Overview of Architectures and Programming Languages for Parallel Computing

François Broquedis[1]
Frédéric Desprez[2]
Jean-François Méhaut[3]

[1]Grenoble INP
[2]INRIA Grenoble Rhône-Alpes
[3]Université Grenoble Alpes

Laboratoire d'Informatique de Grenoble
CORSE Inria Project-team

https://team.inria.fr/corse/

CORSE : Compiler Optimizations and Runtime Systems

- Static/Dynamic Compiler Analysis
  - Hybrid and extensible byte-code
  - Hybrid compilation, static/trace analysis
  - Instruction scheduling, computing and IO complexities
- Runtime Systems and Interface with Compilers
  - Load balancing with compiler analysis information
  - Memory management (thread/data affinity)
  - Scheduling and load balancing directed by power management
- Interactive debugging, properties monitoring
  - Debugging with programming models (gdb, OpenMP, StarPU,…)
  - Dynamic monitoring of program properties

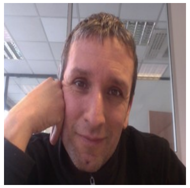
6 CORSE Members + 8 PhDs + 3 Post-Docs

- Fabrice Rastello (DR Inria), CORSE leader
  - Loop transformations for ILP, SSA, IR, Profiling/Feedback, DSL

- Florent Bouchez Tichadou (MCF UGA)
  - Register allocation, backend compiler optimizations, IR

- **François Broquedis** (MCF Grenoble INP)
  - Runtime Systems, OpenMP, Memory Management, FPGA

- **Frédéric Desprez** (DR Inria)
  - Parallel Algorithmic, Scheduling, Data Management



- Ylies Falcone (MCF UGA)
  - Runtime verification, enforcement, Monitoring

- **Jean-François Méhaut** (PR UGA)
  - Runtime Systems, Low Power, Load Balancing, Debugging

Agenda

- Introduction to parallel computing (Jean-François)
  - Architecture Evolutions, Software Stack, Mont-Blanc EU projects

- Multicore Programmning (François)
  - OpenMP, Loop Parallelism, Programming with Tasks

- Heterogeneous (CPU, GPU) Programming (Frédéric)
  - OpenCL, Computing Kernels

Acknowledgements

- Arnaud Legrand (CNRS Grenoble)
- Henri Casanova (Univ. Hawaii)
- Filipo Mantovani, Nikola Rajovic, Alex Ramirez (Barcelona SC)
- Intel, ARM
- CERMACS'2016 (L. Grigori, C. Japhet, P. Moireau, P. Parnaudeau)

General comments

- Feel free to ask questions at any time
- Feel free to provide feedback to improve lectures/slides
- Enjoy !

What is parallel computing ?

- **Parallel Computing** : using multiple processors/cores in parallel to solve problems more quickly than with a single processor/core

- **Examples of parallel machines**
  - A **Chip Multi-processor (CMP)** contains multiple processors (called cores) on a single chip
  - A **Shared memory Multiprocessor** (SMP) by connecting multiple processors to a single memory system
  - A **Cluster Computer** that contains multiple Pcs combined together with a high speed network
  - A **Grid** is a cluster of networked, loosely-coupled computers acting to perform very large tasks

- **Concurrent execution** come from desire for **performance**

Parallel Computing

- We want to run **faster** by using more processors/cores
- **Goal**
  - Write applications that can leverage the performance of multicore systems
  - Write applications that run faster when vendors (Intel/IBM/AMD/ARM) comes out with a new chip that has more cores
    - Application must be parallel
    - Application must be able to use an arbitrary number of cores (within limits)
    - Application should be able to leverage several various accelerators (SIMD extensions, GPUs,...)

Key Challenges : The **3 P's**

- **Performance** challenge
  - How to scale from 1 to 1000 cores – The number of cores is the new Megahertz

- **Power** efficiency
  - Performance per watt is the new metric – systems are often constrained by power and cooling

- **Programming** challenges
  - How to provide a converged solution in a standard programming environment for multi and many core processors

Architecture

CMOS Evolution – Practice

- Vendors have pushed thread performance faster than implied by scaling theory
  - Extra transistors have been used to enhanced microarchitectures
    - Deeper pipelines, branch prediction, Instruction Level Parallelism (ILP)…
    - Larger caches
    - Increased IPC (Instruction per Cycle)
  - Voltage has scaled down more slowly
  - Clock rate has scaled up faster
  - Energy consumption has increased
  - Leakage power has become significant

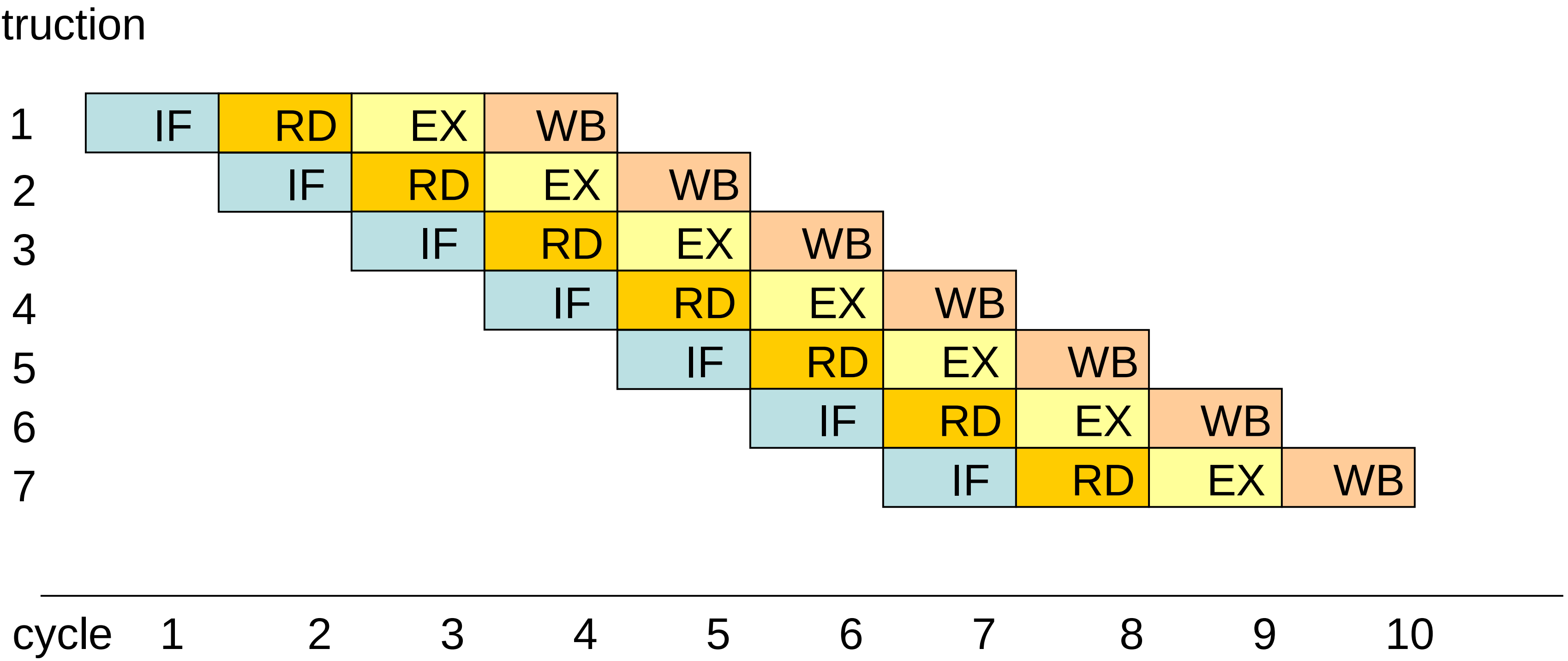
Pipelining

- Overlapping execution of multiple instructions
  - 1 instruction per cycle

- Sub-divide instruction into multiple stages ; Processor handles different stages of adjacent instruction simultaneously

- Suppose 4 stages in an instruction
  - Instruction fetch and decode (IF)
  - Read data (RD)
  - Execution (EX)
  - Write-back results (WB)

Instruction Pipelining



instruction \ cycle	1	2	3	4	5	6	7	8	9	10
1	IF	RD	EX	WB						
2		IF	RD	EX	WB					
3			IF	RD	EX	WB				
4				IF	RD	EX	WB			
5					IF	RD	EX	WB		
6						IF	RD	EX	WB	
7							IF	RD	EX	WB

- Depth of pipeline : number of stages in an instruction

- With pipeline, 7 instructions takes 10 cycles

- Without pipeline, 7 instructions takes 28 cycles

Inhibitors of Pipelining

- Dependencies between instructions interrupts pipelining, degrading performance
  - Control dependance
    - Branching after a conditional branch loop, condition)
      - Avoid excessive branching...
  - Data dependance
    - When an instruction depends on data from previous instruction

```
x = 3*j;
y = x+5.0; // depends on previous instruction
```

-

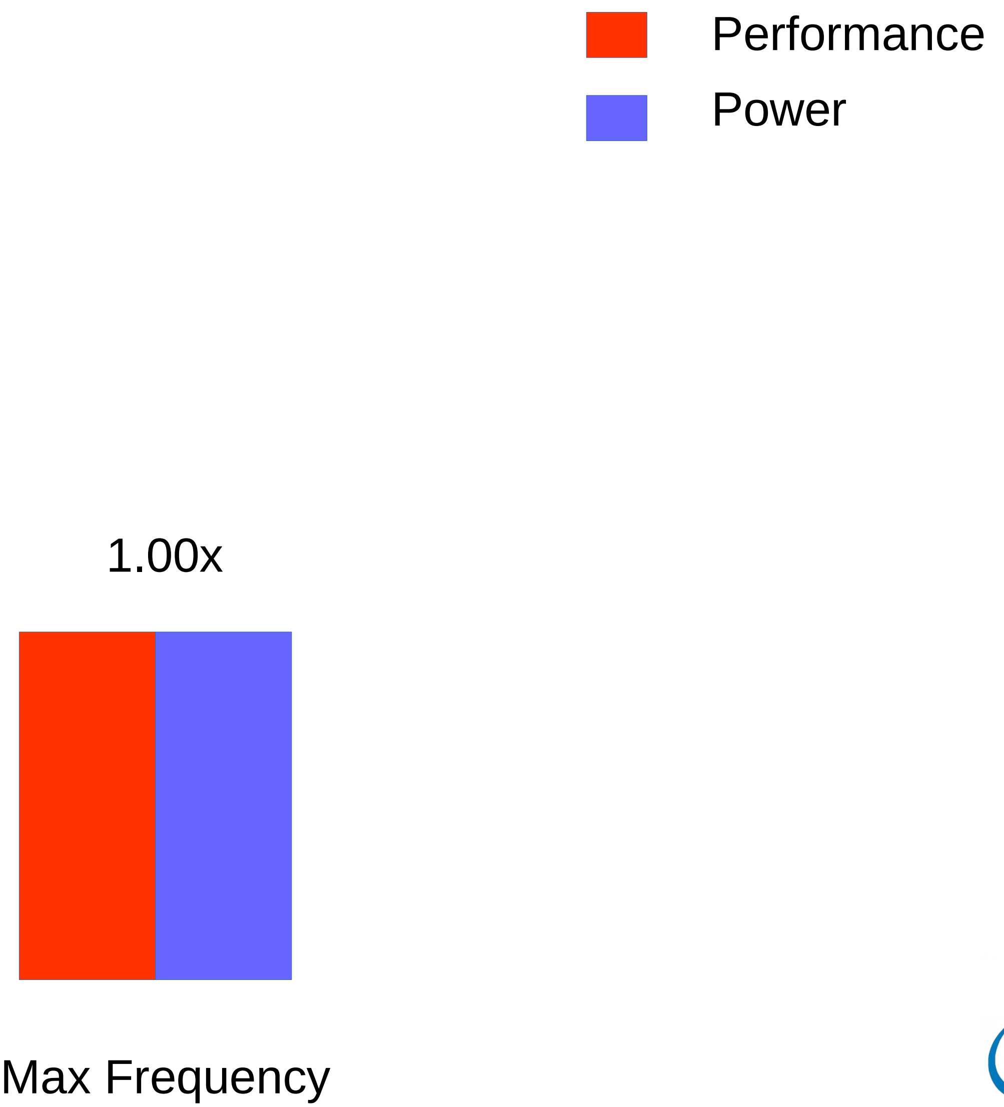
Single Thread Performance Wall

- Cannot increase clock rate
  - Would increase power dissipation
- Cannot increase IPC
  - Cache misses have become relatively more expansive
- Need more parallelism in executing code
  - SIMD Single instruction, Multiple Data – e.g vector operations
  - MIMD Multiple Instruction, Multiple Data – e.g. multithreading
- **Cannot extract parallelism without user support**
- **Increased transistors count is and will be used to be parallel systems on a chip that will run explicitely parallel code**

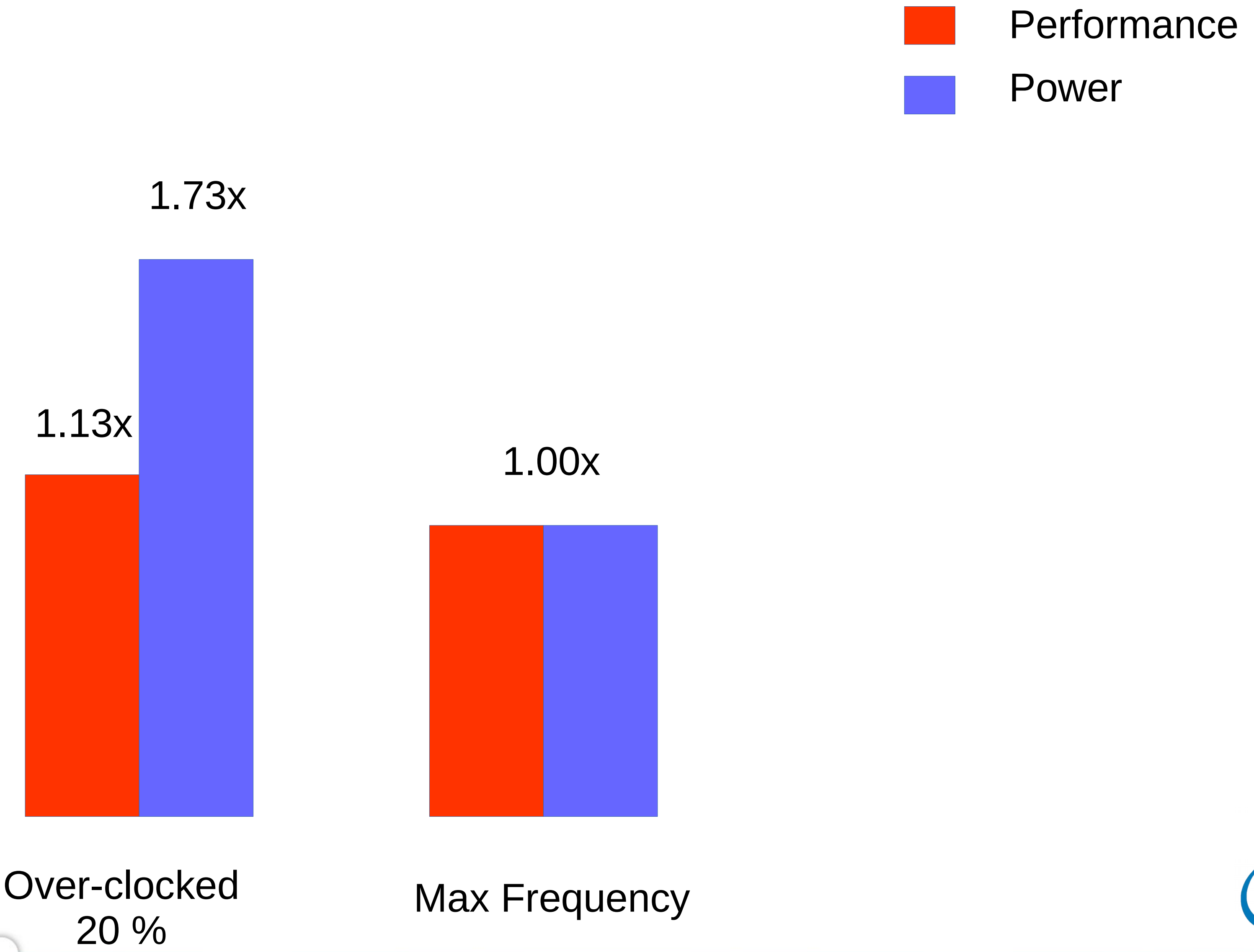
Parallel Hardware

- Multicore
  - Multiple inedependent processors per chip, shared memory
  - Shared memory is possible bottleneck
- Simultaneous Multi-Threading (SMT), Hyper-Threading
  - multiple instruction streams executing on one core (multipe register files sharing same instruction units)
  - better use of core resources
  - slower execution, per stream
- Vector instructions
  - e.g. Intel SSE (processing 128 bits in one instruction)
- Accelerators
  - e.g. GPUs

