CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 26: Thread-level Parallelism

Instructors: Dan Garcia, Justin Yokota

Computing in the News

Python-based compiler achieves orders-of-magnitude speedups

“The user simply writes Python like they’re used to, without having to worry about data types or performance, which we handle automatically — and the result is that their code runs 10 to 100 times faster than regular Python. Codon is already being used commercially in fields like quantitative finance, bioinformatics, and deep learning.”

Specifically, they took roughly 10 commonly used genomics applications written in Python and compiled them using Codon, and achieved five to 10 times speedups over the original hand-optimized implementations. Besides genomics, they explored applications in quantitative finance, which also handles big datasets and uses Python heavily. The Codon platform also has a parallel backend that lets users write Python code that can be explicitly compiled for GPUs or multiple cores, tasks which have traditionally required low-level programming expertise.

Terminology

- A **program** is a sequence of instructions to run (such as an executable)
- A **process** is the actual execution of a program. Each process is a largely separate entity, with its own memory space.
- Each process is composed of **threads**, which are independently running instruction sequences that share most memory.
- The datapath we've discussed so far is a CPU **core**. A single core can run one thread at any given time.
- The CPU is composed of multiple cores, which thus allows multiple threads to be run simultaneously.

Terminology

- A **single-threaded program** is a program that only runs one thread
  - All the programs we've discussed so far are single-threaded
- **Multi-threaded** and **multi-process** programs are programs that use multiple threads or processes.
- The **Operating System**, or OS, is responsible for managing which threads get run on which CPUs (among other tasks)
  - More on this Friday
- On most modern computers, number of active threads >> number of available cores, so most threads are idle at any given time
  - This is one of the big reasons why runtime can vary, even when running the same program
  - For Project 4, we'll run programs on dedicated systems, so you have the full resource allocation

Agenda

- Thread-Level Parallelism
- OpenMP Syntax
- Locks and critical segments

Why Parallelism?

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \quad \frac{\text{Cycles}}{\text{Instruction}} \quad \frac{\text{Time}}{\text{Cycle}}$$

- Instructions/Program
  - Reducing the work/problem involves a lot of theory, and we'll hit theoretical limits eventually.
  - Work per instruction can be increased with SIMD, but that's at most an 8x speedup now that AVX512 is dead
- Cycles/Instruction
  - Clever ordering of instructions can help, but pipelining already optimizes this fairly well
  - Memory instructions have high cycles/instruction, so we can improve this via caches
- Time/Cycle
  - Up until recently, our limit here was manufacturing capacity, so we were able to halve this every ~2 years via Moore's Law
  - Now, we're starting to hit physical limits, so Moore's Law has slowed down quite a bit.
- Only way to move forward is to parallelize
  - Increase the number of processors that get used by a single program

Parallel Computing

- Instead of just one thread, let's write a program that runs with multiple instruction sequences simultaneously
- Why?
  - Our computer has 4 cores, so we can use them all
  - If a thread is waiting on memory accesses, might as well set up another thread to keep working
  - Supercomputers are basically normal computers with hundreds or thousands of cores, so if we want to write code for a supercomputer, we need to parallelize
- Two main choices:
  - Increase the number of threads per process: Today
  - Increase the number of processes for a program: Wednesday

