EECS 570
Lecture 1
Parallel Computer Architecture
Winter 2018
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Austin, Adve, Falsafi, Martin, Narayanasamy, Nowatzyk, Peh, and Wenisch of CMU, EPFL, MIT, UPenn, U-M, UIUC.
Announcements

No discussion this Friday.

Online quizzes (Canvas) on 1st readings due Monday, 1:30pm.

Sign up for piazza.
Readings

For Monday 1/8  (quizzes due by 1:00pm)


For Wednesday 1/10:

EECS 570 Class Info

Instructor: Professor Satish Narayanasamy
- URL: http://www.eecs.umich.edu/~nsatish

Research interests:
- Parallel systems and architecture
- Accelerators for genomics
- Secure systems using hardware and program analysis

GSI:
- Subhankar Pal (subh)

Class info:
- URL: http://www.eecs.umich.edu/courses/eecs570/
- Canvas for reading quizzes & reporting grades
- Piazza for discussions & project coordination
Meeting Times

Lecture
- MW 1:40pm – 3:00pm (1017 Dow)

Discussion
- F 1:40pm – 2:30pm (1200 EECS); starts on 1/12
- Talk about programming assignments and projects
- Make-up lectures
- Keep the slot free, but we often won’t meet

Office Hours
- Prof. Narayanasamy: M 3-4 (4721 CSE) & by appt.
- Subhankar: Tue, Thu 4-5p (Location: BBB atrium) starting 1/9
  - Fri 1:30-2:30p (Location: BBB atrium) when no discussion

Q&A
- Use Piazza for all technical questions
- Use e-mail sparingly
Who Should Take 570?

Graduate Students (& seniors interested in research)
1. Computer architects to be
2. Computer system designers
3. Those interested in computer systems

Required Background
- Computer Architecture (e.g., EECS 470)
- C / C++ programming
Grading

2 Prog. Assignments: 5% & 15%
Reading Quizzes: 10%
Midterm exam: 20%
Final exam: 25%
Final Project: 25%

Attendance & participation count
(your goal is for me to know who you are)
Grading (Cont.)

Group studies are encouraged

Group discussions are encouraged

All programming assignments must be results of individual work

All reading quizzes must be done individually, questions/answers should not be posted publicly

There is no tolerance for academic dishonesty. Please refer to the University Policy on cheating and plagiarism. Discussion and group studies are encouraged, but all submitted material must be the student's individual work (or in case of the project, individual group work).
Exam dates

• Midterm
  ❑ 21\textsuperscript{st} Feb 2018 in class

• Final
  ❑ Tue, April 24\textsuperscript{th} 4-6pm
Some Advice on Reading...

If you carefully read every paper start to finish...
...you will never finish

*Learn to skim past details*
Reading Quizzes

• You must take an online quiz for every paper
  Quizzes must be completed by class start via Canvas

• There will be 2 multiple choice questions
  ❑ The questions are chosen randomly from a list
  ❑ You only have 5 minutes
    ❒ Not enough time to find the answer if you haven’t read the paper
  ❑ You only get one attempt

• Some of the questions may be reused on the midterm/final

• 4 lowest quiz grades (of about 40) will be dropped over the
  course of the semester (e.g., skip some if you are travelling)
  ❑ Retakes/retries/reschedules will not be given for any reason
Final Project

• **Original** research on a topic related to the course
  - Goal: a high-quality 6-page workshop paper by end of term
  - 25% of overall grade
  - Done in groups of 3-4
  - Poster session - April 26, 10:30am-12:30pm

• See course website for timeline

• **Available infrastructure**
  - *FeS2* and *M5* multiprocessor simulators
  - GPGPU*sim*
  - Pin
  - Xeon Phi accelerators

• **Suggested topic list will be distributed in a few weeks**
  You may propose other topics if you convince me they are worthwhile
Course Outline

Unit I – Parallel Programming Models and Applications
- Message passing, shared memory. Data parallelism.
- Parallel applications
- Synchronization, Locks and Transactional Memory

Unit II – Coherency and Consistency
- Snoopy and directory protocols
- Memory Models