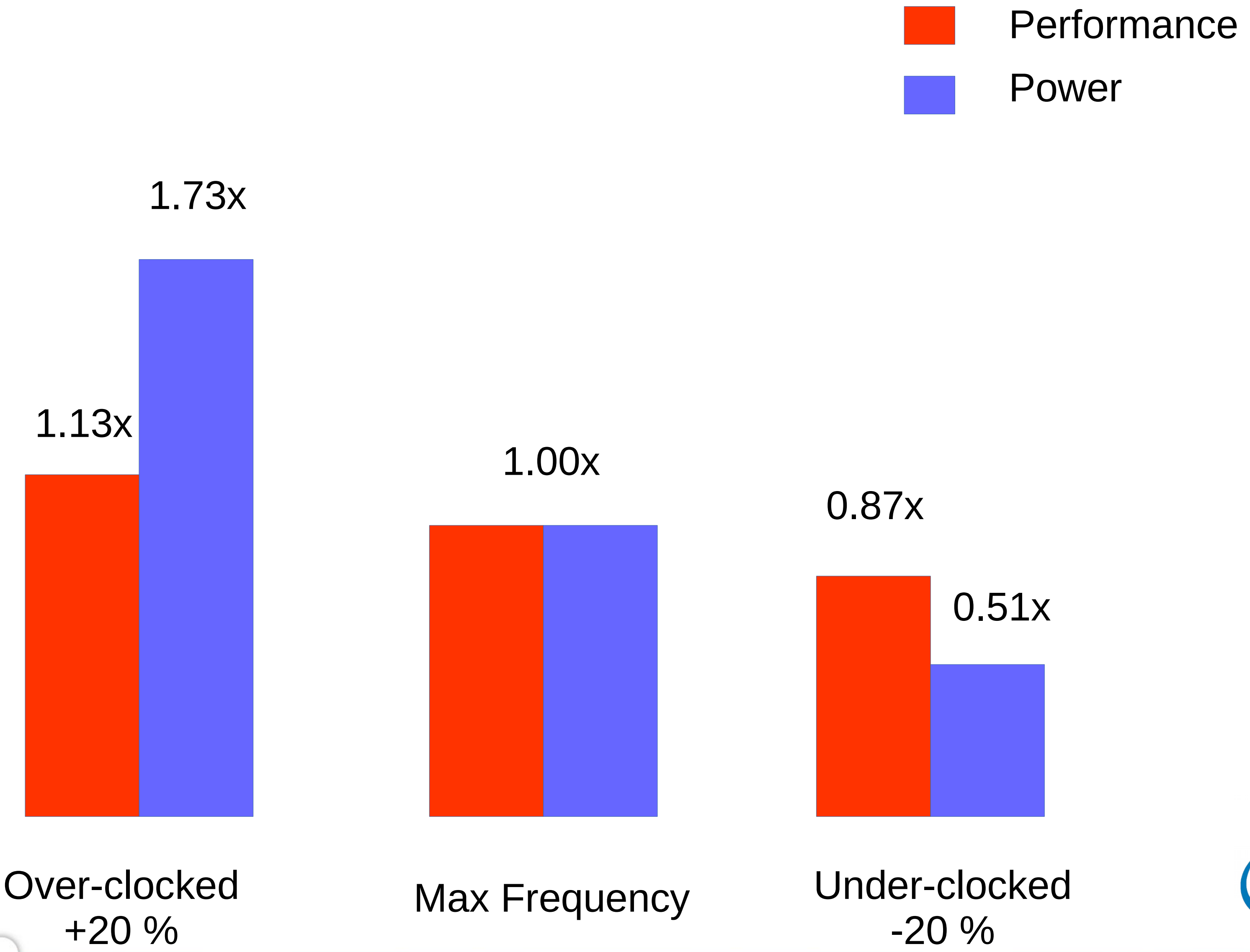
Why multi-core ?



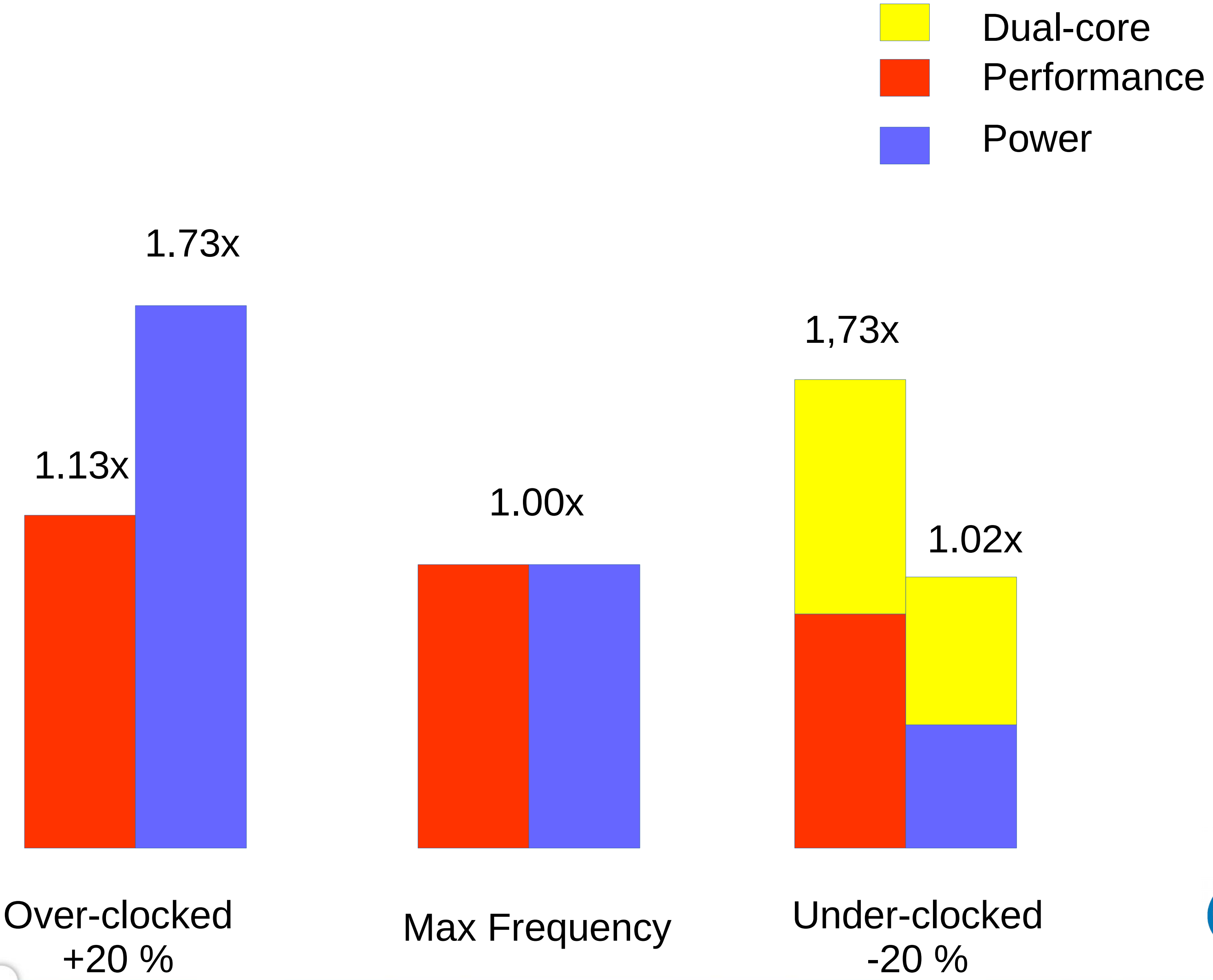
Over-clocking



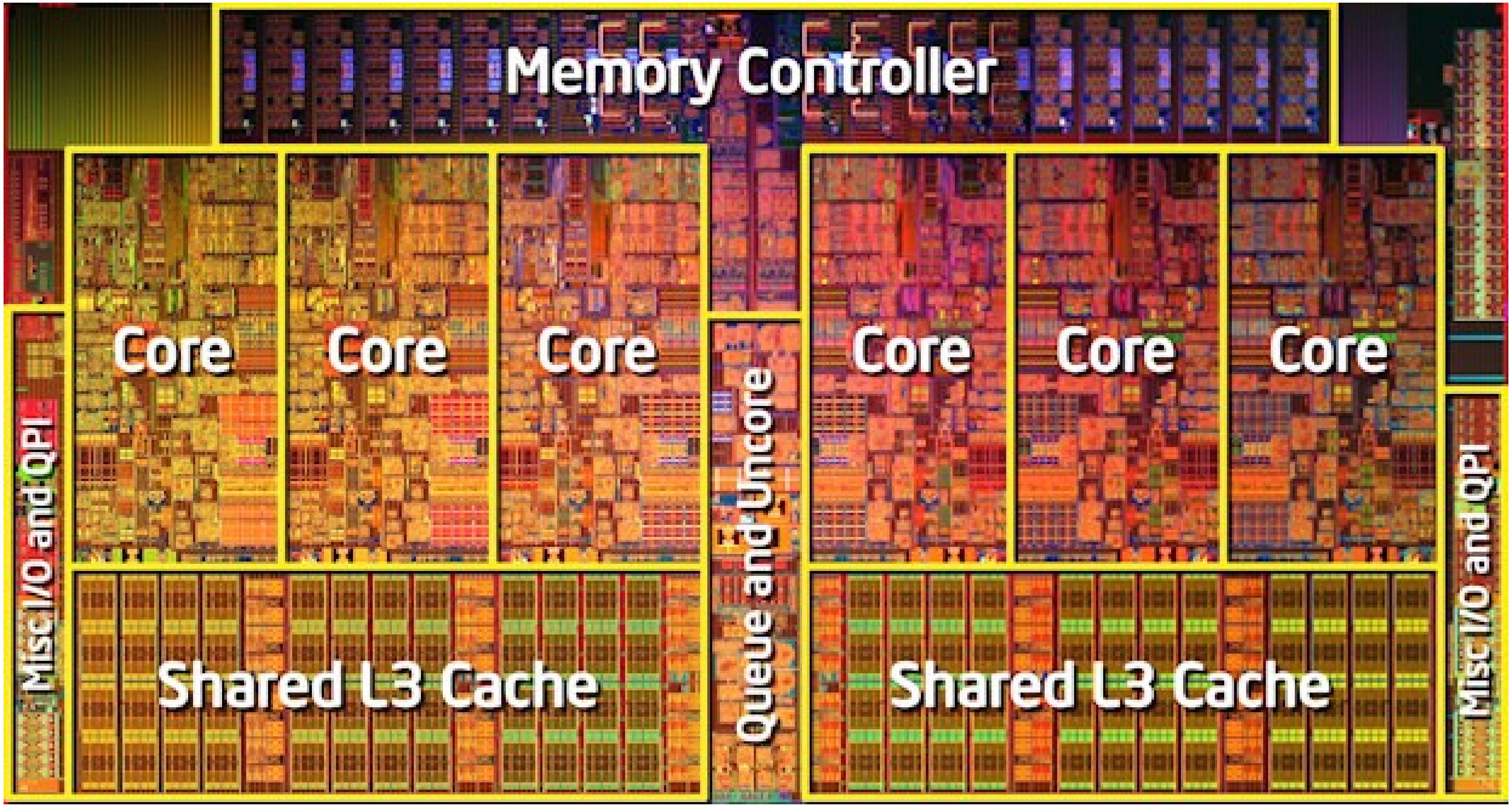
Under-clocking



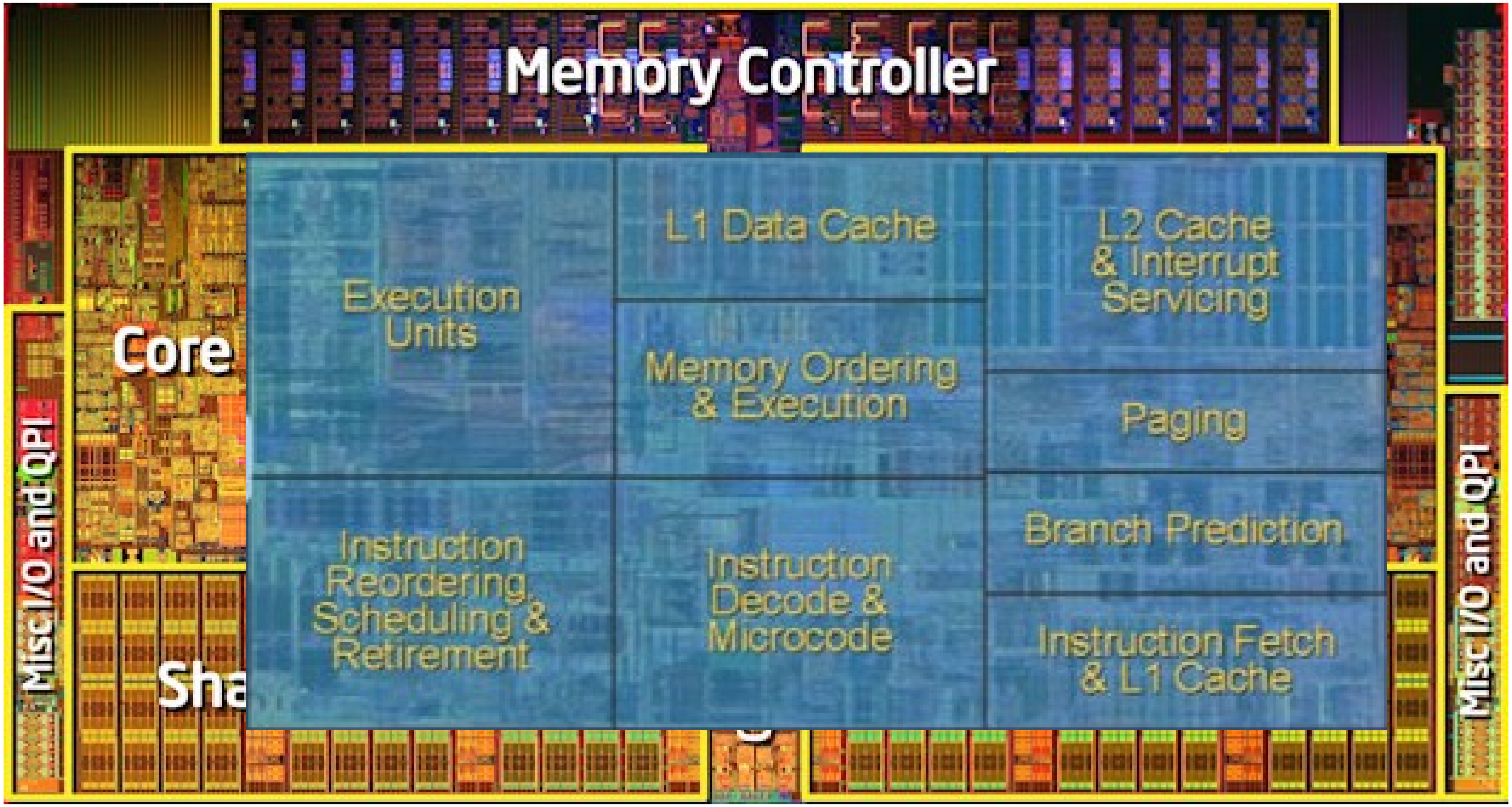
Multi-core : Energy-Efficient Performance



