Concurrency:
Multi-core Programming & Data Processing

Introduction

Prof. P. Felber

Pascal.Felber@unine.ch

http://iiun.unine.ch/

Institut d'informatique
Université de Neuchâtel

Course Objectives

- Cover the foundations of concurrent systems and multiprocessor synchronization
  - Combines theory and practical, hands-on exercises on multi-core computers
- Expected results
  - Understand the major challenges and complexity inherent to concurrent systems
  - Know the major algorithms, paradigms, and theoretical results
  - Learn the basics of **multi-core programming**, the new paradigm of computer science

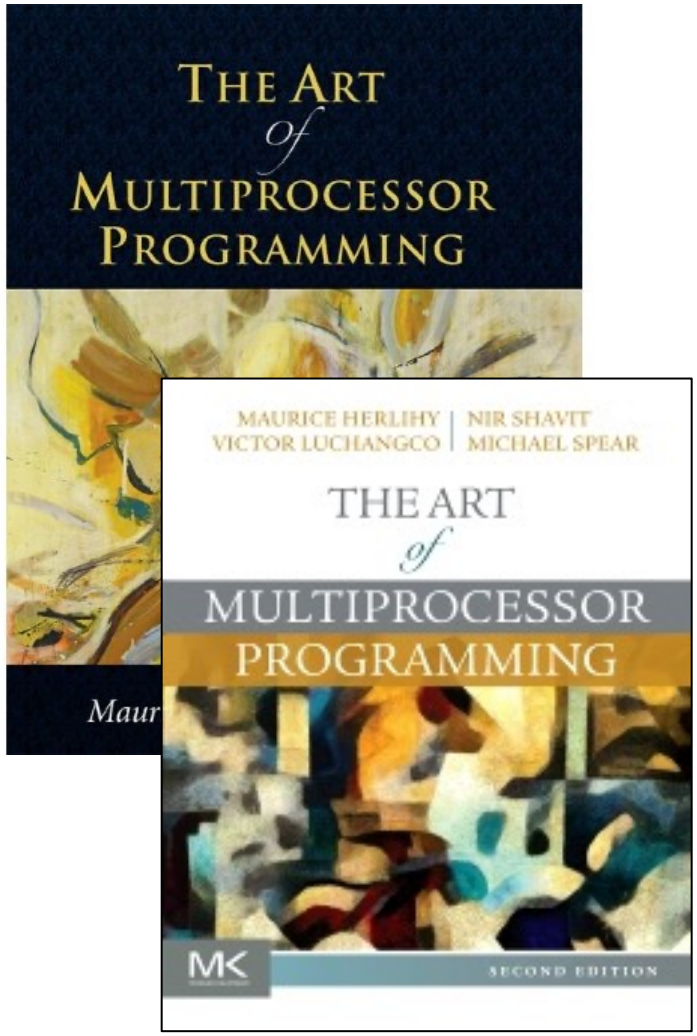
Textbook

- **The Art of Multiprocessor Programming**

  Maurice Herlihy, Nir Shavit

  - 1st edition 2008/2012
  - 2nd edition 2020

- Credits: much of the material kindly provided by authors of the book



Organization

- Course main page on ilias.unibe.ch
  - Slides and support documentation
  - Labs (mandatory)
  - Details on organization and organizational changes
- Discord (optional)
- Labs
  - Assignments every 2-3 weeks, some graded
  - Other labs: programming APIs, exercises, support
  - Programming tools: Java and your laptop (or our servers)
- Exam (written)
- Final grade: 1/3 labs + 2/3 exam