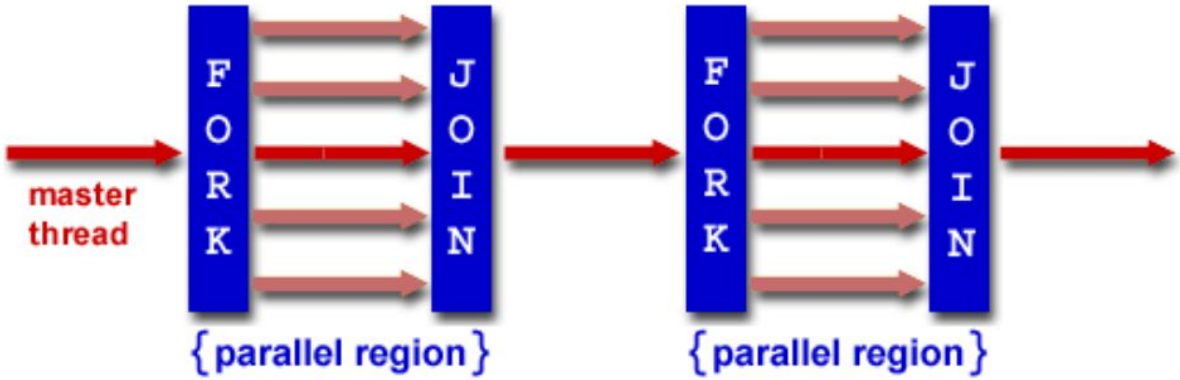
Multithreading vs Multiprocess Code

- Threads: Different instruction sequences on the same process
  - Threads on the same process share memory
  - "Easy" to communicate
  - Limited to a set of cores wired to the same memory block (1 node)
- Processes: Largely independent from each other
  - Different processes can't share memory
  - "Difficult" and time consuming to communicate
  - Can expand to as many cores as you have available, over as many nodes as you want
- Example: The Savio cluster is Berkeley's High Performance Computing system. It's main cluster has 112 nodes, each with 32 cores.
  - With single-threaded code: Max 1 core usable
  - With multi-threaded code: Max 32 cores = 32x speedup possible
  - With multi-process code: 112x32 cores = 3584x speedup possible
- More info: CS 267

Multithreading Framework Overview

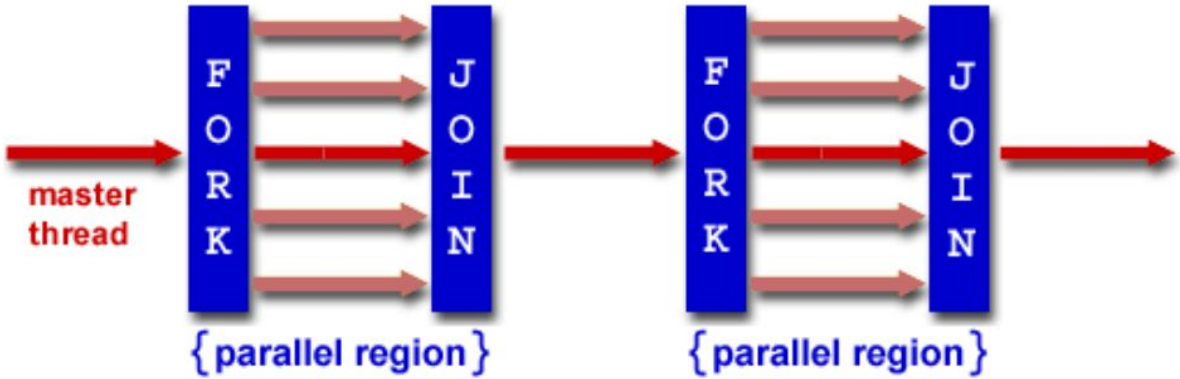
- Each thread has its own registers
- Each thread has its own PC
- Each thread has its own stack
- Each thread shares the same heap
- Communication is done through shared memory
- Threads run simultaneously, so we have no control over the order in which threads do their work

Multithreading: Fork-Join Model



- Many different multithreading models, but for this class, we'll consider the fork-join model
- Program starts serial
- Fork: Master thread creates multiple threads for a segment. Divide the work to all threads
- Join: Wait until all threads finish their work, synchronize, and terminate all but the master thread
- Fork and Join take a while (on the order of a few thousand memory operations), so goal is to minimize the number of forks/joins, and minimize the serial parts.

Multithreading: Fork-Join Model in Human Terms



- Let's say I want to make a big mural
- Step 1: I plan out things on paper, set up the wall for painting, etc.
- Step 2: Find a bunch of other people to help me paint (takes some time)
- Step 3: Assign each person some chunk of the wall to paint
- Step 4: Wait until the last person finishes painting their section
- Step 5: I do some cleanup, last minute checks, etc.