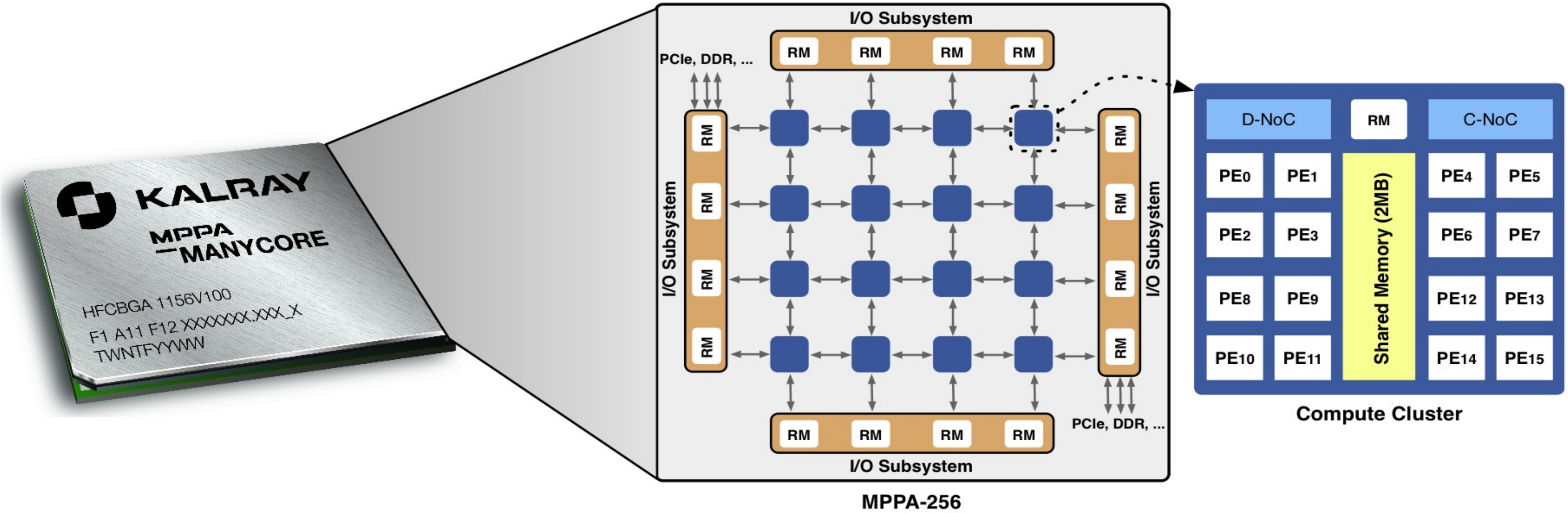
Inside an Intel Nehalem Hex-Core



Inside an Intel Nehalem Hex-Core


Memory Controller
Core
Sha
Misc I/O and QPI
Misc I/O and QPI
Execution Units
L1 Data Cache
L2 Cache & Interrupt Servicing
Memory Ordering & Execution
Paging
Branch Prediction
Instruction Reordering, Scheduling & Retirement
Instruction Decode & Microcode
Instruction Fetch & L1 Cache

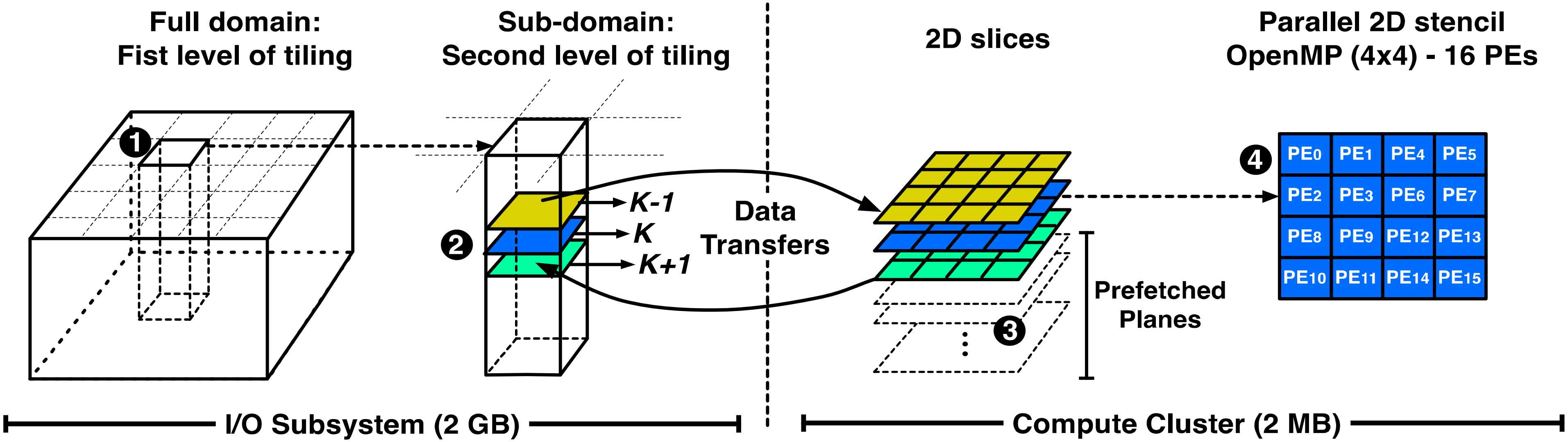
Kalray/MPPA-256 architecture



- **256 cores (PEs) @ 400 MHz** : 16 clusters, 16 PEs per cluster
- PEs **share 2MB of memory**
- **Absence of cache coherence** protocol inside the cluster
- **Network-on-Chip (NoC)** : communication between clusters
- **4 I/O subsystems** : 2 connected to external memory

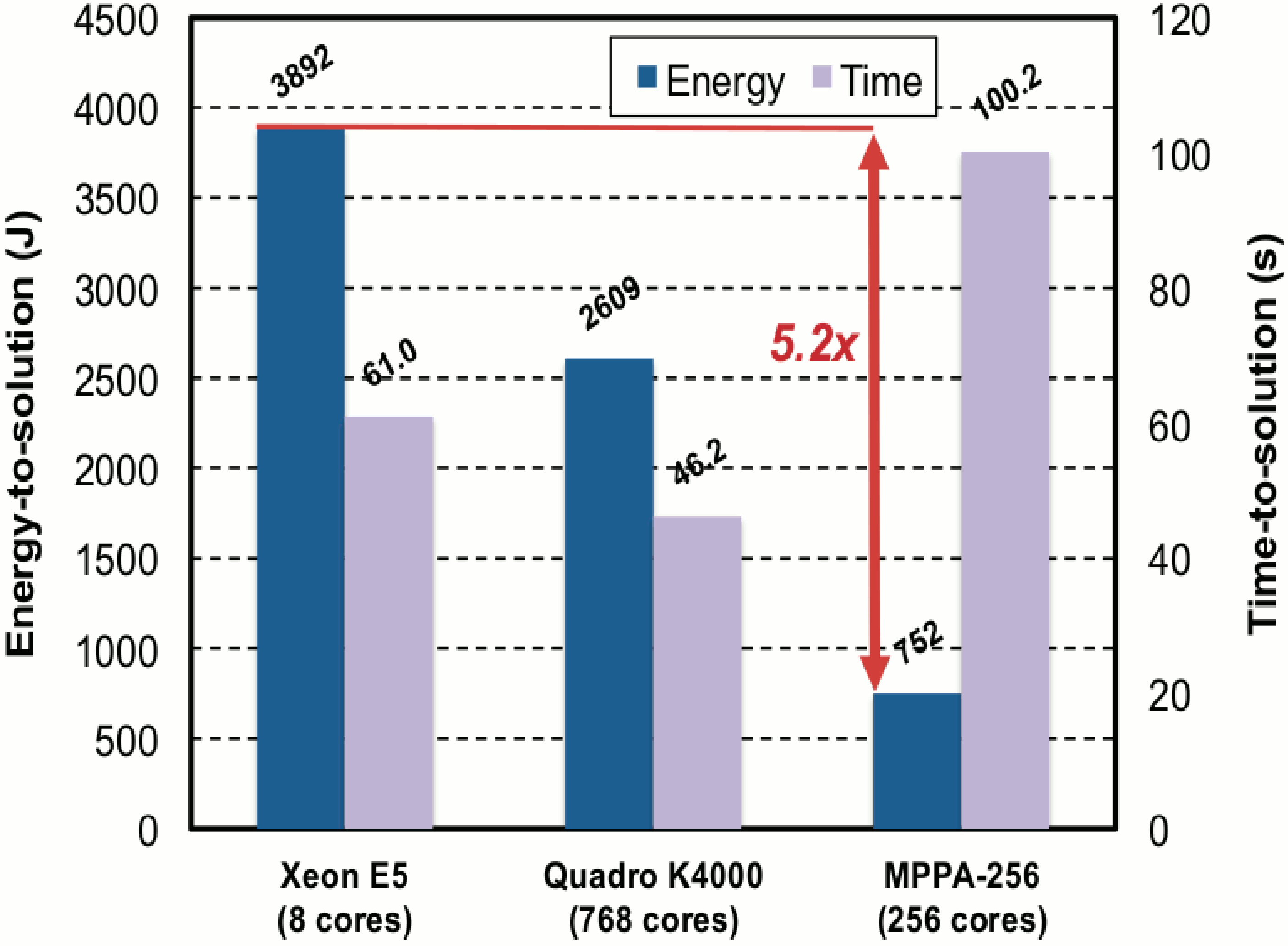
Overview of Parallel Execution of Seismic Application

- **Two-level tiling scheme** to exploit the memory hierarchy of MPPA-256

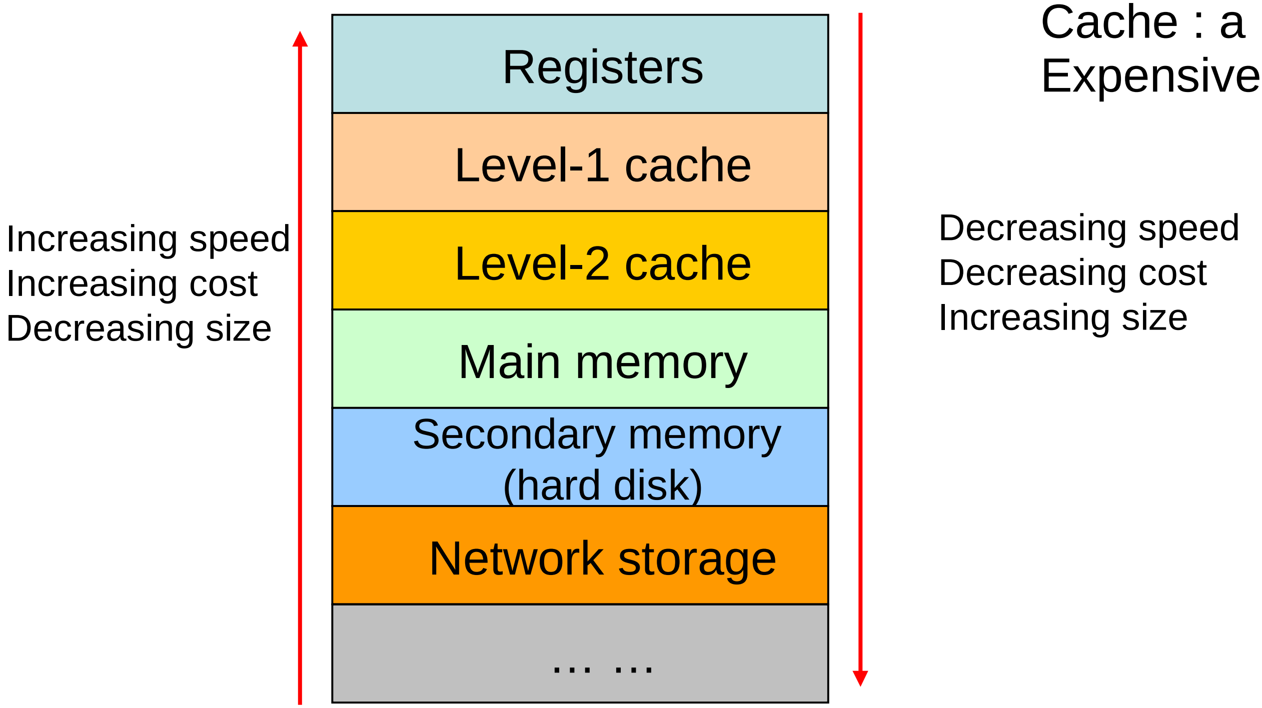


- **Absence of cache coherency inside a cluster**
  - Application programming is difficult !

Results - Input problem size of 2 GB



Memory Hierarchy



Cache : a piece of fast memory
Expensive (transistors, $)

- Performance of different levels can be very different
  - e.g. access time L1 cache can be 5 cycle, L2 can be 12 cycles while main memory can be 170 cycles and secondary memory can be orders of magnitude slower

How Memory Hierarchy Works ?

- (RISC processor) CPU works only data in registers
  - If data is not in register, request data from memory and to register

- Data in register come only from and go only to L1 cache
  - When CPU requests data from memory, L1 cache takes over ;
  - If data is in L1 cache (cache hit), return data to CPU immediately ; end memory access
  - If data is not in L1 cache, cache miss

How Memory Hierarchy Works ?