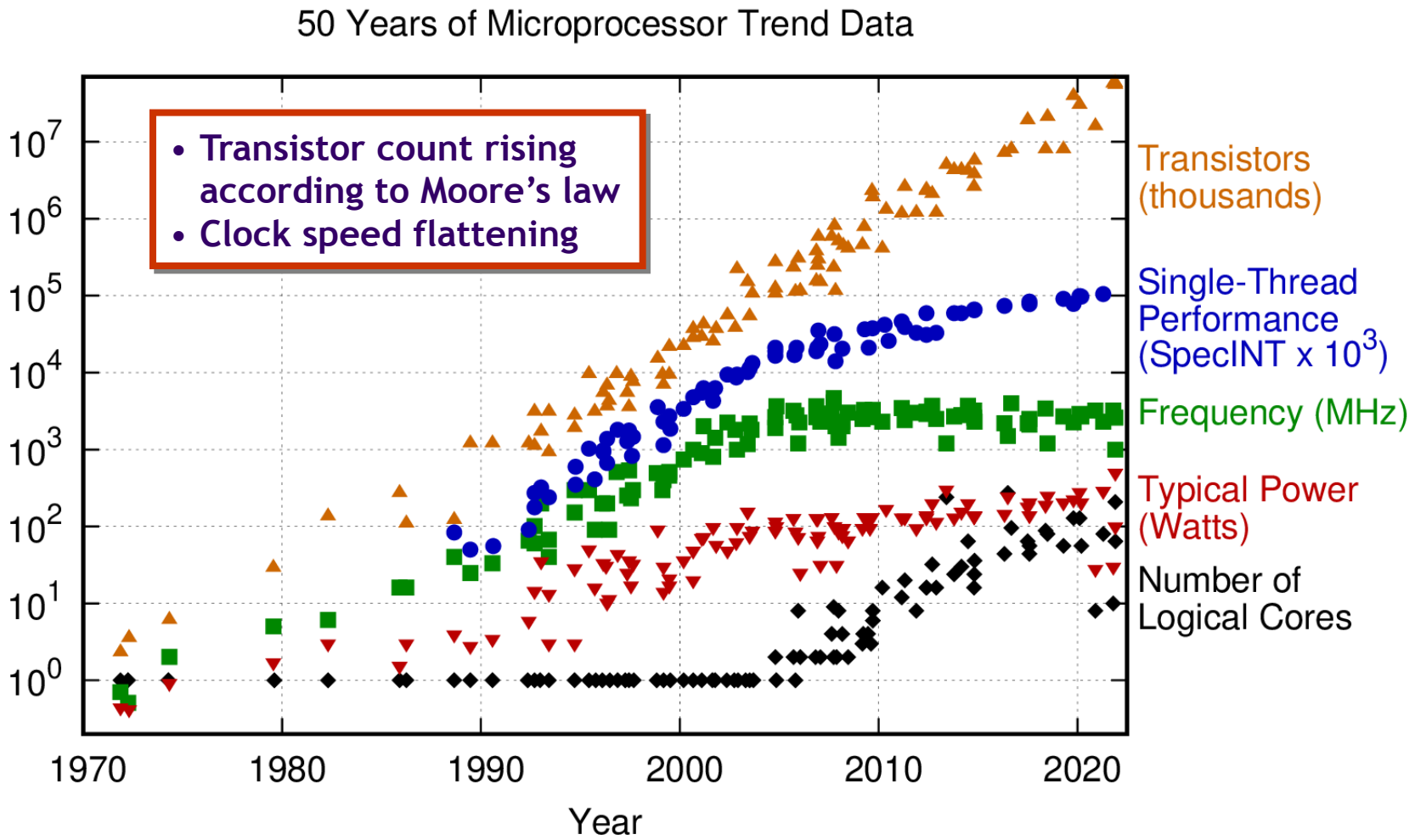
Exercises

- Mostly small coding challenges
  - Java
  - You can use your own laptop
- Some exercises “on paper”
- Code can be deployed on larger server
  - 32-core Sun Niagara
  - 48-core x86
  - 10(x8)-core POWER8
  - 64(x4)-core Xeon Phi

Major Topics

1. Introduction and basics
2. Mutual exclusion
3. Concurrent objects and consistency
4. Multiprocessor architecture basics
5. Spin locks and contention
6. Linked lists
7. Queues and stacks
8. Hashing
9. Futures, scheduling, and work distribution
10. Barriers
11. *Data parallelism*
12. *Software TM*
13. *Hardware TM*

Moore's Law and CPU Speed



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Multicores will be Everywhere

