CS 395T: Topics in Multicore Programming

University of Texas, Austin
Fall 2009

Administration

• Instructors:
  – Keshav Pingali (CS, ICES)
    • 4.126A ACES
    • Email: pingali@cs.utexas.edu

• TA:
  – Aditya Rawal
    • Email: 83.aditya.rawal@gmail.com

Prerequisites

• Course in computer architecture
  – (e.g.) book by Hennessy and Patterson

• Course in compilers
  – (e.g.) book by Allen and Kennedy

• Self-motivation
  – willingness to learn on your own to fill in gaps in your knowledge

Why study parallel programming?

• Fundamental ongoing change in computer industry

• Until recently: Moore’s law(s)
  1. Number of transistors on chip double every 1.5 years
     • Transistors used to build complex, superscalar processors, deep pipelines, etc. to exploit instruction-level parallelism (ILP)
  2. Processor frequency doubles every 1.5 years
     • Speed goes up by factor of 10 roughly every 5 years
     ➔ Many programs ran faster if you just waited a while.

• Fundamental change
  – Micro-architectural innovations for exploiting ILP are reaching limits
  – Clock speeds are not increasing any more because of power problems
     ➔ Programs will not run any faster if you wait.

• Let us understand why.

Gordon Moore
(1) Micro-architectural approaches to improving processor performance

- Add functional units
  - Superscalar is known territory
  - Diminishing returns for adding more functional blocks
  - Alternatives like VLIW have been considered and rejected by the market
- Wider data paths
  - Increasing bandwidth between functional units in a core makes a difference
    - Such as comprehensive 64-bit design, but then where to?

(from Paul Teisch, AMD)

(1) Micro-architectural approaches (contd.)

- Deeper pipeline
  - Deeper pipeline buys frequency at expense of increased branch mis-prediction penalty and cache miss penalty
  - Deeper pipelines => higher clock frequency => more power
  - Industry converging on middle ground... 9 to 11 stages
    - Successful RISC CPUs are in the same range
- More cache
  - More cache buys performance until working set of program fits in cache
  - Exploiting caches requires help from programmer/compiler as we will see

(2) Processor clock speeds

- Old picture:
  - Processor clock frequency doubled every 1.5 years
- New picture:
  - Power problems limit further increases in clock frequency (see next couple of slides)
**Sources of Power Consumption**

**Dynamic**
- \( I_{\text{leakage}} \) (both devices conducting)

**Static**
- \( I_{\text{leakage}} \) (subthreshold, junction leakage)

---

**Technology Scaling**

- Dimensions reduced to increase performance and density
- \( V_{\text{CC}} \) decreases each generation...
  - Limit dynamic power
  - Limit electro fields
- requiring lower \( V_T \)
  - Gate overdrive = \( V_{\text{CC}} = V_T \)
- Leakage increases exponentially
  - \( P_{\text{static}} = V_{\text{CC}} I_{\text{leak}} \times \exp(-V_T) \)

---

Static current rises non-linearly as processors approach max frequency
Recap

• Old picture:
  1. Moore’s law(s):
     a. Number of transistors doubled every 1.5 years
     b. Processor clock frequency doubled every 1.5 years
     ➔ Many programs ran faster if you just waited a while.
  2. Use these to implement micro-architectural innovations for ILP

• New picture:
  1. Moore’s law
     a. Number of transistors still double every 1.5 years
     b. But micro-architectural innovations for ILP are flat-lining
     ➔ Programs will not run faster if you wait a while.

• Questions:
  1. Hardware: What do we do with all those extra transistors?
  2. Software: How do we keep speeding up program execution?

One hardware solution: go multicore

• Use semi-conductor tech improvements to build multiple cores without increasing clock frequency
  – does not require micro-architectural breakthroughs
  – non-linear scaling of power density with frequency will not be a problem

• Predictions:
  – from now on, number of cores will double every 1.5 years

Design choices

• Homogenous multicore processors
  – large number of identical cores

• Heterogenous multicore processors
  – cores have different functionalities

• It is likely that future processors will be heterogenous multicores
  – migrate important functionality into special-purpose hardware (eg. codecs)
  – much more power efficient than executing program in general-purpose core
  – trade-off: programmability
Problem: multicore software

- More aggregate performance for:
  - Multi-tasking
  - Transactional apps: many instances of same app
  - Multi-threaded apps (our focus)

- Problem
  - Most apps are not multithreaded
  - Writing multithreaded code increases software costs dramatically
    - factor of 3 for Unreal game engine (Tim Sweeney, EPIC games)

- The great multicore software question: Can we write programs so that performance doubles when the number of cores doubles?
- Very hard problem for many reasons (see later)

Amdahl's law
- Locality
- Overheads of parallel execution
- Load balancing
- ………

“We are at the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require… I have talked with a few people at Microsoft Research who say this is also at or near the top of their list [of critical CS research problems].” Justin Rattner, CTO Intel

Parallel Programming

- Community has worked on parallel programming for more than 30 years
  - programming models
  - machine models
  - programming languages
  - ………

- However, parallel programming is still a research problem
  - matrix computations, stencil computations, FFTs etc are well-understood
  - few insights for other applications
    - each new application is a “new phenomenon”

- Thesis: we need a science of parallel programming
  - analytic framework for thinking about parallelism in application
  - synthesis: produce an efficient parallel implementation of application

Analogy: science of electro-magnetism

Course objective

- Create a science of parallel programming
  - Structure:
    - understand the patterns of parallelism and locality in applications
  - Analysis:
    - abstractions for reasoning about parallelism and locality in applications
    - programming models based on these abstractions
    - tools for quantitative estimates of parallelism and locality
  - Synthesis:
    - exploiting structure to produce efficient implementations
Approach

- Niklaus Wirth’s aphorism:
  - Algorithms + Data structures = Programs
- Algorithms:
  - a description of the computation, expressed in terms of abstract data types (ADTs) like sets, matrices, and graphs
- Data structures:
  - concrete implementations of ADTs
    - (eg) matrices can be represented using arrays, space-filling curves, etc.
    - (eg) graphs can be represented using adjacency matrices, adjacency lists, etc.
- Strategy:
  - study parallelism and locality in algorithms, independent of concrete data structures
  - What structure can we exploit for efficient implementation?
  - study concrete parallel data structures required to support parallelism in algorithms
  - What structure can we exploit for efficient implementation?

Example: structure in algorithms

We will elaborate on this structure in a couple of weeks.

Course content

- Structure of parallelism and locality in important algorithms
  - computational science algorithms
  - graph algorithms
- Algorithm abstractions
  - dependence graphs
  - halographs
- Multicore architectures
  - interconnection networks, caches and cache coherence, memory consistency models, locks and lock-free synchronization
- Parallel data structures
  - linearizability
  - array and graph partitioning
  - lock-free data structures and transactional memory
- Optimistic parallel execution of programs
- Scheduling and load-balancing

Course content (contd.)

- Locality
  - spatial and temporal locality
  - cache blocking
  - cache-oblivious algorithms
- Static program analysis techniques
  - array dependence analysis
  - points-to and shape analysis
- Performance models
  - PRAM, BPRAM, logP
- Special topics
  - self-optimizing software and machine learning techniques for optimization
  - GPUs and GPU programming
  - parallel programming languages/libraries: Cilk, PGAS languages, OpenMP, TBBs, map-reduce, MPI
Course work

- Small number of programming assignments
- Paper presentations
- Substantial final project
- Participation in class discussions