L18: Parallelism and Concurrency II

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

March 12th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Recap from Last Week

- Parallelism vs. Concurrency
- Parallel for-loop
- Fork-join
- Cost analysis
  - And why should we care about the cost analysis
- Constructs that work well with parallelism
  - Map, filter, reduce
  - Basically, think if we can draw a parallel dataflow graph

Examples of Map

- **fn map<F, R>(self, map_op: F) -> Map<Self, F>**
  - Self need to be iterable

- Let's say we want to do fib(1) to fib(100)

```
let vv: Vec<u32> = (1..100).collect();
let xx: Vec<u64> = vv.par_iter().map(|&n| fib(n)).collect();
```

(1, 2, 3 ..., 100)

map function (what to do)

Input

(f)

↳ Parallel iterator (self)

fn function name (,)

- What else do we need to do?
  - Fork-join

```
rayon::join(|| fib(n-1), || fib(n-2))
```

Examples of Reduce in Rust

- You can use rayon's filter, map and reduce to perform parallel operations

Parallel

(1 + 2 + 3 + .. + n)

- Let's say we want to do a parallel sum

```
let vv: Vec<u32> = (1..50000000).collect();
let x = vv.par_iter().reduce(|| 0, |a, b| a + b);
```

Initial Value   Input   Function

- I will show parallel mergesort after the project presentation
  - Because some of us are working on this code

.reduce Method

(Keep pair number until having 1 element left)

Ex $1 + 2 + 3 + 4 + \dots + 127 + 128$

3 7 251 255

10 506

Examples of Filter in Rust

- How many primes between 1M to 20M

```
fn is_prime(n: u32) -> bool {
  let cutoff = (n as f64).sqrt() as u32 + 1;
 (2u32..cutoff).all(|c| n % c != 0)
}
```

Check if n is prime

→ Check if n % c != 0

Generate all number up to √n

1M - 2M

1M - 2M

```
let count = (1000000u32..20000000).collect().par_iter().filter(|&n|
is_prime(n)).count();
```

true elements

(Count how many primes)

Threadpool in Rust

- We can create a pool of threads waiting for tasks to be allocated for concurrent execution

- Code is longer than this slide → See threadpool.rs

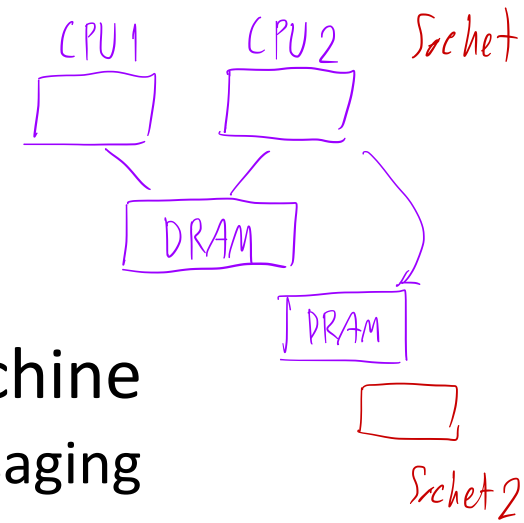
- Note
  - Random scheduler and random duration **is generally bad**
  - Think of it this way: what if you go shopping and then
    - We pick who to checkout randomly
    - Each person can spend a random time getting checked out
  - Still, this is better than sequential execution

Not on Final

Getting More Performance

Traditional Multi-threading - A lot of thread

- Each thread is assigned to run on one of the CPUs
- Threads can run this on a single node machine in parallel
  - Shared memory synchronizes data across threads
    - Using locks to ensure correct load and store



(Not in Exam)

  - This also applies to NUMA machines
- Threads can run across a cluster of machine
  - Shared data is synchronized through messaging
    - Use remote procedure calls

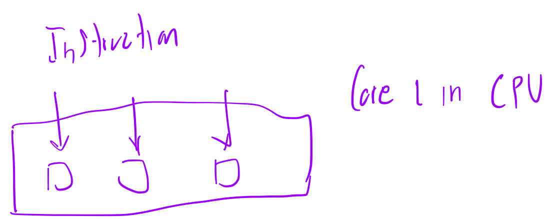
Getting Performance

- Ok, let’s say we write a nice parallel program

- Two big questions:
    - How does this give us performance?
    - What else can we do to get even more throughput?

How Fast Is My Program?

- Instruction throughput
  - How many instructions we can execute within a unit of time
- If you want to know the upper limit
- Assuming load/store is free
  - CPU clock speed
    - Tells the minimum time it takes to execute 1 instruction
  - CPU issue width
    - Tells the number of instructions the CPU can issue per cycle



How Fast Is My Program?

