Educational Foundations for Parallel and Distributed Computing

Randal E. Bryant
School of Computer Science
Carnegie Mellon University
Parallel Computing Platforms

Supercomputers

- Apply massive computation to single problem
- Network-connected processing nodes
- Each node combines CPUs + GPUs

Consumer Devices

- Perform routine tasks with minimum energy
- Multiple processors, each with SIMD vector units
- Multiple GPUs
Distributed Computing Resources

Internet Data Center

- Servers to support millions of customers
- Designed for data collection, storage, and analysis

Cloud Services

- Access to scalable distributed computing
- Students can experience the real deal
Undergraduate Curriculum Goals in Parallel & Distributed Systems

Overall Objectives

- Prepare students for current and future computing platforms
- Parallelism required for performance
- Distributed computing required to provide reliable and scalable services

Diagram:

- CS 440 Distributed Systems
- CS 418 Parallel Computing
- Advanced Electives
- CS 213 Introduction to Computer systems
- CS 210 Data structs. & algorithms
- Core
- CS 122 Imperative computation
- CS 150 Functional programming
- Preparation
- CS 112 Introductory programming
- Math 127 Discrete math
- Introduction
Computational Thinking

- Jeannette Wing, CACM 2006

Algorithmic Thinking
- Refine concrete task into abstract algorithmic problem
- Derive solutions with predictable worst / average case performance

Logical Thinking
- Construct system as set of layered components, each with abstract interfaces
- Reason about program execution: invariants, loops, recursion, …

Systems Thinking
- Designing reliable systems using unreliable components
- Optimize for the common case
Some Curriculum Design History

Introductory Computer Systems

- Introduced in 1998
- To support upper-level systems courses

Sequential & Parallel Data Structures & Algorithms

- Introduced in 2011
- Part of larger undergraduate curriculum revision
  - Stress computational thinking at all levels

Other Issues

- Compatibility with other majors (esp., ECE & Math)
Sequential Programming Preparation

Objectives for Students

- Understand multiple programming models
- Able to reason (semi-formally) about program behavior and performance
- Familiar with basic data structures and algorithms
CS 122 Principles of Imperative Computation

- Writing and reasoning about imperative programs
  - E.g., how to formulate loop invariants
- Taken by ~350 students per year
  - CS + ECE + a few others
- Uses homegrown language C0
  - Similar to C, but type safe
    - No pointer arithmetic
    - No casting
    - Allocate data with specified type
CS 122 Theme

Must approach programming task from multiple viewpoints

Computational Thinking
- Algorithmic principles
  - Invariants, Pre- and Post-conditions

Programming
- Data organization
- Good coding style

Data Structures & Algorithms
- Commonly used building blocks

Computational Thinking

Data structures & algorithms

Programming
Computational Thinking
- Shape invariant
- Ordering invariant

Programming
- Embed tree in array
- Children of node $i$ at $2i$ & $2i+1$

Data Structures & Algorithms
- Efficient insertion & deletion
Data Structure Declaration

```c
struct heap {
    int limit;       /* limit > 0 */
    int next;        /* 1 <= next && next <= limit */
    int[] values;    /* \length(values) == limit */
};
```

```
next = 6
Limit = 7
values
limit
next
```

```
2  4  3  9  7  8
9  7  8
```

```
2
9  7  8
```

```
2  4  3  9  7  8
```
Writing & Checking Assertions

Predicate to Test Heap Property

```c
bool is_heap(heap H)
//@requires H != NULL && \length(H->values) == H->limit;
{
    /* Heap bounds */
    if (!(1 <= H->next && H->next <= H->limit)) return false;
    for (int i = 2; i < H->next; i++)
        /* Parent value <= Child value */
        if (!(H->values[i/2] <= H->values[i])) return false;
    return true;
}
```

Write as Functions in C0

- Not a specialized notation
- Can check dynamically during runtime

Experience

- Helps students learn to think about their programs systematically
CS 150 Principles of Functional Programming

- Concepts of functional programs
  - Looks more like math & less like programs students have written before
  - Must have solid understanding of mathematical foundations
    » E.g., sets & functions

- Based on earlier, 200-level course
  - Question: Would freshmen be able to handle this material?
    » Answer: Yes!

