Amdahl’s Law and Thread Level Parallelism

Instructor: Stephan Kaminsky
We stopped being able to increase clock speeds for our computers...

If we want more performance, we have to come up with new solutions...
Review

• Methods to increase performance
  — Domain-Specific Hardware
  — Parallelism!!
    • Instruction-Level Parallelism
    • Data-Level Parallelism
    • Thread-Level Parallelism
    • Request-Level Parallelism
## Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Single Stage Processor</td>
<td>SIMD: Vector Instructions</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: Nothing really here</td>
<td>MIMD: Multi-core Processors</td>
</tr>
</tbody>
</table>
Great Idea #4: Parallelism

- **Parallel Requests**
  Assigned to computer
  e.g. search “Garcia”

- **Parallel Threads**
  Assigned to core
  e.g. lookup, ads

- **Parallel Instructions**
  > 1 instruction @ one time
  e.g. 5 pipelined instructions

- **Parallel Data**
  > 1 data item @ one time
  e.g. add of 4 pairs of words

- **Hardware descriptions**
  All gates functioning in parallel at same time

**Software**

**Hardware**

Leverage Parallelism & Achieve High Performance

**Computer**

- Core
- ... Core
- Memory
- Input/Output

**Instruction Unit(s)**

- A₀+B₀
- A₁+B₁
- A₂+B₂
- A₃+B₃

**Functional Unit(s)**

**Cache Memory**
Agenda

• **Parallel Computing**
  — Multi-processing
  — Multi-threading
• **Parallelism Challenges**
  — Amdahl’s Law
  — Data Races
• **Synchronization**
• OpenMP
• OpenMP Work Sharing
## Flynn’s Taxonomy

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Single Stage Processor</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: Nothing really here</td>
</tr>
</tbody>
</table>
How do we apply parallelism to processors?

Goal:
Make computer faster by performing tasks concurrently

Solutions:
1. Use multiple cores to run multiple tasks in parallel
2. Run multiple tasks on a single core concurrently
How do we apply parallelism to processors?

Goal:
Make computer faster by performing tasks concurrently

Solutions:
1. Use multiple cores to run multiple tasks in parallel
2. Run multiple tasks on a single core concurrently
Software Tasks: Processes

Program
• Compiled, assembled and linked code that’s ready to be loaded and run

Process
• A program that’s currently running (on a “processor”)
• Has a PC and a memory space
Process State Diagram

- **new**
  - converting to process
  - admitted
  - waiting for CPU
  - I/O or event completion

- **ready**
  - scheduler dispatch
  - interrupt

- **running**
  - I/O or event wait

- **waiting**
  - waiting for I/O or event
  - I/O or event wait

- **terminated**
  - reclaim resource
  - destroy process
  - exit

http://www.csl.mtu.edu/cs3331.ck/common/03-Process.pdf
Multiprocessor Systems

Processor 0
- Control
- Datapath
  - PC
  - Registers
  - (ALU)
- Memory
  - Memory Accesses

Processor 1
- Control
- Datapath
  - PC
  - Registers
  - (ALU)
- Memory
  - Memory Accesses

I/O-Memory Interfaces

Input

Output
Multiprocessor Systems

A computer system with at least 2 processors or cores

- Each core has its own PC and registers
- Each core executes independent instruction streams
- Processors share the same system memory
- Communication through loads and stores to a common location

Deliver high throughput for independent jobs via task-level parallelism
Multiprocessor Example

Run Chrome and Minecraft simultaneously
• Each are separate programs
• Each has a different memory space
• Each can run on a separate core

Don’t even need to communicate...
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading

• Parallelism Challenges
  — Amdahl’s Law
  — Data Races

• Synchronization

• OpenMP

• OpenMP Work Sharing
How do we apply parallelism to processors?

Goal:
Make computer faster by performing tasks concurrently

Solutions:
1. Use multiple cores to run multiple tasks in parallel
2. Run multiple tasks on a single core concurrently
Software Tasks: Threads

Unit of execution *within* a process

Processes we’ve written so far have a single thread
• They “have a single thread of execution”
• They “are single-threaded”

But a single process could have multiple threads...
Thread Memory

Threads have separate:
• PC
• Registers
• Stack memory

Threads share:
• Code memory
• Static memory
Thread Example

Let’s say you’re implementing Chrome:

You want a tab for each web page you open:
• The same code loads each website (shared code section)
• The same global settings are shared by each tab (shared static section)
• Each tab does have separate state for things happening locally though (separate stack and registers)

Disclaimer: Actually, browsers use separate processes for each tab for a variety of reasons including performance and security
Multithreading Processors

• **Basic idea:** Processor resources are expensive and should not be left idle

• Long memory latency to memory on cache miss?
  — Hardware switches threads to bring in other useful work while waiting for cache miss
  — Cost of thread context switch must be much less than cache miss latency
Hardware Support for Multithreading

- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next
Multithreading vs. Multicore

• Multithreading => Better Utilization
  —≈5% more hardware, 1.30X better performance?
  —Share integer adders, floating point adders, caches (L1 I, L1 D, L2 cache, L3 cache), Memory Controller

• Multicore => Duplicate Processors
  —≈50% more hardware, ≈2X better performance?
  —Share some caches (L2 cache, L3 cache), Memory Controller

• Modern machines do both
  —Multiple cores with multiples threads per core
A desktop computer

- 4 total cores
- Each capable of 2 threads
- ≈ 8 processors
Raspberry Pi 4

Quad core processor

- 3-way superscalar pipeline
- L1 Cache
  - 32 KiB 2-way set associative data cache
  - 48 KiB 3-way set associative instruction cache
  - Per core
- L2 Cache
  - 512 KiB to 4 MiB (shared)
- RAM 1-4 GB

$35

Literally all computers are doing parallelism these days
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading

• **Parallelism Challenges**
  — Amdahl’s Law
  — Data Races

• Synchronization

• OpenMP

• OpenMP Work Sharing
Challenges to Parallelism

Parallelism is great! We can do so many things!!

But what’s the downside...?

1. How much speedup can we get from it?
2. How hard is it to write parallel programs?
Imagine a program that takes 100 seconds to run

• 95 seconds in the blue part
• 5 seconds in the green part
Speedup from Improvements

\[
\text{Speedup with Improvement} = \frac{\text{Execution time without improvement}}{\text{Execution time with improvement}}
\]

5s \to 2.5\ s: \quad \text{Speedup} = \frac{100}{97.5} = 1.026

5s \to 1\ s: \quad \text{Speedup} = \frac{100}{96} = 1.042

5s \to 0.001s: \quad \text{Speedup} = \frac{100}{95.001} = 1.053

Performance improvements are only impactful if they are on the important part!
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{(1 - F) + \frac{F}{S}}
\]

- **Non-speed-up part**: \((1 - F)\)
- **Speed-up part**: \(\frac{F}{S}\)

**F** = Fraction of execution time speed up

**S** = Scale of improvement

**Example**: 2x improvement to 25% of the program

\[
\frac{1}{0.75 + 0.25 \cdot 2} = \frac{1}{0.75 + 0.125} = 1.14
\]
Amdahl’s (Heartbreaking) Law

• The amount of speedup that can be achieved through parallelism is limited by the non-parallel portion of your program!
Amdahl’s Law tells us that to achieve linear speedup with more processors, none of the original computation can be scalar (non-parallelizable).

To get a speedup of 90 from 100 processors, the percentage of the original program that could be scalar would have to be 0.1% or less.

\[
\text{Speedup} = \frac{1}{(0.001 + .999/100)} = 90.99
\]

Parallel Speed-up Examples (1/3)

\[
\text{Speedup with enhancement} = \frac{1}{(1-F) + F/S}
\]

- Consider an enhancement which runs 20 times faster but which is only usable 15% of the time.
  \[
  \text{Speedup} = \frac{1}{(0.85 + 0.15/20)} = 1.166
  \]

- What if it’s usable 25% of the time?
  \[
  \text{Speedup} = \frac{1}{(0.75 + 0.25/20)} = 1.311
  \]

Nowhere near 20x speedup!
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading

• **Parallelism Challenges**
  — Amdahl’s Law
  — **Data Races**

• Synchronization
• OpenMP
• OpenMP Work Sharing
Challenges to Parallelism

Parallelism is great! We can do so many things!!

But what’s the downside...?

1. How much speedup can we get from it?
2. How hard is it to write parallel programs?
Data Races

• Thread scheduling is **non-deterministic**
  — There is no guarantee that any thread will go first or last or not be interrupted at any point

• If different threads write to the same variable
  — The final value of the variable is also non-deterministic
  — This is a *data race*

• Avoid incorrect results by:
  1) not writing to the same memory address
  2) *synchronizing* writing and reading to get deterministic behavior
Example: Sum Reduction

• Sum 100,000 numbers on 100 processors
  — Each processor has ID: 0 ≤ Pn ≤ 99
  — Partition 1000 numbers per processor

• Step 1: Initial summation on each processor
  sum[Pn] = 0;
  for (i=1000*Pn; i<1000*(Pn+1); i++)
    sum[Pn] = sum[Pn] + A[i];
Example: Sum Reduction

• Sum 100,000 numbers on 100 processors
  — Each processor has ID: 0 ≤ Pn ≤ 99
  — Partition 1000 numbers per processor

• **Step 1:** Initial summation on *each* processor

  ```
  sum[Pn] = 0;
  for (i=1000*Pn; i<1000*(Pn+1); i++)
    sum[Pn] = sum[Pn] + A[i];
  ```

  — no data dependencies so far
  — not writing to same address in memory
Example: Sum Reduction

• Sum 10,000 numbers on 10 processors
  — Each processor has ID: 0 ≤ Pn ≤ 9
  — Partition 1000 numbers per processor

• **Step 1:** Initial summation on *each* processor
  
  ```
  sum[Pn] = 0;
  for (i=1000*Pn; i<1000*(Pn+1); i++)
    sum[Pn] = sum[Pn] + A[i];
  ```

• **Step 2:** Now need to add these partial sums
  — *Reduction:* divide and conquer approach to sum
  — Half the processors add pairs, then quarters, ...
  — Data dependencies: Need to synchronize between reduction steps
Sum Reduction with 10 Processors

\[ \text{sum}[P0] \quad \text{sum}[P1] \quad \text{sum}[P2] \quad \text{sum}[P3] \quad \text{sum}[P4] \quad \text{sum}[P5] \quad \text{sum}[P6] \quad \text{sum}[P7] \quad \text{sum}[P8] \quad \text{sum}[P9] \]
Example: Sum Reduction Pseudocode

This is **Step 2**, after all “local” sums computed.

*This code runs simultaneously on all processors.*

```plaintext
half = 10;
repeat
    synch();
    ... /* handle odd elements */
    half = half/2; /* dividing line on who sums */
    if (Pn < half)
        sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading

• Parallelism Challenges
  — Amdahl’s Law
  — Data Races

• Synchronization

• OpenMP

• OpenMP Work Sharing
Synchronization

We can’t always avoid any shared memory (and still perform useful tasks)

Avoid data races by *synchronizing* writing and reading to get deterministic behavior
Analogy: Buying Milk

• Your fridge has no milk. You and your roommate will return from classes at some point and check the fridge
• Whoever gets home first will check the fridge, go and buy milk, and return
• What if the other person gets back while the first person is buying milk?
  —You’ve just bought twice as much milk as you need!

How do we solve this?

• It would’ve helped to have left a note...
Lock Synchronization (1/2)

• Use a “Lock” to grant access to a region (critical section) so that only one thread can operate at a time
  — Need all processors to be able to access the lock, so use a location in shared memory as the lock
• Processors read lock and either wait (if locked) or set lock and go into critical section
  — 0 means lock is free / open / unlocked / lock off
  — 1 means lock is set / closed / locked / lock on
Lock Synchronization (2/2)

- Pseudocode:

  Check lock
  Set the lock
  Critical section
  (e.g. change shared variables)
  Unset the lock
Synchronization with Locks

// wait for lock released
while (lock != 0) ;
// lock == 0 now (unlocked)

// set lock
lock = 1;

    // access shared resource ...
    // **sequential** execution!
    // (Amdahl says ewwwww)

// release lock
lock = 0;
Lock Synchronization

Thread 1

while (lock != 0) ;
lock = 1;
// critical section
lock = 0;
• AND, the lock is set to 0 while Thread 2 is in the critical section!!

Thread 2

while (lock != 0) ;
• Thread 2 finds lock not set, before thread 1 sets it
• Both threads believe they got and set the lock!
lock = 1;
// critical section
lock = 0;
while(locklock != 0)
locklock = 1;
    // wait for lock released
    while (lock != 0) ;
    // lock == 0 now (unlocked)

    // set lock
    lock = 1;
locklock = 0;

    // access shared resource ...

    // release lock
lock = 0;

This isn’t going to work...

Try as you like, this problem has no software solution because the lock itself is shared memory.

Unless we introduce new instructions, that is!
Hardware Synchronization

• Solution:
  − Atomic read/write
  − Read & write in single instruction
    ▪ No other access permitted between read and write

• Common implementations:
  − Atomic swap of register ↔ memory
  − Pair of instructions for “linked” read and write
    ▪ write fails if memory location has been “tampered” with after linked read

• RISCV has variations of both, but for simplicity we will focus on the former
RISCV Atomic Memory Operations (AMOs)

- AMOs atomically perform an operation on an operand in memory and set the destination register to the original memory value.
- R-Type Instruction Format: Add, And, Or, Swap, Xor, Max, Max Unsigned, Min, Min Unsigned.

Note: Only one Load actually occurs!!

amoadd.w rd,rs2,(rs1):
  temp = M[R[rs1]];
  R[rd] = temp
  M[R[rs1]] = temp + R[rs2]

Load value into Rd from memory
Add to value and store to memory
Memory Ordering

Atomic instructions include memory ordering

aq - acquiring lock
  • execute no memory accesses after this instruction until it completes

rl - releasing lock
  • finish all memory accesses before this instruction completes
RISCV Critical Section

- Assume that the lock is in memory location stored in register a0
- The lock is “set” if it is 1; it is “free” if it is 0 (it’s initial value)

```
addi t0, x0, 1    # Get 1 to set lock
Try: amoswap.w.aq t1, t0, (a0) # t1 gets old lock value
    # while we set it to 1
    bnez t1, Try      # if already 1, another
    # thread has lock, so
    # we must try again
... critical section goes here ...
amoswap.w.rl x0, x0, (a0) # store 0 in lock to
    # release
```
Lock Synchronization

Broken Synchronization

while (lock != 0) ;
lock = 1;
// critical section
lock = 0;

Fix (lock is at location (a0))

addi t0, x0, 1
Try: amoswap.w.aq t1, t0, (a0)
    bnez t1, Try
Locked:
    # critical section
Unlock:
    amoswap.w.rl x0, x0, (a0)
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading
• Parallelism Challenges
  — Amdahl’s Law
  — Data Races
• Synchronization
• **OpenMP**
• OpenMP Work Sharing
OpenMP

• API used for multi-threaded, shared memory parallelism
  — Compiler Directives
  — Runtime Library Routines
  — Environment Variables
• Portable
• Standardized
OpenMP Specification

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

- governed by parallel directive
- do/parallel do and section directives
- distributes work among threads
- scopes variables
- shares and private clauses
- coordinates thread execution
- critical and atomic directives
- barrier directive
- runtime environment
- omp_set_num_threads()
- omp_get_thread_num()
- OMP_NUM_THREADS
- OMP_SCHEDULE
Shared Memory Model with Explicit Thread-based Parallelism

• Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization

• Pros:
  — Takes advantage of shared memory, programmer need not worry (that much) about data placement
  — Compiler directives are simple and easy to use
  — Legacy serial code does not need to be rewritten

• Cons:
  — Code can only be run in shared memory environments
  — Compiler must support OpenMP (e.g. gcc 4.2)
OpenMP in CS61C

• OpenMP is built on top of C, so you don’t have to learn a whole new programming language
  — Make sure to add `#include <omp.h>`
  — Compile with flag: `gcc -fopenmp`
  — Mostly just a few lines of code to learn

• You will NOT become experts at OpenMP
  — Use slides as reference, will learn to use in lab

• **Key ideas:**
  — Shared vs. Private variables
  — OpenMP directives for parallelization, work sharing, synchronization
OpenMP Programming Model

• Fork - Join Model:

• OpenMP programs begin as single process (master thread) and executes sequentially until the first parallel region construct is encountered
  — FORK: Master thread then creates a team of parallel threads
  — Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  — JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread
OpenMP Extends C with Pragmas

• *Pragmas* are a preprocessor mechanism C provides for language extensions
• Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered in 61C)
• Good mechanism for OpenMP -- compilers that don't recognize a pragma just ignore them
  — Runs on sequential computer even with embedded pragmas
parallel Pragma and Scope

• Basic OpenMP construct for parallelization:

```c
#pragma omp parallel
{
    /* code goes here */
}
```

— *Each* thread runs a copy of code within the block
— Thread scheduling is *non-deterministic*

• Variables declared outside pragma are *shared*
— To make private, need to declare with pragma:

```c
#pragma omp parallel private (x)
```

This is annoying, but curly brace MUST go on separate line from `#pragma`
Thread Creation

• Defined by **OMP_NUM_THREADS** environment variable (or code procedure call)
  — Set this variable to the *maximum* number of threads you want OpenMP to use

• Usually equals the number of cores in the underlying hardware on which the program is run
OMP_NUM_THREADS

• OpenMP intrinsic to set number of threads:
  
  \texttt{omp\_set\_num\_threads(x);} 

• OpenMP intrinsic to get number of threads:
  
  \texttt{num\_th = omp\_get\_num\_threads();} 

• OpenMP intrinsic to get Thread ID number:
  
  \texttt{th\_ID = omp\_get\_thread\_num();}
#include <stdio.h>
#include <omp.h>

int main () {
    int nthreads, tid;

    /* Fork team of threads with private var tid */
    #pragma omp parallel private(tid)
    {
        tid = omp_get_thread_num(); /* get thread id */
        printf("Hello World from thread = \%d\n", tid);

        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = \%d\n", nthreads);
        }
    } /* All threads join master and terminate */
}
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading
• Parallelism Challenges
  — Amdahl’s Law
  — Data Races
• Synchronization
• OpenMP
• OpenMP Work Sharing
OpenMP Directives (Work-Sharing)

- These are defined within a parallel section

 Shares iterations of a loop across the threads

 Each section is executed by a separate thread

 Serializes the execution of a thread
Parallel Statement Shorthand

```c
#pragma omp parallel
{
    #pragma omp for
    for(i=0; i<len; i++) { ... }
}
can be shortened to:
#pragma omp parallel for
    for(i=0; i<len; i++) { ... }
```

This is the only directive in the parallel section.
Building Block: for loop

for (i=0; i<max; i++) zero[i] = 0;

• Break for loop into chunks, and allocate each to a separate thread
  — e.g. if max = 100 with 2 threads: assign 0-49 to thread 0, and 50-99 to thread 1

• Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  — Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
  • No premature exits from the loop allowed
  — i.e. No break, return, exit, goto statements

In general, don’t jump outside of any pragma block
Parallel for *pragma*

```c
#pragma omp parallel for
for (i=0; i<max; i++) zero[i] = 0;
```

- Master thread creates additional threads, each with a separate execution context
- All variables declared outside for loop are shared by default, except for loop index which is *private* per thread (Why?)
- Divide index regions sequentially per thread
  - Thread 0 gets 0, 1, ..., (max/n)-1;
  - Thread 1 gets max/n, max/n+1, ..., 2*(max/n)-1
  - Why?
- Implicit synchronization at end of for loop

Cache blocking -- better AMAT
Agenda

• Parallel Computing
  — Multi-processing
  — Multi-threading
• Parallelism Challenges
  — Amdahl’s Law
  — Data Races
• Synchronization
• OpenMP
• OpenMP Work Sharing
Summary

We can take advantage of parallel programming in order to further improve our performance.

Need to be careful of data races and solve them with synchronization techniques.

Amdahl’s Law lets you measure speedup and determine how much of an effect you can possibly have.