CS 378/CSE 392: Programming for Performance

Administration

- Instructors:
  - Keshav Pingali (Professor, CS department & ICES)
    - 4.126 ACES
    - Email: pingali@cs.utexas.edu
  - Andrew Lenharth (Research Associate, ICES)
    - 4.124 ACES
    - Email: lenharth@ices.utexas.edu
- TA:
  - Makarand Damle (Grad student, CS department)
    - Email: mdamle@cs.utexas.edu

Prerequisites

- Knowledge of basic computer architecture
  - (e.g.) PC, ALU, cache, memory
- Basic calculus and linear algebra
  - differential equations and matrix operations
- Software maturity
  - assignments will be in C/C++ on Linux computers
  - ability to write medium-sized programs (~1000 lines)
- Self-motivation
  - willingness to experiment with systems

Coursework

- 4 or 5 programming projects
  - These will be more or less evenly spaced through the semester
  - Some assignments will also have short questions
- One or two mid-semester exams
- Final project
  - Substantial project that you should plan to start work on by the spring break

Text-book for course

No official book for course

This book is a useful reference.

"Parallel programming in C with MPI and OpenMP", Michael Quinn, McGraw-Hill Publishers. ISBN 0-07-282256-2

Lots of material on the web

What this course is not about

- This is not a tools/libraries course
  - We will use a small number of tools and micro-benchmarks to understand performance, but this is not a course on how to use tools and libraries
- This is not a clever hacks course
  - We are interested in general scientific principles for performance programming, not in squeezing out every last cycle for somebody's favorite program

What this course IS about

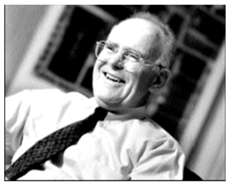
- Hardware guys invent lots of hardware features that can boost program performance
- However, software can only exploit these features if it is written carefully to do that
- Our agenda:
  - understand key performance-critical architectural features at a high level
  - develop general principles and techniques that can guide us in writing programs for exploiting those architectural features
- Major focus:
  - parallel architectures and parallel programming
    - multicore processors
    - distributed-memory computers
  - managing the memory hierarchy
    - high-performance programs must also exploit caches effectively
    - exploiting locality is as important as exploiting parallelism

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

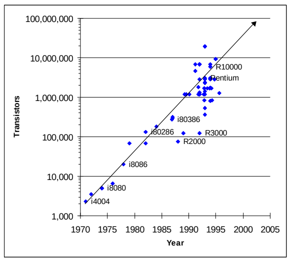
  ➔ Many programs ran faster if you just waited a while.
- Fundamental change
  - Micro-architectural innovations for exploiting ILP are reaching limits
  - Clock speeds are not increasing any more because of power problems

  ➔ 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?

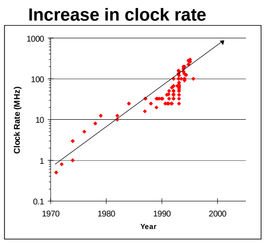


(1) Micro-architectural approaches (contd.)

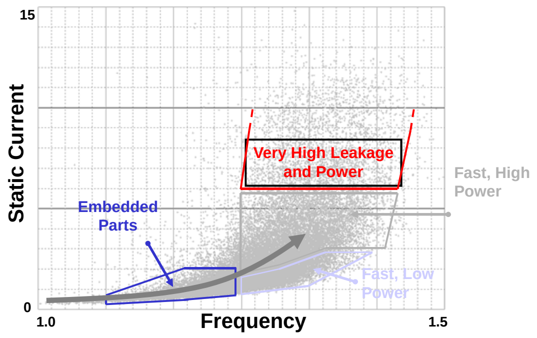
- Deeper pipeline
  - Deeper pipeline buys frequency at expense of increased branch mis-prediction penalty and cache miss penalty
  - Deeper pipelines => higher clock frequency => more power
  - Industry converging on middle ground…9 to 11 stages
    - Successful RISC CPUs are in the same range
- More 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)

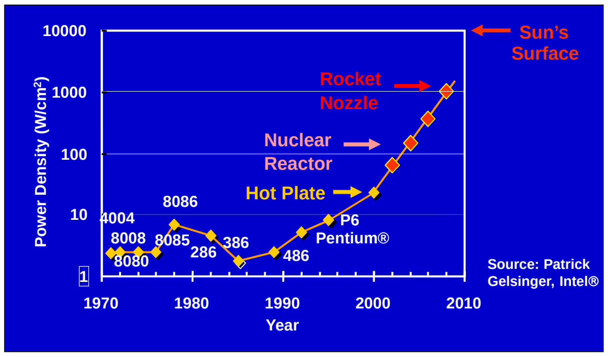


(2) Processor clock speeds (contd.)



Static current rises non-linearly as processors approach max frequency

(2) Processor clock speeds (contd.)



