Parallel Programming

CS149
11:00-12:15 TT
Herrin T195

Administrivia

• Everything is on the class Web site
  http://courseware.stanford.edu

• Syllabus will be on-line, of course
  • Assignment dates should not change
  • Midterm
    • Thursday, 2/11
      in class
  • Final
    • Thursday, 3/18
      • 2:30-5:30

• Communication
  • Use Courseware, email, phone, office hours
  • But definitely prefer Courseware!

Grading

• 5% class participation
• 10% problem sets (2)
• 40% programming assignments (5)
• 20% midterm
• 25% final

• See handout for late day policy

Class Structure

• Class will be taught from notes
  • No text

• Problem sets = practice for exams

• Do your own work!
  • We may use plagiarism detection software

Staff

• Instructors
  • Alex Aiken
  • Kunle Olukotun

• TAs
  • Michael Bauer
  • Nathan Bronson

• Office hours, contact info, on course site

Prerequisites

• What the syllabus says
  • Intro course sequence through CS 110
  • At least one other 14X systems course

• What we mean
  • Basic knowledge of compilers & architecture
  • A certain programming maturity
    • Comfortable picking up new languages/systems
    • Able to deal with “hardhat” debugging environments
Parallel Programming

• The world is going parallel

• General belief most new software in the future will be written as parallel software

• Why?

Dateline: 1990

• My program must run 2X faster. I can

• Work hard for several months
  - Assuming my program is already well-tuned

• Wait 18 months and buy a new computer

  What do you think people did?

The Free Lunch

• With no work, my program kept getting faster

• Huge force for innovation
  - When performance of out-of-reach applications became just-within-reach

• But only in some areas . . .
  - Little incentive to rock the architecture boat
  - Or languages/compilers/OS

Intel Microprocessor Trends

Dateline: 2010

• My program must run 2X faster. I can:

• Do nothing. My program may run slower.

• Work hard for several months.

Parallelism

• Or, I can rewrite my program in parallel.

• Transistor density is still increasing
  - More parallel units on a chip
  - But speed of the units no longer increasing
No More Free Lunch

• Bad News
  - Must work for performance improvements

• Good News
  - But potentially can get much more than 2X!
  - And once parallel, program may get faster on new hardware

Today

• We are in the midst of the transition to parallel programming

• New everything
  - Architectures, languages, compilers

• Bold prediction
  - There will likely be several winners
  - More than one parallel programming model will survive

Observation

Parallel programming is hard.

Good parallel programming is even harder.

Corollary

• The reason to write parallel programs is for improved performance

• Performance programming is hard
  - Management of resources
  - How resources interact
  - How to find, fix, and avoid bottlenecks

This Course

• Understand this new landscape
  - Survey various approaches to parallel programming
  - Understand corresponding architectural/language trends

• Learn parallel programming
  - Assignments on a subset of language paradigms

• Learn performance programming
  - How do we structure programs to run fast?

The Rest of this Lecture

• Understand the resources
  - What are they?
  - Where and when are they managed?
    - Programmer, compiler, run-time system, or hardware?

• Introduce terminology
An Example

Goal: For all \( i,j \) compute \( x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1}) \)

Issue 1

Can I refer to \( x_{i,j}, x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1} \) at the same time?

Why Not?

Resource: Memory    (Hardware Level)

Distributed Memory
- Hardware exposes physically disjoint memories

Shared Memory
- Hardware provides a single hardware address space

Resource: Memory    (Program Level)

Global Address Space
- Programming language allows any piece of data to be named anywhere in the machine

Local Address Space
- Programming language only allows data to be named that is "near" the processor

Software vs. Hardware

- Global address space is easy to implement on shared memory hardware
- Hardware is complex

- Global address space is much more complex to implement on distributed memory hardware
- Language system is complex
**The Example Again**

Goal: For all \( i,j \) compute \( x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1}) \)

---

**Issue 1b**

What is the cost of referring to \( x_{i,j}, x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1} \)?

---

**Locality**

• Is the data “close” to the processor?

