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

• Lecture: Liran Funaro
• TA in charge: Ido Hakimi (Thursday)
• TA: Eylon Shoshan (Wednesday)
• EXE checkers: Itay Elizur and Anat Tetroashvili

• Syllabus on the site – possible changes
Grading

• 70% Exam, 30% Home assignments
  • Pass conditioned on Exam grade >= 55
  • And assignment grade >= 55 for each one

• 4 assignments, syllabus check deadlines (each assignment worth 7.5 final-grade points)
  • Threads (Java)
  • OpenMP (C)
  • MPI (C)
  • Map-Reduce (Java)

• Assignments may overlap
• No postponement (unless force major)
• Submissions will be checked for copying
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 (apparently)

Multiple instructions in parallel
Why parallel programming?

• In general, most software will eventually have to make use of parallelism
  • Assuming performance matters
• Why?
Free lunch...

35 YEARS OF MICROPROCESSOR TREND DATA

Free lunch – is over 😞

35 YEARS OF MICROPROCESSOR TREND DATA

- Transistor number grows (Moore’s law)
- Sequential performance no longer improves
- Cores number grows

Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore
Unfortunately, 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 better alternative!!!
Concurrent and Distributed Programming Course

• Learn:
  • Basic Principles
  • Parallel and distributed architectures
  • Parallel programming techniques
  • Basics of programming for performance

• This is a practical course – you will fail unless you complete all home assignments
Flynn's HARDWARE Taxonomy
SIMD

- Example: vector operations (e.g., Intel SSE/AVX, GPU)
MIMD

- Example: multicores
Problem partitioning

- Domain decomposition
  - Single Program, Multiple Data
  - Input domain
  - Output domain
  - Both

- Functional decomposition
  - Multiple Programs, Multiple Data
  - Independent tasks
  - Pipelining
Test case – parallelizing Game of Life (Cellular automaton)

• Given a 2D grid
• \( v_t(i, j) = F(v_{t-1} \text{ of all its neighbors}) \)
Question: how to partition the problem?

- Which model fits our problem the best?
- How will we partition the problem?

<table>
<thead>
<tr>
<th></th>
<th>SIMD</th>
<th>MIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>Single Instruction stream Single Data stream</td>
<td>Multiple Instruction stream Single Data stream</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction stream Multiple Data stream</td>
<td>Multiple Instruction stream Multiple Data stream</td>
</tr>
</tbody>
</table>
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...

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

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

- **Shared memory**: easier to program, harder to build hardware

- **Distributed memory**: harder to program, easier to build hardware

- **Tradeoff**: Programmability vs. Scalability
Memory architecture

• Question: for CPU0,
  • Time to access $v(i+1,j)$ = Time to access $v(i-1,j)$ ?
Hardware shared memory flavor #1

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

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

**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**
  - From hardware perspective, scaling out (more servers) is easier than scaling up (more cores)
- Access time difference: **local** vs. **remote** data: x100-10000
- **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?
Task management overhead

- **Task**: sequence of instructions yielding a single result
  - Independent tasks are easiest to parallelize
  - Higher level abstraction than *thread*, which abstracts a core
- 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: $v_t(i, j) = F(v_{t-1}(of\ all\ its\ neighbors))$
Question: is it correct?

- If 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....
Check point

• Fundamental hardware-related issues affecting Programmability, 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

\[ G = \frac{\text{Computation}}{\text{Communication}} \]

- **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?

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}}
\]
Amdahl’s law

• Speedup is bound by serial component
• Split program serial time ( $T_{\text{serial}} = 1$ ) into
  • Ideally parallelizable portion: $A$
    • assuming perfect load balancing, identical speed, no overheads
  • Cannot be parallelized (serial) portion : $1 - A$
• Parallel time:

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

$$\text{Speedup(\#CPUs)} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{\frac{A}{\#\text{CPUs}} + (1 - A)}$$
Bad news

So why do we need machines with 1000x CPUs?

Living with Amdahl's law: Gustafson's law

- **Underlying assumptions:**
  1. Workload size usually increased when more cores are available
  2. Serial work remains the same, hence parallel portion grows
  3. So, more cores $\rightarrow$ larger workloads $\rightarrow$ larger parallel portion

\[
T_{\text{parallel}} = T_{\text{parallel}} \cdot \left( A + (1 - A) \right) = T_{\text{parallel}} \cdot A + T_{\text{parallel}} \cdot (1 - A) \tag{1}
\]

- \( T_{\text{best serial}} \leq \#\text{CPUs} \cdot T_{\text{parallel}} \cdot A + T_{\text{parallel}} \cdot (1 - A) \)
  - A bound on the best serial program.

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

- \( \Rightarrow \text{Speedup} \leq \#\text{CPUs} \cdot A + (1 - A) \)
- \( \Rightarrow \text{Speedup} \leq \#\text{CPUs} - (1 - A) \cdot (\#\text{CPUs} - 1) \)
Amdahl vs. Gustafson

- \( N = \#CPUs, \ S = \text{serial portion} = 1 - A \)

- Amdahl's law: \( \text{Speedup}(N) = \frac{1}{\frac{A}{N} + S} \)
  - **Strong scaling:** \( \text{Speedup}(N) \) calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and \#CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

- Gustafson's law: \( \text{Speedup}(N) = N - (N - 1) \cdot S \)
  - **Weak scaling:** \( \text{Speedup}(N) \) calculated given amount of work per CPU is fixed
  - Keep the amount of work per CPU when adding more CPUs to keep the granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence:** speedup upper bound much higher
Question: is superlinear speedup possible?

• Can we achieve speedups higher than #CPUs?
  • I.e., Superlinear

![Diagram showing speedup vs. number of processors with categories: Superlinear, Linear, Sublinear.](image)
Fake superlinear speedup: Serial algorithm does more work

- Unordered tree search problem can be solved using DFS.
- DFS might be inefficient for some input, but BFS is efficient.
- DFS parallel algorithm resembles BFS, achieving superlinear speedup.
- However, BFS can be simulated with a serial algorithm.
Always use best serial algorithm as a baseline

- Another example: sorting an array
- Efficient **bubble sort** takes:
  - Parallel 40s
  - Serial 150s
  - \( Speedup = \frac{150}{40} = 3.75 \)?

- NO!
  - Serial quicksort runs in 30s
  - \( \Rightarrow Speedup = 0.75 \)
True superlinear speedup

• There are more to parallel machine than just CPUs
• **Example: cache**
  • Parallelization results in smaller problem: \( \frac{\text{size}}{\#\text{CPU}} \)
  • If fits the cache \( \Rightarrow \) non-linear performance boost!
• Cannot be efficiently simulated on a serial machine
• (see more examples in the Grama’s 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: references on website and book