

CS 4414/5416: Systems Programming

Intro to Concurrent Programming

David Bindel

2026-10-20

Logistics

- P4 (netsim) checkin due Monday, 10/26
- Lab 9 is Friday: working on P4

- Final is Dec 16 at 9 AM
- Alternate time request is linked here (and is posted on Ed)

- Attendance (from paper era) should be up to date
- PollEv through 10/6 is freebies
 - If you still have trouble post 10/6, reply #225 on Ed
- P4 should go up tonight
- Makeup midterm ETA is now end of week (with apology)

Interlude

Why Are You Here?

Halfway through the semester: why are you (still) taking this class?

Why Am I Here?

*There are so many things that I think you should know.
And that's why I'm bothering telling you so. ...*
– Dr. Seuss, *Oh Say Can You Say*

- I like to switch things up sometimes!
- It was a chance to really learn Rust
- And someone needs to teach this stuff

Why Am I Asking This?

The best thing for being sad .. is to learn something. That is the only thing that never fails. ... That is the only thing which the mind can never exhaust, never alienate, never be tortured by, never fear or distrust, and never dream of regretting.

– T.H. White, The Once and Future King

Courtesy *The Marginalian*.

Recap

This module: Concurrent programming with threads

- Message-based concurrent programming
- Shared memory concurrent programming
- Synchronization and deadlock
- Concurrent network programming

- Today
 - Concurrency and parallelism
 - Threads in Rust
 - (Start) message passing with channels
- Thu: More advanced message passing + P4

Concurrent programming

- Overlapping logical execution flows
- Could be multiple processes or threads
 - Arguably also coroutines, green threads
- Drivers:
 - Useful work while dealing with I/O
 - Responsiveness of shared resources
 - Faster computation (parallelism)

- When and where does computation happen?
- How does data move between processes/threads?
- How do we manage synchronization?

- Dataflow programming
- Message passing
- Bulk synchronous programming
- Synchronized shared data structures

and more!

Threads vs Processes

Process	Thread
Own PC and stack	Own PC and stack
Own heap (can share pages)	Shared heap
Own file descriptors	Shared file descriptors
Pre-emptive multitasking	Usually pre-emptive

Parallelism

Concurrency vs Parallelism

- Concurrency: *logically* overlapping execution
 - But may involve time-sharing a core
- Parallelism: *actually* overlapping execution
 - Think different cores for the moment
 - Though recall ILP from earlier in the course!

Consider executing on p cores vs one core:

$$\text{Speedup} = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)}$$

Usually don't get perfect speedup p , so

$$\text{Efficiency} = \frac{\text{Speedup}}{p}$$

NB: Can rarely get apparent efficiency > 1 ! Why?

Amdahl's Law



Overhead

Parallelizable work



Overhead

Parallelized work ($p=20$)

How fast can things potentially go?

- Suppose fraction s of work is serial
- Rest speeds up perfectly (by a factor of p)

Amdahl's law shows s can be a bottleneck:

$$\text{Speedup} = \frac{1}{s + (1 - s)/p} < \frac{1}{s}$$

Strong and Weak Scaling

Strong scaling: Fixed problem size, varying p

$$\text{Speedup} = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}$$

Weak scaling: Vary size, keep work/processor constant

$$\text{Scaled speedup} = \frac{T_{\text{serial}}(n(p))}{T_{\text{parallel}}(n(p), p)}$$

A very different metric!

For scaled speedup, if serial work is s :

$$\text{Scaled speedup} \leq s + (1 - s)p = p - s(p - 1)$$

This is *Gustafson's Law* (compare Amdahl).

- Get near perfect speedup on crap code!
- Keep your eye on *time to solution*
- Honest comparison should be vs tuned serial code

CS 5220: Applications of Parallel Computers

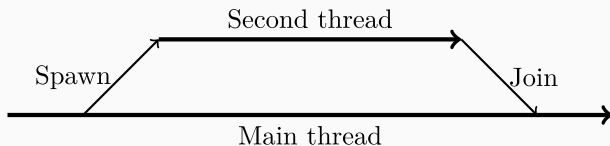
- I taught this for many years (last in 2024)
- Prof Giulia Guidi more recently (and in S27)

Which of these is true about parallel efficiency?