- Programs written in Standard ML
The Power of Functional Languages

Inserting value into ordered list

\[ \text{ins} : (\alpha \times \alpha \rightarrow \text{order}) \rightarrow (\alpha \times \alpha \text{ list}) \rightarrow \alpha \text{ list} \]

- Over data of arbitrary data type
- Given order function:
  - Mapping from \((x, y)\) to \{ LESS, EQUAL, GREATER \}
- Function \text{ins} generates function
  - Takes value & list and returns new list

Important Concepts

- Polymorphism
  - Single function declaration applies to any data type
  - As long as it has defined comparison function
- Higher order functions
  - Pass functions as arguments
  - Generate functions as return values
fun ins cmp (x, [ ]) = [x]  \quad \text{Insertion into empty list}
| ins cmp (x, y::L) =  \quad \text{Insertion into list with head } y
  \quad \text{Comparing } x \text{ to } y:\n  \quad \text{case } cmp(x, y) \text{ of}
  \quad \quad \text{GREATER } \Rightarrow y::\text{ins cmp } (x, L) \quad x > y
  \quad \quad _ \quad \Rightarrow x::y::L; \quad x \leq y

Empty List $\quad$ \rightarrow \quad x

$x > y$

\begin{align*}
\text{y} & \quad \text{L} \\
\text{y} & \quad \text{ins cmp } (x,\text{L})
\end{align*}

$x \leq y$

\begin{align*}
\text{y} & \quad \text{L} \\
\text{x} & \quad \text{y} & \quad \text{L}
\end{align*}
CS 213: Introduction to Computer Systems

Summary

- Randal E. Bryant & David R. O’Hallaron, 1998
- Introduce systems from programmer’s perspective
  - What aspects of a system must a sophisticated application programmer understand?
- Draws material from computer architecture, compilers, operating systems, and networking

Textbook

- Computer Systems: A Programmer’s Perspective
- Widespread adoption
Builder’s Perspective

- Assume programs are fixed, but hardware may be changed
- Must make many difficult design decisions
- Complex tradeoffs and interactions between components
Memory System
Programmer’s Perspective

