# CS 61C

## Discussion 9: Flynn Taxonomy, Parallelism

John Yang
Summer 2019

# Announcements

- Congrats, done with the midterm!
- Check In: tinyurl.com/john61c
- Homework 6 Due 7/30 (Tomorrow!)
- Project 3 Due 7/31 (Wednesday)

# Today’s Goal

- Given the diverse ecosystem of different computer architectures, we’ll see how Flynn’s Taxonomy classifies architectures by instruction and data streams
- Explore techniques and implementations behind different forms of parallelism

# Flynn Taxonomy

Classifying approaches to computer architecture by instruction + data streams

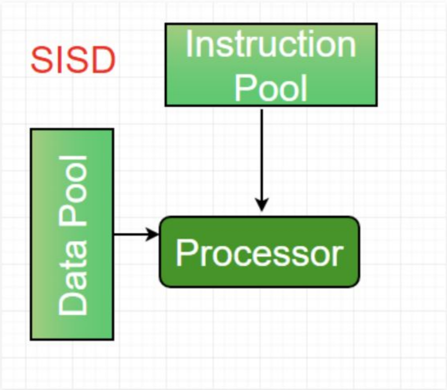
# Flynn Taxonomy

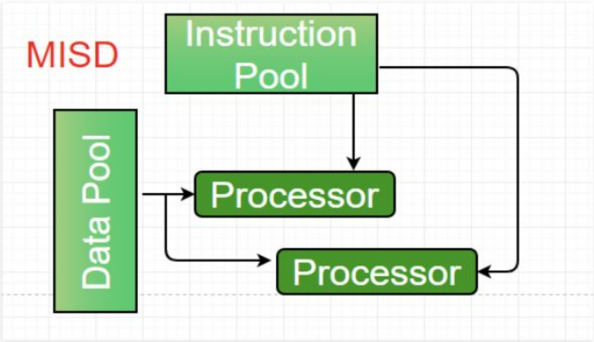
Motivation: Classify different computer systems and machines by the number of instruction and data streams that can be processed simultaneously

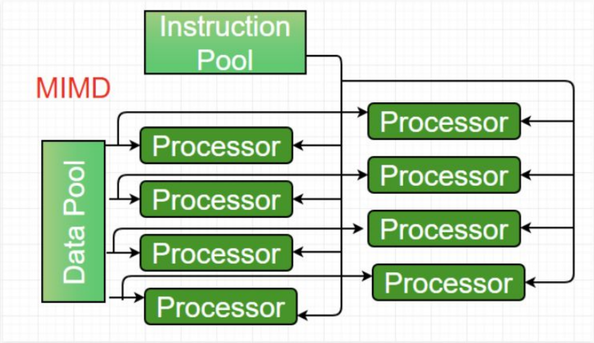
- SISD: Single Instruction, Single Data Stream System
  - Sequential Computation, no use of parallelism in any form.
  - Traditional “von-Neumann”, single CPU computer
- SIMD: Single Instruction, Multiple Data Stream System
  - Available on most modern processors
  - Data is processed as vector operations, not single units
- MISD: Multiple Instruction, Single Data Stream System
  - Not many systems fall under this category.
  - Example: Z = sin(x)+cos(x)+tan(x)
- MIMD: Multiple Instruction, Multiple Data Stream System
  - Multiple processors executing different sets of instructions on different data
  - Examples: Multicore processors, Warehouse Scale Computing

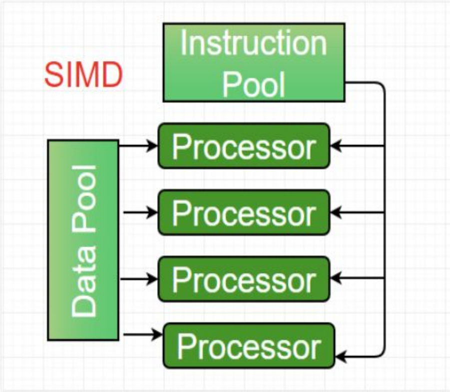
Food for thought: What are some of the challenges of different forms of concurrency?

# Flynn Taxonomy



# Parallelism

Accelerating computations at different stages by reformatting data and utilizing multiple processors

# Parallelism

By splitting a program into smaller chunks, then processing them simultaneously, we can speed up the overall runtime

- Instruction Level: Pipelining ✔
- Data Level (DLP): SIMD (Single Instruction, Multiple Data) ⭕
- Thread Level (TLP): OpenMP ⭕
- Request Level: Spark + Mapreduce ⭕

In general, when are programs parallelizable?