- If data is not in L1 cache, L1 cache forwards memory request down to L2 cache.
  - If L2 cache has the data (cache hit), it returns the data to L1 cache, which in returns data to CPU, end memory access
  - If L2 cache does not have the data (cache miss)
- If data is not in L2 cache, L2 cache forwards memory request down to memory
  - If data is in memory, main memory passes data to L2 cache, which then passes data to L1 cache, which then passes data to CPU.
  - If data is not in memory,…
- Then request is passed to OS to read data from the secondary storage (disk), which then is passed to memory, L2 cache, L1 cache, register...
  - If data is not in register, request data from memory and to register

Cache Effect on Performance

- Cache miss → degrading performance
  - When there is a cache miss, CPU is idle waiting for another cache line to be brought from lower level of memory hierarchy

- Increasing cache hit rate → higher performance
  - Efficiency directly related to reuse of data in cache

- To increase cache hit rate, access memory sequentially ; avoid strides, random access and indirect addressing in programming

```
for(i=0;i<100;i++)
  y[i] = 2*x[i];
```

sequential access

```
for(i=0;i<100;i=i+4)
  y[i] = 2*x[i];
```

strides

```
for(i=0;i<100;i++)
  y[i] = 2*x[index[i]];
```

Indirect addressing

Private vs shared caches

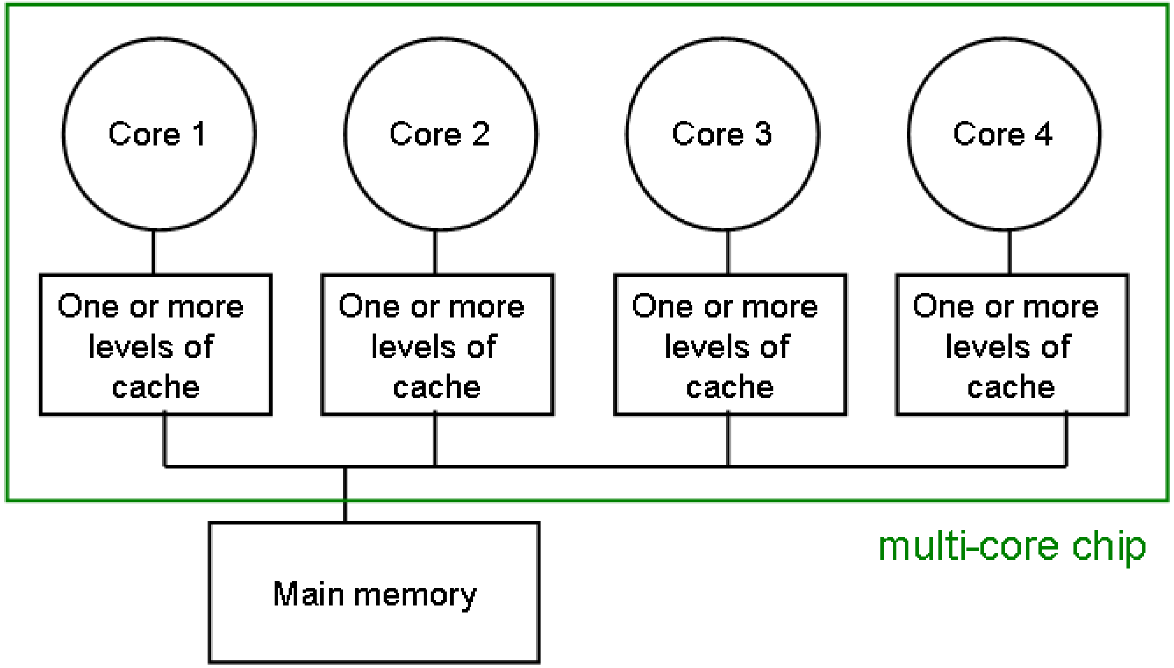
- Advantage of private :
  - They are closer to core, so faster access
  - Reduces contention
  - slower execution, per stream

- Advantages of shared :
  - Threads on different core can share the same cache date
  - More cache space data if a single (or a few) high performance thread run on the system

The cache coherence problem

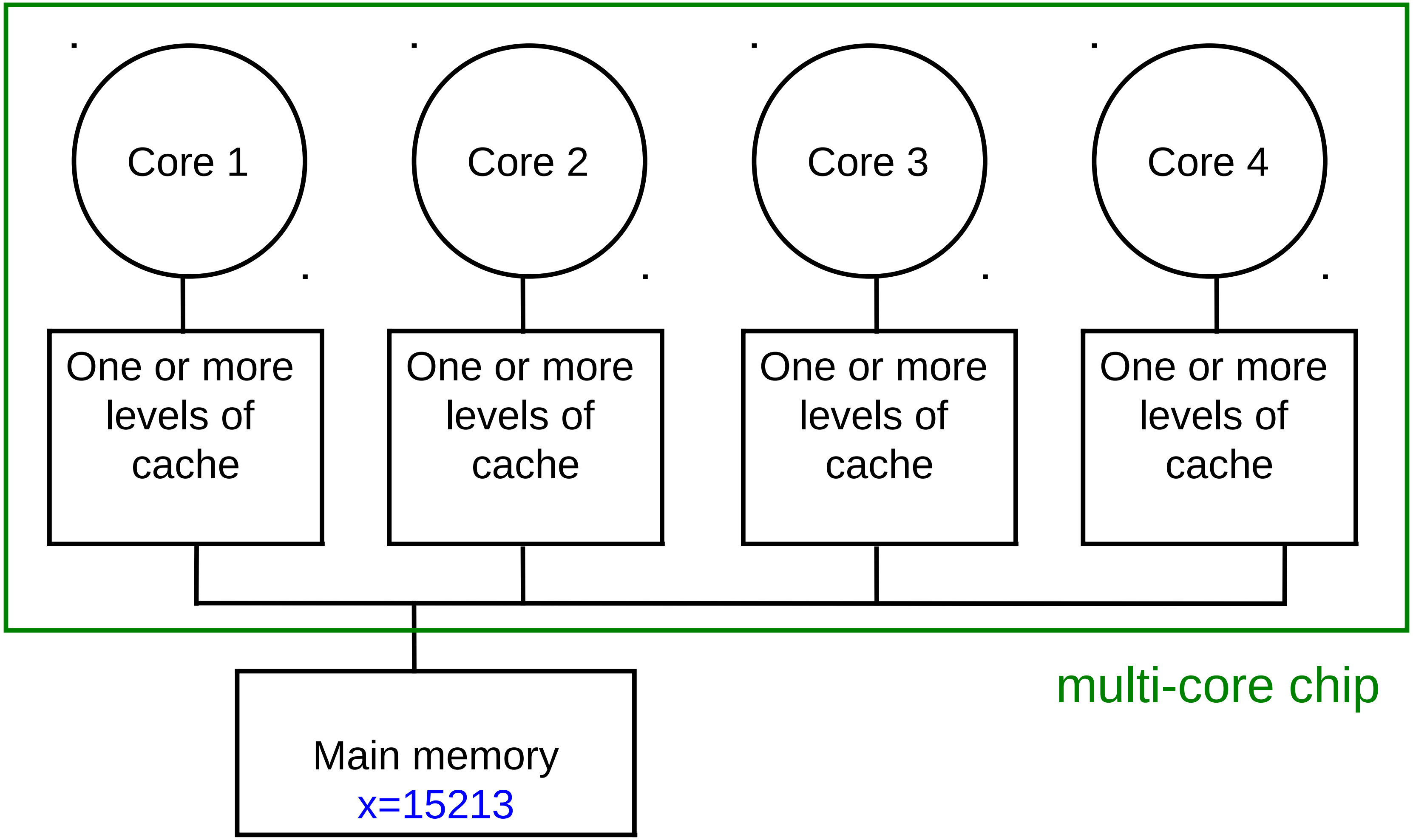
- Since we have private cache :
  - How to keep the data consisten across the caches ?

- Each core should perceive the memory as a monolithic array, shared by all the cores


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
multi-core chip

The cache coherence problem

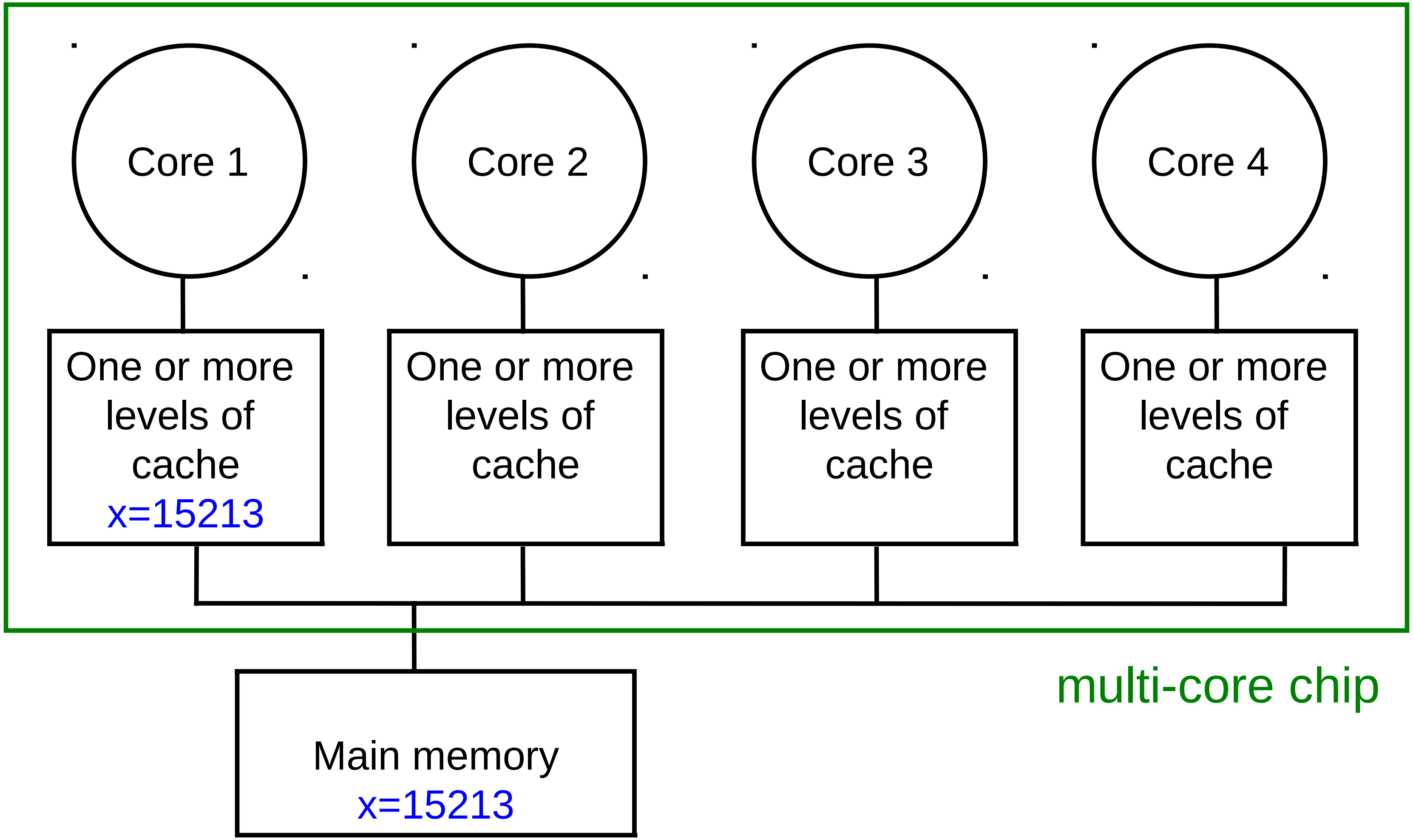
- Suppose variable x initially contained 15213


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
x=15213

multi-core chip

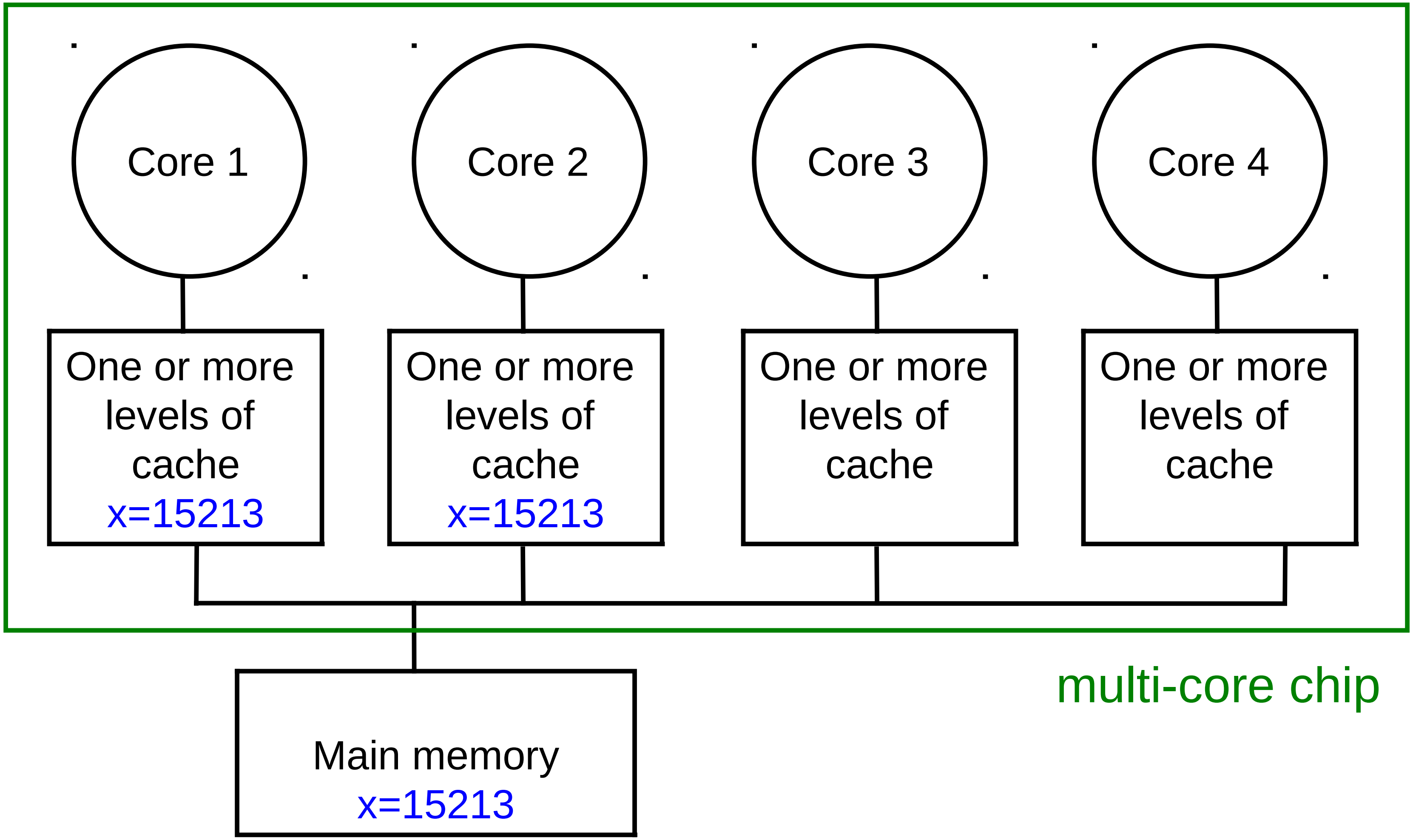
The cache coherence problem

- Core 1 reads x


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
x=15213
multi-core chip

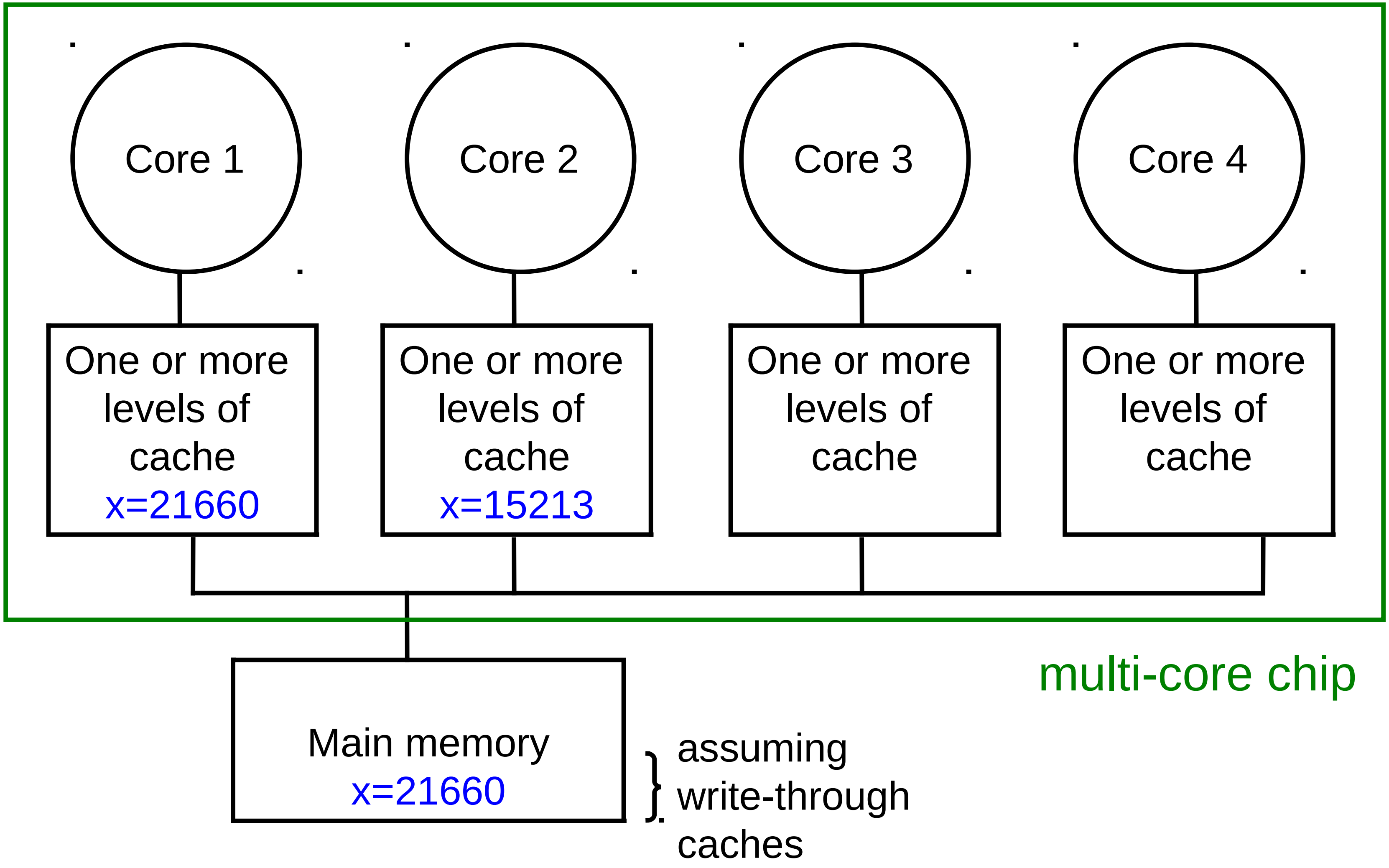
The cache coherence problem

- Core 2 reads x


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=15213
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
Main memory
x=15213
multi-core chip

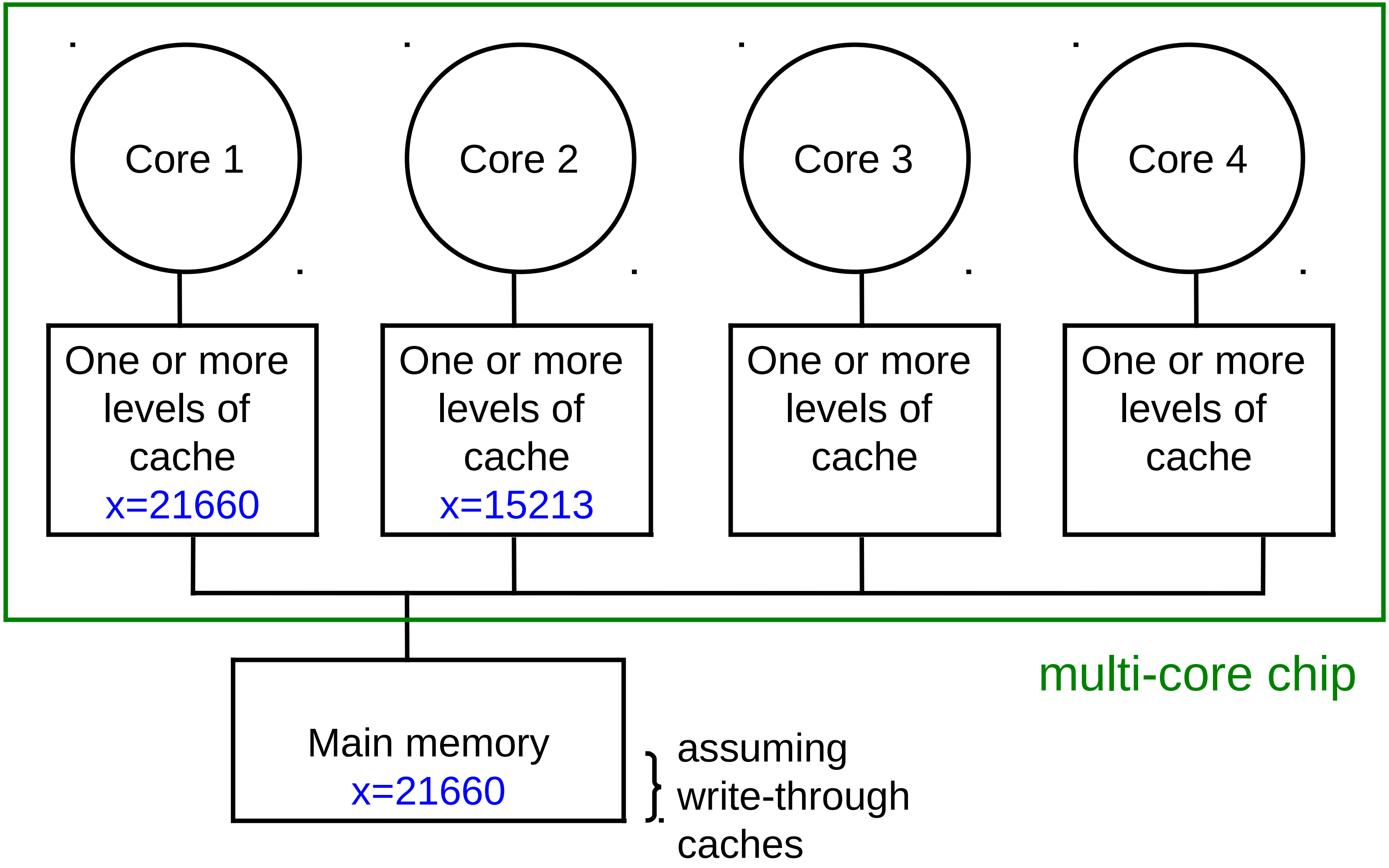
The cache coherence problem

- Core 1 writes to x, setting it to 21660


Core 1
Core 2
Core 3
Core 4
One or more levels of cache x=21660
One or more levels of cache x=15213
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory x=21660
assuming write-through caches

The cache coherence problem

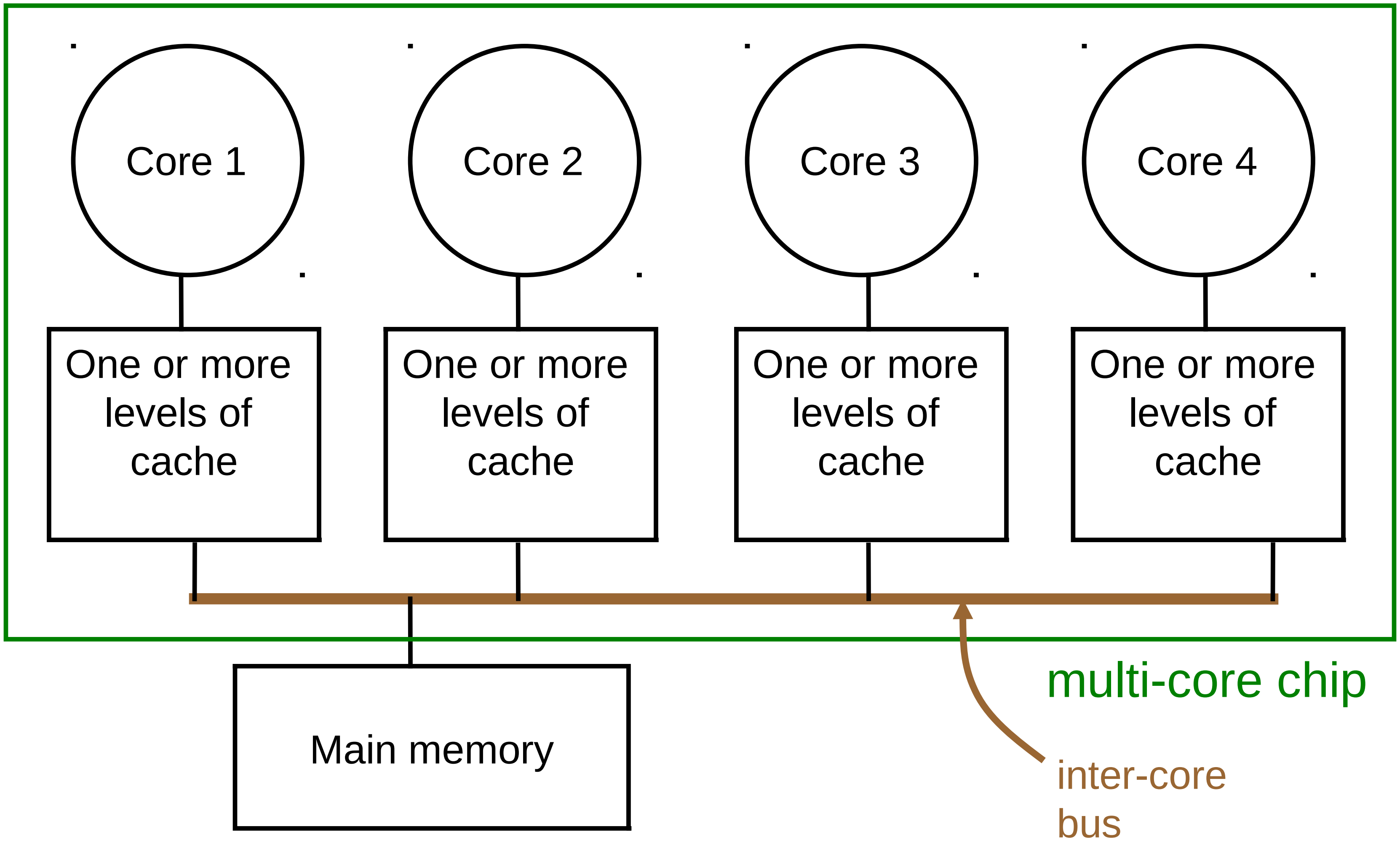
- Core 2 attempts to read x, … get a stale copy


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory
x=21660
assuming write-through caches

Solutions for cache coherence

- This 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


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
multi-core chip
inter-core bus

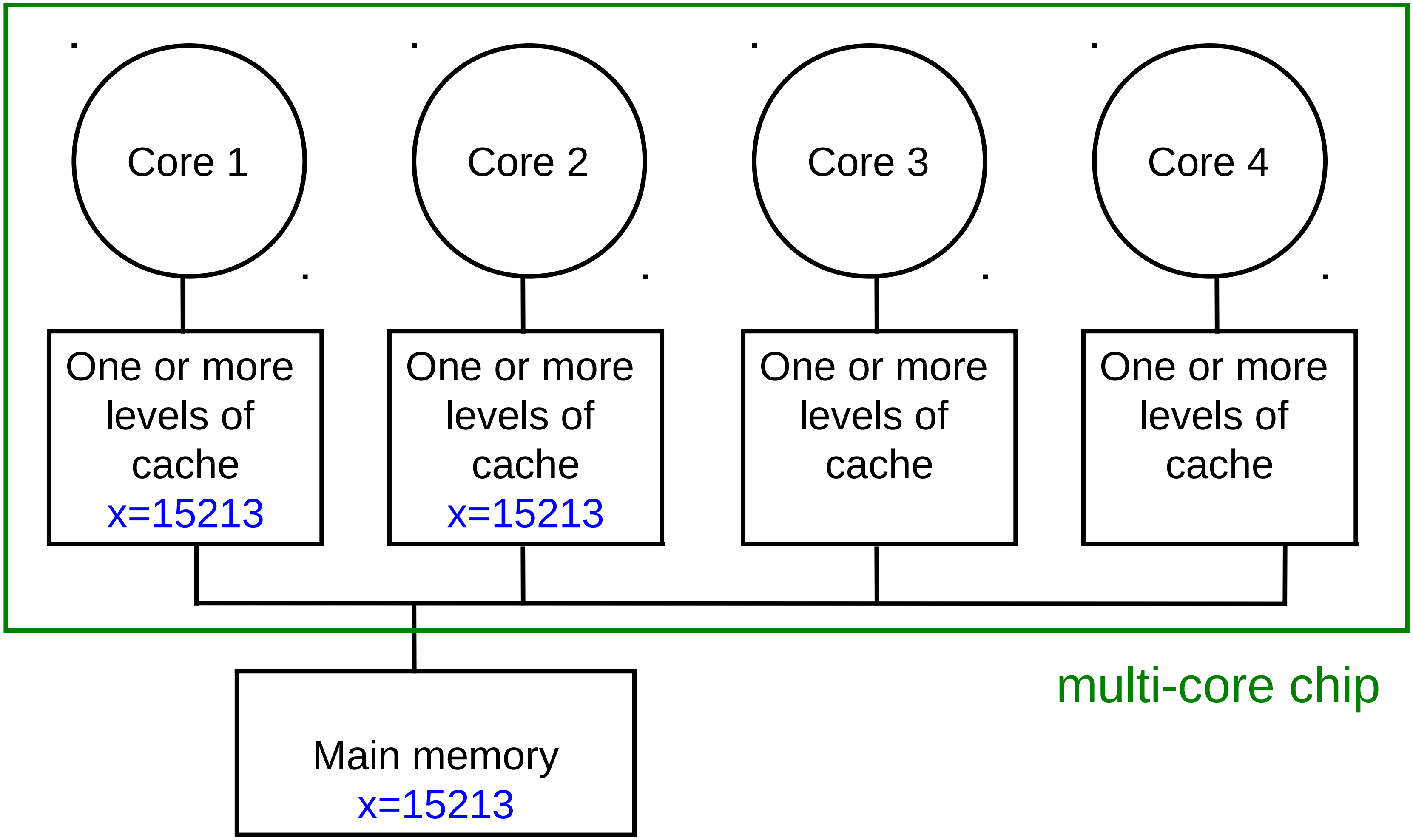
Invalidation protocol with snooping

- Invalidation
  - If a core write 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

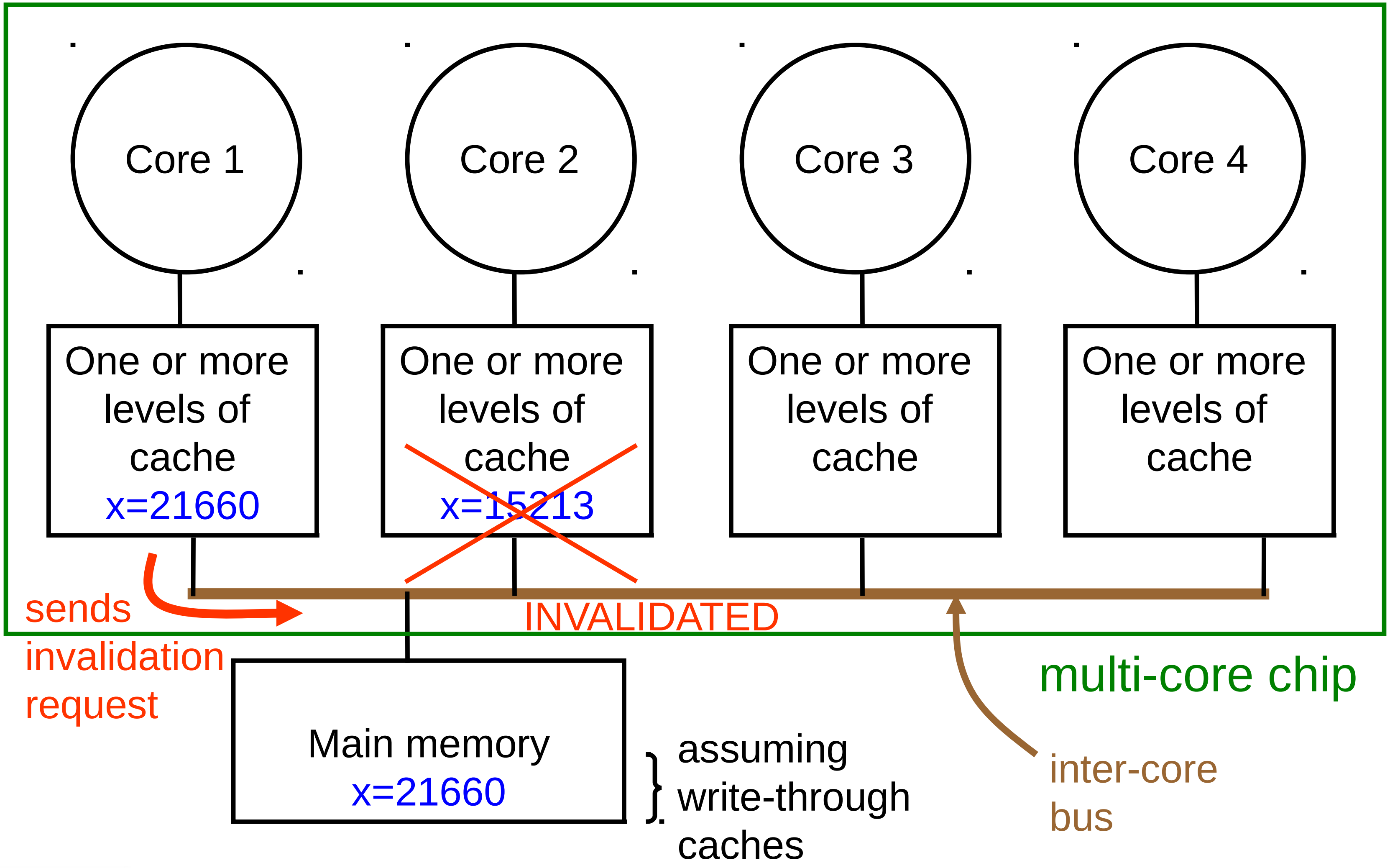
The cache coherence problem

- Revisited : Cores 1 and 2 have both read x


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=15213
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
Main memory
x=15213
multi-core chip

