18-742 Spring 2011
Parallel Computer Architecture
Lecture 4: Multi-core

Prof. Onur Mutlu
Carnegie Mellon University
Research Project

- Project proposal due: Jan 31

- Project topics
  - Does everyone have a topic?
  - Does everyone have a partner?
  - Does anyone have too many partners?
Last Lecture

- Programming model vs. architecture
  - Message Passing vs. Shared Memory
  - Data Parallel
  - Dataflow

- Generic Parallel Machine
Readings for Today

- **Required:**

- **Recommended:**
Reviews

- **Due Today (Jan 21)**

- **Due Next Tuesday (Jan 25)**

- **Due Next Friday (Jan 28)**
Recap of Last Lecture
Review: Shared Memory vs. Message Passing

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
  - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors except (multitasking uniprocessor)
    - Operations on shared data require synchronization
Programming Models vs. Architectures

- Five major models
  - Shared memory
  - Message passing
  - Data parallel (SIMD)
  - Dataflow
  - Systolic

- Hybrid models?
Scalability, Convergence, and Some Terminology
Scaling Shared Memory Architectures
Interconnection Schemes for Shared Memory

- Scalability dependent on interconnect
UMA/UCA: Uniform Memory or Cache Access

- All processors have the same uncontended latency to memory
- Latencies get worse as system grows
- Symmetric multiproducting (SMP) ~ UMA with bus interconnect
Uniform Memory/Cache Access

+ Data placement unimportant/less important (easier to optimize code and make use of available memory space)
- Scaling the system increases latencies
- Contention could restrict bandwidth and increase latency
Example SMP

- Quad-pack Intel Pentium Pro
How to Scale Shared Memory Machines?

- Two general approaches

- Maintain UMA
  - Provide a scalable interconnect to memory
  - Downside: Every memory access incurs the round-trip network latency

- Interconnect complete processors with local memory
  - NUMA (Non-uniform memory access)
    - Local memory faster than remote memory
  - Still needs a scalable interconnect for accessing remote memory
    - Not on the critical path of local memory access
NUMA/NUCA: NonUniform Memory/Cache Access

- Shared memory as local versus remote memory
  + Low latency to local memory
  - Much higher latency to remote memories
  + Bandwidth to local memory may be higher
  - Performance very sensitive to data placement
Example NUMA Machines

- Sun Enterprise Server
- Cray T3E
Convergence of Parallel Architectures

- Scalable shared memory architecture is similar to scalable message passing architecture
- Main difference: is remote memory accessible with loads/stores?
Historical Evolution: 1960s & 70s

• Early MPs
  – Mainframes
  – Small number of processors
  – crossbar interconnect
  – UMA
Historical Evolution: 1980s

• **Bus-Based MPs**
  - enabler: processor-on-a-board
  - economical scaling
  - precursor of today’s SMPs
  - UMA
Historical Evolution: Late 80s, mid 90s

- **Large Scale MPs (Massively Parallel Processors)**
  - multi-dimensional interconnects
  - each node a computer (proc + cache + memory)
  - both shared memory and message passing versions
  - NUMA
  - still used for “supercomp
Historical Evolution: Current

- Chip multiprocessors (multi-core)
- Small to Mid-Scale multi-socket CMPs
  - One module type: processor + caches + memory
- Clusters/Datacenters
  - Use high performance LAN to connect SMP blades, racks

- Driven by economics and cost
  - Smaller systems => higher volumes
  - Off-the-shelf components
- Driven by applications
  - Many more throughput applications (web servers)
  - ... than parallel applications (weather prediction)
  - Cloud computing
Historical Evolution: Future

- Cluster/datacenter on a chip?
- Heterogeneous multi-core?
- Bounce back to small-scale multi-core?
- ???
Multi-Core Processors
Moore’s Law

Multi-Core

- **Idea**: Put multiple processors on the same die.

- Technology scaling (Moore’s Law) enables more transistors to be placed on the same die area.

- What else could you do with the die area you dedicate to multiple processors?
  - Have a bigger, more powerful core
  - Have larger caches in the memory hierarchy
  - Simultaneous multithreading
  - Integrate platform components on chip (e.g., network interface, memory controllers)
Why Multi-Core?

- **Alternative: Bigger, more powerful single core**
  - Larger superscalar issue width, larger instruction window, more execution units, large trace caches, large branch predictors, etc

  + Improves single-thread performance transparently to programmer, compiler
  - Very difficult to design (Scalable algorithms for improving single-thread performance elusive)
  - Power hungry – many out-of-order execution structures consume significant power/area when scaled. Why?
  - Diminishing returns on performance
  - Does not significantly help memory-bound application performance (Scalable algorithms for this elusive)
Large Superscalar vs. Multi-Core


Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

Figure 3. Floorplan for the four-way single-chip multiprocessor.
Multi-Core vs. Large Superscalar

- Multi-core advantages
  + Simpler cores → more power efficient, lower complexity, easier to design and replicate, higher frequency (shorter wires, smaller structures)
  + Higher system throughput on multiprogrammed workloads → reduced context switches
  + Higher system throughput in parallel applications

- Multi-core disadvantages
  - Requires parallel tasks/threads to improve performance (parallel programming)
  - Resource sharing can reduce single-thread performance
  - Shared hardware resources need to be managed
  - Number of pins limits data supply for increased demand
Large Superscalar vs. Multi-Core


- Technology push
  - Instruction issue queue size limits the cycle time of the superscalar, OoO processor → diminishing performance
    - Quadratic increase in complexity with issue width
  - Large, multi-ported register files to support large instruction windows and issue widths → reduced frequency or longer RF access, diminishing performance

- Application pull
  - Integer applications: little parallelism?
  - FP applications: abundant loop-level parallelism
  - Others (transaction proc., multiprogramming): CMP better fit
Why Multi-Core?

- **Alternative: Bigger caches**
  
  + Improves single-thread performance transparently to programmer, compiler
  + Simple to design

  - Diminishing single-thread performance returns from cache size. Why?
  - Multiple levels complicate memory hierarchy
Cache vs. Core

- **Cache**
- **Microprocessor**
Why Multi-Core?

- **Alternative: (Simultaneous) Multithreading**

  + Exploits thread-level parallelism (just like multi-core)
  + Good single-thread performance when there is a single thread
  + No need to have an entire core for another thread
  + Parallel performance aided by tight sharing of caches

- Scalability is limited: need bigger register files, larger issue width (and associated costs) to have many threads → complex with many threads
- Parallel performance limited by shared fetch bandwidth
- Extensive resource sharing at the pipeline and memory system reduces both single-thread and parallel application performance