- Hardware is fixed, but programs may change
- Must understand both hardware and program mapping
  - Hierarchical memory organization
  - How multidimensional array mapped onto memory

```c
void copyi(int src[2048][2048],
           int dst[2048][2048])
{
    int i,j;
    for (i = 0; i < 2048; i++)
        for (j = 0; j < 2048; j++)
            dst[i][j] = src[i][j];
}
```

```
void copyj(int src[2048][2048],
           int dst[2048][2048])
{
    int i,j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

5.2 ms

162 ms

(Measured on 2.7 GHz Intel Core i7)
The Memory Mountain
Intro to Computer Systems at CMU

Goals
- Teach students to be sophisticated application programmers
- Prepare students for upper-level systems courses

Taught every semester to 500+ students
- All CS undergrads (core course)
- All ECE undergrads (core course)
- Many masters students
  - To prepare them for upper-level systems courses
- Variety of others from math, physics, statistics, ...

Logistics
- 3 instructors: 1 lecture + video
- 22 teaching assistants
- Automated systems for assignment grading, exam delivery & grading
Labs

Key teaching insight:

- Cool Labs ⇒ Great Course

A set of 1 and 2 week labs define the course.

Guiding principles:

- Be hands on, practical, and fun.
- Be interactive, with continuous feedback from automatic graders
- Find ways to challenge the best while providing worthwhile experience for the rest
- Use healthy competition to maintain high energy.
Shell Lab

Goal: Write a Unix shell with job control
  - (e.g., ctrl-z, ctrl-c, jobs, fg, bg, kill)

Lessons:
  - First introduction to systems-level programming and concurrency
    - Asynchronous events
    - Race conditions
  - Learn about processes, process control, signals, and catching signals with handlers

Infrastructure
  - Students use a scripted autograder to incrementally test functionality in their shells
Proxy Lab

Goal: write concurrent Web proxy.

Ties together many ideas from course
- Data representations, byte ordering, memory management, concurrency, processes, threads, synchronization, signals, I/O, network programming, application-level protocols (HTTP)

Infrastructure:
- Implement with sockets and pthreads
- Plugs directly between existing browsers and Web servers
- Grading is done via autograders and one-on-one demos
- Very exciting for students, great way to end the course
CMU Courses that Build on ICS

- CS
  - Parallel Systems
  - Dist. Systems
  - Networks
  - Operating Systems
  - Storage Systems
  - Databases
  - Compilers
  - Secure Coding
  - Software Engin.

- Robotics
  - Cog. Robotics
  - Comp. Photo.
  - Computer Graphics

- ECE
  - Embedded Control
  - Real-Time Systems
  - Embedded Systems
  - Computer Arch.

ICS
Shameless Promotion

- http://csapp.cs.cmu.edu
- Third edition published 2015
- In use at 297+ institutions worldwide
Worldwide Adoptions

297 total
CS 210: Sequential & Parallel Data Structures & Algorithms

Guy Blelloch

Performance Parameters

- Work $W$ Total number of operations required
- Span $S$ Time required if had unlimited parallelism
- Processors $P$ Number of available processors

Area $= W$
Execution Time

\[ T = \text{Max}(S, \frac{W}{P}) \]

- Unlimited Parallelism: \[ T = S \]
- Limited Parallelism: \[ T = \frac{W}{P} \]
- Sequential Execution: \[ T = W \]

Area = W
Obfuscating Parallelism