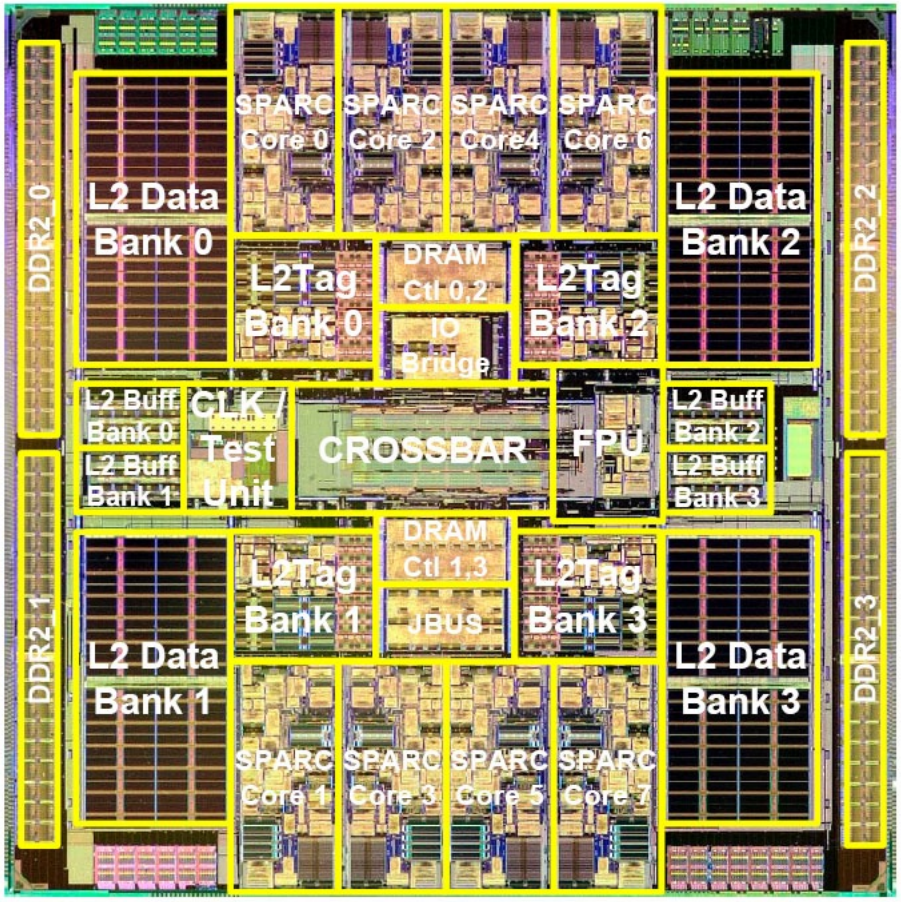
- Multicores are the answer to keeping up with increasing CPU performance despite:
  - The **memory wall** (gap between CPU and memory speeds)
  - The **ILP wall** (not enough instruction-level parallelism to keep the CPU busy)
  - The **power wall** (higher clock speeds require more power and create thermal problems)
- Consequence:
  - Single-thread performance does not improve…

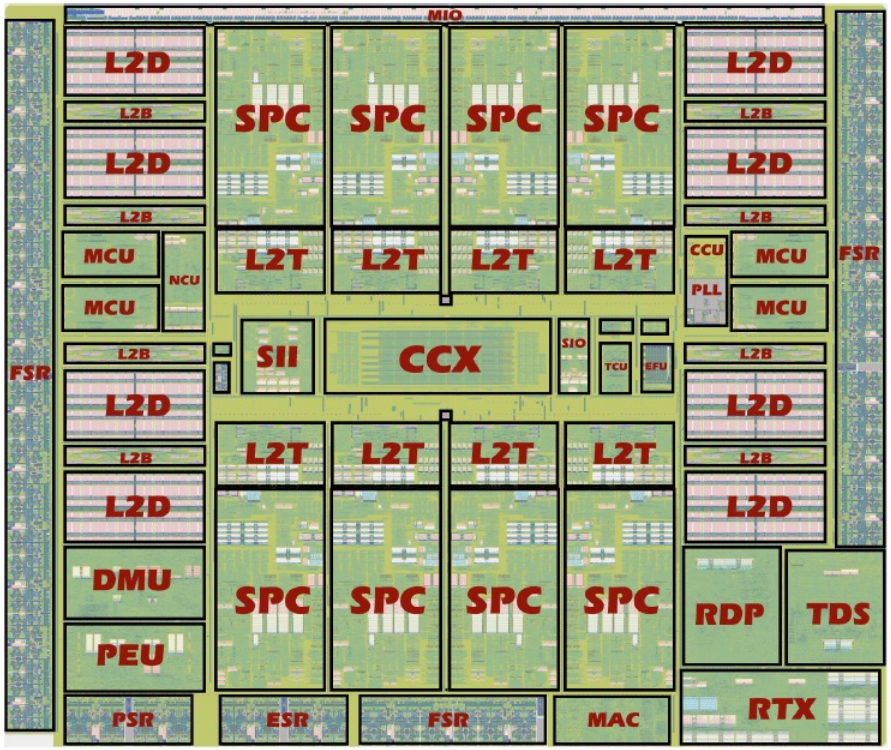
  … but we can put more cores on a chip

Multicores are Everywhere

- Dual-core commonplace in laptops
- Quad-core in desktops
- Dual quad-core in servers
- All major chip manufacturers produce multicore CPUs
  - AMD Opteron (up to 16 cores)
  - Intel Xeon (up to 18 cores)
  - IBM POWER8 (12 cores, 96 concurrent threads)
  - SUN SPARC (16 cores, 128 concurrent threads)
  - …

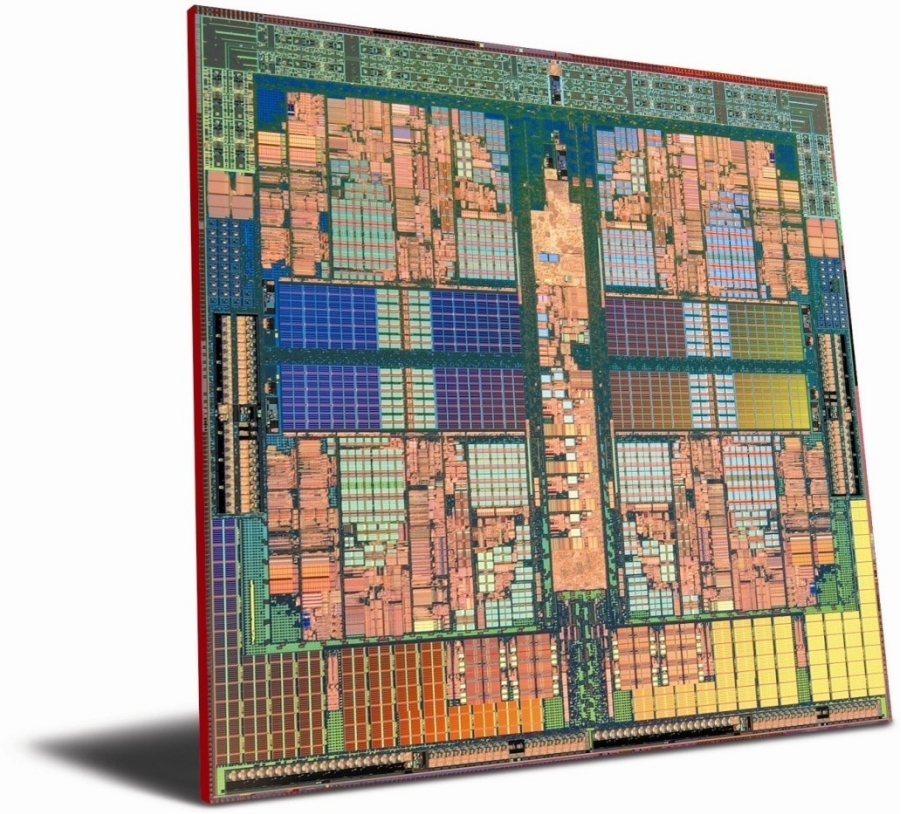
SUN's Niagara CPU2 (8 cores)

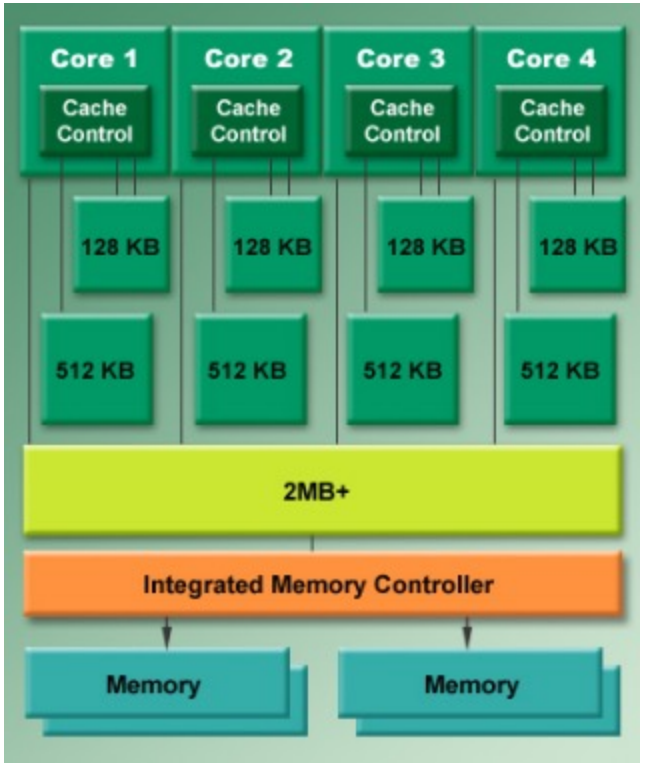




CCX – Crosssbar
CCU – Clock control
DMU/PEU – PCI Express
EFU – Efuse for redundancy
ESR – Ethernet SERDES
FSR – FBD SERDES
L2B – L2 write-back buffers
L2D – L2 data arrays
L2T – L2 tag arrays
MCU – Memory controller
MIO – Miscellaneous I/O
PSR – PCI Express SERDES
RDP/TDS/RTX/MAC – Ethernet
SII/SIO – I/O data path to and from memory
SPC – SPARC cores
TCU – Test and control unit

AMD Opteron (4 cores)



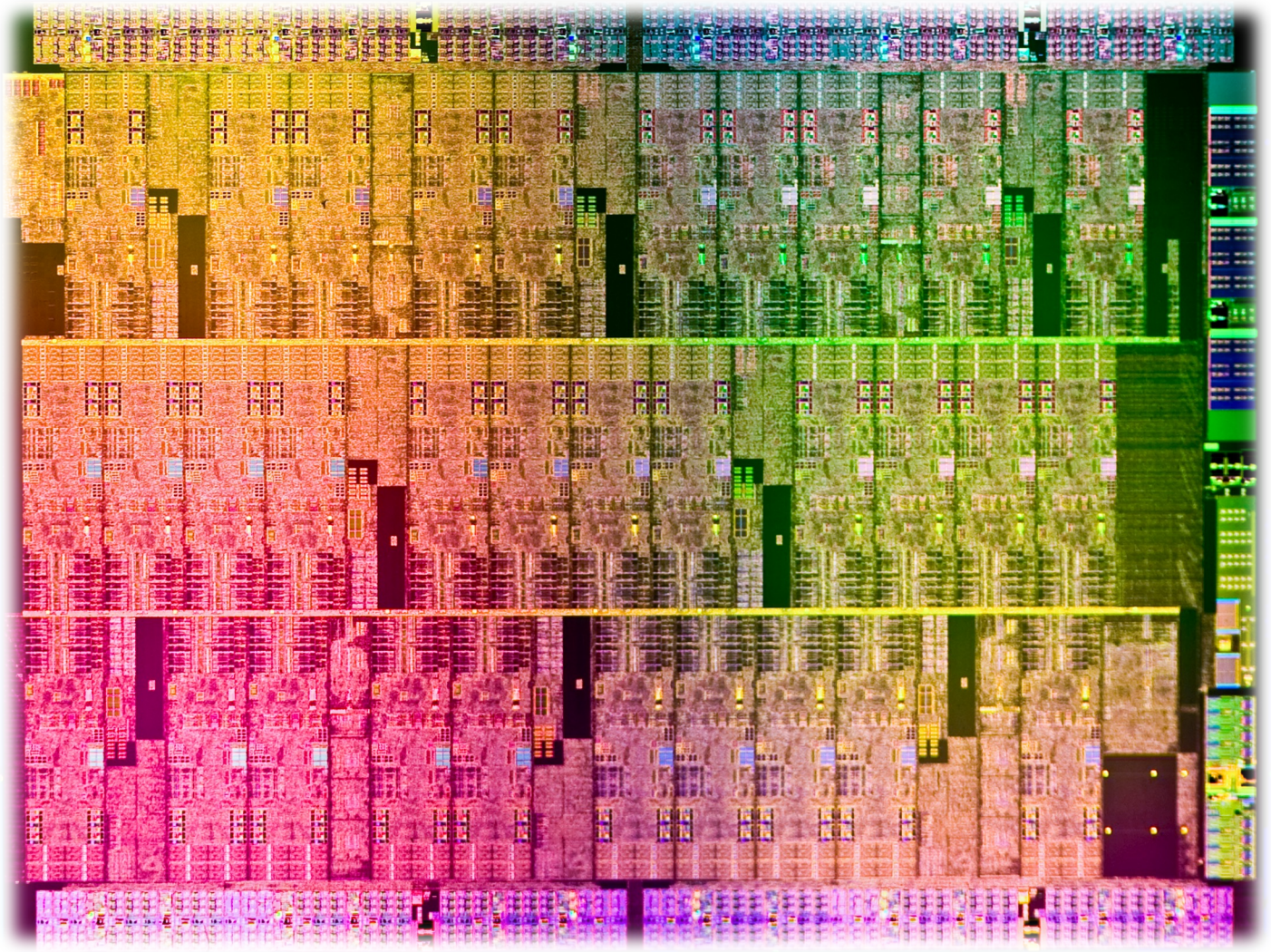


L1 cache

L2 cache

L3 cache (shared)

Xeon Phi (72 cores, 288 HWT)

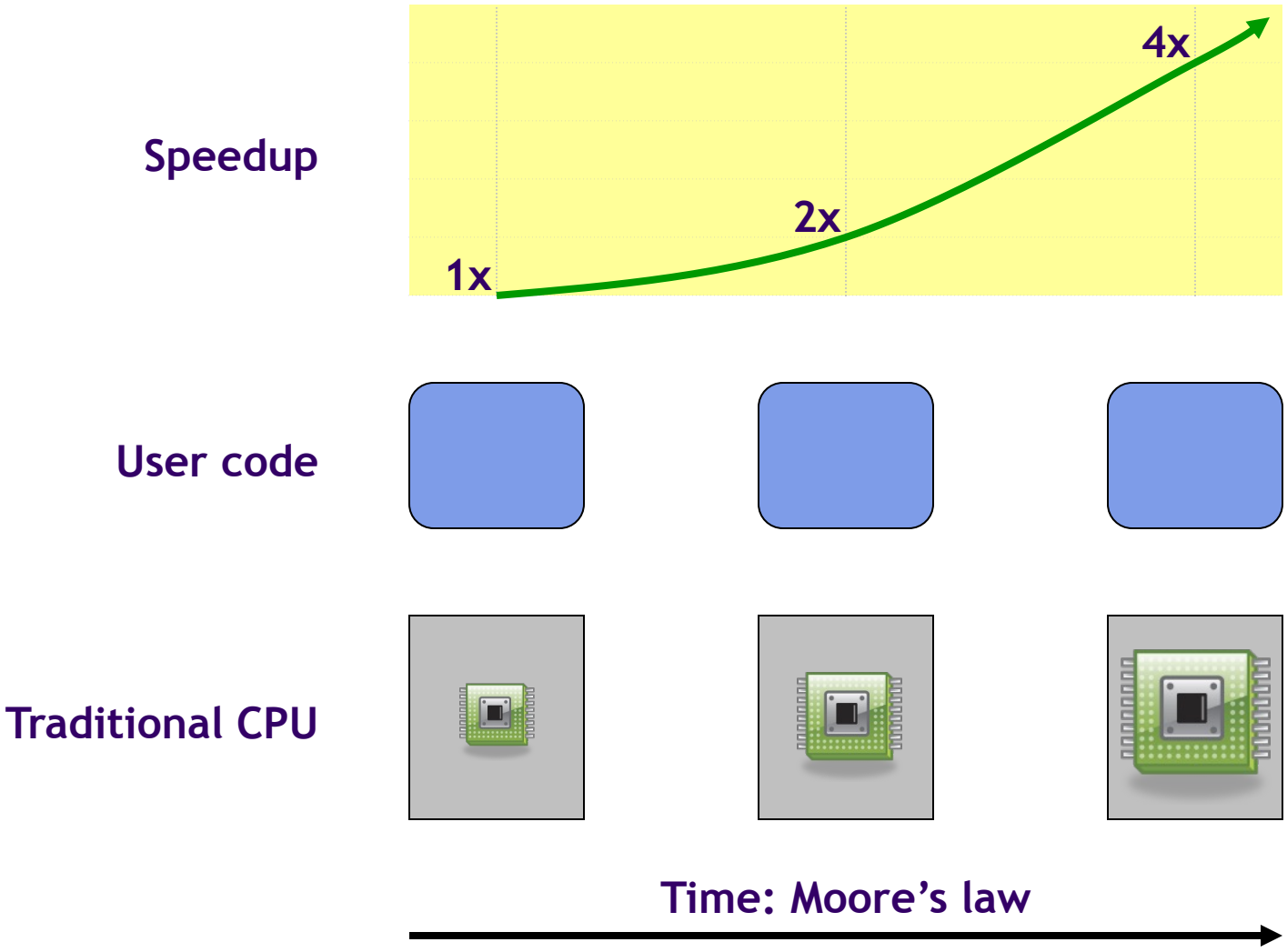


The “Free Ride” is Over

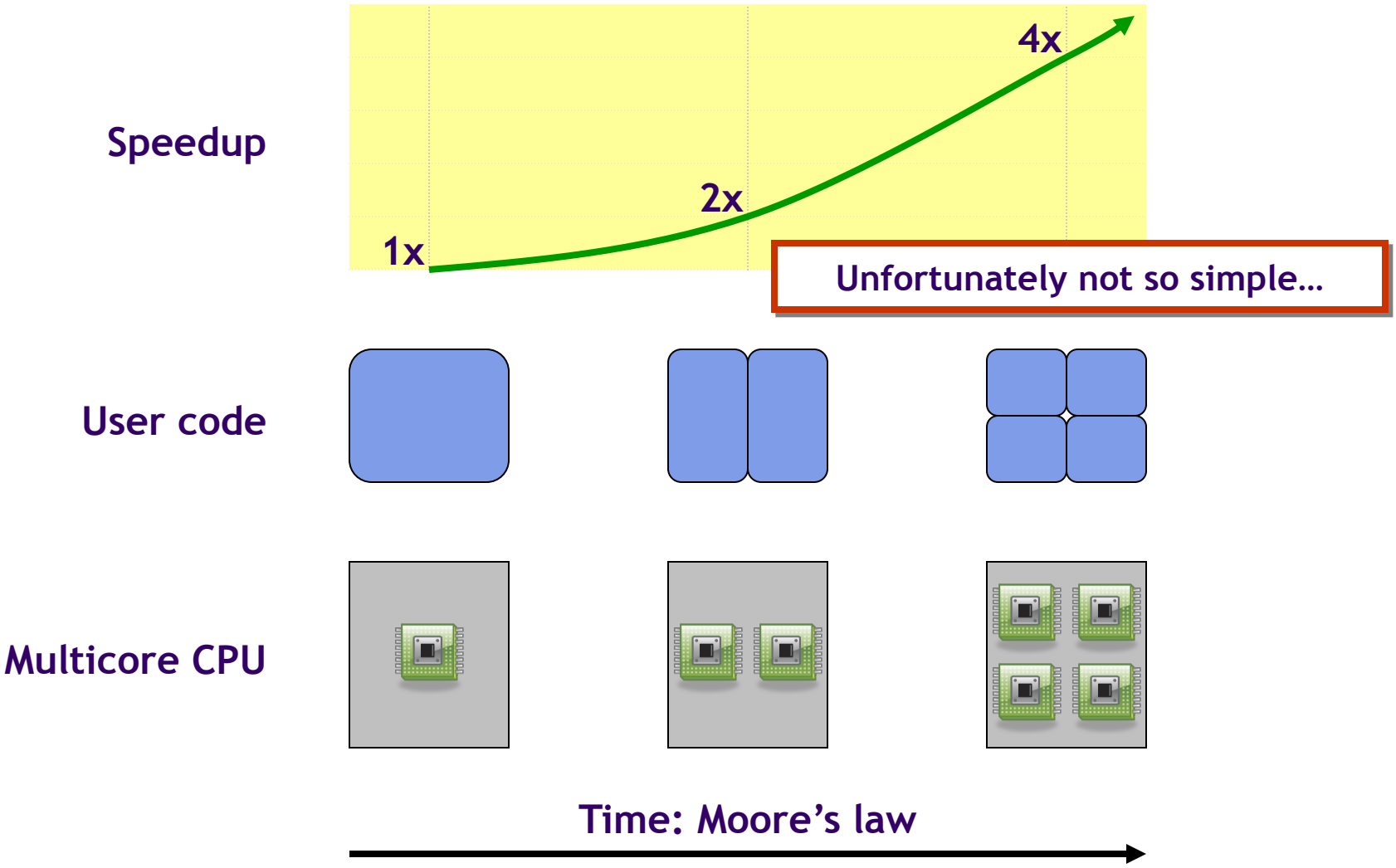
- Cannot rely on CPUs getting faster in every generation
- Utilizing more than one CPU core requires thread-level parallelism (TLP)
- One of the biggest future software challenges: **exploiting concurrency**
  - Every programmer will have to deal with it
  - Affects HW/SW system architecture, programming languages, algorithms, …
  - Concurrent programming is hard to get right

