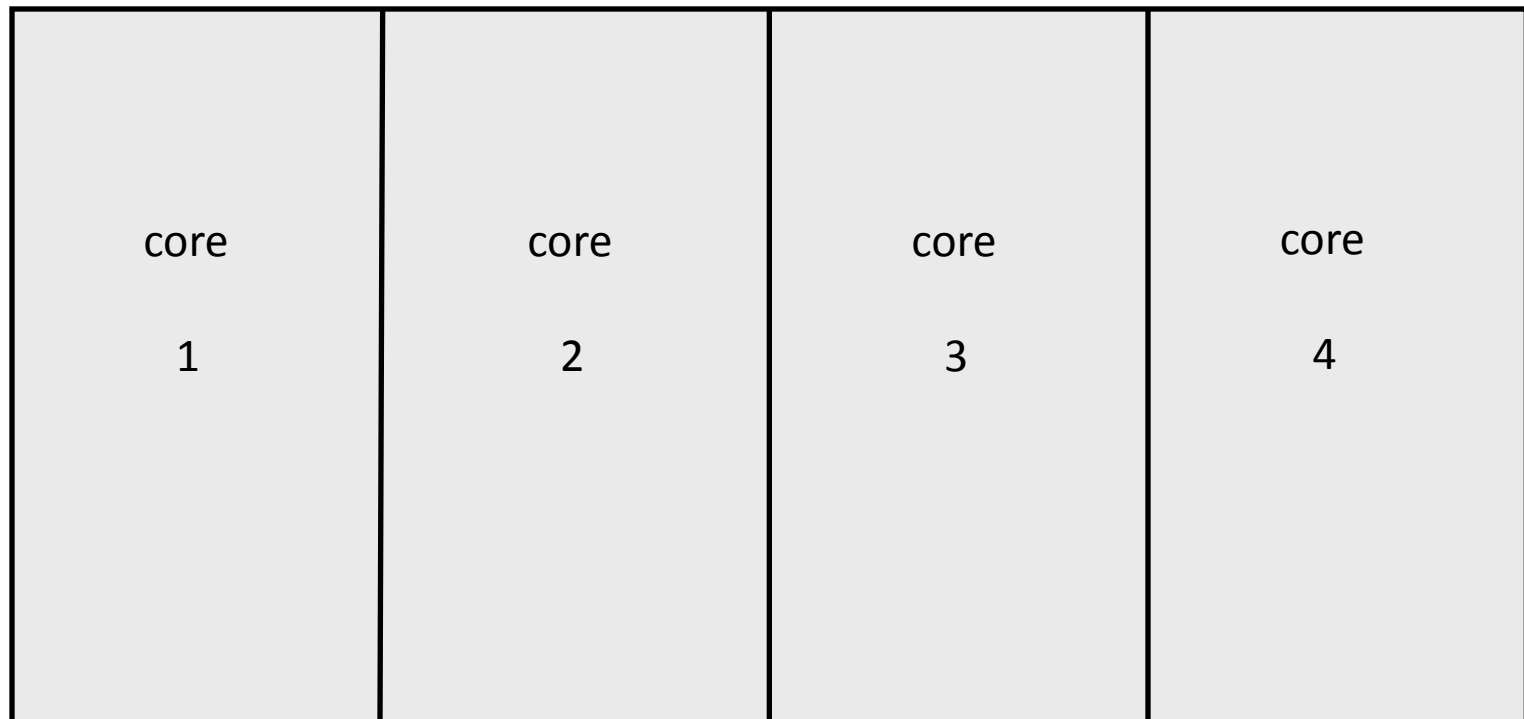
Replicate multiple processor cores on a single die



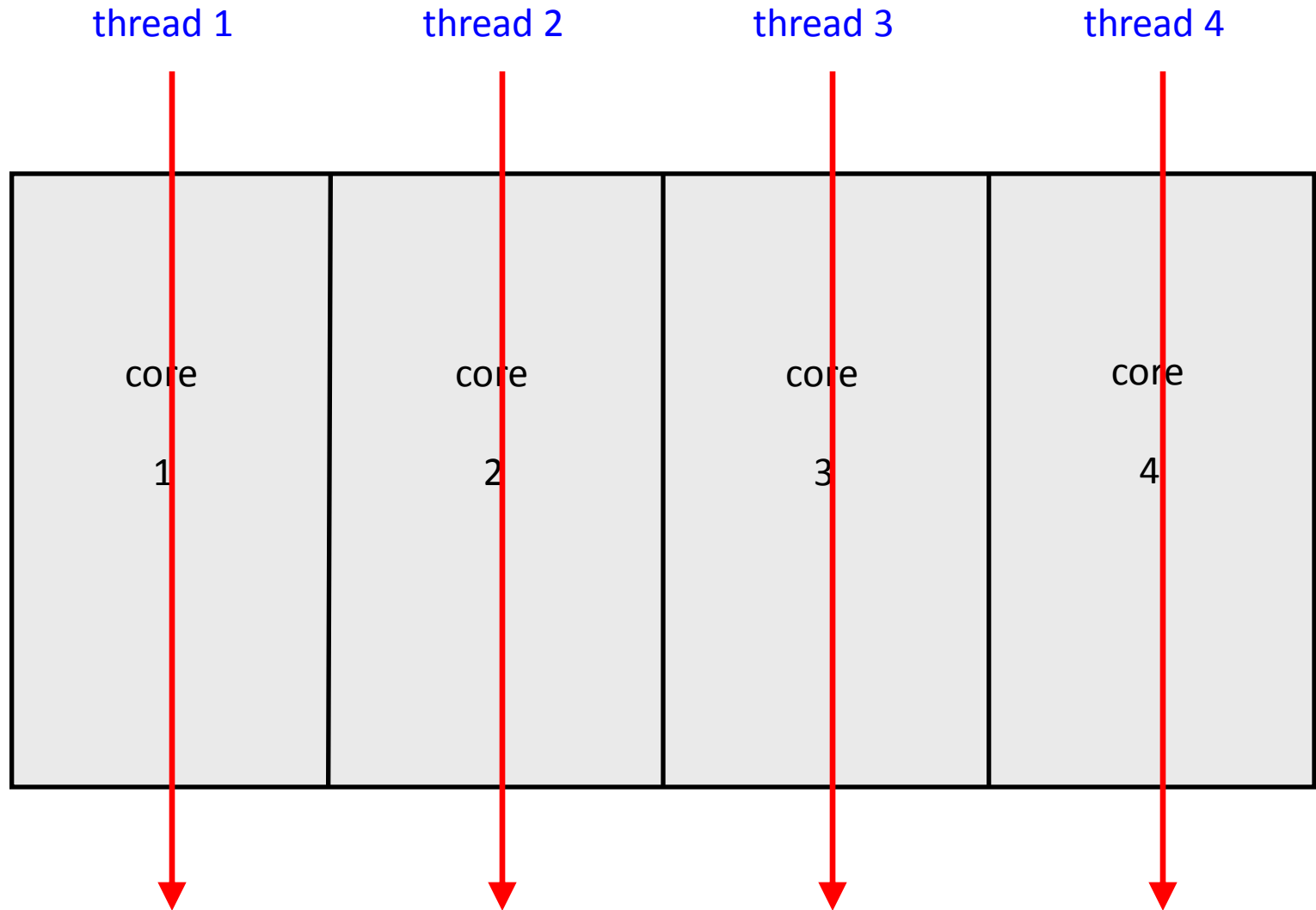
Multi-core CPU chip

Multi-core CPU chip

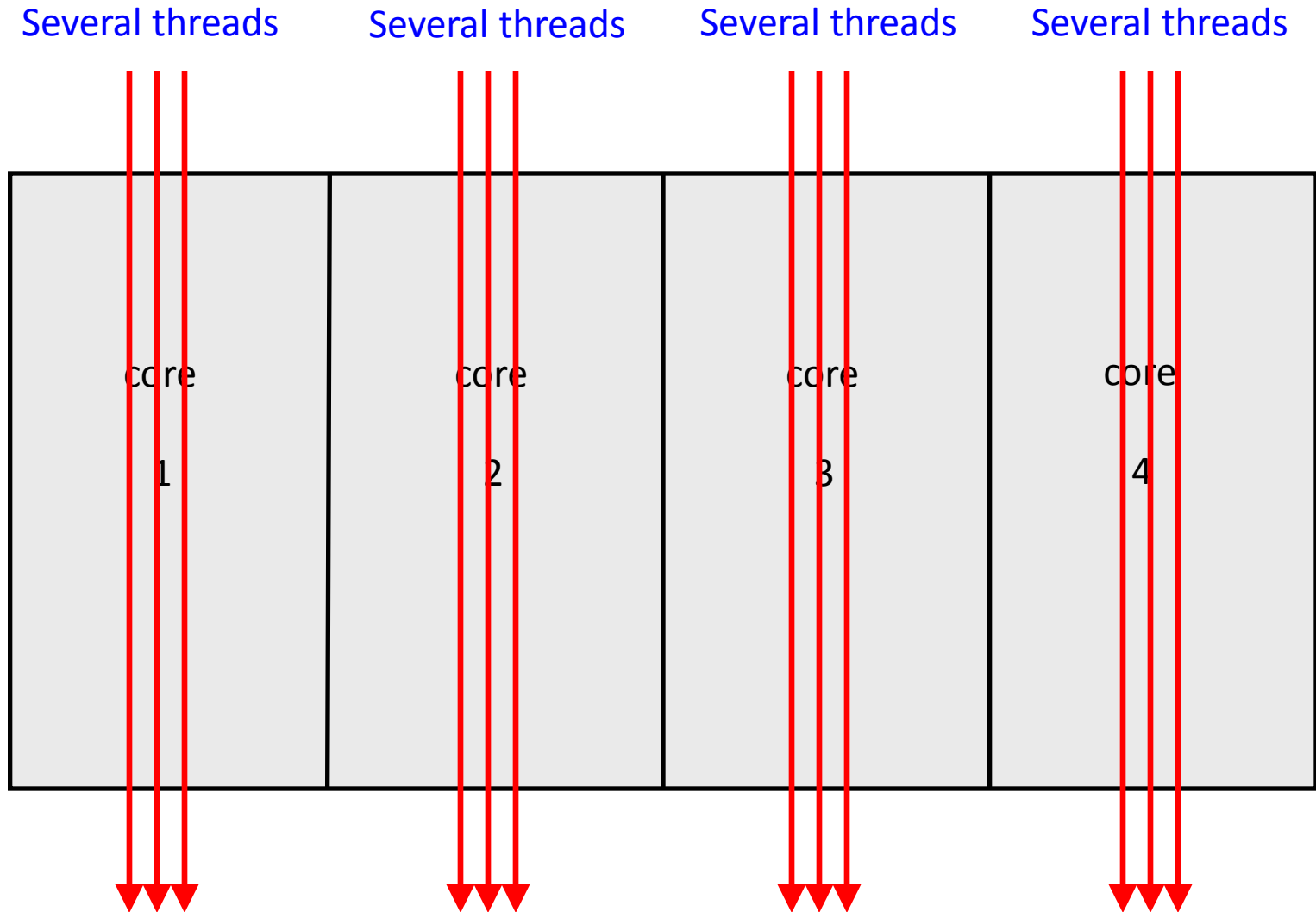
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Within each core, threads are time-sliced (like on a uniprocessor)



Interaction with OS

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, ...

Why Multi-core?

Why multi-core?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
 - heat problems
 - speed of light problems
 - difficult design and verification
 - large design teams necessary
 - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)
- Save power