Multithreading: Scaling Efficiency

- Main problems: Minimize the nonparallelizable portion, and balance the load
- Minimize the time spent doing solo work (and overhead in finding people)
  - If the mural is small enough, it'll take more time to find people to help out, so might as well do it myself
- Strong scaling: If I double the number of people working, how much faster does the problem get (ideally close to 2x faster)?
- Weak scaling: If I double the number of people working AND double the mural size, how fast is it now (ideally close to 1x)?
- Load balancing
  - We're limited by the person who takes the most time to finish
  - Not everyone paints at the same speed, some parts of the mural might have more detail than others and therefore take longer to paint
  - Often impossible to perfectly load balance, so we have to make do with "close enough" and "statistically, everyone should have about the same amount of work"

OpenMP

- OpenMP is an extension of C used for multi-threaded code (shared memory, so no multi-node computation)
- Compiled with the additional flag "gcc -fopenmp foo.c"
  - "#include <omp.h>"
- Standardized over many languages
- Generally follows the fork-join framework
- Each thread has its own stack for private variables, but otherwise shares memory with other threads
- OpenMP code generally is written using lines like "#pragma omp <command>"

#pragma omp parallel

- In order to create a parallel section:

```
#pragma omp parallel
{
    //parallel code
}
```

- C syntax note: brackets mean "previous declaration applies to the all the lines inside me". This means that if we write only one line of code in a parallel section (or if clause, or for loop), we don't need to write brackets.
- The code in the parallel section gets run on all threads, so we need some way to distinguish threads
- In a parallel segment:
    - `omp_get_num_threads()` returns the number of threads running
    - `opm_get_thread_num()` returns a unique number from 0-num_threads per thread

Parallel Hello World


```
#include <stdio.h>
#include <omp.h>
int main () {
    int x = 0; //Shared variable
    #pragma omp parallel
    {
        int tid = omp_get_thread_num(); //Private variable
        x++;
        printf("Hello World from thread %d, x = %d\n", tid, x);
        if(tid==0) {
            printf("Number of threads = %d\n", omp_get_num_threads());
        }
    }
    printf("Done with parallel segment\n");
}
```

Shared vs Private variables

- By default, any variable declared outside the parallel segment is shared: all threads write/read from the same variable
- Any variable declared inside the parallel segment is private: Each thread has its own version of the variable.
- Can overrule this with the private and shared keywords:

```
#pragma omp parallel private(var) shared(var2)
```

- Note that heap memory is always shared, though we can have private pointers and private mallocs (if we do the malloc in the parallel segment, and free them within the parallel segment)

For loops

- Problem: You have to do some work over an array of 1 million numbers, with 4 people. How do you split the work?
- Assumptions:
  - We need to decide this before we run the code
  - Each element of the array is independent, so we can do this in any order
  - The threads are about equally fast at work, so we want to assign each of them 250k numbers

For loops

- Option 1: Do every 4
  - `for(int i = tid; i<1000000;i+=4)`
  - "Interweaving"
- Option 2: Thread 0 does 0-249999, 1 does 250000-499999, etc.
  - `for(int i = tid*250000;i<(tid+1)*250000;i++)`
  - "Blocking"
- Which one's better?
  - With standard multithreading, Option 1 actually is as slow as the serial version of this code due to cache coherency issues… More on this when we cover caching.
  - Option 2 speeds things up correctly
  - Aside: Option 1 does end up being the preferred option when dealing with GPU programming, though that's due to how GPU threads differ from normal threads.

For loops

- Option 3: Let the compiler do it for you with the `#pragma omp for` keyword.
- `#pragma omp for` must be written inside an already existing parallel segment
- If a parallel segment consists only of one for loop, we can combine the two declarations with `#pragma omp parallel for`
  - ```
    #pragma omp parallel for
        for(int i = 0; i<1000000;i++)
    ```
    ```

Data Races

- Note that when we ran Hello World parallel, we ended up with the threads running in random order
  - In fact, every time we run Hello World, we get a different order!
  - The x values stayed largely in-order, but didn't always strictly increase
- Recall the OS can choose whichever threads it wants to run, and change threads at any time
- This is one of the biggest downsides to multithreading: A multithreaded program is no longer deterministic, and will have a random execution order every time we run the program.
- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

Data Race: Example

- If we run this code on 4 threads, what possible values could x be at the end?

```
int x = 0;
#pragma omp parallel {
    x = x + 1;
}
```

If we run this code on 4 threads, how many different possible values could x be at the end?

1

2

3

4

5

6

If we run this code on 4 threads, how many different possible values could x be at the end?

1

2

3

4

5

6

If we run this code on 4 threads, how many different possible values could x be at the end?

1
2
3
4
5
6

Data Race: Example

- To analyze this, we need to see the equivalent assembly code
  - C will compile to x86, but we can still do a correct analysis by compiling to RISC-V, since we're mainly trying to reduce the code to atomic instructions. We can assume that no two atomic instructions happen simultaneously.
- Only the loads and stores affect shared memory, so we only need to consider the different ways we can order the loads and stores
- Even with this, there are $8!/(2!)^4$=2520 different possible orders
  - Can use the fact that all the threads are identical to reduce this to 105 orders, but still too many to check manually

```
                    sw x0 0(sp)
lw t0 0(sp)     lw t0 0(sp)     lw t0 0(sp)     lw t0 0(sp)
addi t0 t0 1    addi t0 t0 1    addi t0 t0 1    addi t0 t0 1
sw t0 0(sp)     sw t0 0(sp)     sw t0 0(sp)     sw t0 0(sp)
```

Data Race: Example

- Case 1: All the threads run one at a time
  - Purple thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread reads x = 1
  - Brown thread stores x = 2
  - Red thread reads x = 2
  - Red thread stores x = 3
  - Blue thread reads x = 3
  - Blue thread stores x = 4
- Final value: 4

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
```

Data Race: Example

- Case 2: The threads are perfectly interleaved
  - Purple thread reads x = 0
  - Red thread reads x = 0
  - Brown thread reads x = 0
  - Blue thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread stores x = 1
  - Red thread stores x = 1
  - Blue thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Case 3: Same as case 1, except purple's store happens last
  - Purple thread reads x = 0
  - Brown thread reads x = 0
  - Brown thread stores x = 1
  - Red thread reads x = 1
  - Red thread stores x = 2
  - Blue thread reads x = 2
  - Blue thread stores x = 3
  - Purple thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Some other ordering?
  - We can find orderings that give x=2,3
- Can we do any more/less?
- Can't go above 4
  - Only 4 "+1s" overall, so can't increase to 5 or more
- Can't go below 1
  - The smallest value that can be loaded by a thread is 0, so the smallest value that can be stored is 1. Therefore, the last store must be at least 1.
- Therefore, we can get any value between 1 and 4

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Avoiding Data Races

- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.
- For today: if you make sure that each thread works on independent data (no two threads write to the same value, or read a value that another thread wrote to), you can guarantee correctness
- Wednesday: how to handle cases where coordination is mandatory
- The hardest part of multithreading is maintaining correctness while also speeding up the code, but this ends up being a fairly transferable skill to management
    - If you can coordinate a group of threads to perform a task, you can coordinate a group of people to perform a task more easily.

The Coordination Game

- As a class, get exactly 5 votes in each answer choice
- For this poll, you will NOT be able to change your vote

