Lecture 01: Introduction

Computer Science Department, University of Crete

Multicore Processor Programming

Course Objectives

- Train students on parallel programming
  - Use 7 parallel programming models at various levels of abstraction
  - Use 3 parallel multi-core architectures
- Train students on reading, discussing, criticizing research papers

Course Logistics

- Graduate course with research training component
  - Discuss research papers in class on a weekly basis (30% of class grade)
    - Read papers, write summary and review, email to instructor before the class
    - Papers for each class on website
    - Paper for next time: Stencil Computation Optimization and Autotuning on State-of-the-Art Multicore Architectures (`https://crd.lbl.gov/assets/pubs_presos/sc08-stencil.pdf`)
  - Perform 4 programming assignments using 4 programming models to implement parallel applications of varying degrees of complexity (30% of class grade)
    - First assignment will be online by next class: Posix and Java Threads
    - 1-2 weeks per assignment
    - Homework assignments are personal
    - Academic integrity: do not share code or discuss details
    - Discussing speedup and performance (and competing on them) is OK
  - Paper presentation (10% of class grade)

Course Logistics

- Course web page
  - `http://www.csd.uoc.gr/~hy529`
  - Online soon
- Subscribe to the course mailing list by sending an e-mail to `majordomo@csd.uoc.gr` with body:
  - `subscribe hy529-list`
- Coordinate with instructor in first week of class to get accounts for accessing a multi-core system on which you will perform your class projects

Moore's law

- 1965:
  "The complexity for minimum component costs has increased at a rate of roughly a factor of two **per year** *...Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years."*
- 1975: New projection: doubling every 2 years
- 2013: Doubling every 3 years, expected to slow down more
- In 2003, Intel predicted the end by 2020
  - Limit estimate: 16 nanometer process, 5 nanometer gates
  - Quantum tunneling effects at smaller sizes
  - Predictions are hard: "the end is in 10 years", 30 years now

Moore’s law in a picture

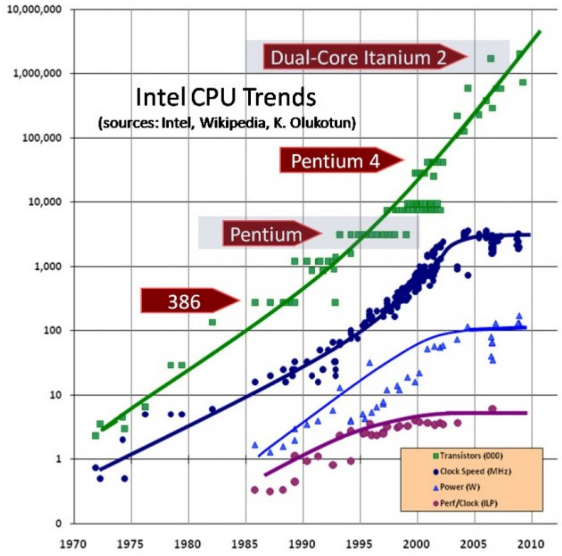


Chart originally from “The Free Lunch is Over” by H. Sutter.

Parallel computing for performance

- Moore’s law: The number of components in an integrated circuit doubles every X months. X = 12, . . ., 24, . . .
- Until recently, performance through frequency
  - Higher frequencies gave performance “for free”
  - Transistor size limited by speed of light given frequency
- Power wall: frequency is so high and transistors so small that they melt
  - Frequency stopped increasing
  - Need for computing power keeps increasing
  - Industry solution: More transistors, same frequency, more proceessor cores
- New interpretation of Moore’s Law
  - The number of cores doubles with every technology generation
- Shift to parallel computing changes fundamentally the way we program, debug, and analyze the performance of computers

Parallel computing then and now

- Parallel computing is an old idea
  - Not very widespread in the past
  - Cost/Performance ratio was better for sequential than parallel machines
- Cost/performance ratio more favorable now:
  - Can pack multiple cores on the same chip
  - Graceful technology scaling through replication
- Parallel computing may fail again if we do not find a way to exploit parallelism in software
  - Challenges of parallelizing software:
    - Understanding data dependencies
    - Synchronize accesses to shared data
    - Minimize communication, balance load
    - Difficult for humans to think about all possible executions
  - Often optimal parallel algorithm not similar to equivalent sequential
    - May need complete redesign

New kinds of parallelism

- 100s of cores per chip already available
  - Single-board GPU systems with up to 1600 NVIDIA cores
  - 50+ general purpose cores in Intel Xeon Phi
- Vendors integrate up to 8 chips in a single node, 64-core x86 machines available now
- Computing systems that benefit:
  - Datacenters performing big data analytics
  - Cloud installations offering virtualized services
  - Supercomputers running heavy HPC applications on large data sets
  - Mobile devices that tend to replace PCs as general purpose personal computers
  - Embedded systems that perform real-time intensive data processing

Application-driven parallelism: Then

- Parallel computing has been driven by HPC domain in the past
  - Scientific applications that could afford high cost/benefit of old parallel architectures
  - Dense arithmetic, physics, military, oil industry, stock market
- Dominant programming model: message passing
  - Processors exchange data and synchronize by explicit messages
  - Portable, optimizable by experts
  - Tedious to write, debug, requires experts
  - High cost of programmers still less than higher costs of hardware
- Many alternatives proposed
  - Parallel languages, libraries of patterns, auto-parallelizing compilers
  - Easier to program, but less performance compared to message passing written by experts
  - Message passing still dominant