• Local address space
  - Yes, memory references are always cheap
  - Programmer structures program for locality

• Global address space
  - Memory references may have greatly varying cost
  - E.g., on distributed memory machines

---

**Summary: Memory**

• Memory is a critical resource

• Who deals with reality that memory is physically distributed?
  - Shared memory: the hardware does it
  - Global address space: the compiler/runtime does it
  - Local address space: the programmer does it

• Programs can exhibit good or bad locality

---

**Issue 2**

Goal: For all \( i,j \) compute \( x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1}) \)

In parallel for each \( i,j \).

How many copies of the program do I need?

---

**Control**

• Control is a resource

• Parallel copies of the program require state
  - At least a program counter
  - This state must be stored somewhere and managed

• Note this is different from the question of how many processors there are
  - Number of executing “jobs” not necessarily the same as number of processors
Question

- Recall:
  - For all $i,j$ compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$
- Consider two different functions $F$

**Issue 2b**

Goal: For all $i,j$ compute $x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})$
In parallel for each $i,j$.
In what order to reads and writes happen?

**Synchronization**

- Many read/write orders are possible
- To ensure a particular order, must use synchronization
  - Multiple control contexts must coordinate their actions
- Large variety of synchronization abstractions
  - Locks, semaphores, condition variables, barriers, ...
Summary: Control

- Control is a resource
  - Replicating control is expensive
- Many control contexts
  - Parallel jobs run asynchronously
  - Synchronization required
- One/few control contexts
  - Can still execute on many data elements
  - Synchronization built-in

Summary: Control (Cont.)

- Who deals with the fact that the hardware provides a limited number of control contexts?
  - Compiler/runtime system may provide more contexts than physically available
  - Or not: Let the programmer deal with it
- Who deals with synchronization?
  - Many strategies from hardware, compiler, programmer, to combinations of all three

Discussion

- Two fundamental resources
  - Memory
    - locality
  - Control
    - synchronization

Digression: Sequential Programming?

- Hardware is fundamentally parallel
- So sequential programs are really parallel programs!
  - Global address space
  - Synchronization after each instruction

Performance

- Recall
  - The goal is to make computations run fast
- But, there are overheads
  - Managing memory
    - Moving data from where it is to where it is needed
  - Managing control
    - Synchronization, context-switching

Definition

\[ W = \text{useful work in a computation} \]
\[ O = \text{overhead in a computation} \]
\[ \frac{W}{O} \text{ is the computational intensity} \]

The ratio needs to be large, or we will not benefit from parallelism
The Example Again

\[
\begin{array}{cccc}
X_{i+1,j} & X_{i,j} & X_{i+1,j} & X_{i,j+1} \\
X_{i,j-1} & X_{i,j} & X_{i,j+1} & X_{i,j+2} \\
X_{i,j-1} & X_{i+1,j} & X_{i,j+1} & X_{i,j+2}
\end{array}
\]

Goal: For all \(i,j\) compute \(x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})\)

Granularity

- For all \(i,j\) compute \(x_{i,j} = F(x_{i-1,j}, x_{i,j-1}, x_{i+1,j}, x_{i,j+1})\)
- Consider two different functions \(F\)

Discussion

- Two example sweet spots
- Very fine-grain parallelism
  - But easily identified by compiler, so can be scheduled off-line => low overheads
- Very coarse-grain parallelism
  - But dependent on computation at run-time, so heavyweight run-time scheduling needed
  - Overhead is still low, though

Granularity (Cont.)

- A given application on a given parallel platform has a minimum overhead
  - Combination of program, hardware, compiler, run-time systems
- Implies a minimum granularity
  - High overheads: coarse-grain parallelism required
  - Low overheads: fine-grain parallelism can be used

Parallel Languages

- Design space is huge
- Why?
  - Two dimensions: Memory & Control
  - Spectrum of choices in each dimension
    - Responsibilities of hardware, run-time, compiler, programmer
  - Leads to different inherent overheads
    - And so different choices better suited to different classes of applications

Paradigms We'll Cover

- Threads
- Transactional Memory
- Map-Reduce
- SIMD
- SIMT (CUDA)
- Futures
- Nested Parallelism

Lectures in pairs
- Programming model
- Implementation
- ~1 week/paradigm

Red = Programming Assignment
Conclusion

- Parallel programming is a hot topic
  - Huge (bewildering) amount of current activity

- Goals
  - Teach an increasingly important skill
  - Convey the fundamentals
    - There will be new architectures/languages/compilers, but essential trade-offs won't change
  - Make sense of it all