- Instruction throughput
  - How many instructions we can execute within a unit of time
- If you want to know the upper limit
- Assuming load/store is free
  - CPU clock speed
    - Tells the minimum time it takes to execute 1 instruction
  - CPU issue width
    - Tells the number of instructions the CPU can issue per cycle
  - Possible maximum instruction throughput
    - ~ $[1/CPU_{clock}] * CPU_{issue\_width}$ - Ideal Performance
- Problem? Memory instruction is not free!

Things That Kill Performance

- Memory Instructions
  - Load/store takes > 10x longer to do
  - Especially when you need to share a big data structure
    - Which does not really fit in the cache
  - CPU is designed to hide these long latency loads
    - By imitating a dataflow program
    - I will cover this very important topic in ICCS 221: Comp. Arch.

Execution Order

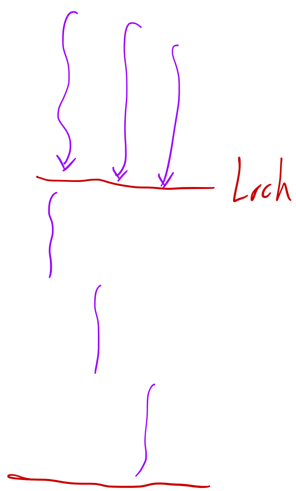
(Dependency)

Things That Kill Performance

- Branches (If-else, loop) Limit the number of for loop and if else
  - Reduce the possible optimization compiler can do
    - We will cover this
  - Hardware suffers from branch misprediction
    - I will cover this very important topic in ICCS 221: Comp. Arch.

Things That Kill Performance

- Data synchronization and barrier
  - Every time you lock things → Lower performance
  - This get complicated as your L1 cache data can be out-of-date
    - I will cover this very important topic in ICCS 221: Comp. Arch.



Fix: Make it read only

↳ Everyone can share data in parallel

Traditional Multi-threading

- Each thread is assigned to run on one of the CPU
- If #threads >>> # CPUs
  - Each thread takes turn running on the CPU
  - Context switch: switching between one thread to another
  - OS and HW → Create the policy to context switch
    - This heavily determines performance

- What about hardware multithreading?
  - 1 CPU can run multiple hardware threads

Multi-Program Execution

- Why are we only execute one thread of execution?
  - Limited intermediate resources
- **Idea:** Maintain multiple contexts and multiple threads of execution
- Each of these thread runs a different process
  - What is the difference between a process and a program?

Simultaneous Multithreading

- When the processor is waiting for the memory
  - Switch to another thread
  - Keep the pipeline full
  - Allow concurrent execution of useful work
    - Intel hyperthreading

Simultaneous Multithreading

- When the processor is waiting for the memory
  - Switch to another thread
  - Keep the pipeline full
  - Allow concurrent execution of useful work
    - Intel hyperthreading

- Tradeoffs:
  - Supporting multiple threads can be costly
    - Need to save the state of each thread
    - Need logic to recover these states quickly when switching
  - Switching upon L1 miss vs. L2 miss
    - Performance vs. cost

Fine-grain Multithreading

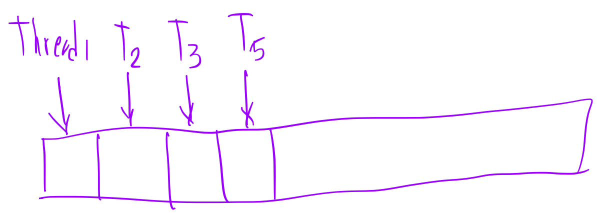
- Simultaneous Multithreading on steroid
- **Idea:** Upon a pipeline stall → Switch to a new thread

- How many threads do you need if:
  - Add → 1 cycle
  - Multiply → 5 cycles
  - L1 access → 1 cycle
  - L2 access → 5 cycles
  - Memory access →20-100 cycles
  - Unlimited memory bandwidth
- Note that this number is false with limited bandwidth
  - Why?

GPU = FMT + SIMD/SIMT

- Group similar instructions into the same unit
  - This is called a warp (NVIDIA) or wavefront (AMD)
  - Threads in a warp/wavefront execute the exact same code

- Perform fine-grain multithreading on these warps



→ Warp

(Execute the same code)

↳ Work on matrix multiply

GPU = FMT + SIMD/SIMT

- Group similar instructions into the same unit
  - This is called a warp (NVIDIA) or wavefront (AMD)
  - Threads in a warp/wavefront execute the exact same code
- Perform fine-grain multithreading on these warps
- Everything that kills multithreaded performance applies here
  - Branches
  - Diverging memory instructions

In-class Exercise 18

- What is the depth and the work for our parallel sum?

```
let vv: Vec<u32> = (1..50000000).collect();
let x = vv.par_iter().reduce(|| 0, |a, b| a + b);
```