```java
public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
        while (a[++i] < a[right]);
        while (a[right]<a[--j])
            if (j==left) break;
        if (i >= j) break;
        swap(a,i,j); }
    swap(a, i, right);
    quickSort(a, left, i - 1);
    quickSort(a, i+1, right); }
```
procedure QUICKSORT(S):
if S contains at most one element then return S
else
begin
choose an element a randomly from S;
let $S_1$, $S_2$ and $S_3$ be the sequences of elements in S less than, equal to, and greater than a, respectively;
return (QUICKSORT($S_1$) followed by $S_2$ followed by QUICKSORT($S_3$))
end
Quicksort Parallelism

- Can sort each subset in parallel
- “Divide and Conquer” parallelism
Quicksort Pseudo Code

Example (Parallel QuickSort)

quickSort(S) =
if |S| = 0 then S
else let p = S[0] % assuming S starts in random order
    SL = {s ∈ S | s < p} SE = {s ∈ S | s = p}
    SG = {s ∈ S | s > p}
    (RL, RG) = (quickSort(SL) || quickSort(SG))
    in append(RL, append(SE, RG)) end

- Present at high level of abstraction
- Potential parallelism left exposed
Analyzing Performance

Sequential Composition: $e_1 ; e_2$

- **Work:** $W = W(e_1) + W(e_2) + 1$
- **Span:** $S = S(e_1) + S(e_2) + 1$

Parallel Composition: $e_1 \parallel e_2$

- **Work:** $W = W(e_1) + W(e_2) + 1$
- **Span:** $S = \max(S(e_1), S(e_2)) + 1$

For Quicksort

**Analysis:**

Expected Work = $O(|S| \log |S|)$

Use standard sum of indicator variables $A_{ij} = i$ compared to $j$

Span = $O(k \log^2 |S|)$ w.h.p, i.e., $p > (1 - 1/|S|^k)$. Use Markov’s inequality, union bound, product of independent expectations.
Parallel Computing Primitives

Mapping Function Over Sequence: Map(f, S)
- Linear work, constant span

Reduction of Sequence: Reduce(f, I, S)
- Associative operator
- Linear work, logarithmic span

Parallel Prefix over Sequence: Scan(f, I, S)
- Associative operator
- Linear work, logarithmic span
CS 418: Parallel Computer Architecture & Programming

Student Background
- Abstract algorithmic principles
- Concrete systems experience
- Minimal understanding of processor implementation

Goal
- How to make programs run fast
- Good algorithms
- Understand how they execute on hardware
- Able to identify & eliminate performance inhibitors
HPC Programming

System Level
- Message-Passing Interface (MPI) supports node computation, synchronization and communication

Node Level
- OpenMP supports thread-level operation of node CPU
- CPUs have SIMD arithmetic units
- CUDA programming environment for GPUs
  - Performance degrades quickly if don’t have perfect balance among memories and processors

Result
- Single program is complex combination of multiple programming paradigms
Parallel Programming Labs

Getting Started

- Exercises using pthreads, ISPC, SIMD intrinsics

GPU Programming

- CUDA implementation of graphics renderer

Shared Memory Programming

- OpenMP implementation of graph algorithms

Distributed Services

- Simulated balancer for distributed web server

Open-Ended Projects

- Mapping applications onto parallel systems
- Language support
- Experimental evaluation of basic primitives
My CUDA Experience

- N x N matrix multiplication
- Performance in GFlops
- Compare to Xeon iCore 7

![Graph showing performance in GFlops for N x N matrix multiplication with different methods (Simple, Transpose, Cuda Simple, Cuda Transpose) across various matrix sizes (8, 16, 32, 64, 128, 256, 512, 1024).]
My CUDA Experience

- Improve memory access patterns
Some Logistics

Increasing Popularity

- Capped enrollment at 160
- Undergraduates + masters
- Will offer 2X / year

Resources

- Multicore servers
  - 12-core Xeon
- nVidia GPUs
  - Varying capabilities and speeds
- Xeon Phi’s
  - First generation
- Not enough to meet needs during crunch times
- Negative experience when used Supercomputer Center resources
Required Understanding

Parallel Algorithms
- Mapping, scans, reductions
- Covered well by CS 210

Program Execution
- Multiple programs executing in shared address space
- Races & synchronization
- Covered well by CS 213

Hardware Support
- Memory hierarchy
- Cache protocols
- GPU implementation
- SIMD instructions
- Supporting simultaneous multithreading
- Had to cover in class
CS 440: Distributed Computing

**Principles**
- Networking and communication
- Time, synchronization, mutual exclusion, consensus
- Fault tolerance
- Distributed file systems and Map/Reduce
- Security protocols

**Practice**
- Implement (parts of) several systems
- Go programming language
Distributed System Projects

Client/Server/Worker System
- Creating reliable protocol on top of UDP
- Farming tasks out to collection of workers
  - That come & go unpredictably

Distributed Hash Table
- To support a social networking server
- Atomicity / consistency / fault tolerance

Paxos
- Distributed consensus
Curriculum Summary

- Build in layers to introduce and reinforce ideas
- Cover both principles and practical implementations
- Hands-on projects essential to learning
Our Contrarian Perspective

Conventional Wisdom

- We need to make CS more appealing by making it more fun
  - Programming games, robots, ...
  - (This is fine for young students)
- Students want to learn the latest technology
  - Android apps, Ruby on Rails, ...
  - This will help them get jobs

Our Viewpoint

- Requirements for software quality are increasing
  - Safety critical systems
  - Reduce vulnerability to cyberattack
- Must create foundations for lifelong learning
- Implication: Core curriculum should focus on fundamentals
  - Mathematical principles, formal reasoning
  - But do in way that prepares them for real systems
Motivating Students

- Premier companies understand the need for greater rigor
- They also want deep programming skills
  - Not language specific
- Their interview questions reinforce this to students
More Information about 2010 Curriculum Changes

Bryant, Sutner, Stehlik

- “Introductory Computer Science Education at Carnegie Mellon University, a Deans’ Perspective”
- CMU-CS-10-140

http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf