LECTURE 21

Theory and Design of PL (CS 538)

April 08, 2020

PARALLELISM

WHAT IS PARALLELISM?

- Multiple tasks executing at same instant in time
  - Think: multicore, datacenter, supercomputer
- Property of *actual execution on hardware*
  - Not property of language/program

WHY PARALLELISM?

1. We want to go really fast!
2. We're getting more and more cores on CPUs
   - CPU clocks aren't getting much faster
3. Custom chips becoming more common
   - GPUs, ASICs, TPUs, …

EXAMPLES

DATA PARALLELISM

- Divide up data into a bunch of pieces
  - Useful when you have a lot of homogeneous data
  - Image data, log files, training examples, ...
- Process parts independently, usually in same way
- Wait for tasks to finish, collect results
- Examples: MapReduce, Hadoop

TASK PARALLELISM

- Divide up task into a bunch of pieces
- Try to run tasks at the same time
- Complications
  - Some tasks may depend on other tasks
  - Often unclear how to split up a complex task
  - Scheduling tasks makes a big difference

BIT-LEVEL

- Instruction-set architecture level
- Use bigger instructions to operate on more data
- Get more done with every instruction
- Examples
  - 16-bit, 32-bit, 64-bit microprocessors
  - SIMD: single instruction, multiple data

INSTRUCTION-LEVEL

- Instruction-set architecture level
- Run instructions themselves in parallel
  - Each clock cycle, execute multiple instructions
- Common in all modern processors
  - Pipelining
  - Gain when instructions don’t interfere

MULTICORE

- Package several processors into one
- 4-, 8-, 16-, 32-cores are not unusual
- Each core is almost a separate CPU

GPUS AND ASICS

- Application Specific Integrated Circuits
- Specialized chips for specialized tasks
  - Really, really efficient for certain tasks
- Examples
  - GPUs: processing graphics
  - TPUs: training neural networks
  - ASICs: mining bitcoin

DISTRIBUTED COMPUTING

- Geographically spread-out computers
- Grid computing: borrow time from idle computers
  - SETI@Home, protein folding, …
- Datacenters

SUPERCOMPUTERS

- Really, really big computers
  - Footprint of several basketball courts
  - Hundreds of miles of cabling
- Weather prediction, computational biology, …
- Massively parallel
  - Millions, or even tens of millions of “cores”

CHALLENGES

DATA RACES

- Two requirements:
  1. Multiple tasks read/write same piece of data
  2. Final state depends on the interleaving
- This is almost never what you want!
  - Interleaving is not under programmer control
  - Data race: result not under programmer control

EXAMPLE

```
X := 0; X := 1        ||        Y := X
```

- Final result depends on when `Y := X` is executed
  - Earlier: `Y` ends up 0
  - Later: `Y` ends up 1
- Easy to see in small programs, harder in big programs

“HEISENBUGS”

- Unpredictable behavior
- Sometimes show up, sometimes don’t
- Very hard to reproduce and debug

> *"We’re hitting this catastrophic bug every 3 months or so. Can you fix it?*

(IN)FAMOUS RACE CONDITIONS

- Many, many security vulnerabilities due to races
- Therac-25 radiation therapy machine
  - Serious injuries to patients
- GE energy management system
  - Caused Northeast blackout of 2003
  - Two day outage, more than 50 million affected

FEEDING THE CORES

- When some steps get faster, bottlenecks shift
  - "Amdahl's law"
- How to effectively use cores?
  - If they sit idle, waste time
  - 4 cores? 16 cores? 128 cores?
- How to get data to where it is needed?
  - Communication takes time
- How to synchronize?