Application-driven parallelism: Now

- Parallel computing hardware costs much less
  - Parallel computing synonym to computing
  - Parallel hardware everywhere, mobile parallel computers
  - Needs of society at large, not just few specific applications
    - Irregular algorithms, unstructured data, arbitrary applications
  - New important factors: power budget, space constraints, latency, real-time
- Programmer cost becomes important
  - High cost of message-passing experts
  - Programmer productivity more important that small sacrifice in performance

Technology-driven parallelism

- Many cores per chip, per device
- High speed I/O links, network links
  - 10Gbps current, 40Gbps emerging, 100Gbps future
  - Network speeds make packet processing processor-bound
  - May need many cores to sustain throughput
- Faster storage
  - Flash replacing or complementing disks, order of magnitude faster
  - Phase change RAM as main memory

Parallel architectures: CMP

- Single chip multi-core processor, up to 8 cores
  - Typical general-purpose architecture: Desktop, laptop
  - Private L2, shared L3 cache memories
  - Hardware cache-coherence

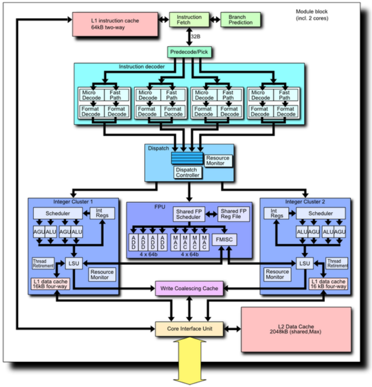
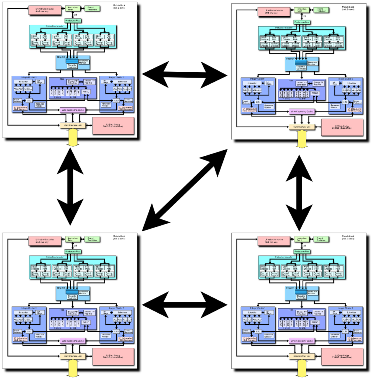


Figure source: Wikipedia.

Parallel architectures: CC-NUMA

- Cache Coherent, Non-Uniform Memory Access architecture (CC-NUMA)
  - Multiple multicore processors per package or per board
  - Multiple memory controllers, memory banks
  - Coherence traffic accross motherboard
  - Not all memory accesses cost the same



Parallel architectures: GPU

- Massively parallel processors
  - Simple scalar cores
  - Software-managed shared memory
  - Groups/tiles of cores running the same code, often lock-step

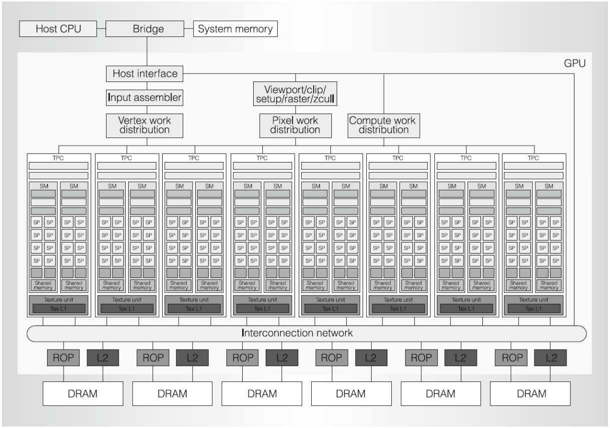


Figure source: rutgers.edu

Writing Parallel Software is Hard

- Finding parallelism is hard
- Selecting proper granularity is hard
- Locality effects are more important, may vary a lot between architectures
- Load balancing is hard
- Coordination and synchronization are hard
- Performance modeling is different for each architecture
- Testing and debugging are hard

- Parallel programming is even harder than sequential programming

Find enough parallelism: Amdahl's Law

- Suppose 90% of the application is parallel, the rest 10% is sequential
- Amdahl's Law says:
  - If s is the fraction of total work done sequentially and $1-s$ is parallelizable
  - And P is the number of processors
  - Then the maximum speedup is

$$Speedup(P) = \frac{Time(1)}{Time(P)}$$

$$\leq \frac{1}{(s + \frac{(1-s)}{P})}$$

$$\leq \frac{1}{s}$$

- Even we execute the parallel part in zero time with infinite cores, the sequential part will limit performance

Overheads and Granularity

- Given enough parallel work, overhead is the biggest problem in getting large speedups
- Overhead may be
  - Cost of starting parallelism: create thread, spawn task, fork process, etc.
  - Cost of communication of shared data: send message, invalidate and transfer cache line, etc.
  - Cost of synchronization: acquiring a lock costs even when it succeeds, blocking or spinning costs, etc.
  - Cost of unnecessary computation: redundant recomputation of data, computation of parallelism, computation cost of load rebalancing, etc.
- Each can be up to milliseconds
  - Waiting for a lock in Pthreads may yield to the OS!
- Tradeoff in the granularity of work:
  - Create enough parallel work for all cores, load balancing – finer granularity
  - But, fine granularity of work adds overhead, slows down the program

Locality effects magnified by parallelism

- Large memories are slow, small memories are faster
- The more cores, the larger the memory hierarchy
- But, more available fast caches
- Not every memory access costs the same
- Not only cache misses, but also accesses to “remote” data are more expensive
- Programs should work mostly with data in the local cache
- Controlling what is “local” depends on other cores too!
  - E.g., a write by a remote core invalidates data from the local cache

Load balance

- Load imbalance: some cores have nothing to do
  - Insufficient parallelism
  - Unequal work
- Predicting “equal work” may be difficult
  - Computations on trees or graphs
  - Computations that adapt granularity depending on data (e.g., galaxy simulation)
  - Unstructured problems
- Parallel algorithms need to balance work load among cores
  - May require additional computation, restructuring