Unit III – Interconnection Networks
- On-chip and off-chip networks

Unit IV – Accelerators
- GPUs, machine learning, genomics, in-memory

Unit V – Modern & Unconventional Multiprocessors
- SMT, Tera, Quantum
Parallel Computer Architecture

The Multicore Revolution
Why is it happening?
If you want to make your computer faster, there are only two options:

1. increase clock frequency

2. execute two or more things in parallel

   Instruction-Level Parallelism (ILP)

   Programmer specified explicit parallelism
• 6-issue has higher IPC than 2-issue, but not by 3x
  - Memory (I & D) and dependence (pipeline) stalls limit IPC

Olukotun et al ASPLOS 96
Single-thread performance

<table>
<thead>
<tr>
<th>Year</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985</td>
<td>1</td>
</tr>
<tr>
<td>1990</td>
<td>10</td>
</tr>
<tr>
<td>1995</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td>2005</td>
<td>10000</td>
</tr>
<tr>
<td>2010</td>
<td>100000</td>
</tr>
</tbody>
</table>


**Conclusion:** Can’t scale MHz or issue width to keep selling chips

*Hence, multicore!*
The Power Wall

- Transistors (100,000's)
- Power (W)
- Performance (GOPS)
- Efficiency (GOPS/W)

Limits on heat extraction
Limits on energy-efficiency of operations
The Power Wall

- Transistors (100,000's)
- Power (W)
- Performance (GOPS)
- Efficiency (GOPS/W)

Era of High Performance Computing

- Stagnates performance growth
- Limits on heat extraction
- Limits on energy-efficiency of operations

Era of Energy-Efficient Computing

c. 2000
Classic CMOS Dennard Scaling: the science behind Moore’s Law

Scaling:

Voltage: \( V/\alpha \)
Oxide: \( t_{\text{ox}}/\alpha \)
Wire width: \( W/\alpha \)
Gate width: \( L/\alpha \)
Diffusion: \( x_d/\alpha \)
Substrate: \( \alpha N_A \)

Results:

Higher Density: \( \sim \alpha^2 \)
Higher Speed: \( \sim \alpha \)
Power/ckt: \( 1/\alpha^2 \)
Power Density: \( \sim \text{Constant} \)

Source: Future of Computing Performance: Game Over or Next Level?, National Academy Press, 2011


\[ P = C V^2 f \]
Post-classic CMOS Dennard Scaling

Post Dennard CMOS Scaling Rule

Scaling:
- Voltage: $\frac{V}{\alpha}$
- Oxide: $\frac{t_{\text{ox}}}{\alpha}$
- Wire width: $\frac{W}{\alpha}$
- Gate width: $\frac{L}{\alpha}$
- Diffusion: $\frac{x_d}{\alpha}$
- Substrate: $\alpha N_A$

Results:
- Higher Density: $\sim \alpha^2$
- Higher Speed: $\sim \alpha$
- Power/ckt: $\frac{1}{\alpha^2}$
- Power Density: $\sim \frac{\alpha^2}{\text{Constant}}$

$P = C V^2 f$

Leakage Killed Dennard Scaling

Leakage:
• Exponential in inverse of $V_{th}$
• Exponential in temperature
• Linear in device count

To switch well
• must keep $V_{dd}/V_{th} > 3$

$\Rightarrow V_{dd}$ can’t go down
Multicore: Solution to Power-constrained design?

Power = $CV^2F$  \( F \propto V \)
Scale clock frequency to 80%

Now add a second core

Roughly same power budget, but 1.6x performance!

But:
- Must parallelize application
- Remember Amdahl’s Law!
What Is a Parallel Computer?

“A collection of processing elements that communicate and cooperate to solve large problems fast.”

Almasi & Gottlieb, 1989
Spectrum of Parallelism

Why multiprocessing?

- Desire for performance
- Techniques from 370/470 difficult to scale further
Why Parallelism Now?

- These arguments are no longer theoretical
- All major processor vendors are producing multicore chips
  - Every machine will soon be a parallel machine
  - All programmers will be parallel programmers???
- New software model
  - Want a new feature? Hide the “cost” by speeding up the code first
  - All programmers will be performance programmers???
- Some may eventually be hidden in libraries, compilers, and high level languages
  - But a lot of work is needed to get there
- Big open questions:
  - What will be the killer apps for multicore machines?
  - How should the chips, languages, OS be designed to make it easier for us to develop parallel programs?
Multicore in Products

- “We are dedicating all of our future product development to multicore designs. ... This is a sea change in computing”
  
  Paul Otellini, President, Intel (2005)

- All microprocessor companies switch to MP (2X cores / 2 yrs)

<table>
<thead>
<tr>
<th></th>
<th>Intel’s Nehalem-EX</th>
<th>Azul’s Vega</th>
<th>nVidia’s Tesla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processors/System</td>
<td>4</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Cores/Processor</td>
<td>8</td>
<td>48</td>
<td>448</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Threads/System</td>
<td>64</td>
<td>768</td>
<td>1792</td>
</tr>
</tbody>
</table>
Revolution Continues..

Azul’s Vega 3 7300
54-core chip
864 cores
768 GB Memory
May 2008

Blue Gene/Q Sequoia
16-core chip
1.6 million cores
1.6 PB
2012

Sun’s Modular DataCenter ‘08
8-core chip, 8-thread/core
816 cores / 160 sq.feet

Lakeside Datacenter (Chicago)
1.1 million sq.feet
~45 million threads
Multiprocessors Are Here To Stay

• Moore’s law is making the multiprocessor a commodity part
  ❞ 1B transistors on a chip, what to do with all of them?
  ❞ Not enough ILP to justify a huge uniprocessor
  ❞ Really big caches? $t_{hit}$ increases, diminishing $%_{miss}$ returns

• Chip multiprocessors (CMPs)
  ❞ Every computing device (even your cell phone) is now a multiprocessor
Parallel Programming Intro
Motivation for MP Systems

• Classical reason for multiprocessing:
  More performance by using multiple processors in parallel

  ❑ Divide computation among processors and allow them to work concurrently

  ❑ Assumption 1: There is parallelism in the application

  ❑ Assumption 2: We can exploit this parallelism
Finding Parallelism

1. Functional parallelism
   - Car: \{engine, brakes, entertain, nav, \ldots\}
   - Game: \{physics, logic, UI, render, \ldots\}
   - Signal processing: \{transform, filter, scaling, \ldots\}

2. Data parallelism
   - Vector, matrix, db table, pixels, \ldots

3. Request parallelism
   - Web, shared database, telephony, \ldots
Computational Complexity of (Sequential) Algorithms

- Model: Each step takes a unit time

- Determine the time (space) required by the algorithm as a function of input size
Sequential Sorting Example

• Given an array of size n

• MergeSort takes $O(n \log n)$ time
• BubbleSort takes $O(n^2)$ time

• But, a BubbleSort implementation can sometimes be faster than a MergeSort implementation
• Why?
Sequential Sorting Example

• Given an array of size n

• MergeSort takes $O(n \log n)$ time

• BubbleSort takes $O(n^2)$ time

• But, a BubbleSort implementation can sometimes be faster than a MergeSort implementation

• The model is still useful
  ▶ Indicates the scalability of the algorithm for large inputs
  ▶ Lets us prove things like a sorting algorithm requires at least $O(n \log n)$ comparisons
We need a similar model for parallel algorithms
Sequential Merge Sort

16MB input (32-bit integers)

Recurse (left)

Recurse (right)

Sequential Execution

Merge to scratch array

Copy back to input array
Parallel Merge Sort (as Parallel Directed Acyclic Graph)

16MB input (32-bit integers)

Recurse(left)  Recurse(right)

Merge to scratch array

Copy back to input array

Parallel Execution
Parallel DAG for Merge Sort (2-core)
Parallel DAG for Merge Sort (4-core)
Parallel DAG for Merge Sort (8-core)
The DAG Execution Model of a Parallel Computation

- Given an input, dynamically create a DAG

- Nodes represent sequential computation
  - Weighted by the amount of work

- Edges represent dependencies:
  - Node A → Node B means that B cannot be scheduled unless A is finished
Sorting 16 elements in four cores
Sorting 16 elements in four cores
(4 element arrays sorted in constant time)
Performance Measures

• Given a graph G, a scheduler S, and P processors

• $T_p(S)$ : Time on P processors using scheduler S

• $T_p$ : Time on P processors using best scheduler

• $T_1$ : Time on a single processor (sequential cost)