COMPILER AND HARDWARE ARE OUT TO GET YOU

- Compiler may reorder instructions to optimize
- Hardware also reorders instructions to go fast
- Rules for which reorderings are OK is… not clear
  - Formally captured by a “memory model”
- Most languages use the “C11 memory model”
  - C11 memory model not really formalized

FORK-JOIN MODEL

EXPRESS PARALLELISM IN PL

- Programmer knows something about the data
- Can help compiler decide how to divide tasks
- Indicate parts that can safely be done in parallel

FORKING

- Parent thread spawns child to execute some function
- Parent thread doesn’t wait for child, keeps going
- Child executes, hopefully in parallel with parent

JOINING

- Wait for another thread to finish before continuing
  - “Block on another thread”
- Example: wait until all child tasks are done
  - Need to synchronize to collect results

IN RUST: RAYON

DATA PARALLELISM CRATE

- Name refers to Cilk: C/C++ parallel extensions
- Simple interface to write data-parallel stuff
- Often: change `into_iter` to `into_par_iter`

EXAMPLE: SEQUENTIAL

- Suppose we have:
  - List of a bunch of shops
  - List of products we care about
- Want to compute: sum of prices across all stores

```
let total = shops.iter()
                 .map(|store| store.compute_price(&products))
                 .sum();
```

EXAMPLE: PARALLEL

- Using Rayon: easy to make this computation parallel
- Each task shares `products`, but read-only: no races!

```
let total = shops.par_iter()
                 .map(|store| store.compute_price(&products))
                 .sum();
```

BUILDING BLOCK: JOIN

- Run two closures in parallel, wait until done

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
   if v.len() > 1 {
       let mid = partition(v);  // pick pivot index, partition v
       let (lo, hi) = v.split_at_mut(mid);
       rayon::join(|| quick_sort(lo),
                   || quick_sort(hi));
   }
}
```

REF RULES: PREVENT RACES

- Can only have one mutable ref to data at a time
- Can't have mutable and immutable refs at same time

```
fn quick_sort<T:PartialOrd+Send>(v: &mut [T]) {
    if v.len() > 1 {
        let mid = partition(v);  // pick pivot index, partition v
        let (lo, hi) = v.split_at_mut(mid);
        rayon::join(|| quick_sort(lo),
                    || quick_sort(lo));  // <-- oops
    }
}
```

UNDER THE HOOD

- Program suggests parallelism, but doesn’t require
- Library free to decide when and where to execute
- Goal: balance out work among all cores
- Work-stealing parallelism
  - Each core has a public queue of tasks
  - If core finishes early, steal from other queues

CONCURRENCY

WHAT IS CONCURRENCY?

- Tasks can make progress over *overlapping* periods
- Concurrency is a property of *two* things:
  1. Low-level execution (hardware level)
  2. High-level concept of a “task” (PL level or higher)

NOT SAME AS PARALLELISM

- Concurrency without parallelism
  - Multiple threads on a single core processor
  - Each task is a thread, tasks overlap in time
- Parallelism without concurrency
  - SIMD parallelism: single instruction, multiple data
  - One task, operate on multiple data at same time

WHY CONCURRENCY?

- Tasks are a useful abstraction for programmers
  - Natural way to organize systems, group code
  - Threads for UI, listening to network, writing file
- Don’t need to manually specify interleaving
  - Programmer usually can’t plan interleaving
  - “Run whatever is ready, I don’t care what order”

CHALLENGES

INTERLEAVING EXECUTION

- Scheduler decides which task to run, for how long
  - Actual execution switches rapidly between tasks
- Scheduler is not controlled by the programmer
  - Tasks can be paused, restarted at any time
  - Order may appear *non-deterministic*, random

HARD TO THINK ABOUT!

- One thread with 100 instructions
  - One possible ordering
- 2 threads with 100 instructions
  - 10^{344} possible orderings
- 1000 threads with 100,000 instructions
  - A whole lot of possible orderings

> *If even one interleaving has a bug, the whole program has a bug*

CONCURRENCY BUGS: BAD

- Can be intermittent
  - Sometimes there, sometimes not
- May be very rare, but still serious
  - Every 7 months, system wipes all files
- Very hard to reproduce
  - Don't know which interleaving caused bug