L16: Introduction to Rust (and More Parallel Programming)

Rachata Ausavarungnirun

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

March 5th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Systems and Parallel Programming

- Why is this typically hard?

- Systems programming
  - Hard to secure
  - Hard to multithread

- Parallel programs - Better Performance
  - Hard to detect data races
  - Hard to debug

Rust: Type Safety

- A well-define program
  - No undefined behavior on all cases of execution

- A type safe language
  - Every program is well defined

- Is C type safe?
  - Then, why should we use C?

Python is not type safe, because of casting

(int -> float)

```
int main() {
    int *a;
    a = malloc(sizeof(int) * 2)
    int *i;
    i = a + 5
    (*i)
}
```

Performance Is King

- Performance differs across languages
  - Why? - Scala → Assembly → CPU (Different to Python)
- Computer executes an the assembly code
- Language somehow translate your program into assembly code
- And we have not even touch how hardware can be very different as well
  - See CPU vs. GPU

    ↳ Something run well in CPU, but slow in GPU

Let’s Compare Performance

- Assume a program that check if a string only consist of whitespace

Ruby Performance

- class :: String
  def blank?
  /\A[[:space:]]*\z/ == self
  end
  end

- You can run 964K of these per second

(times)

C Performance

- Let's Optimize this on a C code
  - I am going to borrow the code from https://github.com/SamSaffron/fast_blank

    L> Run every case in Parrallel

- You can run 10.5M iterations in 1 sec

- This is ~10 times faster

Rust Performance

- **extern** “C” **fn** fast_blank(buf: Buf) -> bool {
  buf.as_slice().chars().all(|c| c.is_whitespace())
  }

- buf.as_slice() gets the string slice
- .char() gets the iterator over each characters - String → Character
- Then the rest just check if there are whitespaces

- This is 11M iterations/sec

Parallel Program in Rust

- Say I want to load many images from my input paths to all the images
- **fn** load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.iter()
  .map(|path| {
  Image::load(path)
  })
  .collect() }
- paths.iter() iterates over each path
- Then you load each path's image
- Create and return a vector of images
- **This is sequentially done**

Parallel Program in Rust

- I want to parallelize this
- **extern crate** rayon - Library to perform task in parallel

```
fn load_images(paths: &[PathBuf]) -> Vec<Image> {
  paths.par_iter()
    .map(|path| {
      Image::load(path)
    })
  .collect() }
```

- rayon is a data-parallel library that convert sequential execution into parallel execution
  - https://docs.rs/rayon/1.3.0/rayon/
- paths.par_iter() iterates over each path in parallel

Detecting Data Races

- Rust compiler will tell you
- **fn** load_images(paths: &[PathBuf]) -> Vec<Image> {

```
(Val) let mut jpegs = 0;
    paths.par_iter()
      .map(|path| {
        if path.ends_with("jpeg") { jpegs += 1; }
        Image::load(path)
      })
      .collect();
  }
```

- This does not compile, why?

↳ Some Thread may read the old value, before adding it

Detecting Data Races

- Rust compiler will tell you
- **fn** load_images(paths: &[PathBuf]) -> Vec<Image> {
  let mut jpegs = 0;
  paths.par_iter()
  .map(|path| {
  if path.ends_with(“jpeg”) { jpegs += 1; }
  Image::load(path)
  })
  .collect();
  }
- Note: let bind a value to a variable
- This will not compile, why?
  - Need to lock this (or use AtomicU32)

Side Note on Atomic

- What is an atom?
- What is an atomic operation? - If something is wrong, go back to the beginning (Kill one of threads, and rewrite again)
- Why is this useful?

Ex

1000 → 0

ATM → 1000

ATM 2 → 1000

} One of them get it

Reading Rust Code

- fn defines a function

Input variable name (mutable) and types (64 bits undefined integer)

- fn gcd(mut n: u64, mut m: u64) -> u64 {
  assert!(n != 0 && m != 0); *This is a macro, ! asserts only if false*
  while m != 0 {
  if m < n { *Note no parenthesis for the condition check is ok*
  let t = m; m = n; n = t; *Create a local variable. Type is inferred*
  }
  m = m % n;
  }
  n *You can also type return n, but the last line is the return value similar to Scala*
  }

Generics

(Requirement)

Total Ordering (Whatever that can sort Ex. Integer, Object)

↓

Need to define how to compare it

- fn min<T: Ord>(a: T, b: T) -> T {
  if a <= b { a } else { b }
  }

- Ord means that T, which is a generic type have total ordering

Enumeration and Sum Types

- enum Option<T> { - Return Nothing or Some
  None,
  Some(T)
  }
- **fn** safe_div(n: i32, d: i32) -> Option<i32> {
  if d == 0 { **return** None; }
  return Some(n / d); }

- **match** safe_div(num, denom) {
  None => println!("No quotient."),
  Some(v) => println!("quotient is {}", v)
  }

Catching Errors using Result<T,E>

- Result<T, E> is basically
  - enum Result<T, E> {
    Ok(T),
    Err(E),
    }

- Let's use this to print the content of the directory and return the number of entries, or std::io::Error

Catching Errors using Result<T,E>

- use std::path::Path;
  fn list_directory(dir: &Path) -> std::io::Result<usize> {
    let mut count = 0;
    let entries = try!(std::fs::read_dir(dir));
    for entry_or_error in entries {
      let entry = try!(entry_or_error);
      println!("{:?}", entry.path());
      count += 1;
    }
    return Ok(count);
  }
- fn main() {
    let path = Path::new("/tmp");
    let dir_count = list_directory(&path);
    match dir_count {
      Ok(count) => println!("Total: {}", count),
      Err(err) => println!("Error: {:?}", err) }
  }

Memory Safety

- Rust provides three promises
- No null-pointer dereferencing
  - No Null value, use Option<T> or Result<T, E>
- No dangling pointers
  - What is a dangling pointer?
  - No garbage collectors as well
    - Use an ownership system
- No buffer overruns
  - No pointer arithmetic
  - Array in rust is not translated to pointers
  - Boundary checking at compile time

In-Class Exercise 15

- Write a rust code that check of any number from n to m is prime
  - (optional) make this parallel