Concurrent and Distributed Programming

Lecture 1

Introduction

References:

Slides by Mark Silberstein, 2011

“Intro to parallel computing” by Blaise Barney

Lecture 1, CS149 Stanford by Aiken & Olukotun,

“Intro to parallel programming”, Gupta
Administration

- Lectures: Assaf Schuster
- TA in charge: Liran Funaro
- TA: Roy Svink
- EXE checker: Roma Shor

- Syllabus on the site – possible changes
Personal Data

Career focus - parallel and distributed computing
- Scalable systems, parallel models, cloud management.
- Last ~10 years focus - scalable data analytics, aka BIG DATA
- Distributed data mining, monitoring rapid data streams.

Current interests:
- Cloud resource allocation.
- Data Streams.
- Big Data Analytics.
- Distributed Monitoring.
- Systems and Virtualization.
- Solid State Disks.
- Cyber Security.
- Internet of Things.
Grading

- 70% Exam, 30% Home assignments
- Pass conditioned on Exam grade $\geq 55$
- And assignment grade $\geq 55$ for each one
- 4 assignments, syllabus check deadlines (each assignment worth 7.5 final-grade points)
  - Threads (Java)
  - OpenMP (C)
  - MPI (C)
  - BSP (C)
- Assignments may overlap
- Will be checking assignment copying
- No postponement (unless force major)
Prerequisites

Formal

- 234123 Operating Systems
- 234267 Digital Computer Structure
- Or EE equivalent

Informal

- Programming skills (makefile, Linux, ssh)
Serial vs. parallel program

One instruction at a time

Multiple instructions in parallel
Why parallel programming?

- In general, most software will have to be written as a parallel software
- Why?
Free lunch ...

- Wait 18 months – get new CPU with x2 speed
- Moore's law: #transistors per chip $\sim 1.8^{\text{years}}$

is over

- More transistors = more execution units
- **BUT** performance per unit does not improve

- **Bad news:** serial programs will not run faster
- **Good news:** parallelization has high performance potential
- May get faster on newer architectures
Parallel programming is hard

- Need to optimize for performance
- Understand management of resources
- Identify bottlenecks
- No one technology fits all needs
- Zoo of programming models, languages, run-times
- Hardware architecture is a moving target
- Parallel thinking is not intuitive
- Parallel debugging is not fun

But there is no alternative!!!
Concurrent and Distributed Programming Course

- basic principles
- Parallel and distributed architectures
- Parallel programming techniques
- Basics of programming for performance

- No one book 😞
- Slides are comprehensive 😊
- Tons of material on the Internet 😊

This is a practical course – you will fail unless you complete all home assignments
Parallelizing Game of Life (Cellular automaton)

- Given a 2D grid
- $v^t(i,j) = F(v^{t-1} \text{(of all its neighbors)})$
Problem partitioning

Domain decomposition
- (SPMD)
- Input domain
- Output domain
- Both

Functional decomposition
- (MPMD)
- Independent tasks
- Pipelining
We choose: domain decomposition

- The field is split between processors
Issue 1. Memory

- Can we access $v(i+1,j)$ from CPU 0 as in serial program?
It depends...

- **YES:** Shared memory space architecture: same memory space

- **NO:** Distributed memory space architecture: disjoint memory space
Someone has to pay

- Easier to program, harder to build hardware

- Harder to program, easier to build hardware

- Tradeoff: *Programability* vs. *Scalability*
Flynn's **HARDWARE** Taxonomy

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction, Single Data</th>
<th>Single Instruction, Multiple Data</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S I S D</strong></td>
<td>Single Instruction, Single Data</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td><strong>M I S D</strong></td>
<td>Multiple Instruction, Single Data</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
<tr>
<td><strong>S I M D</strong></td>
<td>Single Instruction, Multiple Data</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
<tr>
<td><strong>M I M D</strong></td>
<td>Multiple Instruction, Single Data</td>
<td>Multiple Instruction, Multiple Data</td>
</tr>
</tbody>
</table>
SIMD

- Lock-step execution

- Example: vector operations (SSE)
MIMD

- Example: multicores
- Which of SIMD/MIMD requires hw shared memory?
Memory architecture

- CPU0: Time to access $v(i+1,j) = Time$ to access $v(i-1,j)$?
Hardware shared memory flavors 1

- Uniform memory access: **UMA**
- Same cost of accessing **any** data by all processors
Hardware shared memory flavors 2

- NON-Uniform memory access: **NUMA**

  ![Diagram of NUMA architecture]

- **Tradeoff**: *Scalability vs. Latency*
Software Distributed Shared Memory (SDSM)

- Software - DSM: emulation of NUMA in software over distributed memory space
Memory-optimized programming