… better not hit the **productivity wall**

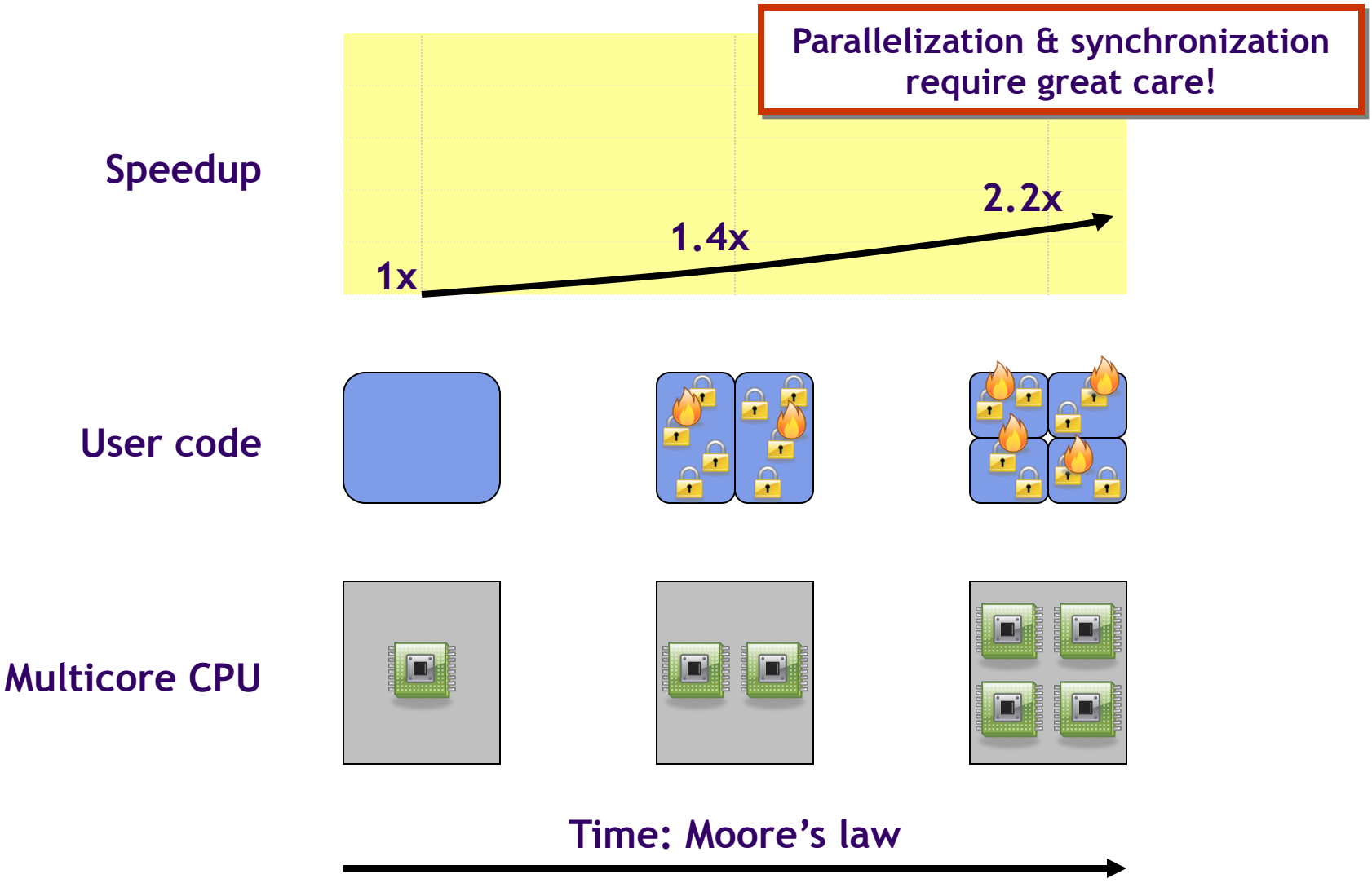
Traditional Scaling Process



Multicore Scaling Process



Real-World Scaling Process



Concurrent programming is hard

- Hard to make **correct** and **efficient**
  - We need to exploit parallelism
- The human mind tends to be sequential
  - Concurrent specifications
  - Non-deterministic executions
- What about races? deadlocks? livelocks? starvation? fairness?
  - Need synchronization (correctness)…

  … but not too much (performance)

Parallelization (1)

- **Data parallelism**
  - Split data into partitions
  - Let threads process partitions
  - Threads join after finishing their work
  - **Example:** image processing, sequencing, etc.
- Good data partitioning is difficult (correctness, load balancing, …)
- Synchronization required
  - Join
  - Load balancing during runtime

Parallelization (2)

- **Pipeline parallelism**
  - Split work into phases
  - Let each thread work on jobs in one of the phases
  - **Example:** event stream processing

    input → parse → process → format → output
- Not every program has such phases, some phases are longer than others
- Synchronization required
  - One phase's results become next phase's input
  - Complex access patterns

Parallelization (3)

- **Speculative/optimistic parallelism**
  - Just execute concurrent jobs
  - Assumption: most jobs do not conflict
  - Partitioning or phases not required but possible
  - **Example:** event handlers, application servers, graph algorithms, etc.
- There must be little contention on shared data structures
- Synchronization required
  - Every access can potentially interfere with accesses from another thread

Shared memory synchronization

- Cores share main memory
  - Some hardware instructions are (or can be made) atomic: **`inc`**, **`dec`**, **`cmpxchg`**, …
- Loads/store usually atomic, not necessarily ordered (no sequential consistency!)
  - Must use memory barriers to enforce order
- Current state of concurrent programming:
  - Use locks built from these instructions
  - Build concurrent (non-blocking) algorithms from these instructions

In this course

- **Concurrency: foundations and algorithms**
  - How to develop concurrent data structures and applications that are:
    - Correct
    - Efficient (scale with the number of processors)
  - How to reason upon concurrent programs
  - How to understand performance tradeoffs and avoid bottlenecks
  - How to choose the right programming abstraction and algorithm for a given problem