- (a) It usually improves with the number of cores
- (b) It is between 0 and 1 by definition
- (c) It is usually between 0 and 1, but could be greater than 1 if the serial baseline is poorly tuned

Thread basics

Spawn and Join



- `thread::spawn` takes `FnOnce() -> T`, returns `JoinHandle<T>`
- `handle.join()` returns `thread::Result<T>`
- Can also drop the handle without join
 - Result is a *detached thread*
 - Detached threads terminate on `main()` exit

```
type thread::Result<T> = Result<T, Box<dyn Any + Send + 'static>>
```

- Unrecoverable: `Err` variant on `JoinHandle`
 - `Any` is a trait to emulate dynamic typing
 - `Send` means safe to move between threads
 - Needs to have `'static` lifetime
- Recoverable: make closure return `T = Result<S,E>`
 - Get `Ok(Err(err))` for `err: E`

Spawn and Join

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Hi {i} from spawned");
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("Hi {i} from main");
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```

Map-Reduce Pattern

```
use std::thread;

fn count_even_par(v: &[u32]) -> usize {

    // Independent mappers run on parts of data
    let mut handles: Vec<thread::JoinHandle<usize>> =
        v.chunks(1024).map(|c| {
            let c = c.to_vec(); // Can do this better!
            thread::spawn(|| -> usize {
                c.into_iter().filter(|x| *x % 2 == 0).count()
            })).collect();

    // Reduce results at the end
    handles.drain(..).map(|h| h.join().unwrap()).sum()
}
```

Monte Carlo Example

```
use std::thread;
use rand::prelude::*;
use rand::rngs::Xoshiro256PlusPlus;

fn run_mc(ntrials: usize, p: f64) -> f64 {
    let mut handles: Vec<thread::JoinHandle<f64>> =
        (0..p).map(|_| {
            let mut rng: Xoshiro256PlusPlus = rand::make_rng();
            thread::spawn(move || {
                let mut x = 0f64;
                for _ in 0..ntrials {
                    x += run_trial(&mut rng);
                }
                x
            })
        }).collect();
    let n = (p*ntrials) as f64;
    handles.drain(..).map(|h| h.join().unwrap()).sum::<f64>() / n
}
```

Ask Me a Question!

Channels

Do not communicate by sharing memory; instead, share memory by communicating.

– *Effective Go*

Putting message-passing first is not accidental!

- Shared memory often *looks* simpler
 - But synchronization is hard
 - And often end up with contention from over-sharing
- Message passing (Actor model) forces being explicit

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        tx.send(String::from("hi")).unwrap();
    });
    let received = rx.recv().unwrap();
    println!("Got: {received}");
}
```

- mpsc = Multiple Producer / Single Consumer
- Unidirectional with (Sender, Receiver) pair

Multiple producers

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();
    let tx2 = tx.clone();
    thread::spawn(move || {
        tx.send(String::from("hi")).unwrap();
    });
    thread::spawn(move || {
        tx2.send(String::from("hello")).unwrap();
    });
    println!("Got: {}", rx.recv().unwrap());
    println!("Got: {}", rx.recv().unwrap());
}
```

What Does a Channel Carry?

```
// mpsc channel signature
fn channel<T>() -> (Sender<T>, Receiver<T>);

// methods in Sender<T> / Receiver<T>
fn sender(&self, x: T) -> Result<(), SendErr<T>>;
fn receiver(&self, x: T) -> Result<T, RecvErr>;
```

- Channel carries *typed* data
- Generally involves transfer of ownership
- T must implement **Send** (most types, not all)
- Errors when other end is dropped

Channels are *unidirectional* – communicate other way by

- Return values from threads (or captures)
- Separate channel passed on input
- Passing an “ack” channel via an existing channel

The Ack Pattern

```
use std::sync::mpsc::{channel, Receiver, RecvError, Sender};
use std::thread;

struct Command(String, Sender<String>);

fn echo_client(id: usize, tx: Sender<Command>) {
    let (response_tx, response_rx) = channel();
    tx.send(Command(format!("Ping {id}"), response_tx)).unwrap();
    println!("{id} received: {}", response_rx.recv().unwrap());
}

fn echo_server(server_rx: Receiver<Command>) -> Result<(), RecvError> {
    loop {
        let Command(s, tx) = server_rx.recv()?;
        println!("Server received: {s}");
        tx.send(s).unwrap();
    }
}
```

The Ack Pattern

```
fn main() {  
    let (server_tx, server_rx) = channel();  
    let mut clients: Vec<thread::JoinHandle<()>> = Vec::new();  
    thread::spawn(move || echo_server(server_rx));  
    for i in 0..10 {  
        let tx = server_tx.clone();  
        clients.push(thread::spawn(move || echo_client(i, tx)));  
    }  
    for c in clients.drain(..) {  
        c.join().unwrap()  
    }  
}
```

Ask Me a Question!

Wrap-Up

- *Concurrency*: logically overlapping execution
 - Useful for covering I/O, sharing resources
- *Parallelism*: actual overlapping execution
 - Useful for performance (and concurrency)
- Threads are one mechanism for concurrency
 - Can coordinate by fork/join, message passing, shared mem
 - Message passing (“actor model”) has some advantages

- More advanced message passing
- Intro to P4 (which brings lots together)

- Take a walk with a friend this week!
- Remember to fill the alternate final time form if relevant