Multicore Processors Save Power

$$\text{Power} = C * V^2 * F$$

$$\text{Performance} = \text{Cores} * F$$

Let's have two cores

$$\text{Power} = 2 * C * V^2 * F$$

$$\text{Performance} = 2 * \text{Cores} * F$$

But decrease frequency by 50%

$$\text{Power} = 2 * C * V^2 / 4 * F / 2$$

$$\text{Performance} = 2 * \text{Cores} * F / 2$$

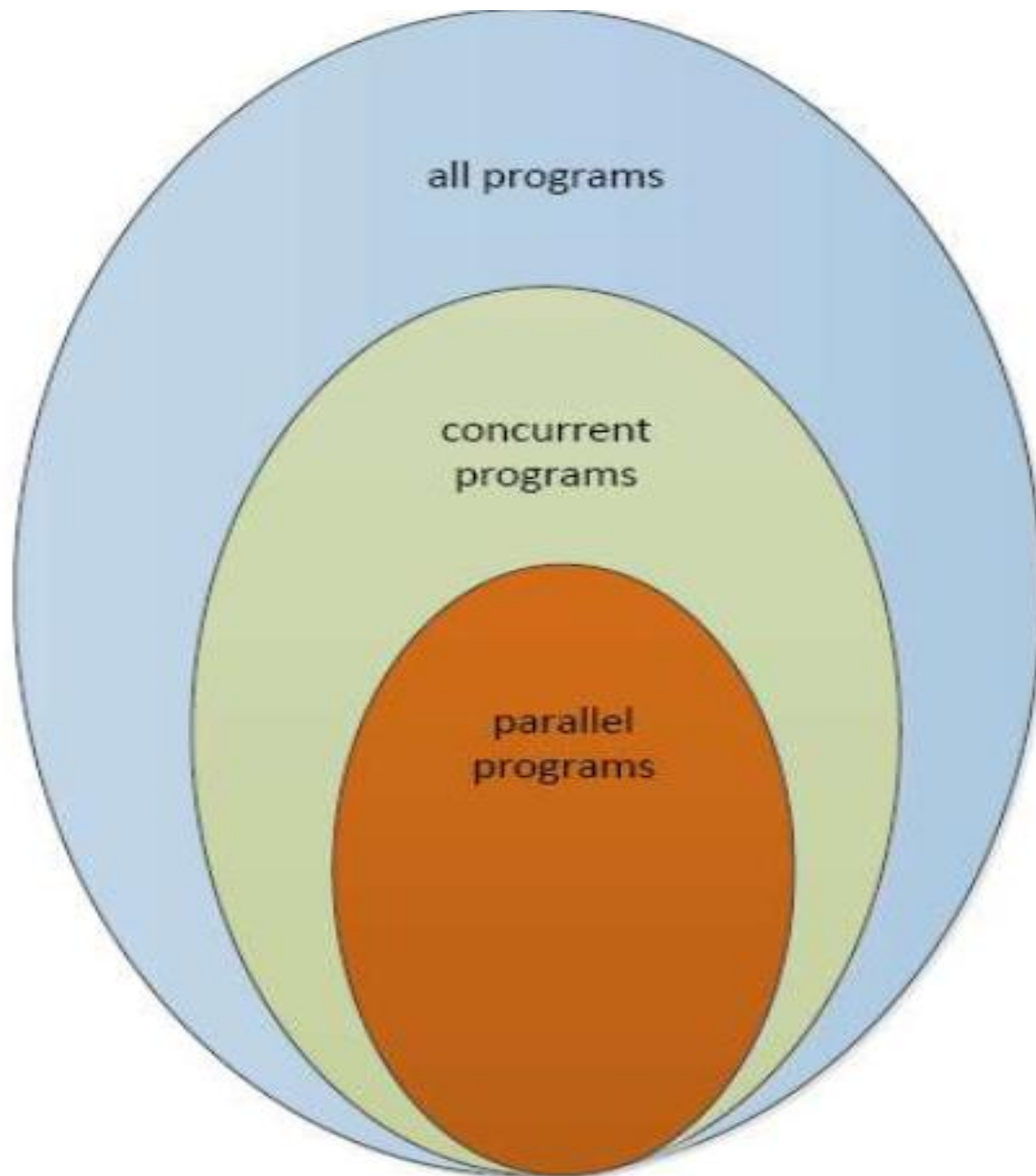


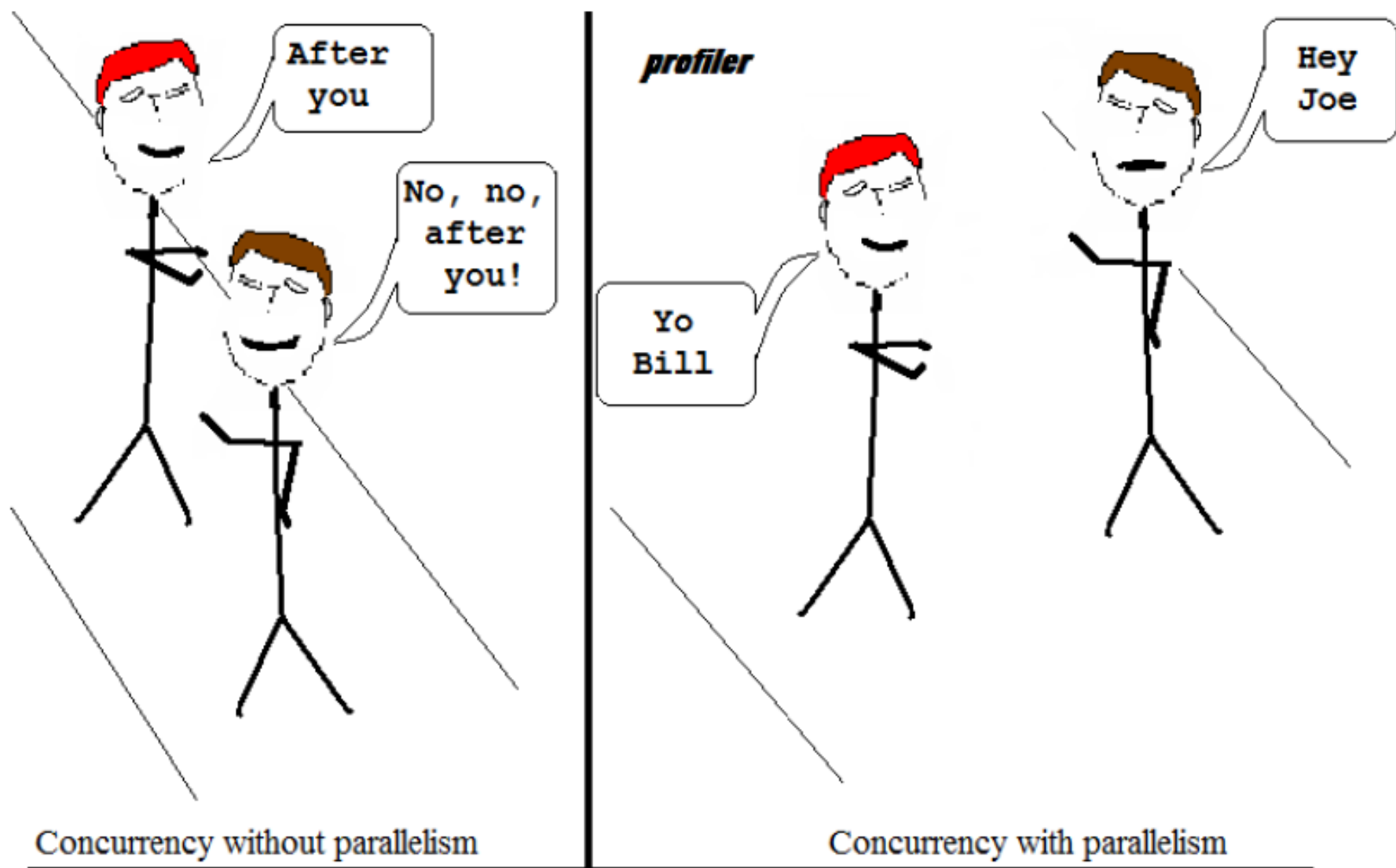
$$\text{Power} = C * V^2 / 4 * F$$

$$\text{Performance} = \text{Cores} * F$$

Concurrency vs. Parallelism: Same meaning?

- Concurrency: At least two tasks are making progress at the same time frame.
 - Not necessarily at the same time
 - Include techniques like time-slicing
 - Can be implemented on a single processing unit
 - Concept more general than parallelism
- Parallelism: At least two tasks execute literally at the same time.
 - Requires hardware with multiple processing units





Performance tuning technique number 106: Concurrency vs. Parallelism

Copyright © Fasterj.com Limited

Questions

- If we have as much hardware as we want, do we get as much parallelism as we wish?
- If we have 2 cores, do we get 2x speedup?

Amdahl's Law



Gene M. Amdahl

- How much of a speedup one could get for a given parallelized task?

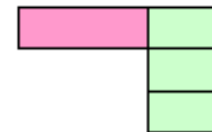
If F is the fraction of a calculation that is sequential then the maximum speed-up that can be achieved by using P processors is $1/(F+(1-F)/P)$



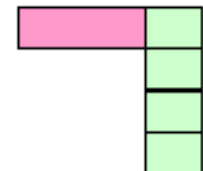
1CPU



2CPUs



3CPUs



4CPUs

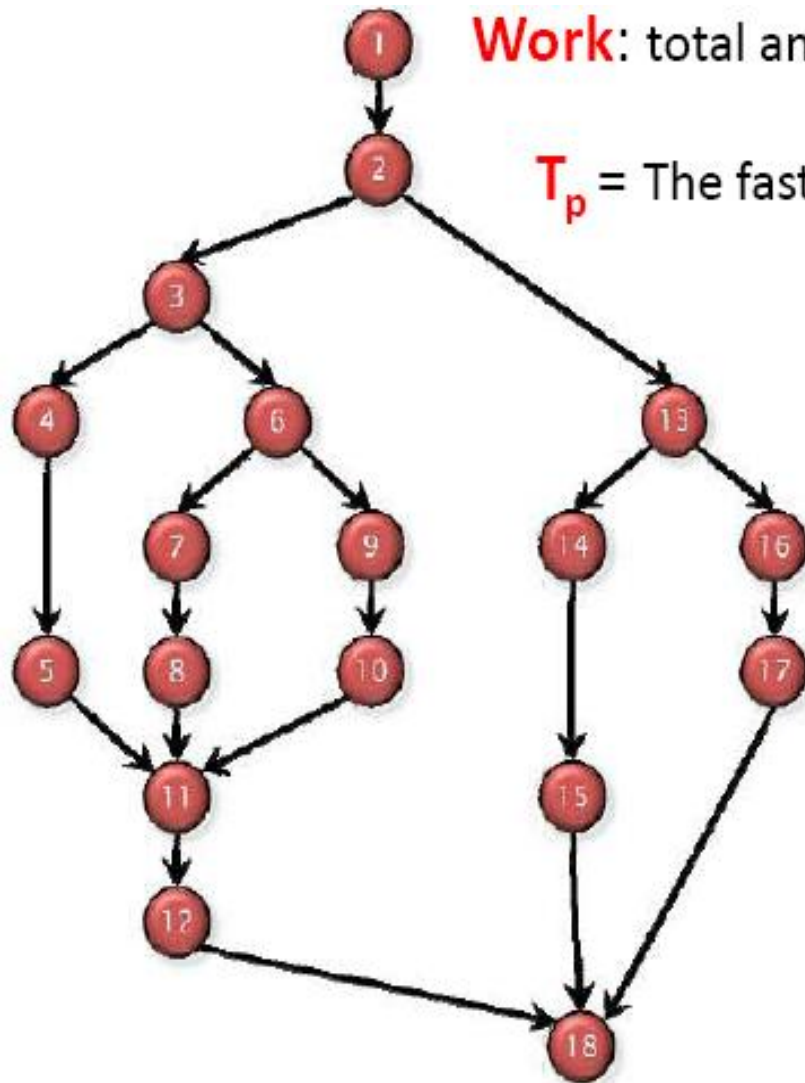
What was Amdahl trying to say?

- Don't invest blindly on large number of processors
- Having faster core (or processor at his time) makes more sense than having many cores.
- Was he right?
 - At his days (the law appeared 1967) many programs have long sequential parts
 - This is not necessarily the case nowadays
 - It is not very easy to find F (sequential portion)

So...

- Decreasing the serialized portion is of greater importance than adding more cores
- Only when a program is mostly parallelized, does adding more processors help more than parallelizing the remaining rest
- **Gustafson's law**: computations involving arbitrarily large data sets can be efficiently parallelized
- Both Amdahl and Gustafson do not take into account:
 - The overhead of synchronization, communication, OS, etc.
 - Load may not be balanced among cores
- So you have to use these laws as guidelines and theoretical bounds only.

DAG Models for Multithreading



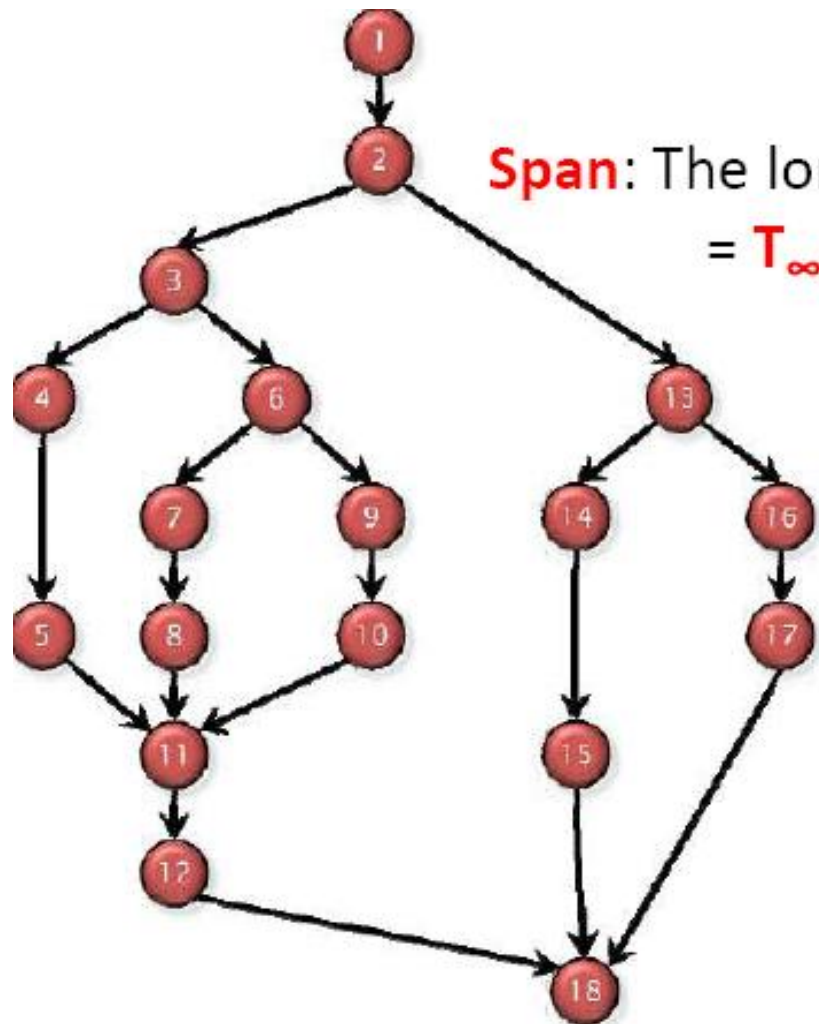
Work: total amount of time spent on all instructions

T_p = The fastest possible execution time on P processors

Work Law:

$$T_p \geq T_1/P$$

DAG Models for Multithreading




Span: The longest path of dependence in the DAG
 $= T_{\infty}$

Span Law:

$$T_p \geq T_{\infty}$$

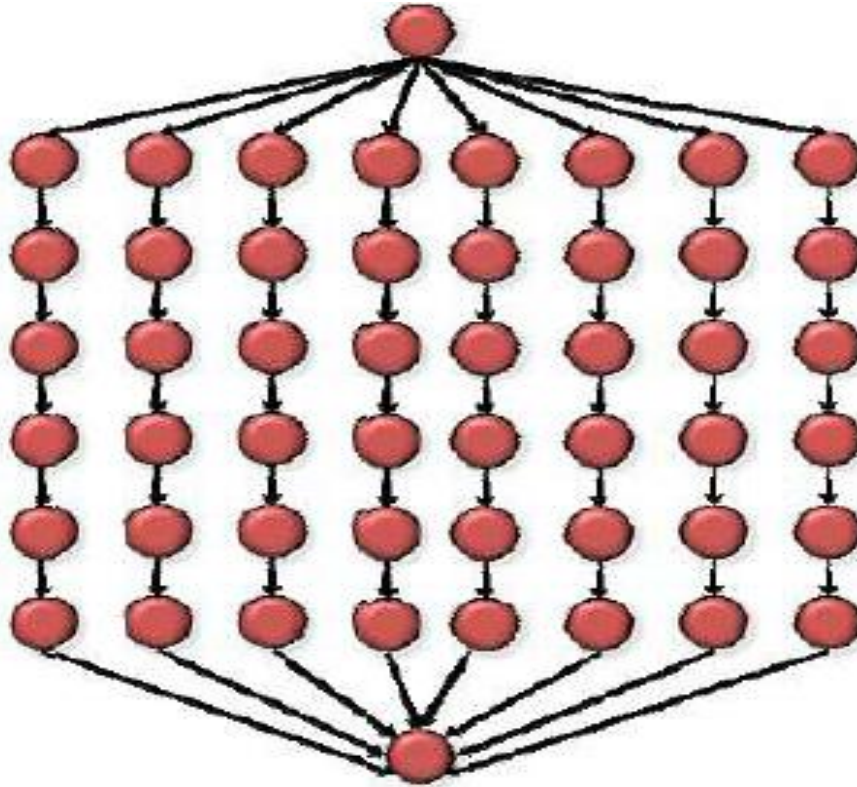
Can we define parallelism now?

How about? T_1/T_∞



Ratio of work to span

Can we define parallelism now?



Work: $T_1 = 50$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 6.25$

Parallelism Granularity

- Instruction Level Parallelism (ILP)
- Thread Level Parallelism (TLP)

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can **re-order, pipeline instructions, split** them into microinstructions, do aggressive **branch prediction**, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- *Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP*

General context: multiprocessors

- Multiprocessor is any computer with several processors
- SIMD
 - Single instruction, multiple data
 - Modern graphics cards
- MIMD
 - Multiple instructions, multiple data

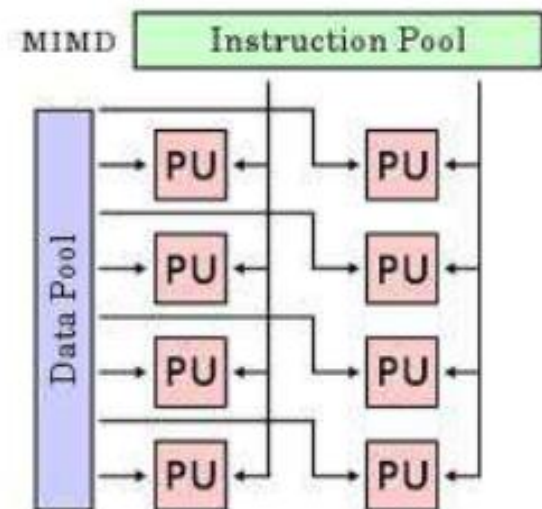
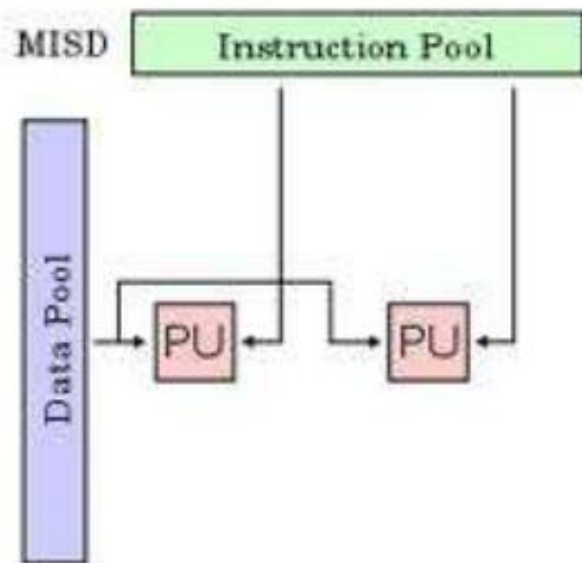
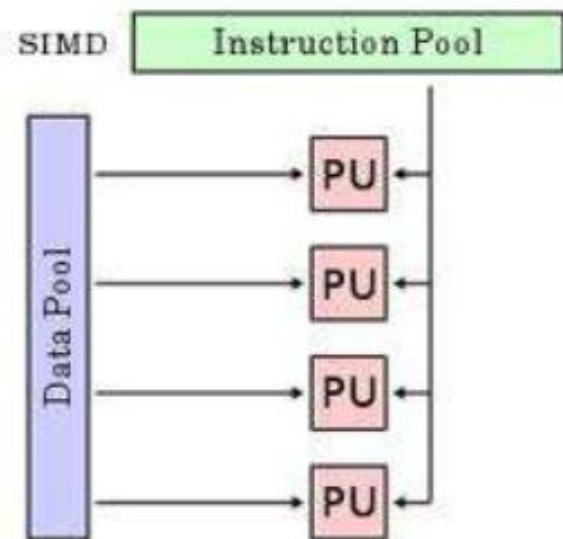
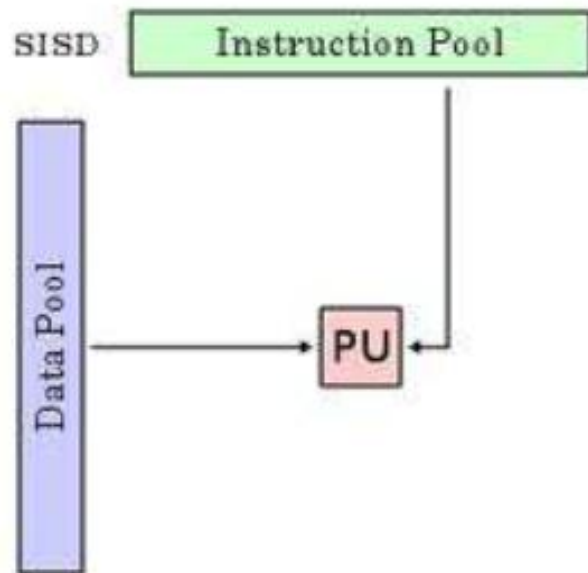


Lemieux cluster,
Pittsburgh
supercomputing
center

Flynn Classification: Computer Architecture

- Proposed by Michael Flynn in 1966
- Based on: Instructions & Data

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD



PU = Processing Unit

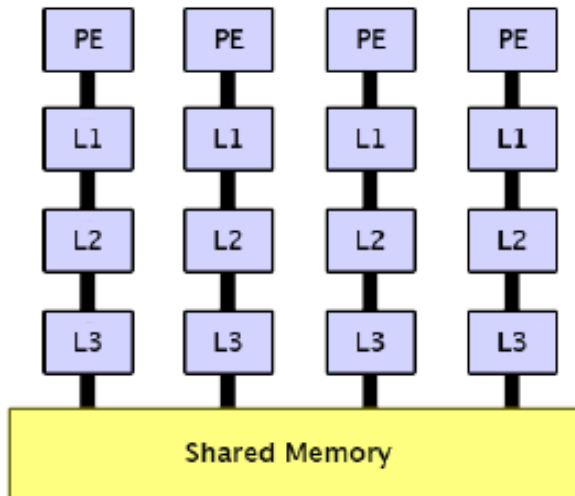
SIMD vs. MIMD

- MIMD = Multiple Instruction, Multiple Data
 - “traditional” parallel processing
 - N processors all doing their own thing
- SIMD = Single Instruction, Multiple Data
 - All processors do exactly the same thing
 - Simple hardware

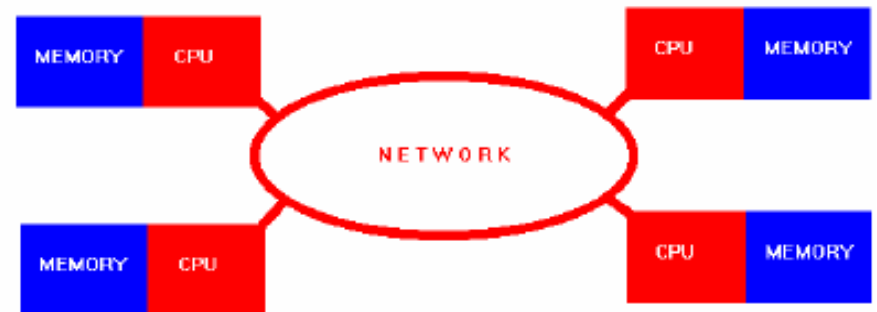
Multiprocessor (MIMD) memory types

- Shared memory:
In this model, there is one (large) common shared memory for all processors
- Distributed memory:
In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else
- Hybrid

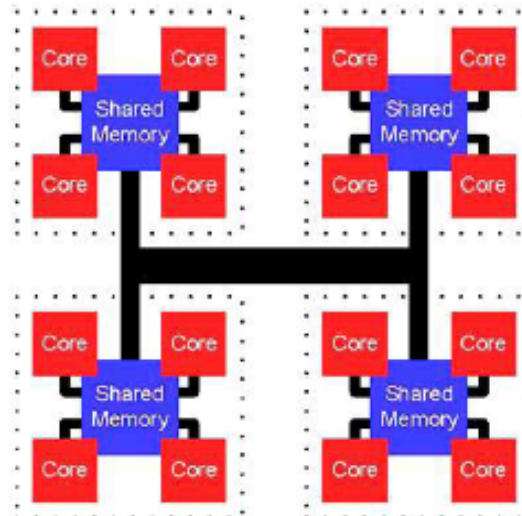
Shared Memory



Distributed Memory



Hybrid



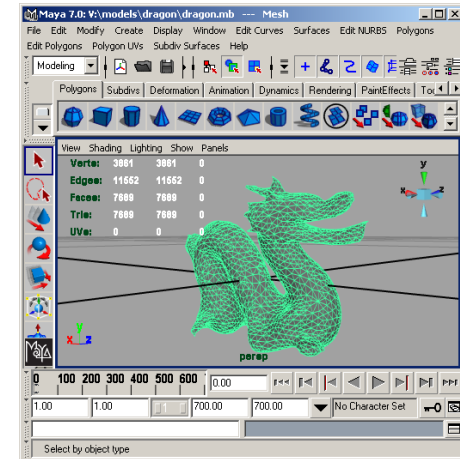
Multi-core processor is a special kind of a multiprocessor:

All processors are on the same chip

- Multi-core processors are MIMD:
Different cores execute different threads (**M**ultiple **I**nstructions), operating on different parts of memory (**M**ultiple **D**ata).
- Multi-core is a shared memory multiprocessor:
All cores share the same memory

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)



Each can
run on its
own core



More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- “Anything that can be threaded today will map efficiently to multi-core”
- BUT: some applications difficult to parallelize

Single superscalar core

Equipped with SMT

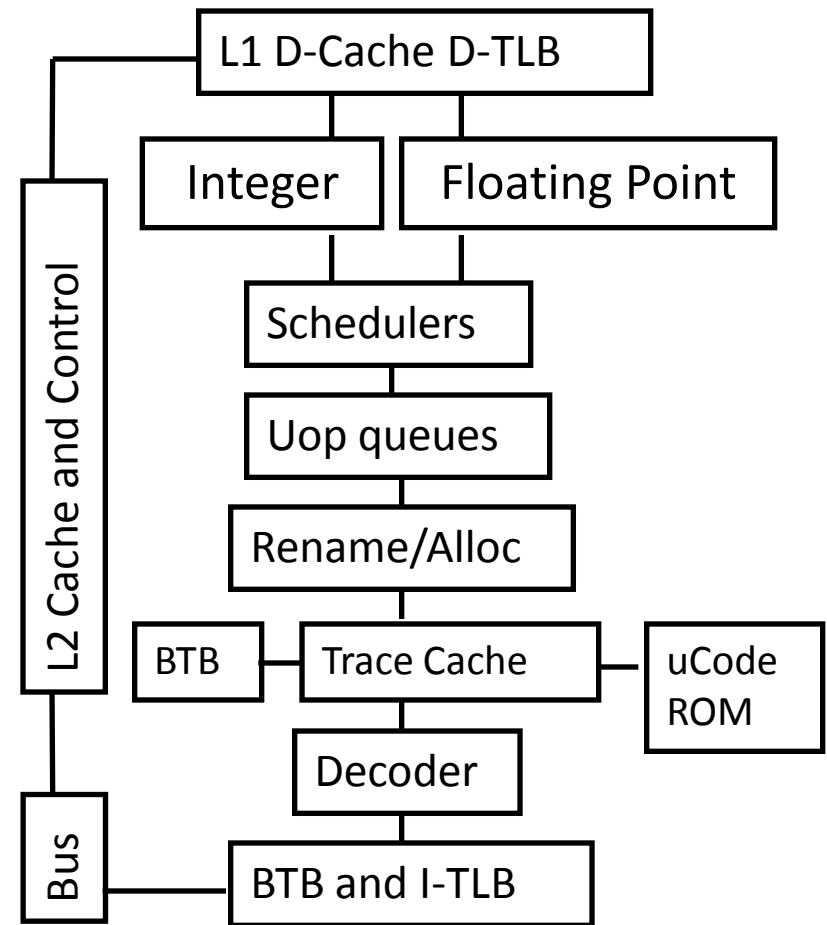
- Simultaneous MultiThreading

Simultaneous multithreading: complementary to multi-cores

- Problem addressed:
The processor pipeline can get stalled:

- Waiting for the result of a long floating point (or integer) operation
- Waiting for data to arrive from memory

Other execution units
wait unused!

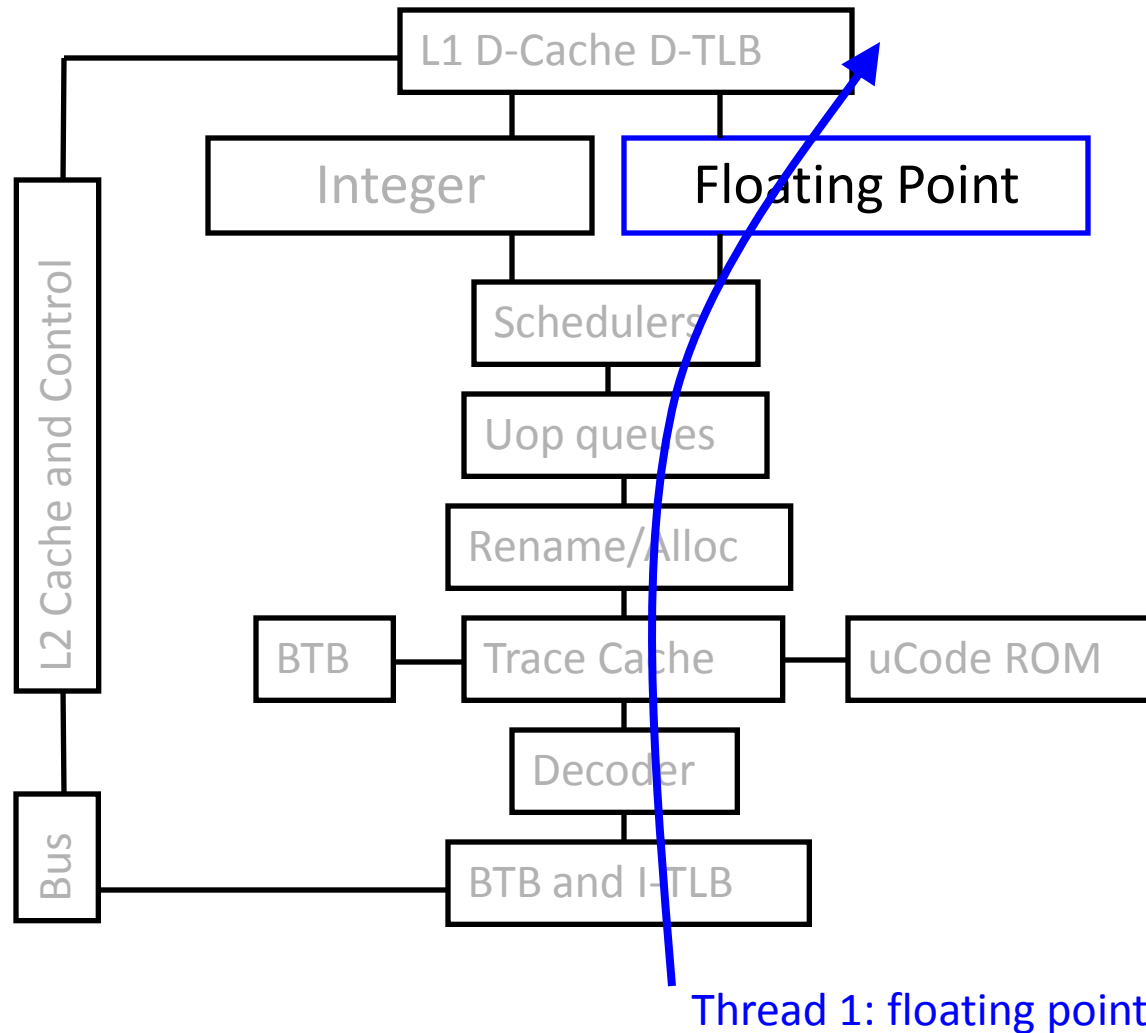


Source: Intel

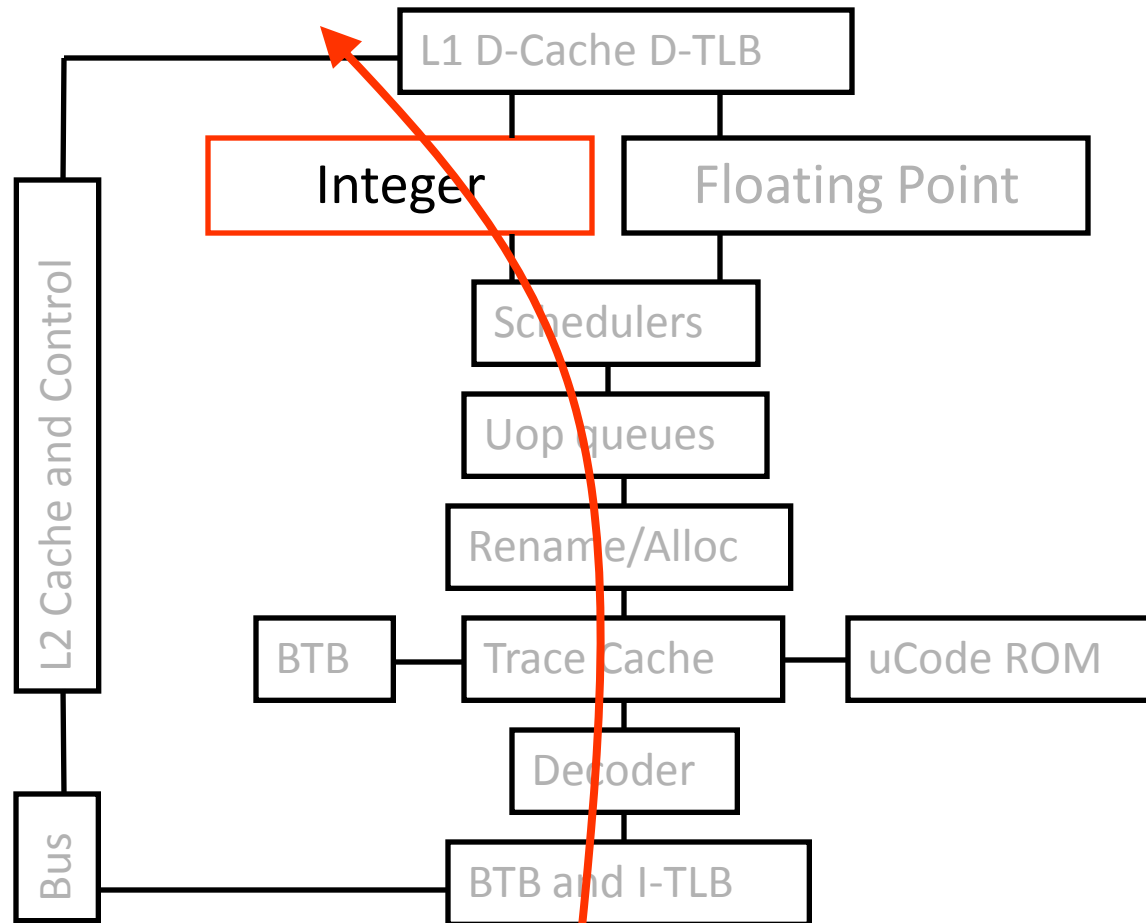
Simultaneous multithreading (SMT)

- Permits multiple independent threads to execute **SIMULTANEOUSLY** on the **SAME** core
- Weaving together multiple “threads” on the same core
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

Without SMT, only a single thread can run at any given time

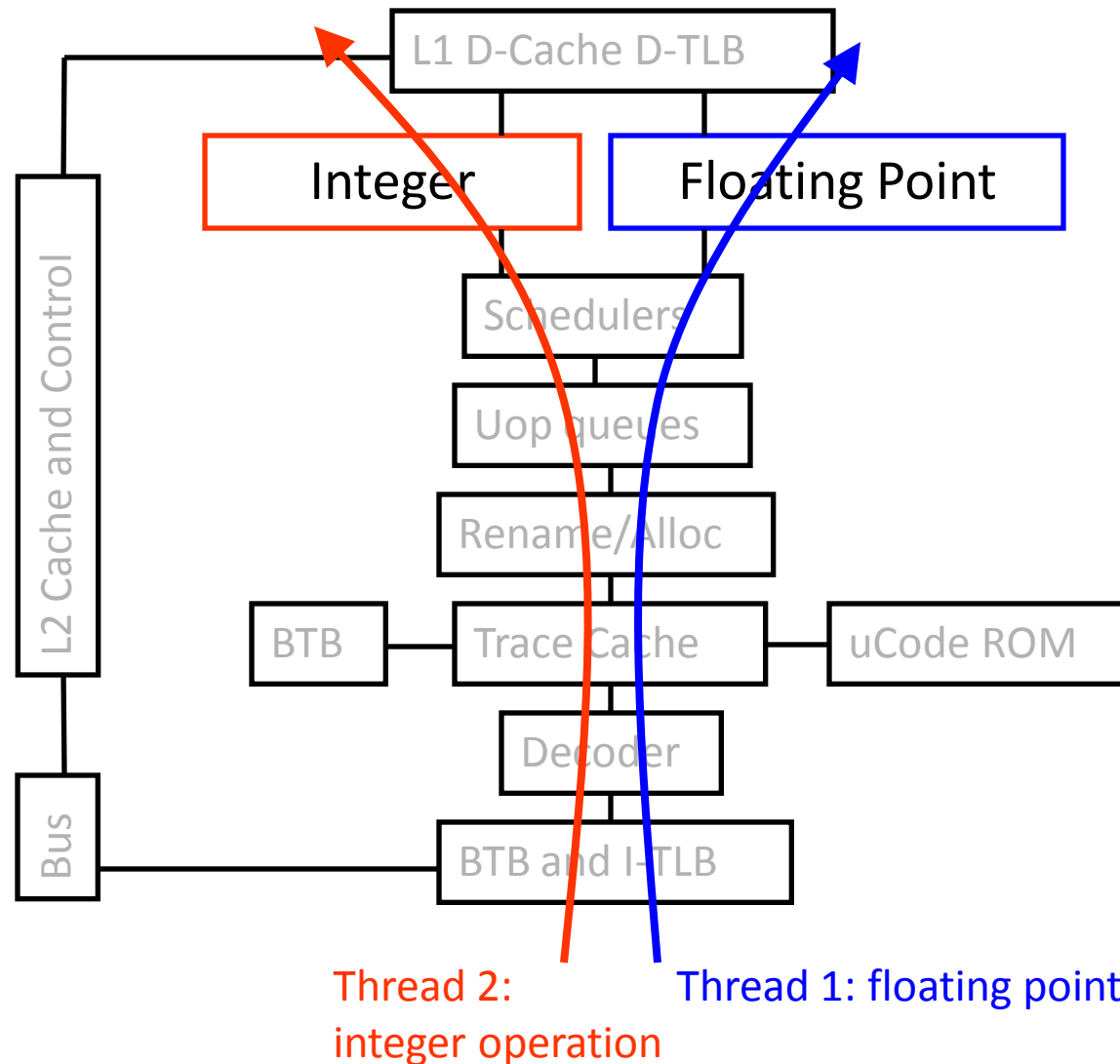


Without SMT, only a single thread can run at any given time

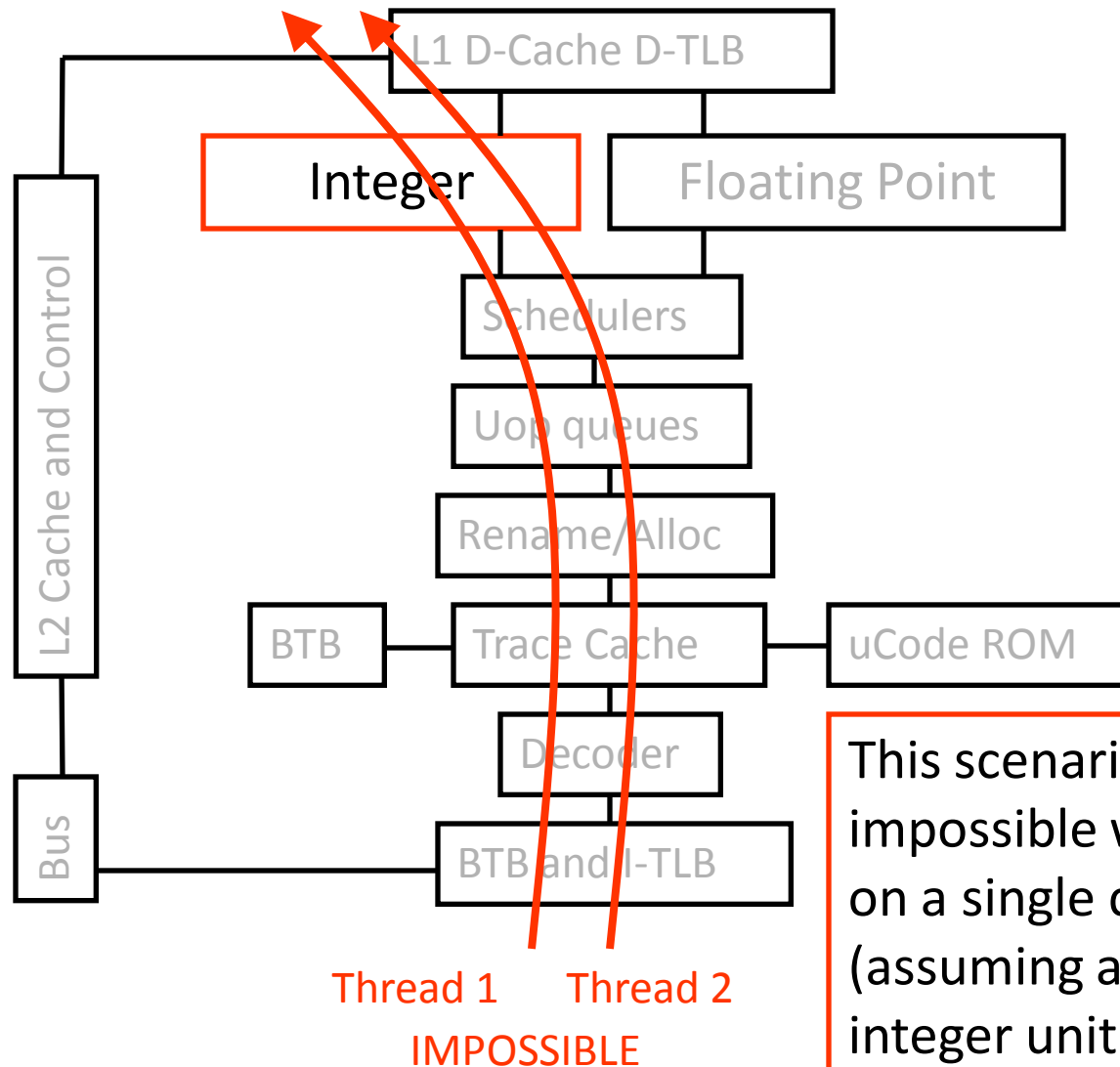


Thread 2:
integer operation

SMT processor: both threads can run concurrently



But: Can't simultaneously use the same functional unit

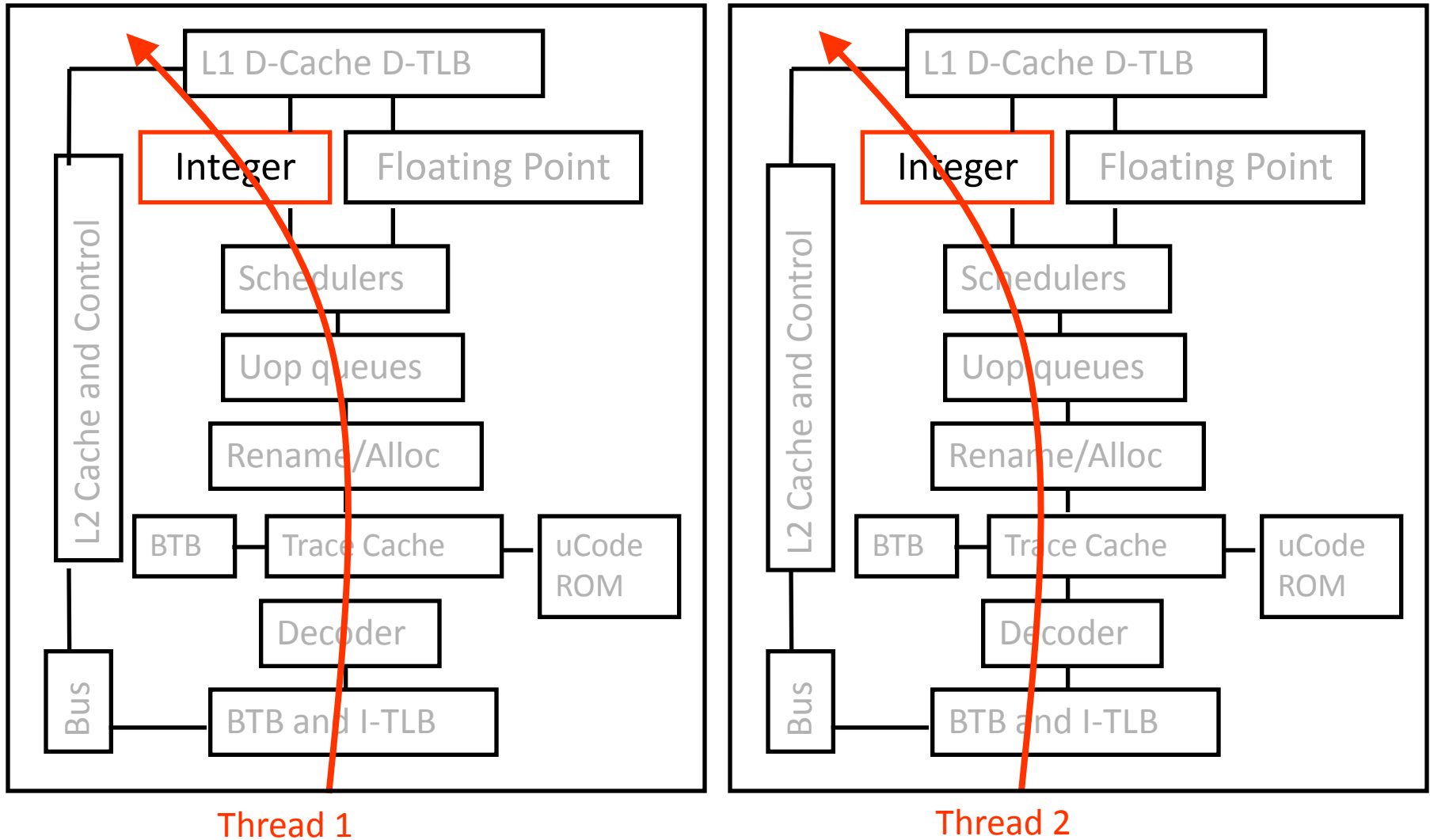


SMT not a “true” parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate “virtual processor”
- The chip has only a single copy of each resource
- Compare to multi-core:
each core has its own copy of resources

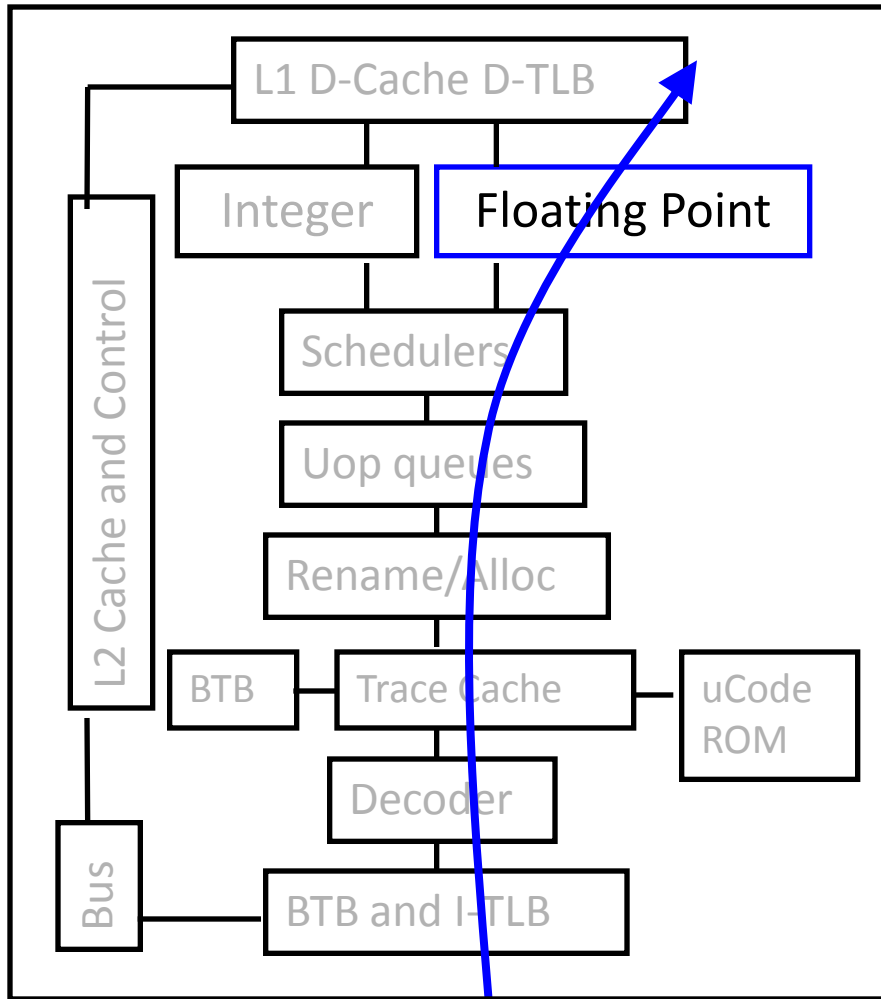
Multi-core:

threads can run on separate cores

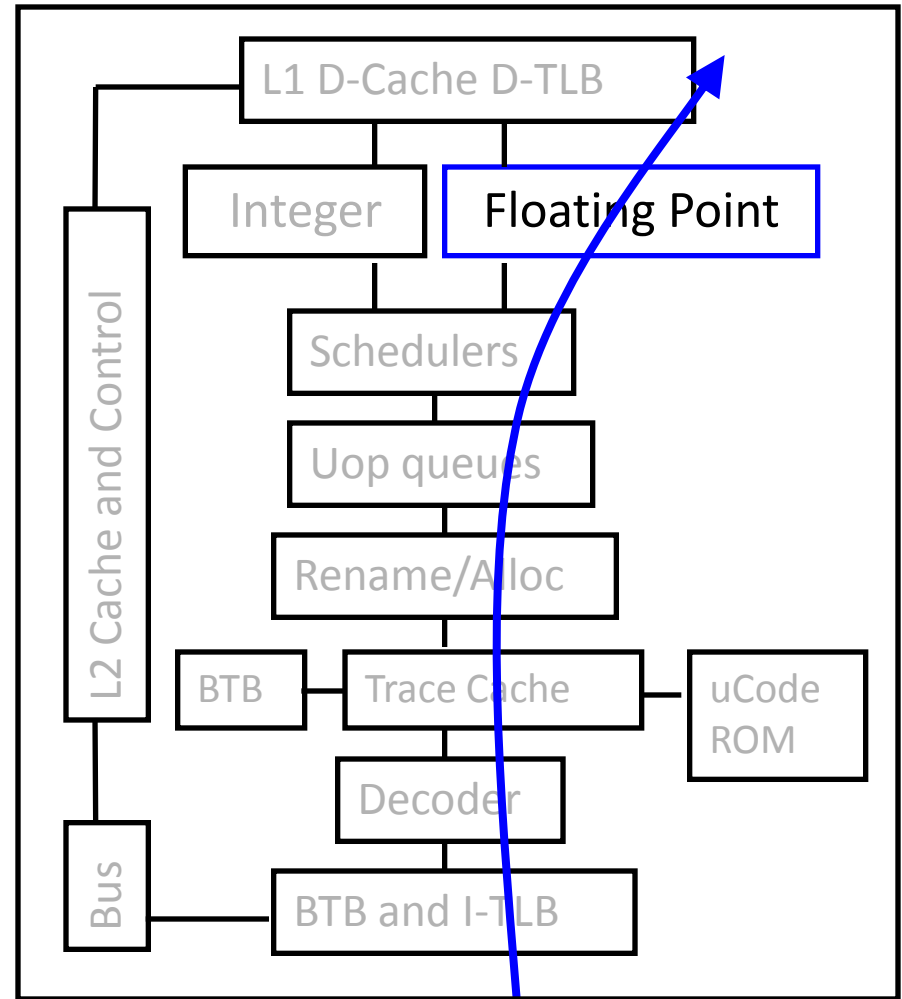


Multi-core:

threads can run on separate cores



Thread 3

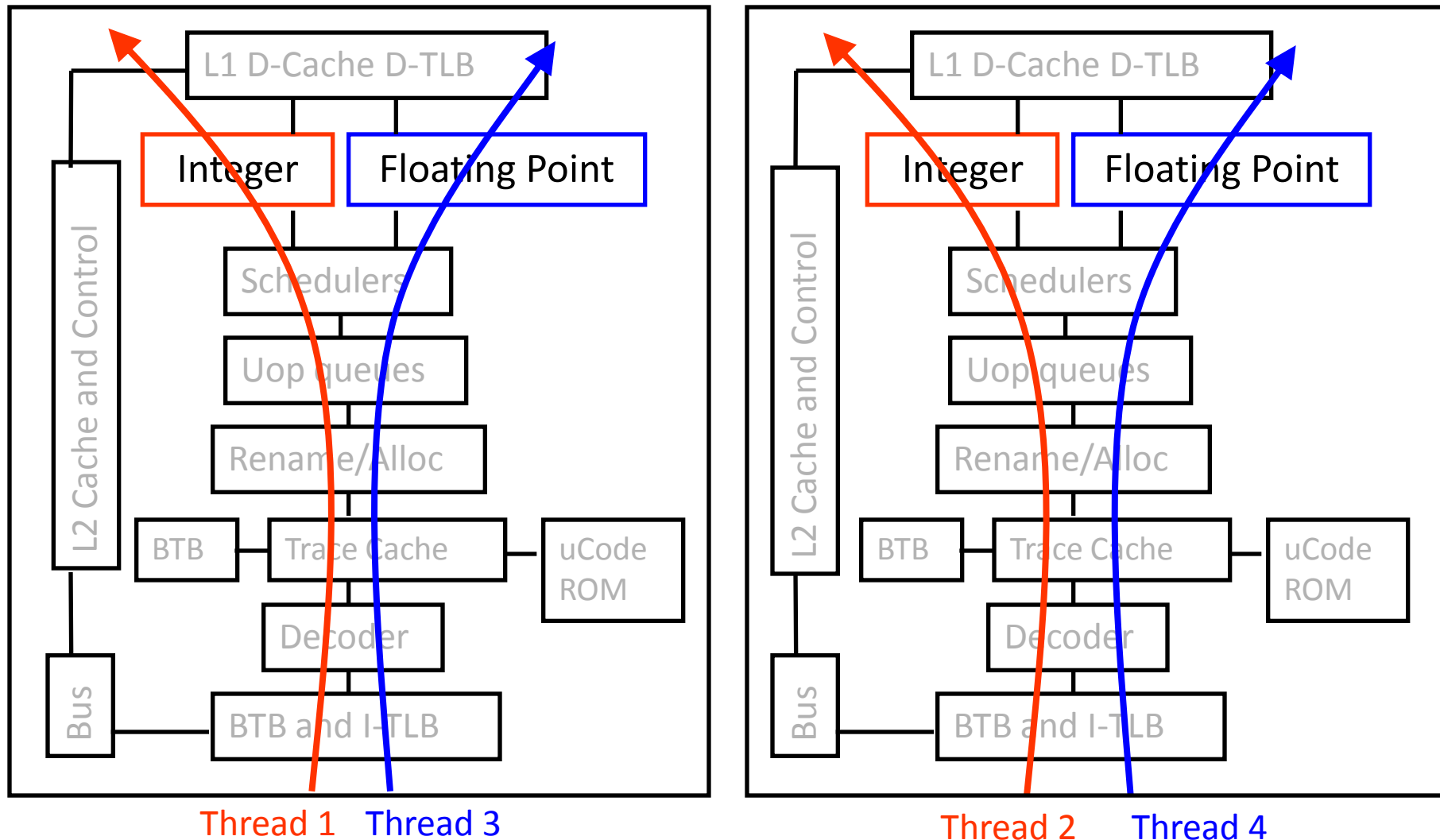


Thread 4

Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
 - Single-core, non-SMT: standard uniprocessor
 - Single-core, with SMT
 - Multi-core, non-SMT
 - Multi-core, with SMT
- The number of SMT threads:
2, 4, or sometimes 8 simultaneous threads
- Intel calls them “hyper-threads”

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Advantages/disadvantages?

Comparison: multi-core vs SMT

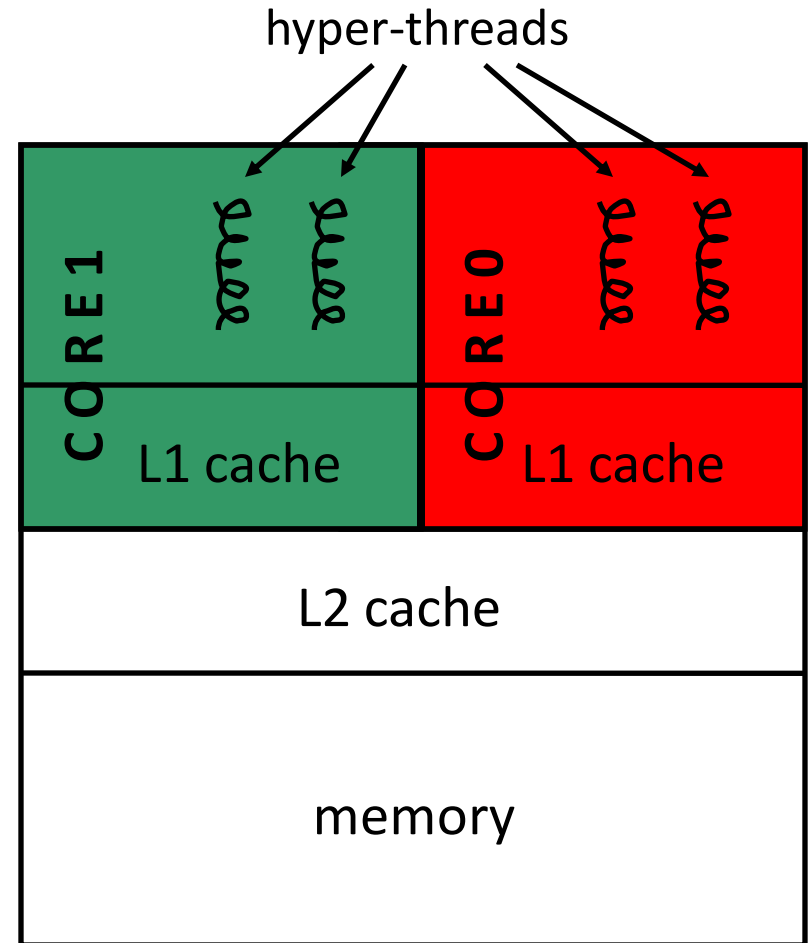
- Multi-core:
 - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
 - However, great with thread-level parallelism
- SMT
 - Can have one large and fast superscalar core
 - Great performance on a single thread
 - Mostly still only exploits instruction-level parallelism

The memory hierarchy

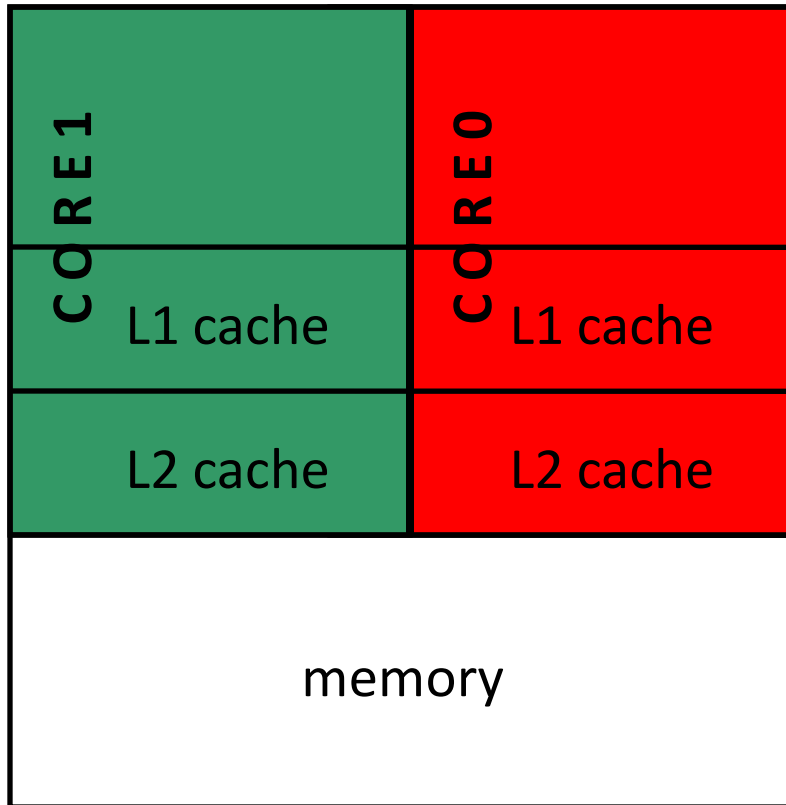
- If simultaneous multithreading only:
 - all caches shared
- Multi-core chips:
 - L1 caches private
 - L2 caches private in some architectures and shared in others
- Memory is always shared

“Fish” machines

- Dual-core Intel Xeon processors
- Each core is hyper-threaded
- Private L1 caches
- Shared L2 caches

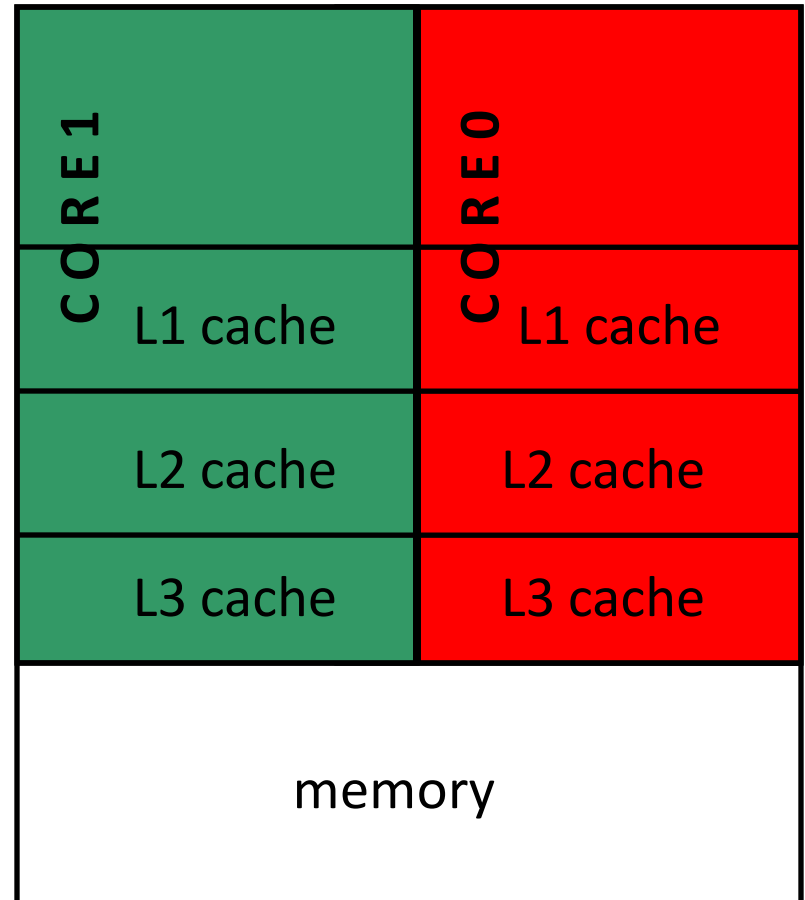


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

Private vs shared caches?

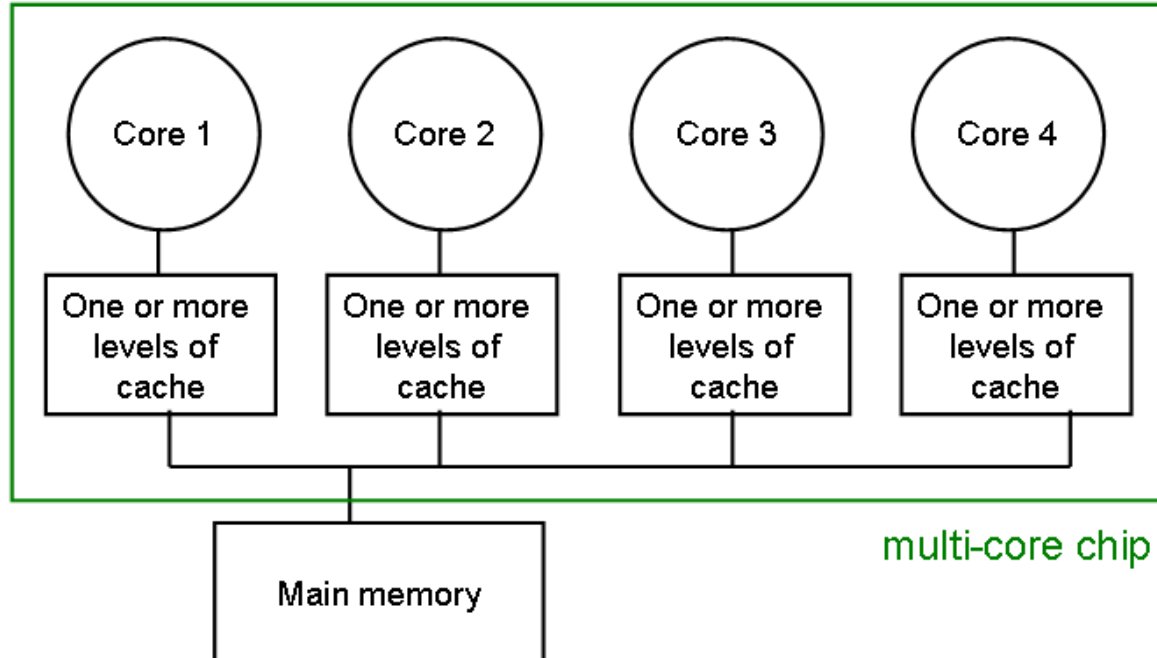
- Advantages/disadvantages?

Private vs shared caches

- Advantages of private:
 - They are closer to core, so faster access
 - Reduces contention
- Advantages of shared:
 - Threads on different cores can share the same cache data
 - More cache space available if a single (or a few) high-performance thread runs on the system

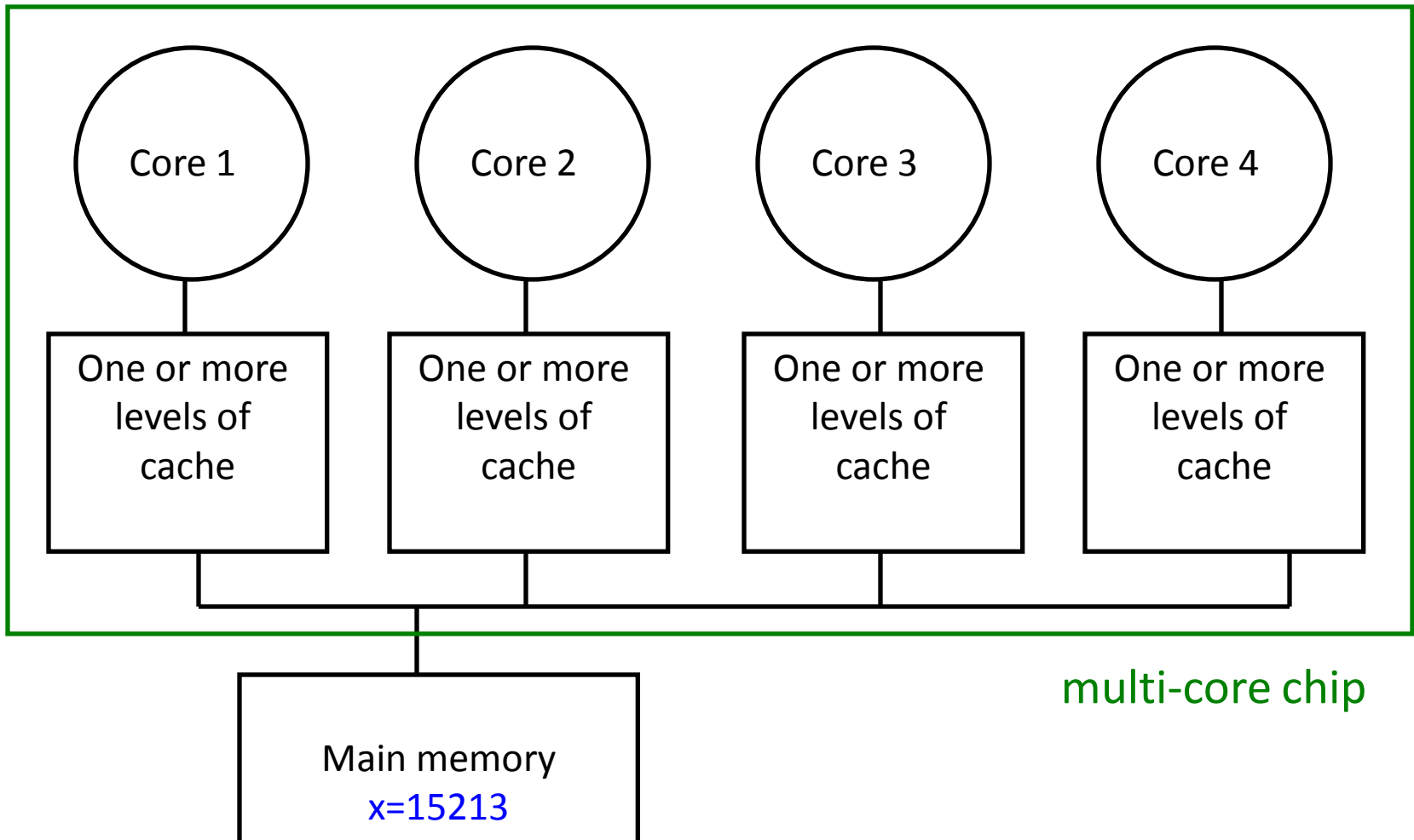
The cache coherence problem

- Since we have private caches:
How to keep the data consistent across caches?
- Each core should perceive the memory as a monolithic array, shared by all the cores



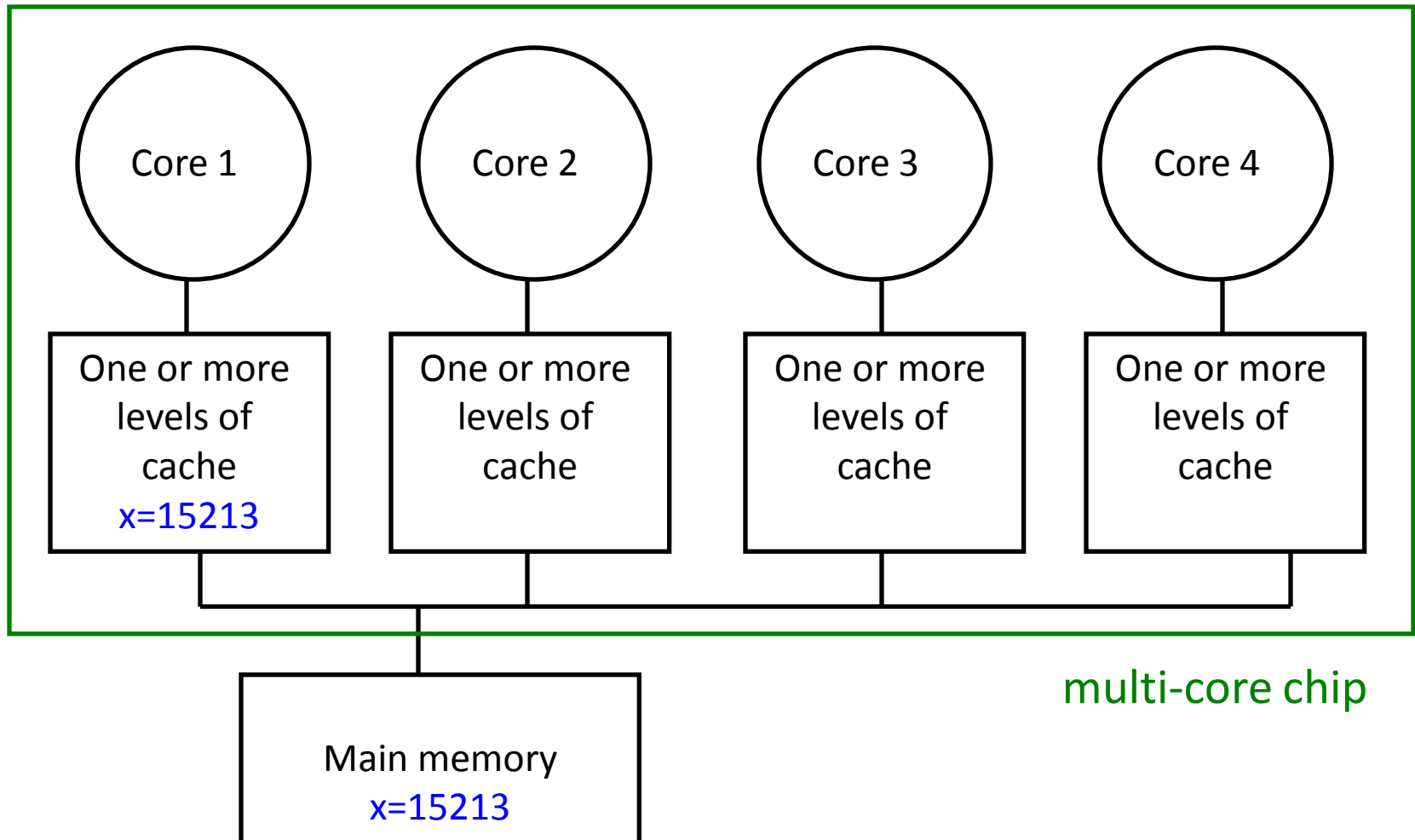
The cache coherence problem

Suppose variable x initially contains 15213



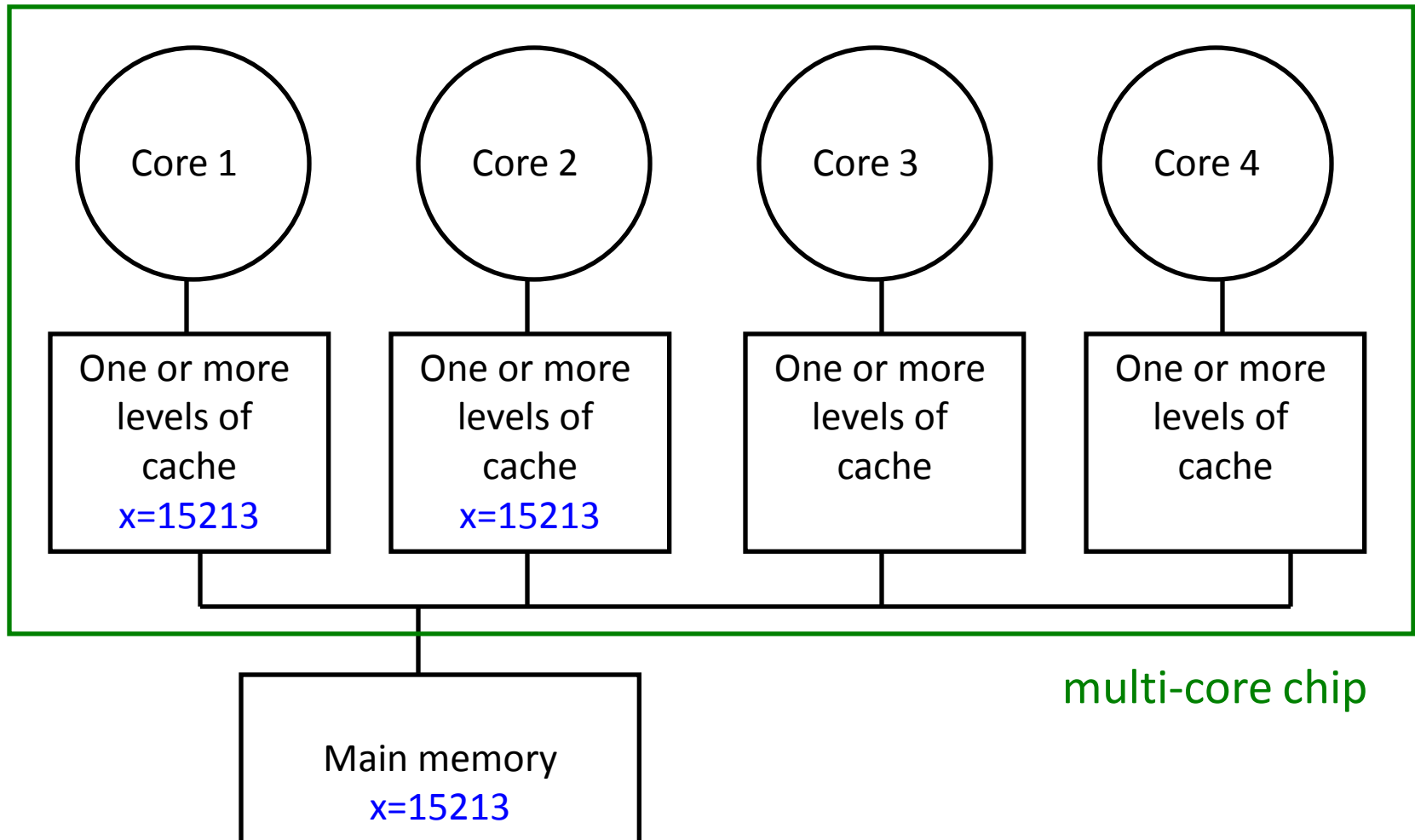
The cache coherence problem

Core 1 reads x



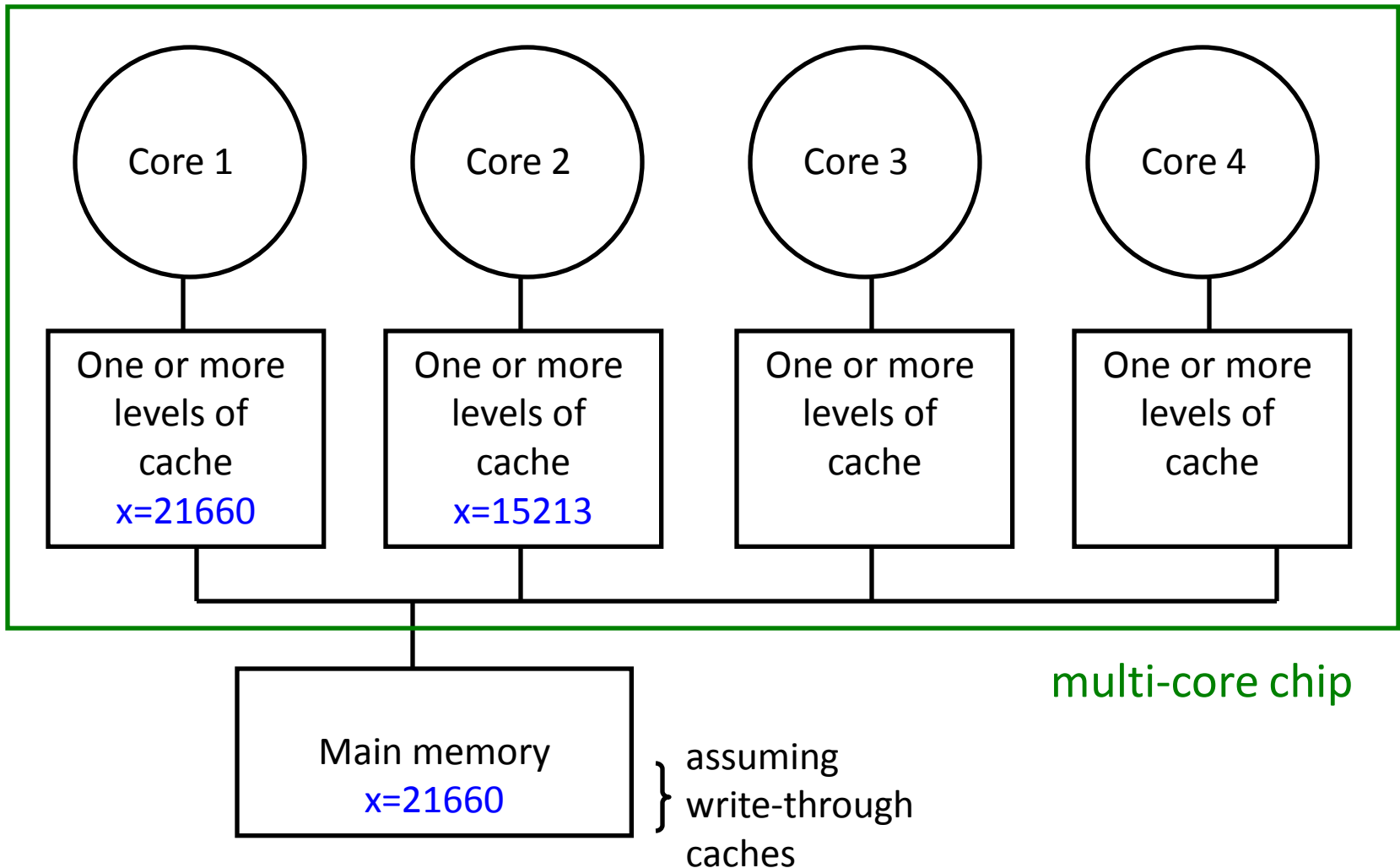
The cache coherence problem

Core 2 reads x



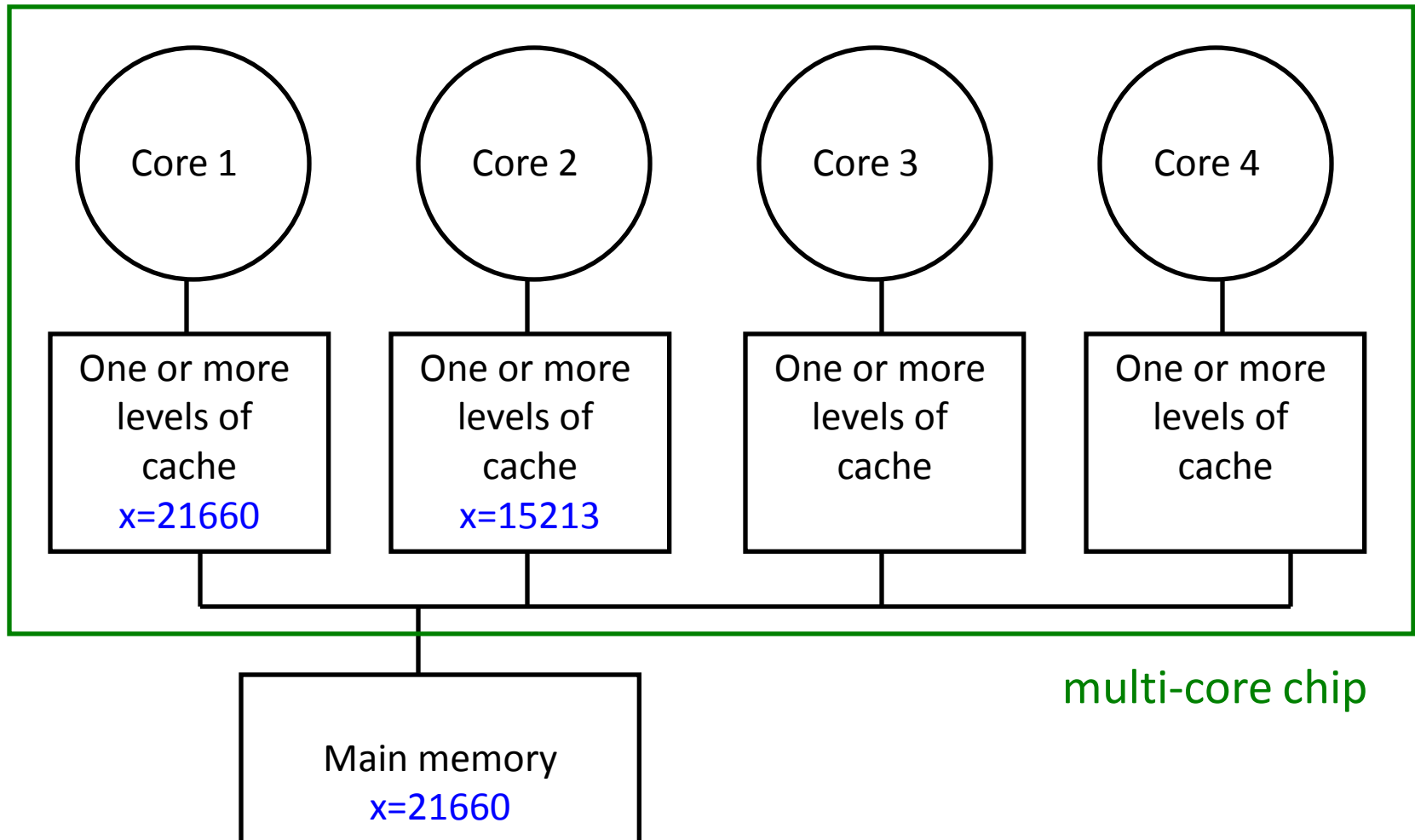
The cache coherence problem

Core 1 writes to x, setting it to 21660



The cache coherence problem

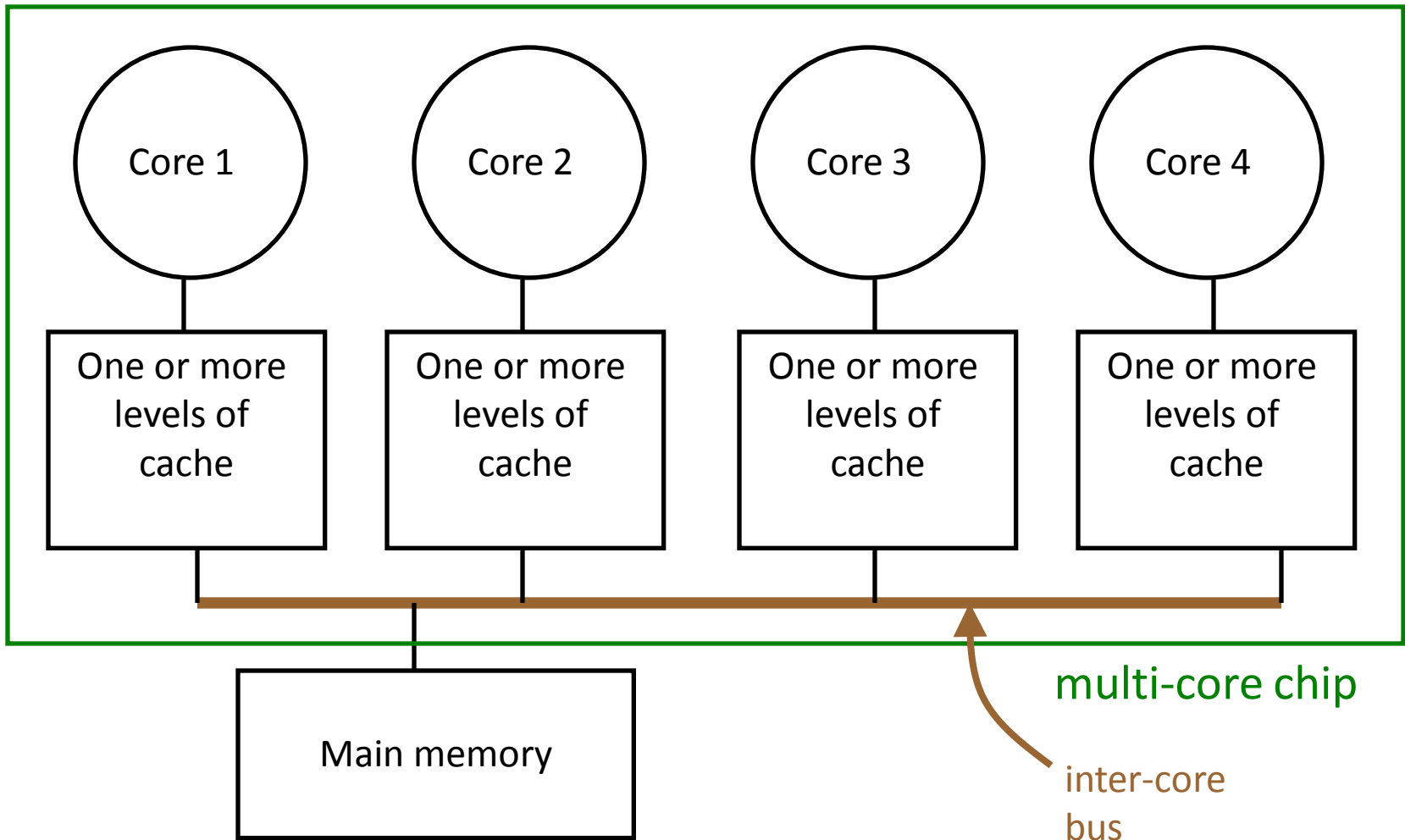
Core 2 attempts to read x... gets a stale copy



Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.
- A simple solution:
invalidation-based protocol with *snooping*

Inter-core bus



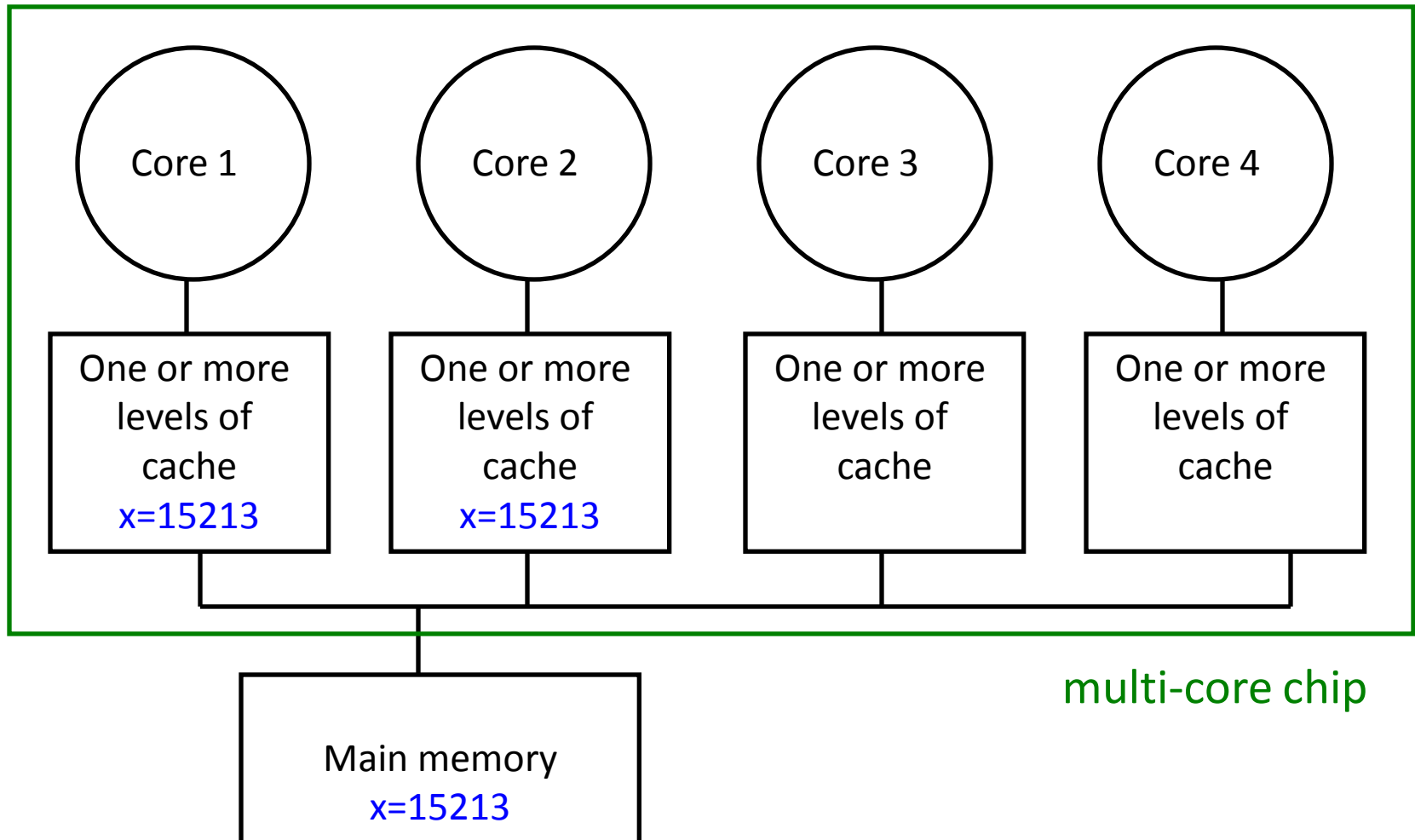
Invalidation protocol with snooping

- Invalidation:
If a core writes to a data item, all other copies of this data item in other caches are *invalidated*
- Snooping:
All cores continuously “snoop” (monitor) the bus connecting the cores.

Snooping is the process where the individual caches monitor address lines for accesses to memory locations that they have cached. When a write operation is observed to a location that a cache has a copy of, the cache controller invalidates its own copy of the snooped memory location.

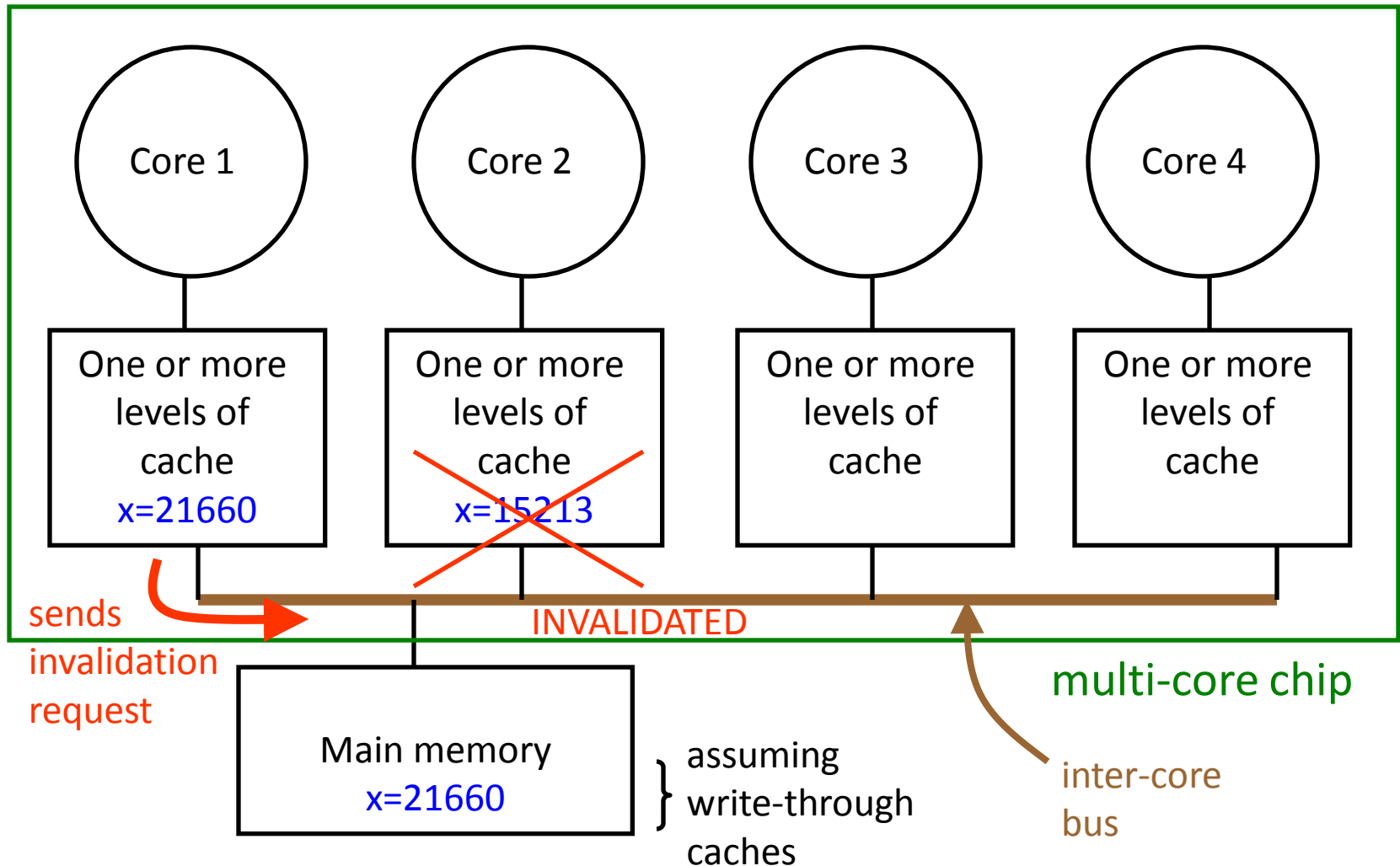
The cache coherence problem

Revisited: Cores 1 and 2 have both read x



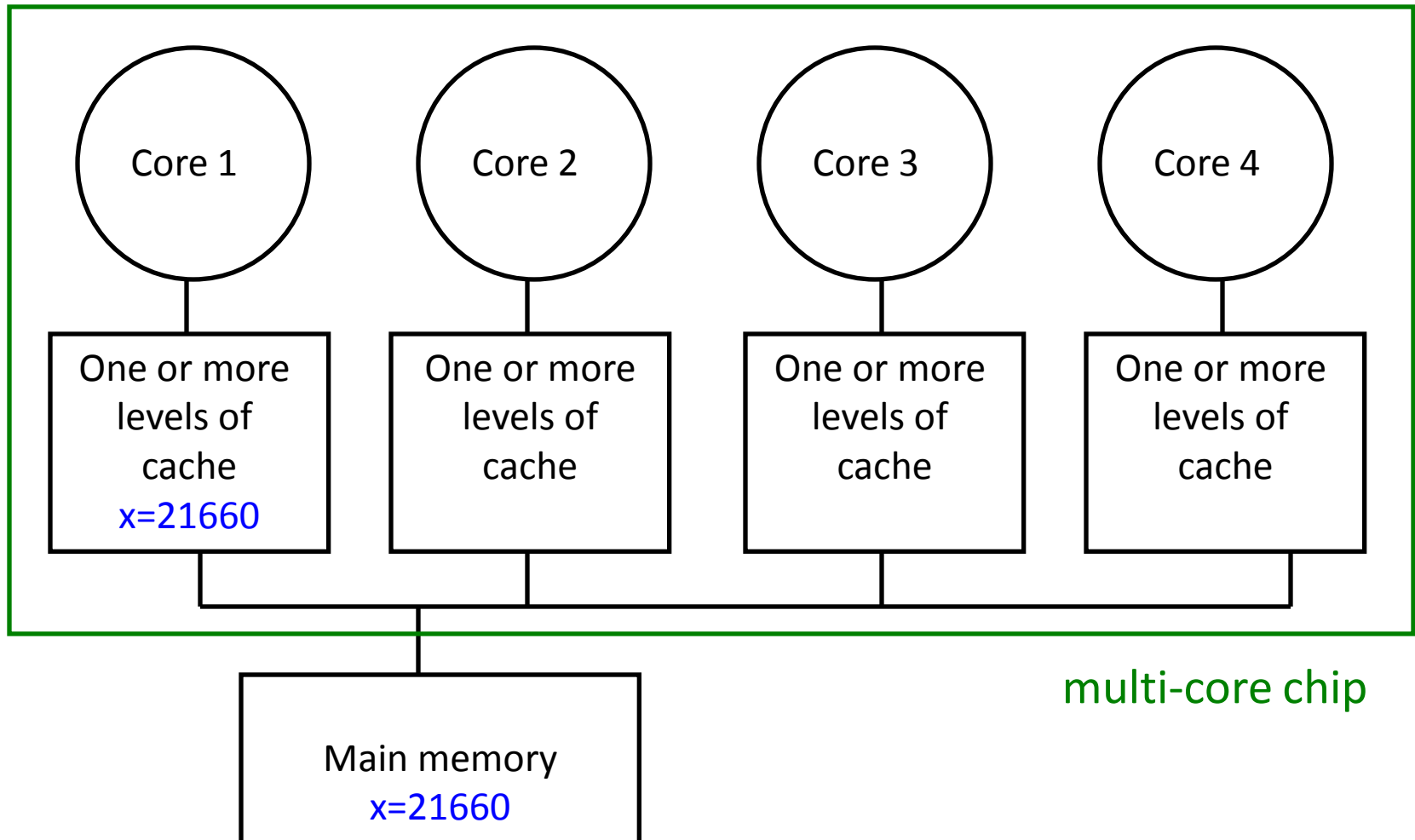
The cache coherence problem

Core 1 writes to x, setting it to 21660



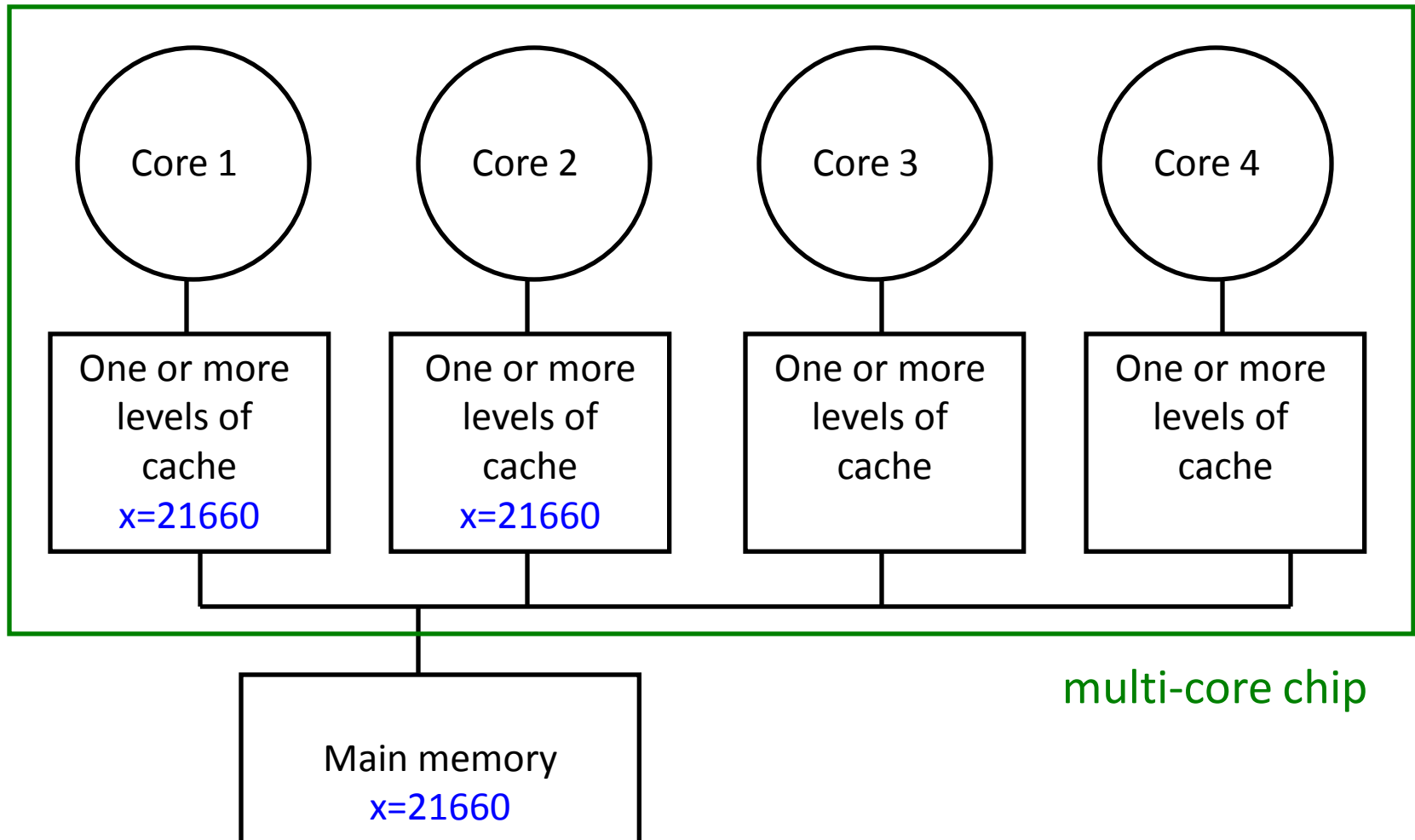
The cache coherence problem

After invalidation:



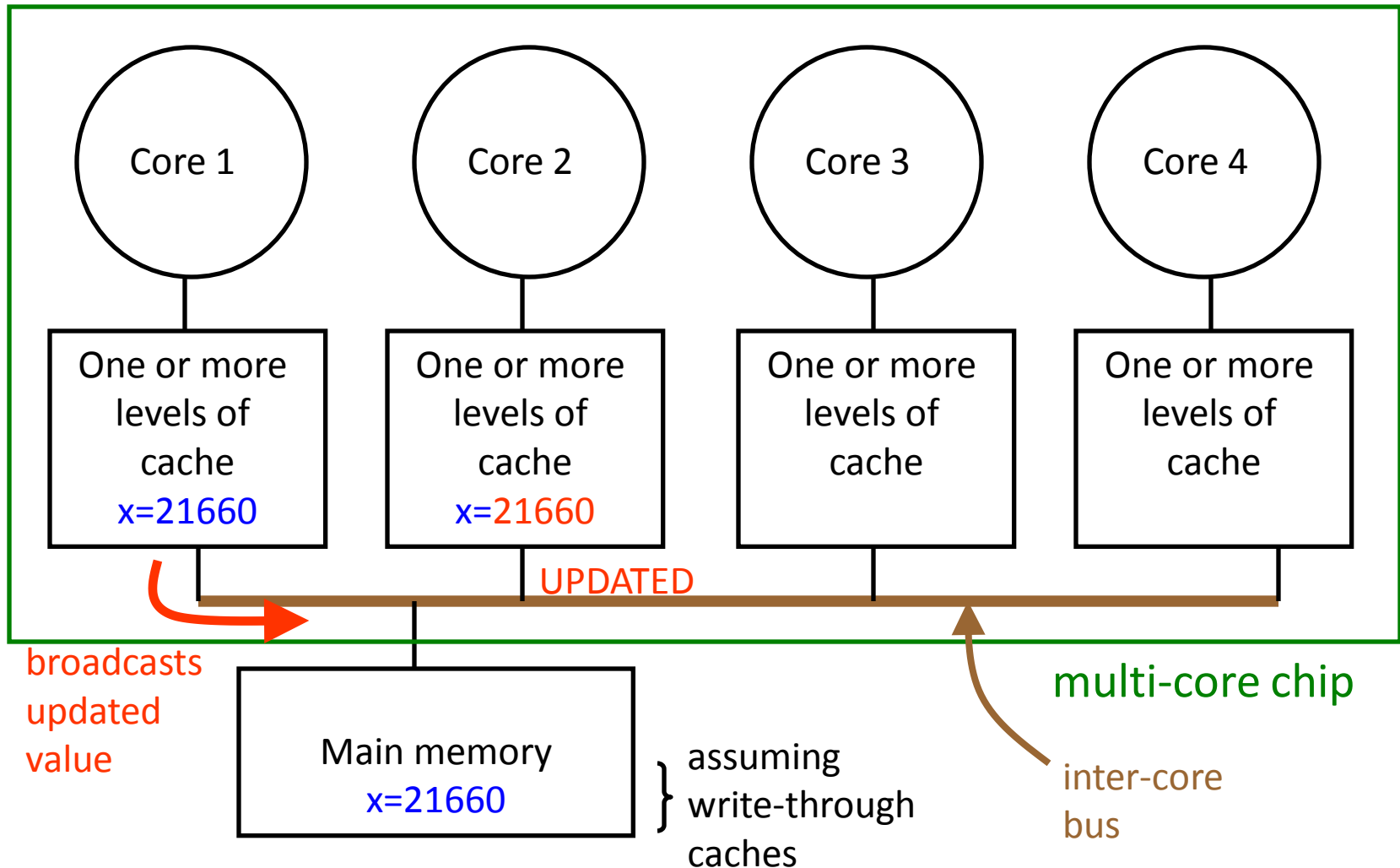
The cache coherence problem

Core 2 reads x. Cache misses, and loads the new copy.



Alternative to invalidate protocol: update protocol

Core 1 writes $x=21660$:



Which do you think is better?
Invalidation or update?

Invalidation vs update

- Multiple writes to the same location
 - invalidation: only the first time
 - update: must broadcast each write
(which includes new variable value)
- Invalidation generally performs better:
it generates less bus traffic

Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

Thread safety very important

- Pre-emptive context switching:
context switch can happen AT ANY TIME
- True concurrency, not just uniprocessor time-slicing
- Concurrency bugs exposed much faster with multi-core

However: Need to use synchronization even if only time-slicing on a uniprocessor


```
int counter=0;
```

```
void thread1() {  
    int temp1=counter;  
    counter = temp1 + 1;  
}
```

```
void thread2() {  
    int temp2=counter;  
    counter = temp2 + 1;  
}
```


Need to use synchronization even if only time-slicing on a uniprocessor

```
temp1=counter;  
counter = temp1 + 1;  
temp2=counter;  
counter = temp2 + 1
```



gives counter=2

```
temp1=counter;  
temp2=counter;  
counter = temp1 + 1;  
counter = temp2 + 1
```



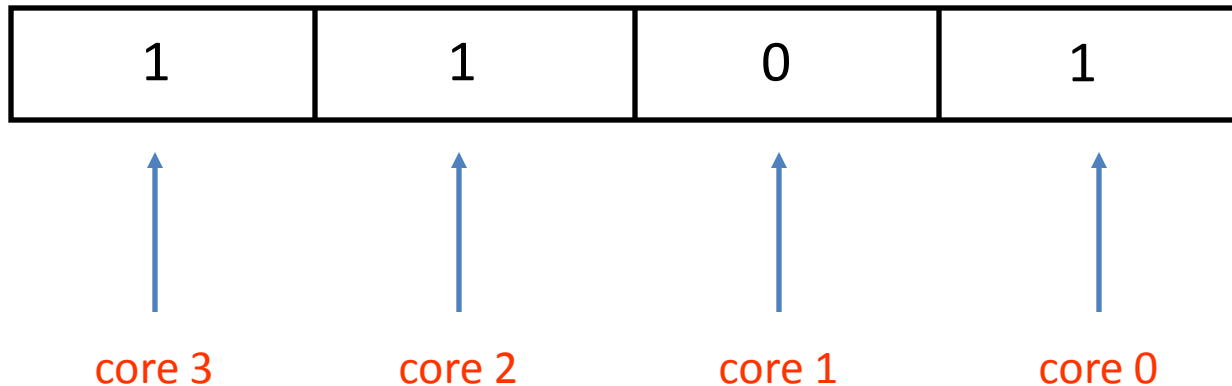
gives counter=1

Assigning threads to the cores

- Each thread/process has an *affinity mask*
- Affinity mask specifies what cores the thread is allowed to run on
- Different threads can have different masks
- Affinities are inherited across `fork()`

Affinity masks are bit vectors

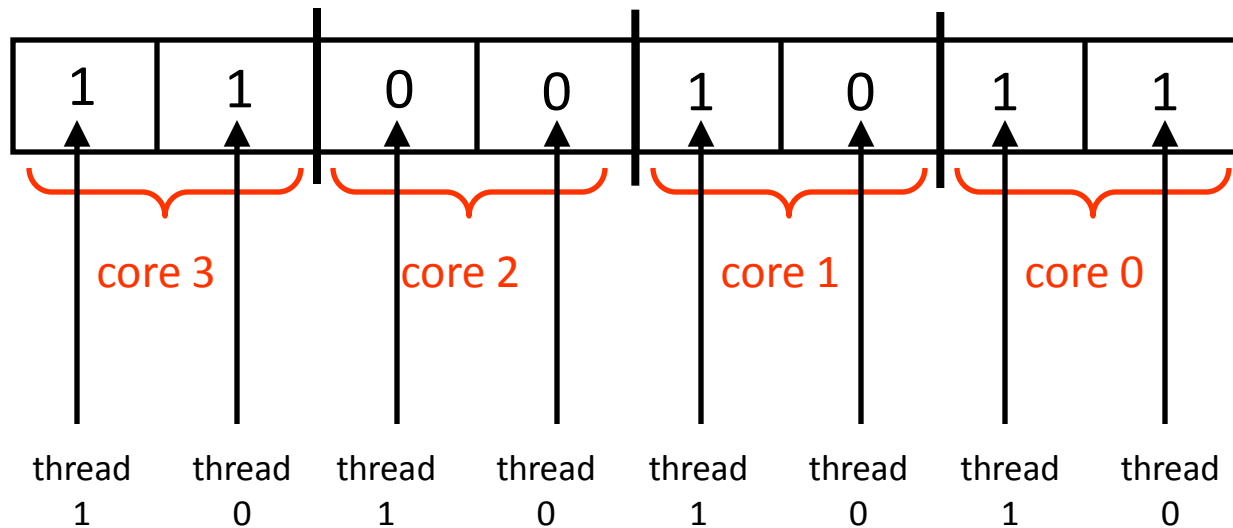
- Example: 4-way multi-core, without SMT



- Process/thread is allowed to run on cores 0,2,3, but not on core 1

Affinity masks when multi-core and SMT combined

- Separate bits for each simultaneous thread
- Example: 4-way multi-core, 2 threads per core



- Core 2 can't run the process
- Core 1 can only use one simultaneous thread

Default Affinities

- Default affinity mask is all 1s:
all threads can run on all processors
- Then, the OS scheduler decides what threads run on what core
- OS scheduler detects skewed workloads, migrating threads to less busy processors

Process migration is costly

- Need to restart the execution pipeline
- Cached data is invalidated
- OS scheduler tries to avoid migration as much as possible:
it tends to keep a thread on the same core
- This is called *soft affinity*

Hard affinities

- The programmer can prescribe her own affinities (hard affinities)
- Rule of thumb: use the default scheduler unless a good reason not to

When to set your own affinities

- Two (or more) threads share data-structures in memory
 - map to same core so that can share cache
- Real-time threads:
Example: a thread running a robot controller:
 - must not be context switched, or else robot can go unstable
 - dedicate an entire core just to this thread



Source: Sensable.com

Kernel scheduler API

```
#include <sched.h>
```

```
int sched_getaffinity(pid_t pid,  
    unsigned int len, unsigned long * mask);
```

Retrieves the current affinity mask of process 'pid' and stores it into space pointed to by 'mask'.

'len' is the system word size: sizeof(unsigned int long)

Kernel scheduler API

```
#include <sched.h>

int sched_setaffinity(pid_t pid,
    unsigned int len, unsigned long * mask);
```

Sets the current affinity mask of process 'pid' to *mask

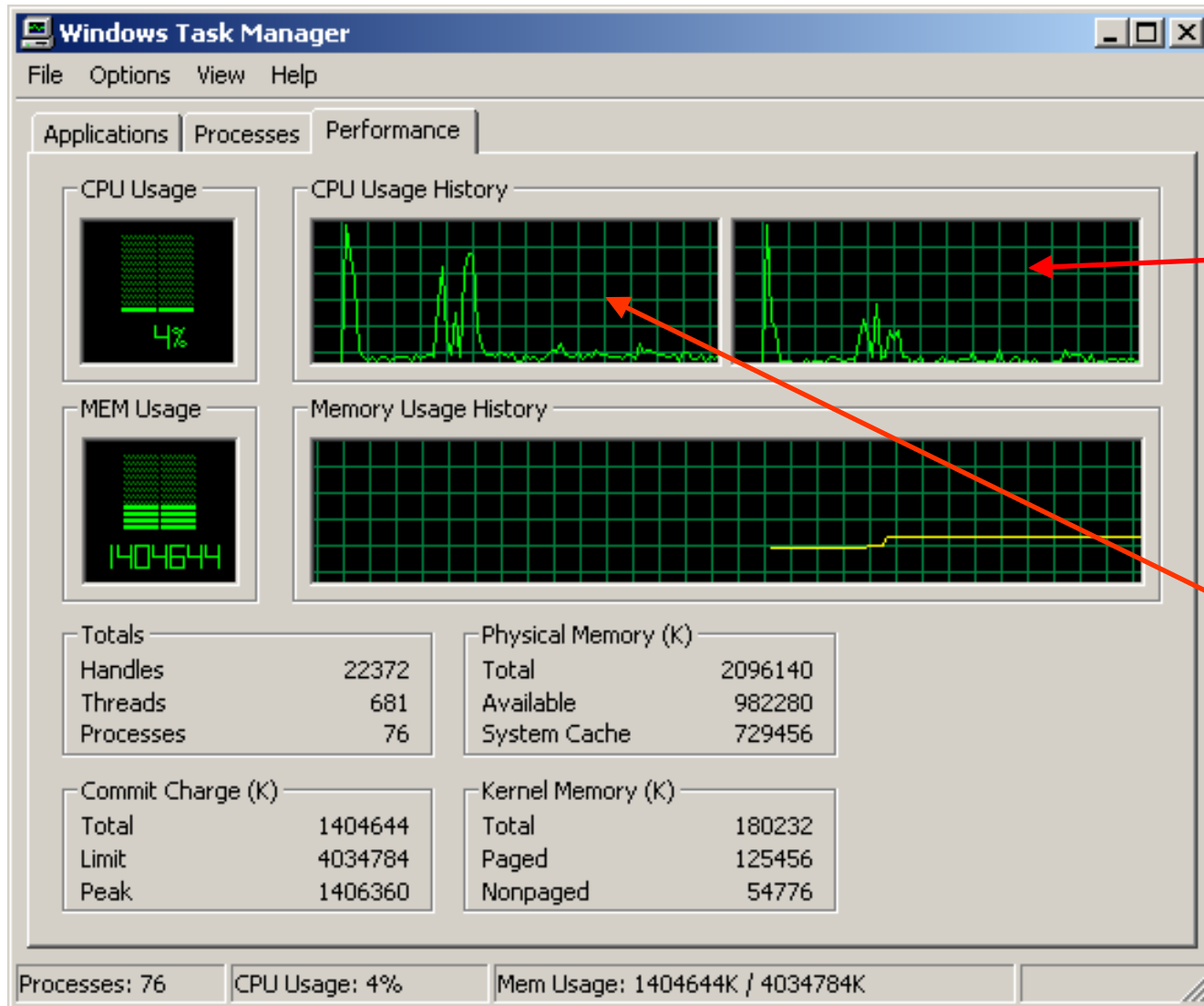
'len' is the system word size: sizeof(unsigned int long)

To query affinity of a running process:

```
$ taskset -p 3935
```

```
pid 3935's current affinity mask: f
```

Windows Task Manager



core 2

core 1

Legal licensing issues

- Will software vendors charge a separate license per each core or only a single license per chip?
- Microsoft, Red Hat Linux, Suse Linux will license their OS per chip, not per core

Attempts to Make Multicore Programming Easy

Attempts to Make Multicore Programming Easy

- **1st idea:** The right computer language would make parallel programming straightforward
 - **Result so far:** Some languages made parallel programming easier, but none has made it as fast, efficient, and flexible as traditional sequential programming.

Attempts to Make Multicore Programming Easy

- **2nd idea:** If you just design the hardware properly, parallel programming would become easy.
 - **Result so far:** no one has yet succeeded!

Attempts to Make Multicore Programming Easy

- **3rd idea:** Write software that will automatically parallelize existing sequential programs.
 - **Result so far:** Success here is inversely proportional to the number of cores!

Mutlicore and Manycore

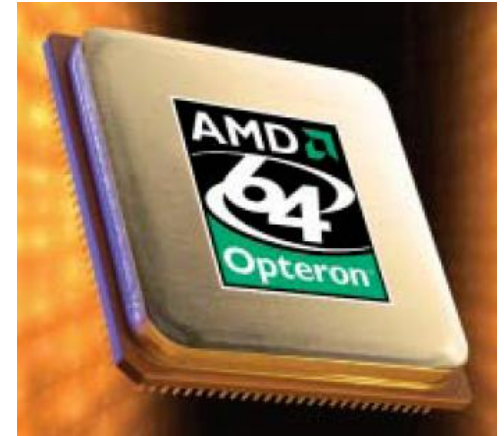
- Dilemma
 - Parallel hardware is ubiquitous
 - Parallel software is not!
 - After over 25 years of research, we are not closer to solving the parallel programming model!

We have arrived at many-core solutions not because of the success of our parallel software but because of our failure to keep increasing CPU frequency.*

Tim Mattson

Conclusions

- Multi-core chips an important new trend in computer architecture
- Several new multi-core chips in design phases
- Parallel programming techniques likely to gain importance



Acknowledgement

- Some slides are adapted from
 - Jernej Barbic
 - Mohamed Zahran