• $T_\infty$ : Time assuming infinite resources
Work and Depth

- $T_1 = \text{Work}$
  - The total number of operations executed by a computation

- $T_\infty = \text{Depth}$
  - The longest chain of sequential dependencies (critical path) in the parallel DAG
$T_\infty$ (Depth): Critical Path Length (Sequential Bottleneck)
$T_1 (work)$: Time to Run Sequentially
Sorting 16 elements in four cores
(4 element arrays sorted in constant time)
Some Useful Theorems
Work Law

• “You cannot avoid work by parallelizing”

\[ \frac{T_1}{P} \leq T_p \]
Work Law

• “You cannot avoid work by parallelizing”

\[ \frac{T_1}{P} \leq T_P \]

Speedup = \[ \frac{T_1}{T_P} \]
Work Law

• “You cannot avoid work by parallelizing”

\[ \frac{T_1}{P} \leq T_P \]

Speedup = \( \frac{T_1}{T_P} \)

• Can speedup be more than 2 when we go from 1-core to 2-core in practice?
Depth Law

• More resources should make things faster
• You are limited by the sequential bottleneck

\[ T_P \geq T_\infty \]
Amount of Parallelism

Parallelism = $T_1 / T_\infty$
Maximum Speedup Possible

Speedup \( \frac{T_1}{T_P} \leq \frac{T_1}{T_\infty} \) Parallelism

“speedup is bounded above by available parallelism”
Greedy Scheduler

- If more than P nodes can be scheduled, pick any subset of size P
- If less than P nodes can be scheduled, schedule them all
Theorem [Graham '68 & Brent '75]. Any greedy scheduler achieves

\[ T_P \leq \frac{T_1}{P} + T_\infty. \]

Proof.

- \# complete steps \( \leq \frac{T_1}{P} \), since each complete step performs \( P \) work.
- \# incomplete steps \( \leq T_\infty \), since each incomplete step reduces the span of the unexecuted dag by 1.

Performance of the Greedy Scheduler

\[ T_P(\text{Greedy}) \leq \frac{T_1}{P} + T_\infty \]

Work law \[ \frac{T_1}{P} \leq T_P \]

Depth law \[ T_\infty \leq T_P \]
Greedy is optimal within factor of 2

\[ T_P \leq T_P(\text{Greedy}) \leq 2\ T_P \]

Work law \[ \frac{T_1}{P} \leq T_P \]

Depth law \[ T_\infty \leq T_P \]
Work/Depth of Merge Sort (Sequential Merge)

- Work \( T_1 \): \( O(n \log n) \)
- Depth \( T_\infty \): \( O(n) \)
  - Takes \( O(n) \) time to merge \( n \) elements

- Parallelism:
  - \( T_1 / T_\infty = O(\log n) \) \( \rightarrow \) really bad!
Main Message

• Analyze the Work and Depth of your algorithm
• Parallelism is Work/Depth
• Try to decrease Depth
  ❑ the critical path
  ❑ a *sequential* bottleneck
• If you increase Depth
  ❑ better increase Work by a lot more!
Amdahl’s law

• Sorting takes 70% of the execution time of a sequential program

• You replace the sorting algorithm with one that scales perfectly on multi-core hardware

• How many cores do you need to get a 4x speed-up on the program?
Amdahl’s law, $f = 70\%$

Speedup$(f, c) = \frac{1}{1-f} + \frac{f}{c}$

- $f$ = the **parallel** portion of execution
- $1 - f$ = the **sequential** portion of execution
- $c$ = number of cores used
Amdahl’s law, $f = 70\%$
Amdahl's law, \( f = 70\% \)

Desired 4x speedup

Speedup achieved (perfect scaling on 70%)

Limit as \( c \to \infty \) = \( 1/(1-f) \) = 3.33
Amdahl’s law, $f = 10\%$

Amdahl’s law limit, just 1.11x

Speedup achieved with perfect scaling

Speedup

#cores

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Amdahl’s law, $f = 98\%$
Lesson

- Speedup is limited by **sequential** code

- Even a small percentage of **sequential** code can greatly limit potential speedup