- No Dependencies (code doesn’t have to wait on another operation)
- No Data Races (code doesn’t access or alter same data values simultaneously)

# Data Level Parallelism

- SIMD (Single Instruction, Multiple Data)
  - Execute one operation on *multiple* data streams
  - **Vectors** store multiple pieces of data, then operate on it all at once
  - Special Assembly Instructions for constructing vectors (Intel “Intrinsics”, link)

**Current**

```
for (int i = 0; i < 16; i++) {
    A[i] += A[i];
}
```

A[1] = A[1] + A[1]

Repeat instruction x16

**SIMD**

```
for (int i = 0; i < 4; i+=4) {
    A[i..(i+3)] += A[i..(i+3)];
}
```

| | | | | |
|---|---|---|---|---|
| A[0]<br>A[1]<br>A[2]<br>A[3] | = | A[0]<br>A[1]<br>A[2]<br>A[3] | + | A[0]<br>A[1]<br>A[2]<br>A[3] |

Repeat instruction x4

# Thread Level Parallelism

- Threads: Collection of instructions
  - Formerly, one program = one thread of execution (i.e. All program instructions on single thread)
  - Parallelization = assign subsets of program's instructions to different threads...
    - To be executed simultaneously by multiple cores
    - Multiple threads share memory + resources (i.e. L3 cache, instruction / execution units)
  - Only possible with no dependencies between threads (subsets of instructions don't require data from one another, similar to pipeline *data* hazards)
- Potential Issues:
  - False Sharing
  - Data Dependencies (mentioned above)
  - Data Races + Race Conditions (Introduces non-determinism)

# Open MP

- Popular C, C++ Plugin for thread level parallelism (multithreading)
- Features
  - Direct access to individual threads constituting a single program
  - Divide work across threads by thread ID
  - Indicators specify how to wrap + handle code
- Commands Overview
  - get_thread_num(): returns number of threads on machine
  - get_thread_id(): returns id of thread working on the program
  - #pragma omp parallel for: each thread handles a sub-range of entire for loop
  - #pragma omp parallel: everything within enclosing brackets of this command is parallelized

```
for (int i = 0; i < n; i++) { … }
```

```
#pragma omp parallel for
for (int i = 0; i < n; i++) { … }
```

```
#pragma omp parallel {
#pragma omp for
for (int i = 0; i < n; i++) { … }
}
```

*Parallelized* ➡️

# False Sharing

## Example

- 16 Byte Block Cache for each process
- 4 Byte (32 Bit) Integers

| Thread A | Thread B |
|---|---|
| `arr[1] = 2`<br>`arr[3] = 4` | `arr[0] = 1`<br>`arr[2] = 3` |

Recall, if one piece of data in a block is changed, the *entire* block is updated in the cache.

Here, arr[0] … arr[3] exists in same block

- Thread A => Updates B's Cache
- Thread B => Updates A's Cache

Even though, A, B don't use one another's data

## Definition

1. Two Programs access data within same block, but accessed data is different
2. Cache coherency protocol requires us to update entire block in cache of each process, even though data used by each thread is unique + disjoint

## Impact

Performance degradation, causes distributed programs to run in slower, serial-like speed. (This doesn't impact correctness)

# Data Races

## Example

```
int sum = 0;
#pragma omp parallel for
for (int i = 1; i <= 2; i++)
    sum += i;
```

Our expectation...

| Thread A | Thread B |
|---|---|
| read sum = 0 | - |
| write sum = 1 | - |
| - | read sum = 1 |
| - | write sum = **3** |

But what stops our distributed code from executing according to a different pattern...

| Thread A | Thread B |
|---|---|
| read sum = 0 | - |
| - | read sum = 0 |
| write sum = 1 | - |
| - | write sum = **2** |

| Thread A | Thread B |
|---|---|
| read sum = 0 | - |
| - | read sum = 0 |
| | write sum = 2 |
| write sum = **1** | |

## Definition

- “Time of Check to Time of Use” Problem
- 2+ Threads attempt to modify the same variable (almost) simultaneously, and in the process, overwrite one another. Non-deterministic results.
- Common issues: “count”, or variable storing aggregation of some form

# Amdahl's Law

**Motivation**: Formula for quantifying the amount of speed up that should be attributed to a particular enhancement

**Generally**, Speed Up= Enhanced Execution Time / Normal Execution Time

**Formula**: $\text{Speedup} = \dfrac{1}{(1-F) + F/S}$

F: Fraction of program with speed up
S: Speed Up Factor

**Observation**: Max amount of speedup that can be achieved is limited by non-parallel / non-sped up portion of the program