Recap

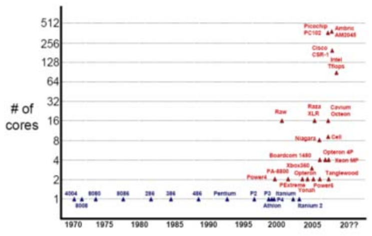
- Old picture:
  - Moore's law(s):
    1. Number of transistors doubled every 1.5 years
       - Use these to implement micro-architectural innovations for ILP
    2. Processor clock frequency doubled every 1.5 years

  ➔ Many programs ran faster if you just waited a while.
- New picture:
  - Moore's law
    1. 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



(from Saman Amarasinghe, MIT)

New problem: multi-core software

- More aggregate performance for:
  - Multi-threaded apps (our focus)
  - Transactions: many instances of same app
  - Multi-tasking
- 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 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
  - 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, Senior Fellow, Intel

Amdahl’s Law

- Simple observation that shows that unless most of the program can be executed in parallel, the benefits of parallel execution are limited
  - serial portions f program become bottleneck
- Analogy: suppose I go from Austin to Houston at 60 mph, and return infinitely fast. What is my average speed?
  - Answer: 120 mph, not infinity

Amdahl’s Law (details)

- In general, program will have both parallel and serial portions
  - Suppose program has N operations
    - r*N operations in parallel portion
    - (1-r)*N operations in serial portion
- Assume
  - Serial execution requires one time unit per operation
  - Parallel portion can be executed infinitely fast by multicore processor, so it takes zero time to execute.
- Speed-up:

$$\frac{\text{(execution time on single core)}}{\text{(execution time on multicore)}} = \frac{N}{(1-r)*N} = \frac{1}{(1-r)}$$

- Even if r = 0.9, speed-up is only 10.

Our focus

- Multi-threaded programming
  - also known as *shared-memory* programming
  - application program is decomposed into a number of “threads” each of which runs on one core and performs some of the work of the application: “many hands make light work”
  - threads communicate by reading and writing memory locations (that’s why it is called shared-memory programming)
  - we will use a popular system called OpenMP
- Key issues:
  - how do we assign work to different threads?
  - how do we ensure that work is more or less equitably distributed among the threads?
  - how do we make sure threads do not step on each other (synchronization)?
  - ….

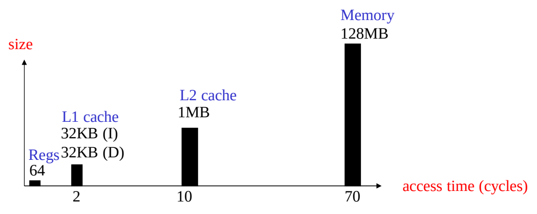
Distributed-memory programming

- Some application areas such as computational science need more power than will be available in the near future on a multi-core processor
- Solution: connect a bunch of multicore processors together
  - (e.g.) Ranger machine at Texas Advanced Computing Center (TACC): 15,744 processors, each of which has 4 cores
- Must use a different model of parallel programming called
  - *message-passing* (or)
  - *distributed-memory programming*
- Distributed-memory programming
  - units of parallel execution are called processes
  - processes communicate by sending and receiving messages since they have no memory locations in common
  - most-commonly-used communication library: MPI
- We will study distributed-memory programming as well and you will get to run programs on Ranger

Software problem (II): memory hierarchy

- Complication for parallel software
  - unless software also exploit caches, overall performance is usually poor
  - writing software that can exploit caches also complicates software development

Memory Hierarchy of SGI Octane



- R10 K processor:
  - 4-way superscalar, 2 fpo/cycle, 195MHz
- Peak performance: 390 Mflops
- Experience: sustained performance is less than 10% of peak
  - Processor often stalls waiting for memory system to load data

Software problem (II)

- Caches are useful only if programs have locality of reference
  - temporal locality: program references to given memory address are clustered together in time
  - spatial locality: program references clustered in address space are clustered in time
- Problem:
  - Programs obtained by expressing most algorithms in the straight-forward way do not have much locality of reference
  - How do we code applications so that they can exploit caches?.

Software problem (II): memory hierarchy

"…The CPU chip industry has now reached the point that instructions can be executed more quickly than the chips can be fed with code and data. Future chip design is memory design. Future software design is also memory design. ….

Controlling memory access patterns will drive hardware and software designs for the foreseeable future."

Richard Sites, DEC

Abstract questions

- Do applications have parallelism?
- If so, what patterns of parallelism are there in common applications?
- Do applications have locality?
- If so, what patterns of locality are there in common applications?
- We will study sequential and parallel algorithms and data structures to answer these questions

Course content

- Analysis of applications that need high end-to-end performance
- Understanding parallel performance: DAG model of computation, Moore's law, Amdahl's law
- Measurement and the design of computer experiments
- Micro-benchmarks for abstracting performance-critical aspects of computer systems
- Memory hierarchy:
  - caches, virtual memory
  - optimizing programs for memory hierarchies
  - cache-oblivious programming
- ……..

Course content (contd.)

- …..
- GPUs and GPU programming
- Multi-core processors and shared-memory programming
  - OpenMP
  - Galois
- Distributed-memory machines and message-passing programming
  - MPI
- Advanced topics:
  - Optimistic parallelism
  - Self-optimizing software
    - ATLAS,FFTW
  - Google map-reduce