Get exactly 5 votes in each answer choice

A

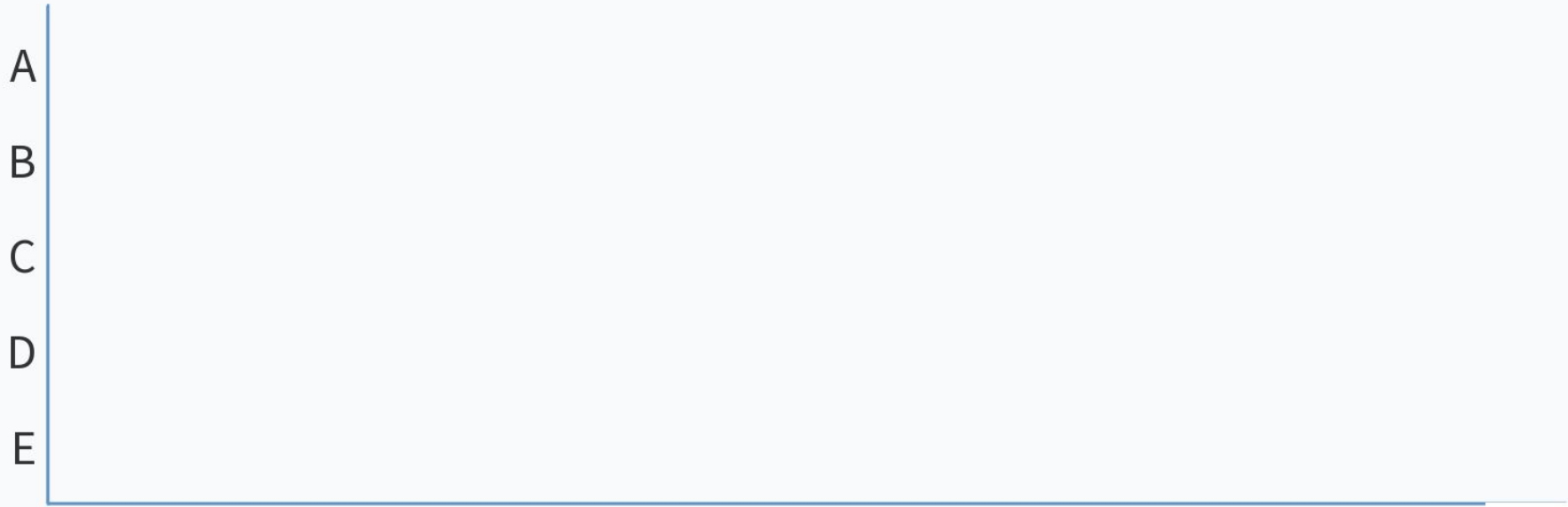
B

C

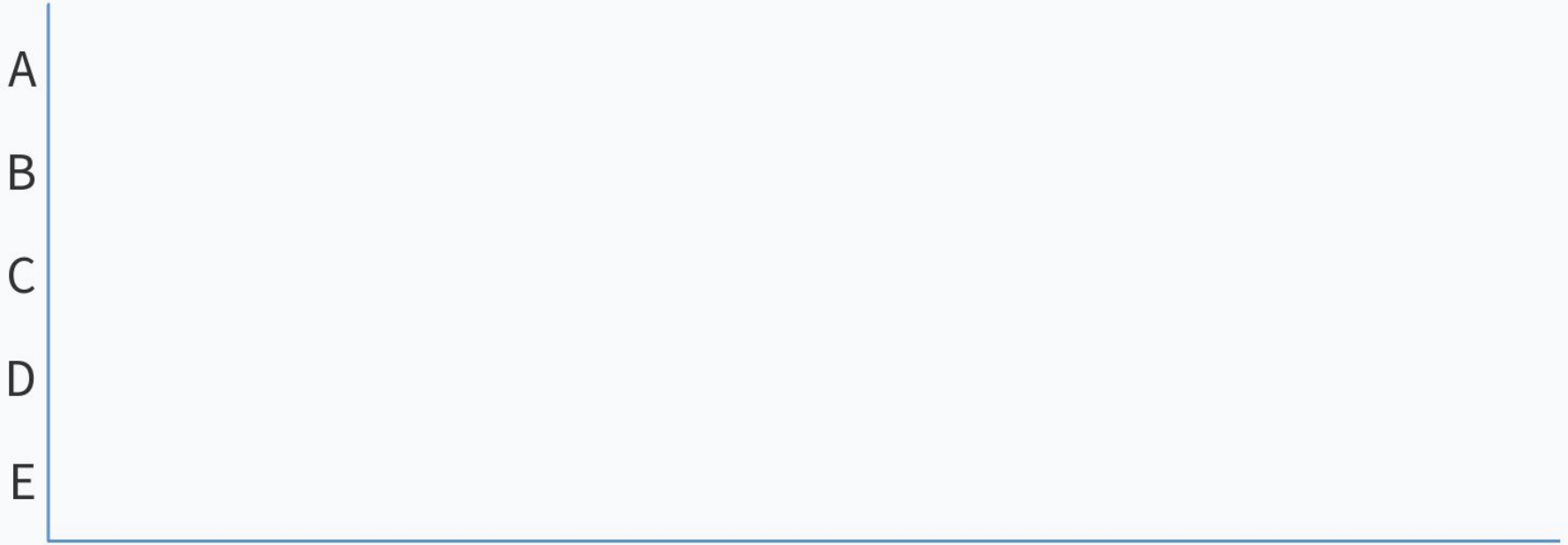
D

E

Get exactly 5 votes in each answer choice



Get exactly 5 votes in each answer choice



The Coordination Game

- At the start of Wednesday's and Friday's lectures, we'll play a similar game (not the same win condition, but a similar coordination problem).
- If you succeed, all students in the class will receive 3x(fraction of the class that responds) extra credit points
- You may discuss strategies as much as you wish
- Good luck!