CS 395T: Topics in Multicore Programming

University of Texas, Austin
Spring 2012

Administration

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

• TA:
  – Reza Mahjourian
    • Email: reza@cs.utexas.edu

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

Course Material

• All course material in online at this URL:
  http://www.cs.utexas.edu/~pingali/CS395T/2012sp/

• Lots of material on the web
  – you are encouraged to find and study relevant material on your own
  – if you find a really useful paper or webpage for some topic, let me know
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.

(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?

(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**
- $C \frac{dV}{dt}$: charging of capacitive load

**Static**
- $I_{leakage}$: subthreshold, junction leakage

**Technology Scaling**
- $V_{CC}$ decreases each generation...
  - Limit dynamic power
  - Limit electric fields
- $V_{T}$: Gate overdrive = $V_{CC} - V_{T}$
- Leakage increases exponentially
  - $P_{static} = V_{CC} I_{leak} = \exp(-V_{T})$
Recap

- **Old picture:**
  - Moore’s law(s):
    1. Number of transistors doubled every 1.5 years
    2. Processor clock frequency doubled every 1.5 years
  - Many programs ran faster if you just waited a while.

- **New picture:**
  - Number of transistors still double every 1.5 years
  - But micro-architectural innovations for ILP are flat-lining
  - Processor clock frequencies are not increasing very much
  - Programs will not run faster if you wait a while.

- **Questions:**
  - Hardware: What do we do with all those extra transistors?
  - 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 (e.g., 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
    – [Tim Sweeney, EPIC games]
  – The great multicore software quest: 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 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
  – analysis: framework for thinking about parallelism in application
  – synthesis: produce an efficient parallel implementation of application

“We are 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

Analogy: science of electro-magnetism

“Seemingly unrelated phenomena Unifying abstractions Specialized models that exploit structure

“The Alchemist” Cornelius Bega (1663)
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
  - operator formulation of algorithms
- 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
- 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
  - approximate computing

**Course work**

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