- Most modern systems are NUMA or distributed
- Architecture is easier to scale up by scaling out
- Access time difference: local vs. remote data: $x_{100-10000}$
- Memory accesses: main source of optimization in parallel and distributed programs
- Locality: most important parameter in program speed (serial or parallel)
Issue 2: Control

- Can we assign one $v$ per CPU?
- Can we assign one $v$ per process/logical task?

\[
\begin{align*}
& i-1 & i, j & i+1 \\
& j-1 & & j+1
\end{align*}
\]
Task management overhead

- Each task has a state that should be managed
- More tasks – more state to manage

- Who manages tasks?
- How many tasks should be run by a cpu?
- Does that depend on $F$?
- Reminder: $v(i,j) = F(\text{all } v's \text{ neighbors})$
Question

- Every process reads the data from its neighbors
- Will it produce correct results?
Synchronization

- The order of reads and writes in different tasks is non-deterministic
- Synchronization is required to enforce the order
- Locks, semaphores, barriers, conditionals…. 

---

30
Check point

Fundamental hardware-related issues affecting Programability, Correctness, Scalability, Performance

- Memory accesses
- Optimizing locality of accesses
- Control
- Overhead
- Synchronization
Parallel programming issues

- We decide to split this 3x3 grid like this:
Issue 1: Load balancing

- Always waiting for the slowest task
- Solutions?
Issue 2: Granularity

- G = Computation/Communication

- Fine-grain parallelism
  - G is small
  - Good load balancing
  - *Potentially* high overhead

- Coarse-grain parallelism
  - G is large
  - *Potentially* bad load balancing
  - Low overhead

What granularity works for you?
It depends..

For each combination of computing platform and parallel application

- The goal is to minimize overheads and maximize CPU utilization

High performance requires

- Enough parallelism to keep CPUs busy
- Low relative overheads (communications, synchronization, task control, memory accesses versus computations)
- Good load balancing
Summary

Parallelism and scalability do not come for free

- Overhead
- Memory-aware access
- Synchronization
- Granularity

But... let's assume for one slide we did not have these issues....
Can we estimate an upper bound on speedup? *Amdahl's law*

- Sequential component **limits the speedup**

- Split program serial time $T_{\text{serial}}=1$ into
  - *Ideally parallelizable*: $A$ (fraction parallel): ideal load balancing, identical speed, no overheads
  - *Cannot be parallelized*: $1-A$

- Parallel time $T_{\text{parallel}}= A/#\text{CPUs}+(1-A)$

- Speedup($#\text{CPUs}$)=$T_{\text{serial}}/T_{\text{parallel}}=$

  $1 / ( A/#\text{CPUs}+(1-A) )$
Bad news

So - why do we need machines with 1000x CPUs?
Living with Amdahl's law: Gustafson's law

\[ T_{\text{parallel}} = T_{\text{parallel}} \times (A + (1-A)) \]

\[ T_{\text{bestserial}} \leq \#\text{CPUs} \times T_{\text{parallel}} \times A + T_{\text{parallel}} \times (1-A) \]

[by simulation, a bound on the best serial program]

\[ \text{Speedup} = \frac{T_{\text{bestserial}}}{T_{\text{parallel}}} \leq \#\text{CPUs} \times A + (1-A) = \#\text{CPUs} - (1-A) \times (\#\text{CPUs} - 1) \]

It all depends how good the simulation is

Simulation usually improves with problem size
Amdahl vs. Gustafson – both right

It is all about granularity

- When problem size is fixed granularity diminishes with #CPU
- When granularity diminishes, simulation departs from "bestserial", and Gustafson's upper-bound departs from the actual speedup

Amdahl's law: strong scaling

- Solve same problems faster for fixed problem size and #CPU grows
- If A is fixed, granularity diminishes

Gustafson's law: weak scaling

- Problem size grows: solve larger problems
- Keep the amount of work per CPU when adding more CPUs to keep the granularity fixed
Question

- Can we achieve speedups higher than #CPUs?
- “Superlinear”
Fake superlinear speedup: Serial algorithm does more work

- Parallel algorithm is BFS
- DFS is inefficient for this input
- BFS can be simulated with a serial algorithm
Always use best serial algorithm as a baseline

Example: sorting an array

- Efficient parallel bubble sort takes 40s, serial 150s. Speedup = 150/40?

- **NO. Serial** quicksort runs in 30s. Speedup 0.75!
True superlinear speedup

Example: cache

Parallelization results in smaller problem size/CPU
=> if fits the cache
=> non-linear performance boost!

- Cannot be *efficiently* simulated on a serial machine
- See more examples in the Graha book
Summary

- Parallel performance is a subtle matter
- Need to use a good serial program
- Need to understand the hardware

Amdahl's law: understand the assumptions!!!

Too pessimistic
- Fraction parallel independent of #CPUs
- Assumes fixed problem size

Too optimistic
- Assumes perfect load balancing, no idling, equal CPUs speed

See more material: refs/ on website and book