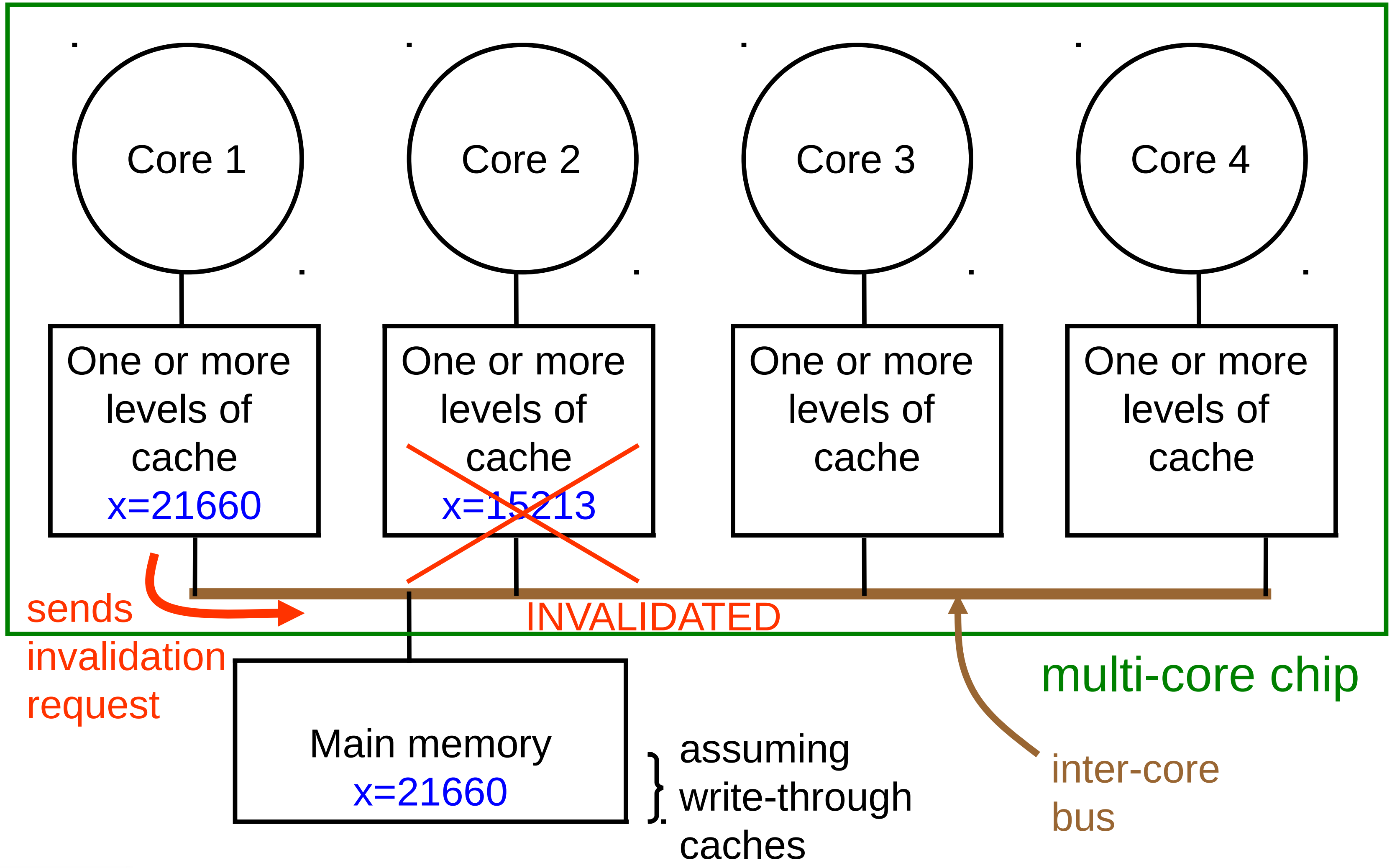
The cache coherence problem

- Core 1 writes to x, setting it to 21660


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
sends invalidation request
INVALIDATED
multi-core chip
Main memory
x=21660
assuming write-through caches
inter-core bus

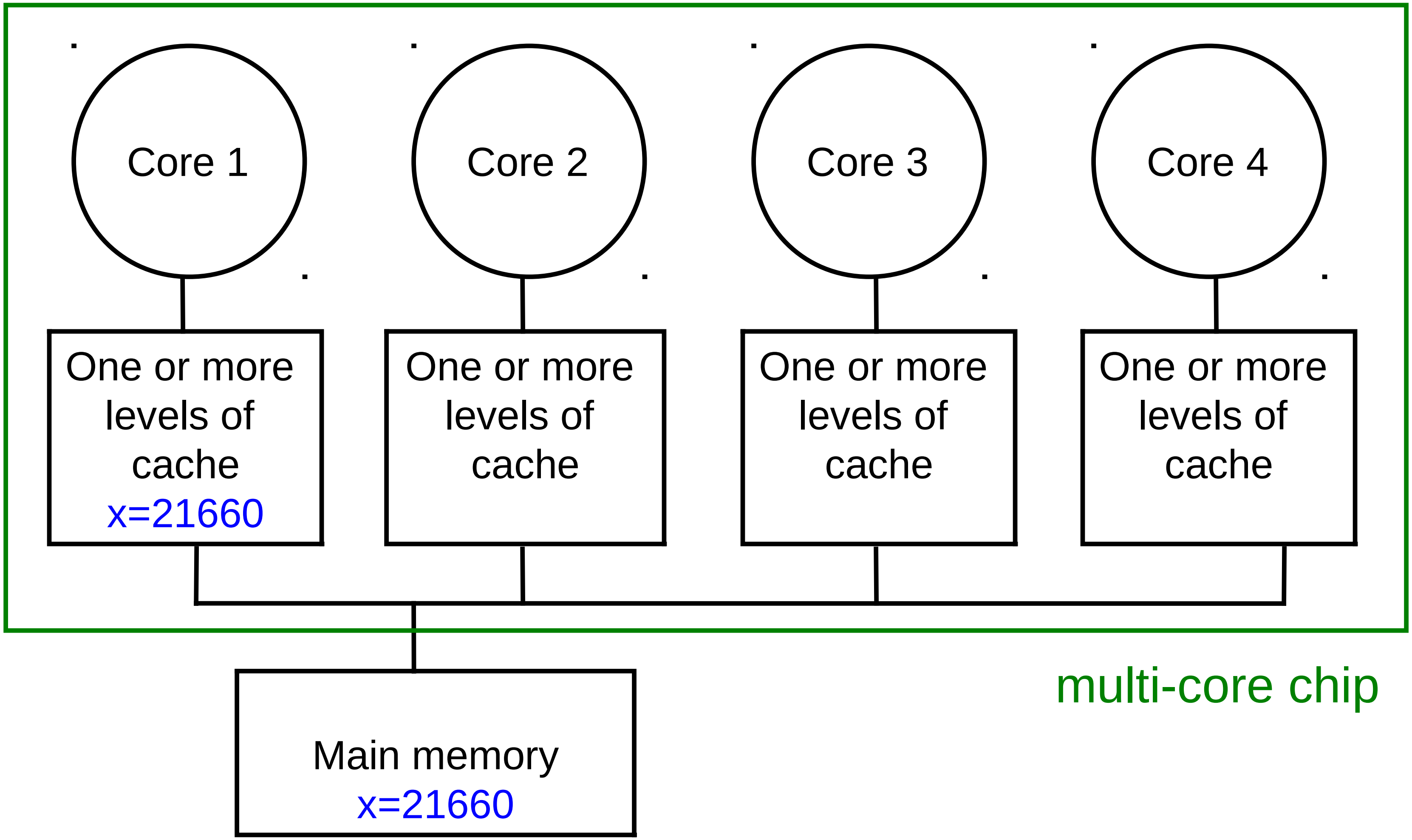
The cache coherence problem

- Core 1 writes to x, setting it to 21660


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
x=15213
One or more levels of cache
One or more levels of cache
sends invalidation request
INVALIDATED
multi-core chip
Main memory
x=21660
assuming write-through caches
inter-core bus

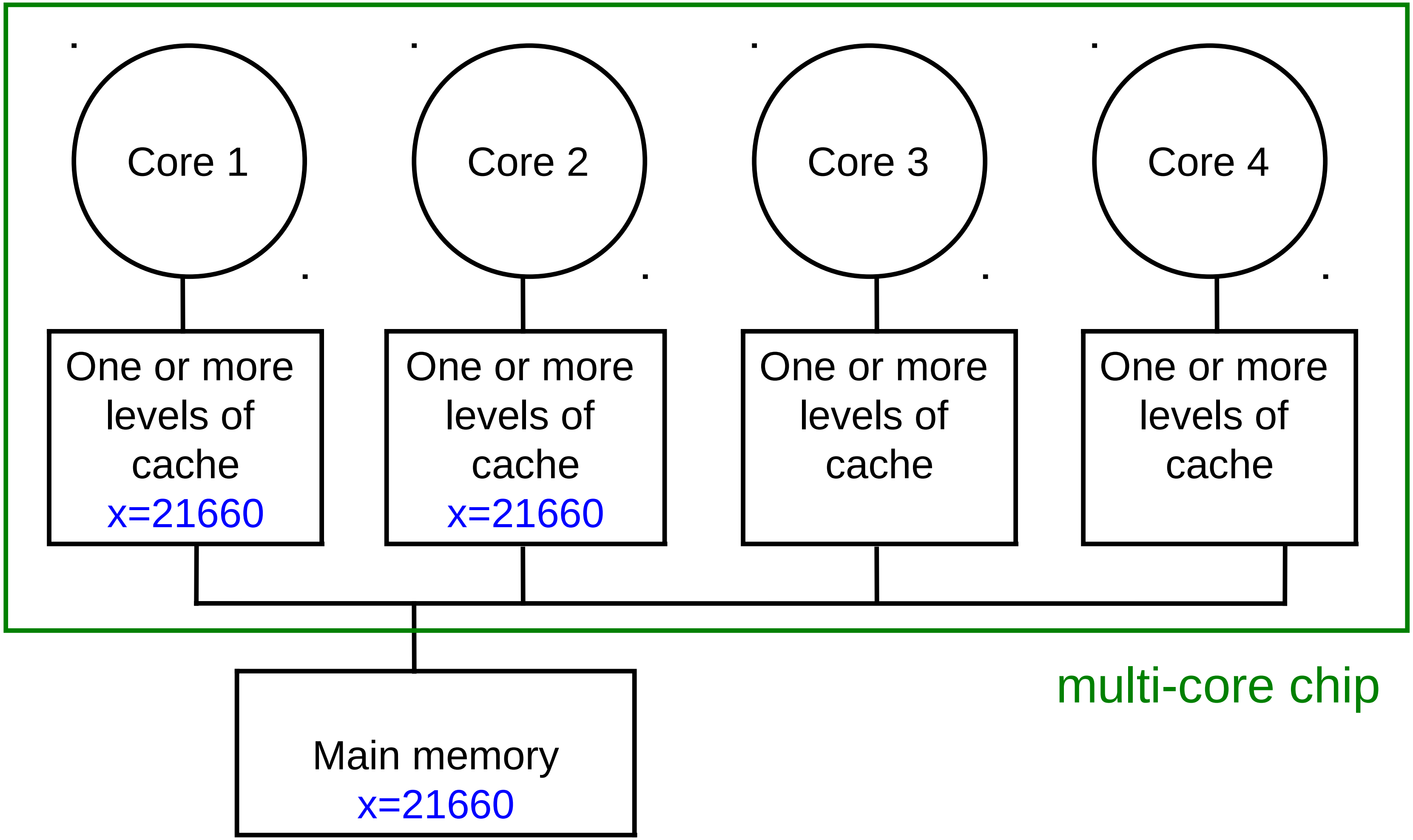
The cache coherence problem

- After invalidation :


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
One or more levels of cache
One or more levels of cache
multi-core chip
Main memory
x=21660

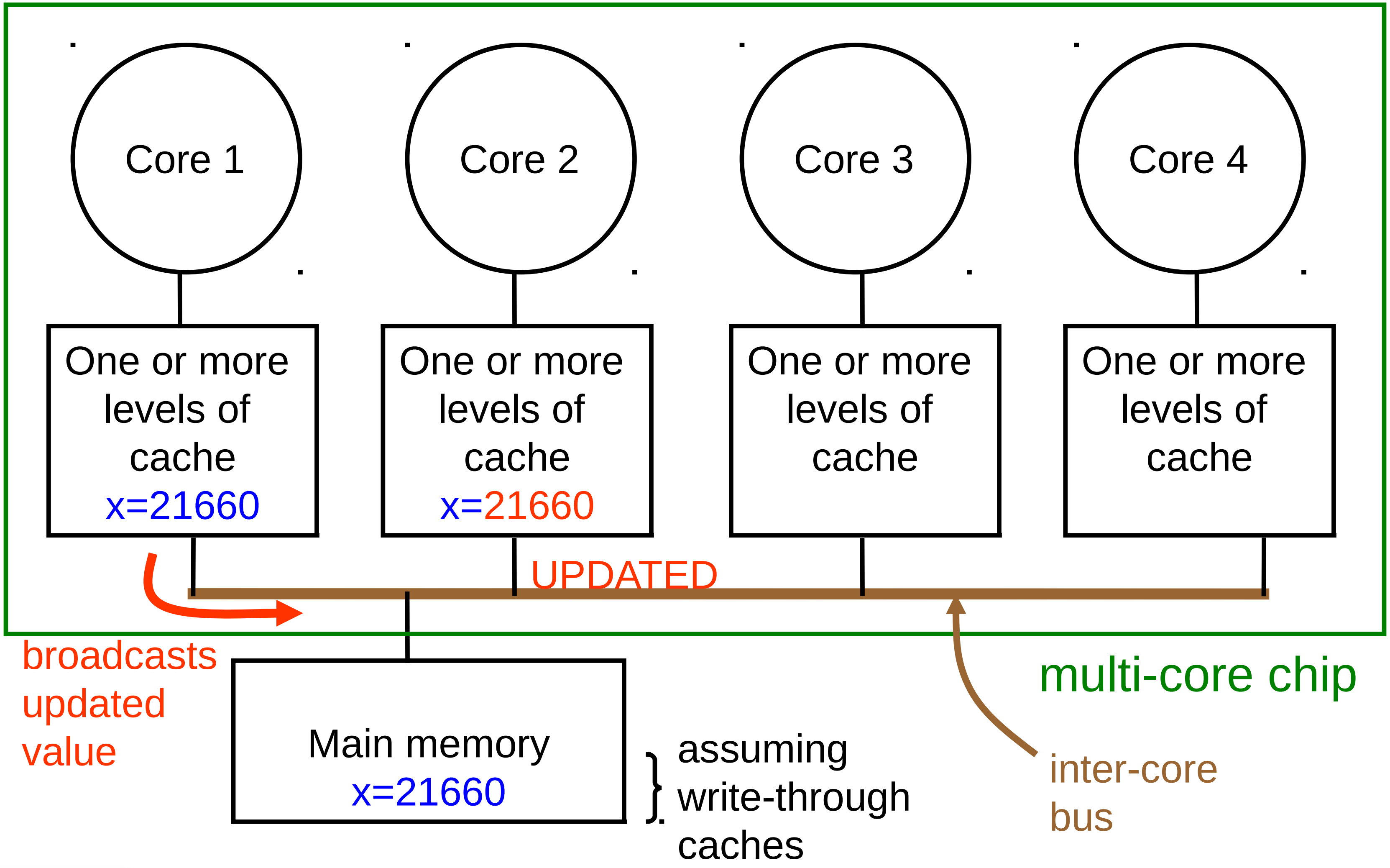
The cache coherence problem

- Core 2 reads x. Cache misses and load the new copy :


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
x=21660
One or more levels of cache
One or more levels of cache
Main memory
x=21660
multi-core chip

Alternative to invalidate protocol : update protocol

- Core 1 writes x = 21660 :


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
x=21660
One or more levels of cache
x=21660
One or more levels of cache
One or more levels of cache
UPDATED
broadcasts updated value
Main memory
x=21660
assuming write-through caches
multi-core chip
inter-core bus

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

Invalidation protocols

- This was just the basic invalidation protocol

- More sophisticated protocols use extra cache state bits

- MSI, MESI
  (Modified, Exclusive, Shared, Invalid)

4 C's : Sources of Cache Misses

- Compulsory misses (aka cold start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced blocked is later accessed again
- Conflict misses (aka collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache
- Coherence Misses

Hardware Performance Bottlenecks

- Cores
  - Do we suffer from cache misses ?
- Multicores
  - Do we keep all core busy ?
- Vector units
  - Is the code vectorized ?
- SMT, Hyperthreading
  - More threads, with lesser performance
  - Contention issues to computing units
- Memory
  - Do core interfere with each other in shared memory accesses ?

It's the Memory !

- Most performance bottlenecks have to do with data movement
  - Most optimizations have to do with locality
    - Temporal Locality
      - Re-use of data recently used
    - Spatial locality
      - Using data nearby that recently used
    - Processor locality
      - Resource sharing between the cores
      - Including avoidance of false sharing

Future Evolution

- Number of cores per chips double every 18-24 months
  - Up to 256 cores in 2016

- Cores may be heterogeneous
  - Fat an thin cores
  - GPU-like streaming accelerators

- Non Uniform Cache Architectures (NUCA)
  - Faster (smaller) caches shared by fewer cores
  - Possibly non coherent caches (coherence whithin smaller subdomains)

- Varying performance in time and space
  - Power management
  - Error recovery and fault masking

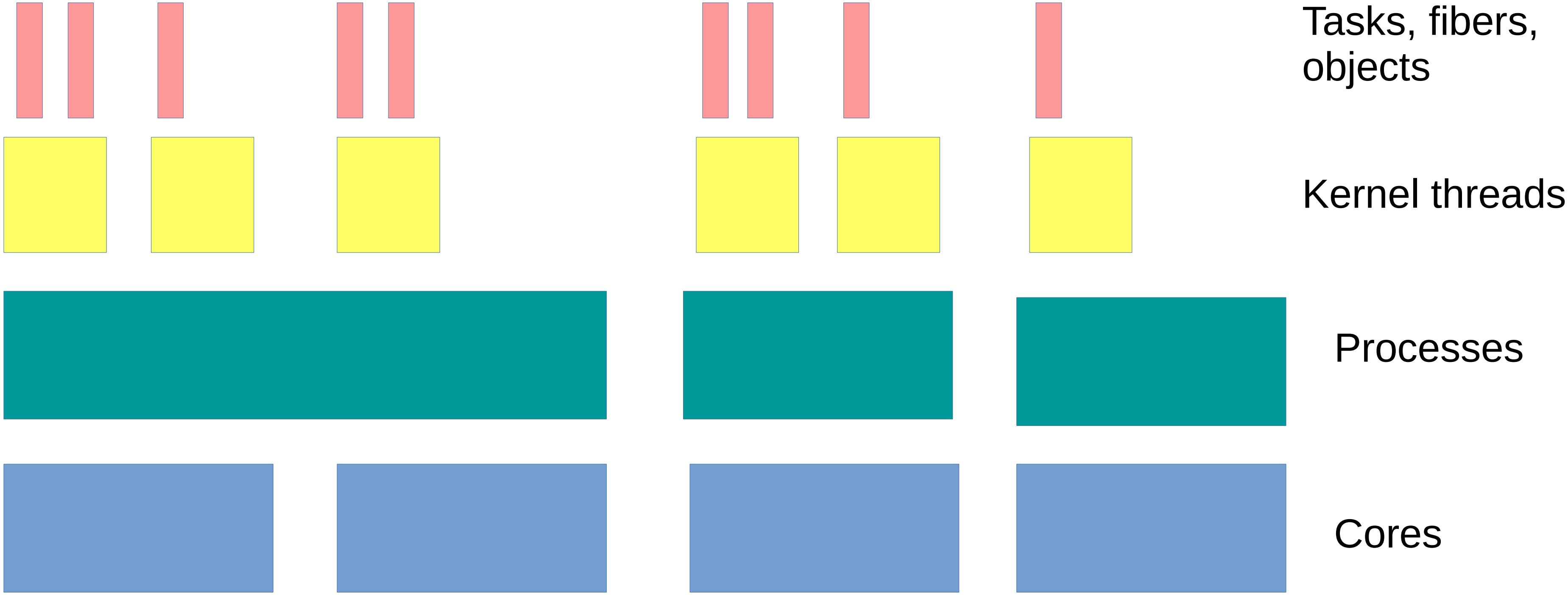
Software Stack

Programming for multi-core

- Programmer must use processes, threads or tasks.
  - Processes are the OS abstractions (Unix, Linux,…)
  - Kernel threads (Posix API) are parallel activities in processes
  - Tasks come with OpenMP, StarPU, QUARK
- Spread the workload across multiple cores/processors
- Write parallel algorithms
- OS will map processes/threads to processors/cores

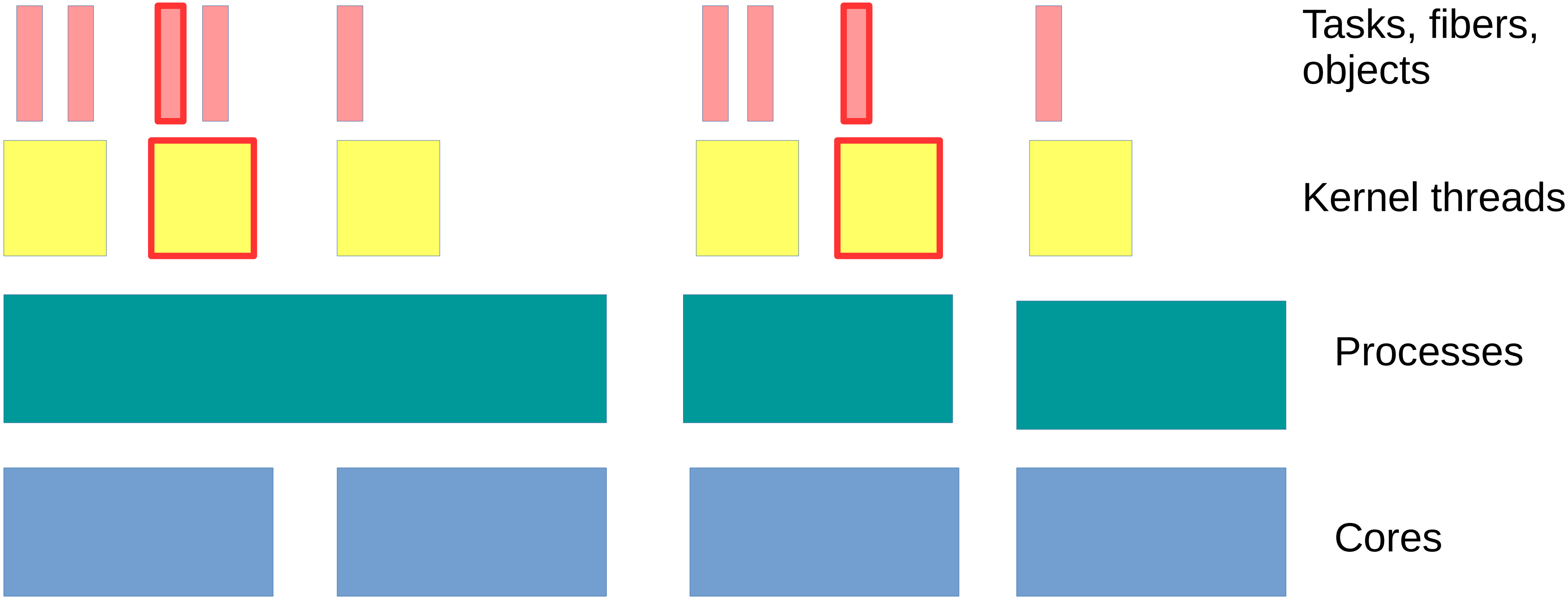
Resource Allocation (1)

- **Process**
  - Address space, virtual memory, file descriptors, queues, pipes, etc.
  - Binary code
  - One or more kernel threads



Resource Allocation (2)

- **Task** :
  - Stack pointer, registers – no system state
  - Scheduled by user library (runtime) onto pool of kernels threads
  - Nonpremptible – has to complete or yield
  - Blocking system blocks executing kernel threads



Task vs Kernel Thread

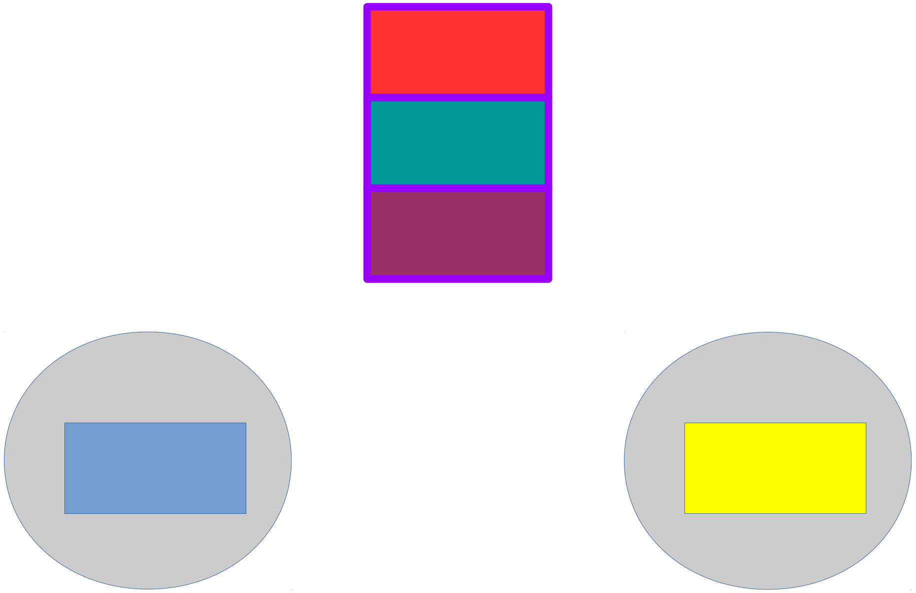
- Overhead for task creation and scheduling much lower
  - Lighter object
  - Collaboration objects (yield, not interrupt)
- Scheduling polycy can be application/programming model dependent

- Blocking system calls block all tasks scheduled on HW thread
- Hard to schedule on resources that come and go

- When tasks are used
  - Avoid, as much as possible, blocking system calls
  - Make sure that kernel threads run continously in one place
    - One kernel thread per core
    - Affinity scheduling
    - High priority
    - Left over resources for background activities

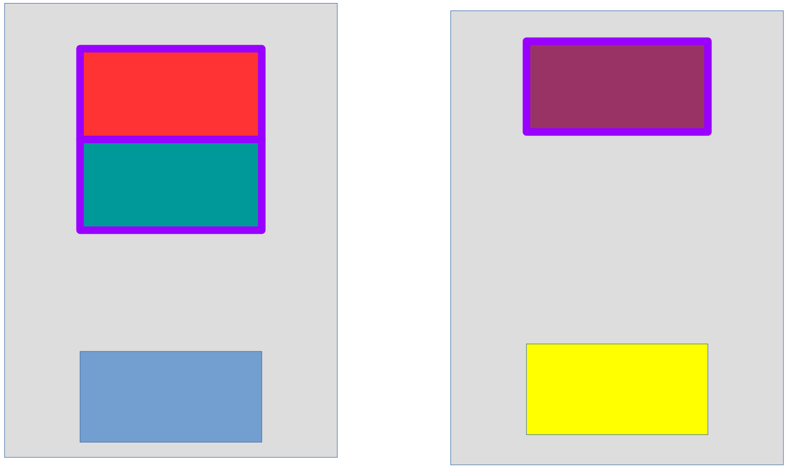
Scheduling Policies for Tasks (1)

- Central Queue
  - A thread appends a newly created task to the queue
  - An idle thread picks a task from the queue
  - Possible optimizations : chunk small taks ; use LIFO queue



Scheduling Policies for Tasks (2)

- Work Stealing
  - a thread appends a newly created task to local queue
  - an idle thread picks a task from its local queue
  - If local queue is empty, it steals a task from another queue



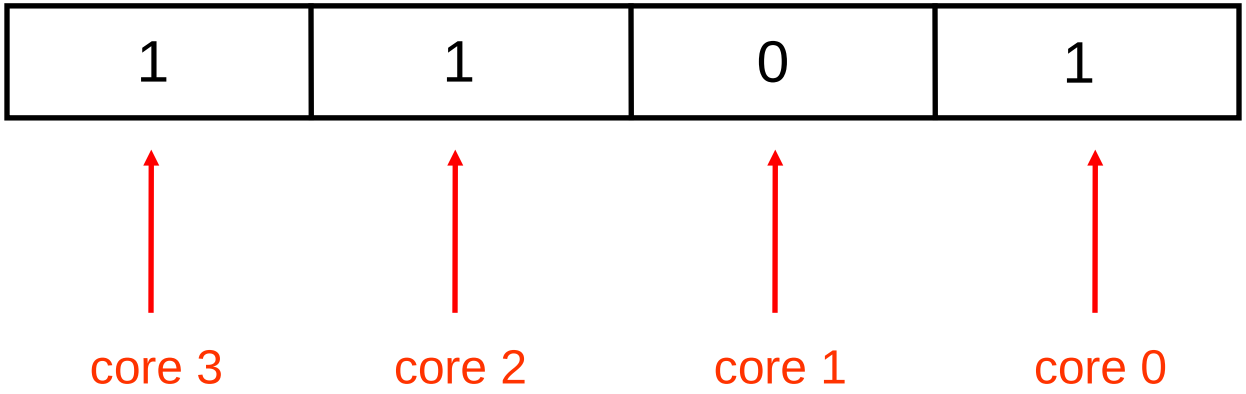
- Less communication/synchronization with central queue
- Better locality

Assigning Kernel Threads to the cores

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

Affinity masks are bit vectors

- Example : a 4-way multicore, without hyperthreading



1	1	0	1
core 3	core 2	core 1	core 0

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

Default Affinities

- Default affinity mask is all 1s :
  All kernel threads can run on all cores

- Then, the OS scheduler decides what treads run on what core

- Os scheduler skewed workloads, migrating kernel threads to less busy cores

Process/Kernel Thread 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 a same core

- This is called ***soft affinity***

When to set your own affinitues

- Two (or more) kernel threads share data structures in memory
  - Map to the cores of same processor,

    so that can share cache (L3)

- Kernel scheduler API

```
#include <sched.h>
int sched_setaffinity(pid_t pid,
unsigned int len, unsigned long * mask);
```

- Set the affinity mask of process/thread (**pid**) to ***mask**

Mont-Blanc Projects
2011-20xx

MB Projects Goals

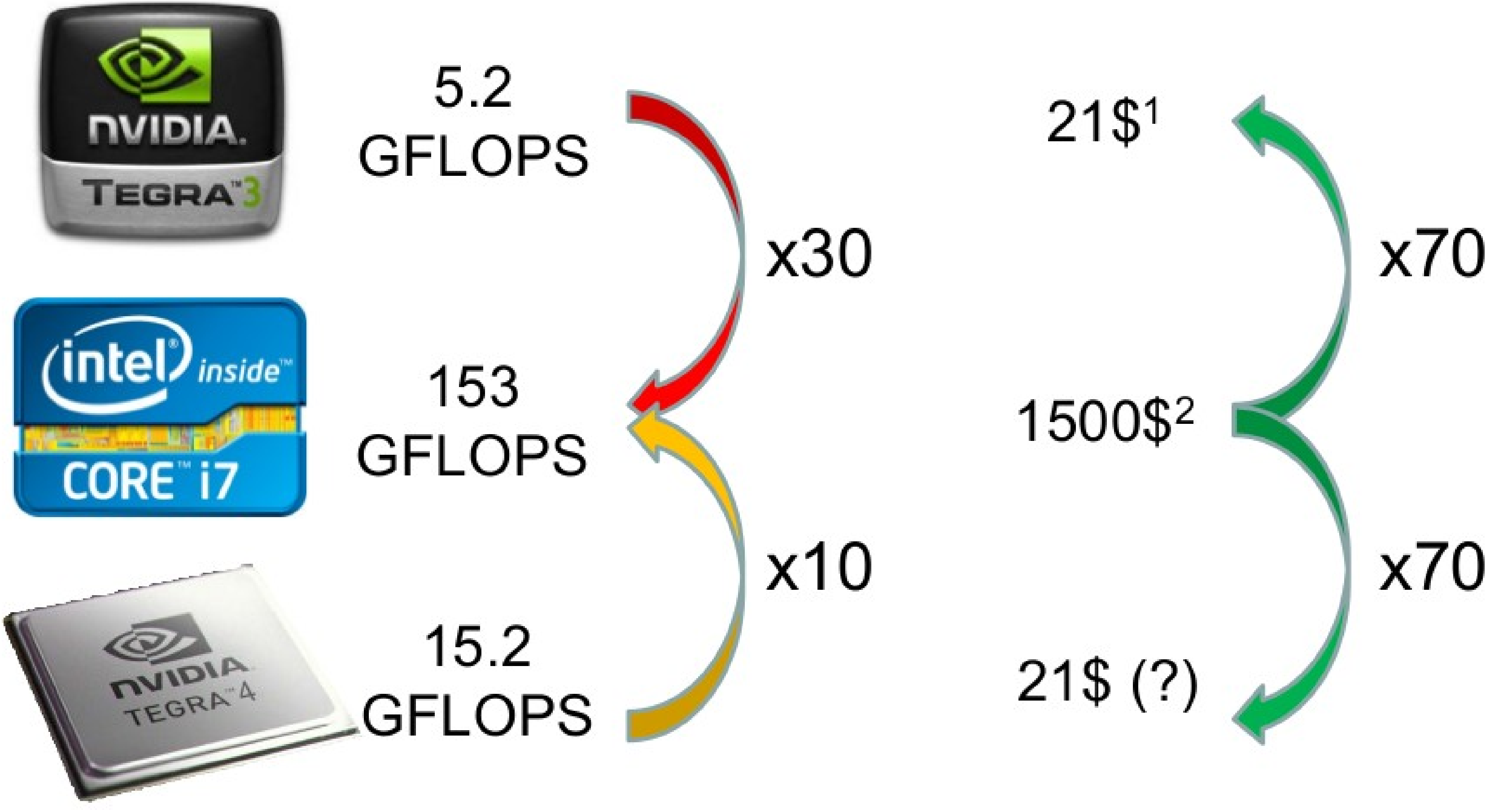
- To develop an **European** Exascale Approach

- Based on embedded **power-efficient technology**
  - **ARM :** Energy Efficient Processors

Mobile SoC vs Server processor

Performance

Cost



5.2 GFLOPS

153 GFLOPS

15.2 GFLOPS

x30

x10

21$[1]

1500$[2]

21$ (?)

x70

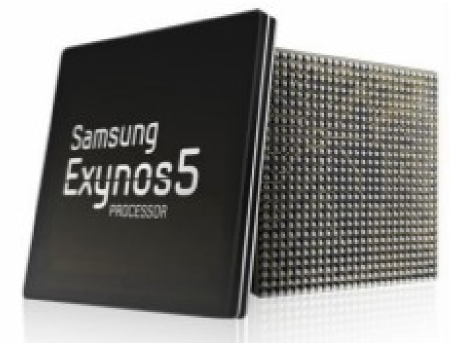
x70

1. Leaked Tegra3 price from the Nexus 7 Bill of Materials
2. Non-discounted List Price for the 8-core Intel E5 SandyBrdige

SoC under study: CPU and Memory

NVIDIA Tegra 2
2 x ARM Cortex-A9 @ 1GHz
1 x 32-bit DDR2-333 channel
32KB L1 + 1MB L2

NVIDIA Tegra 3
4 x ARM Cortex-A9 @ 1.3GHz
2 x 32-bit DDR23-750 channels
32KB L1 + 1MB L2



Samsng Exynos 5 Dual
2 x ARM Cortex-A15 @ 1.7GHz
2 x 32-bit DDR3-800 channels
32KB L1 + 1MB L2

Intel Core i7-2760QM
4 x Intel SandyBrdige @ 2.4GHz
2 x 64-bit DDR3-800 channels
32KB L1 + 1MB L2 + 6MB L3

EU Mont-Blanc projects

- Objectives
  - To deploy a prototype HPC system based on currently available energy-efficient, embedded technology
    - Deploy a full HPC system software stack
  - To design a next-generation HPC system and new embedded technologies targeting HPC systems that would overcome most of the limitations of the prototype
  - To port and optimize a small number of representative Exascale applications capable of exploiting this new generation of HPC systems
    - Up to 11 full-scale applications

The Mont-Blanc prototype

Exynos 5 compute card
2 x Cortex-A15 @ 1.7GHz
1 x Mali T604 GPU
6.8 + 25.5 GFLOPS
15 Watts
2.1 GFLOPS/W

Carrier blade
15 x Compute cards
485 GFLOPS
1 GbE to 10 GbE
300 Watts
1.6 GFLOPS/W

Blade chassis 7U
9 x Carrier blade
135 x Compute cards
4.3 TFLOPS
2.7 kWatts
1.6 GFLOPS/W

Rack
8 BullX chassis
72 Compute blades
1080 Compute cards
2160 CPUs
1080 GPUs
4.3 TB of DRAM
17.2 TB of Flash

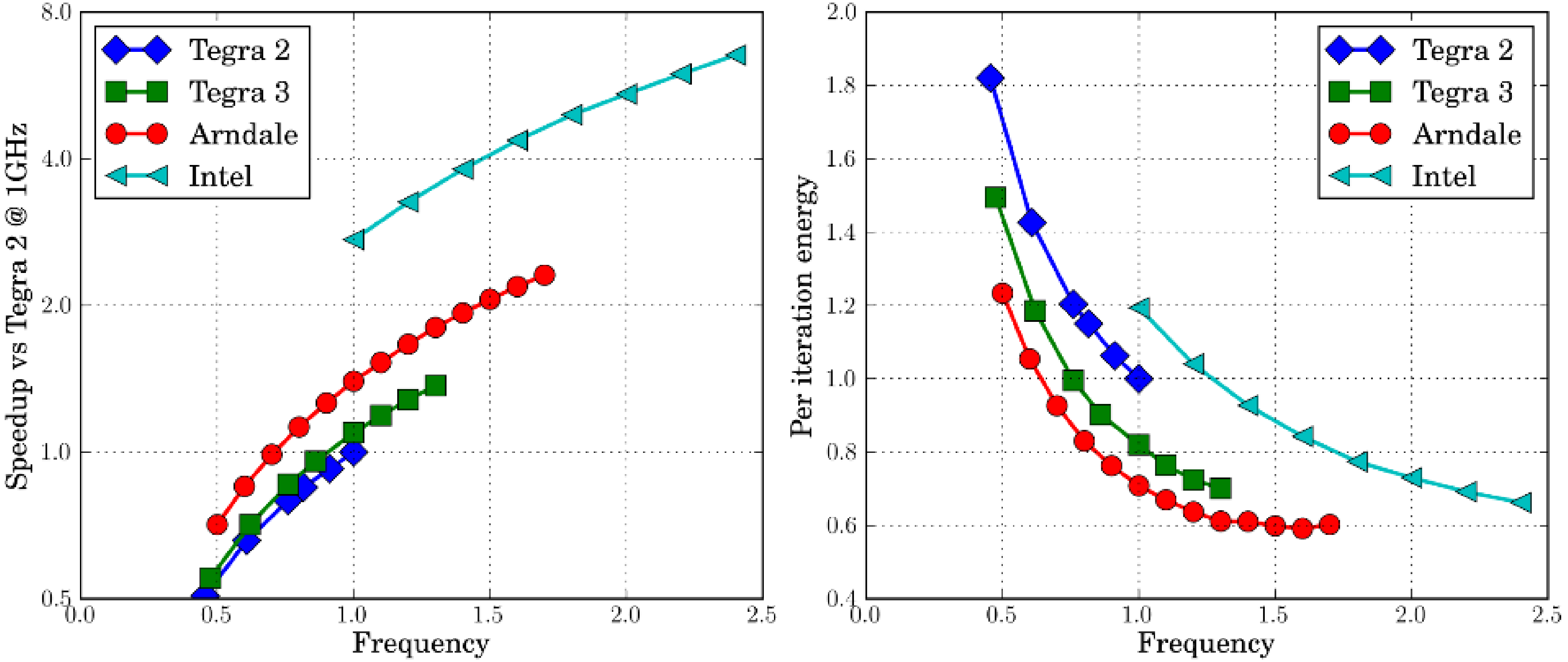
35 TFLOPS
24 kWatt

	Mont-Blanc [GFLOPS/W]	Green500 [GFLOPS/W]
Nov 2011	0.15	2.0
Nov 2014	1.5	5.2

Evaluated kernels

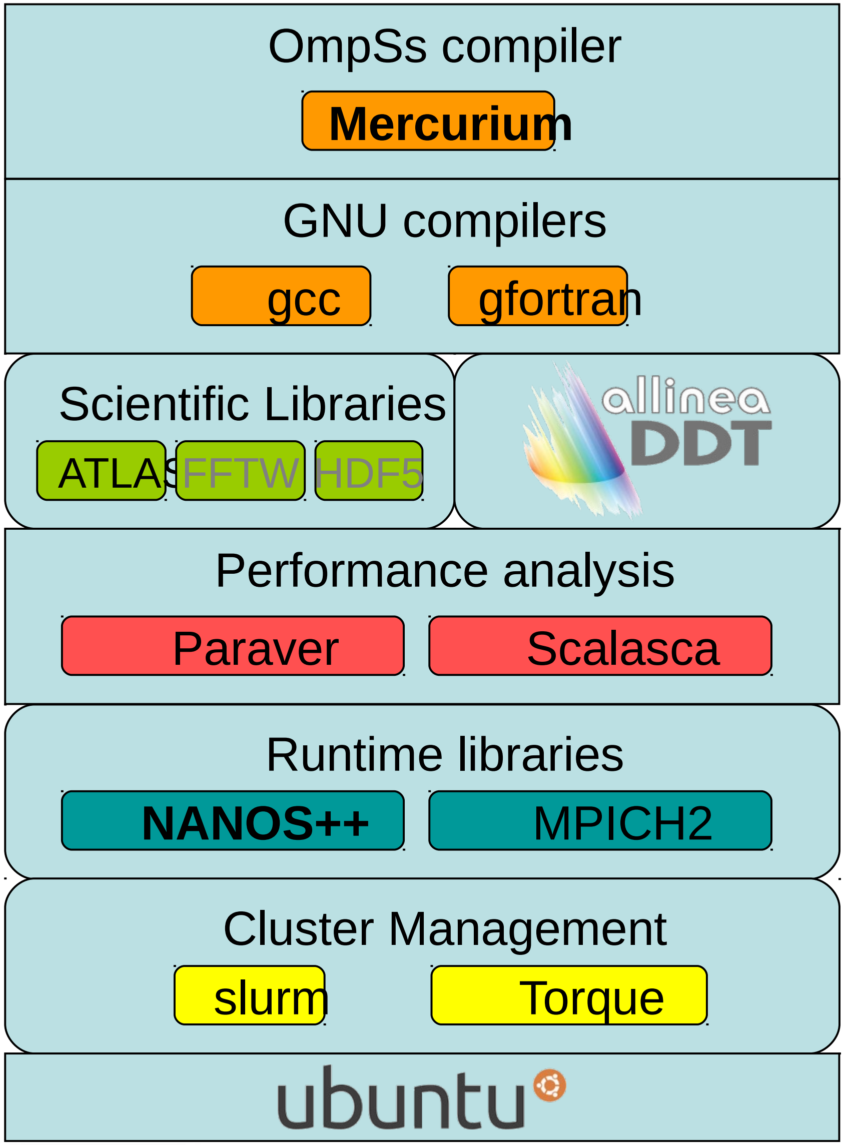
Tag	Full name	Properties	pthreads	OpenMP	OmpSs	CUDA	OpenCL
vecop	Vector operation	Common operation in numerical codes	✓	✓	✓	✓	✓
dmmm	Dense matrix-matrix multiply	Data reuse an compute performance	✓	✓	✓	✓	✓
3dstc	3D volume stencil	Strided memory accesses (7-point 3D stencil)	✓	✓	✓	✓	✓
2dcon	2D convolution	Spatial locality	✓	✓	✓	✓	✓
fft	1D FFT transform	Peak floating-point, variable stride accesses	✓	✓	✓	✓	×
red	Reduction operation	Varying levels of parallelism	✓	✓	✓	✓	×
hist	Histogram calculation	Local privatization and reduction stage	✓	✓	✓	✓	×
msort	Generic merge sort	Barrier synchronization	✓	✓	✓	✓	×
nbody	N-body calculation	Irregular memory accesses	✓	✓	✓	✓	×
amcd	Markov chain Monte-Carlo method	Embarassingly parallel	✓	✓	✓	✓	×
spvm	Sparse matrix-vector multiply	Load imbalance	✓	✓	✓	✓	×

Single core performance and energy



- Tegra3 is 1.4x faster than Tegra2
  - Higher clock frequency
- Exynos 5 is 1.7x faster than Tegra3
  - Better frequency, memory bandwidth, and core microarchitecture
- Intel Core i7 is ~3x better than ARM Cortex-A15 at maximum frequency
- ARM platforms more energy-efficient than Intel platform

Mont-Blanc System Software Stack



- Open source system software stack
  - Ubuntu Linux OS
  - GNU compiler
    - gcc 4.4.5
    - gfortran
  - Scientific libraries
    - ATLAS, FFTW, HDF5,...
  - Cluster management
- Runtime libraries
  - MPICH2, OpenMPI
  - **OmpSs toolchain**
- Performance analysis tools
  - Paraver, Scalasca
- Allinea DDT 3.1 debugger
  - Ported to ARM

Inria

INVENTEURS DU MONDE NUMÉRIQUE