Parallel Programming with Thread pools and iterators

Stefan Schindler (@dns2utf8)

March 21, 2019

Rust Zürichsee, Schweiz, CH - hosted by Cloud Solutions Amsterdam, NL

Index

About me

Timetable

- 18:30 => Venue opens, pizza's arrive
- now => Talk Stefan: Parallel Programming with Rust
- 19:30 => Break
- 19:45 => Maarten: How to speed up your builds with Bazel
- 20:15 => Discussions
- 21:00 => Venue closes
- tomorrow => ???
- the day after => parallelize the World!

About:me

Hello my name is Stefan and I work on and with computers.

I organize

- RustFest.eu Next: probably in September 2019 with ”impl days” before or after the conference
- Meetups in and around Zürich, CH
- ErnstEisprung.ch (in de Zwitserse Alpen Juli 2019)

Some of my side projects

- rust threadpool (maintainer)
- Son of Grid Engine (SGE) interface
- run your own infrastructure - DNS, VPN, Web, ...

What will we learn tonight?

- Loops
- Iterators
- Different modes of execution
- Single vs. Multi Threading
- How to synchronize pools
- Hot to translate linear into parallel code

Loops

Loops 0 - What happened so far

```
const char *data[] = { "Peter Arbeitsloser", ... };

  const int length = sizeof(data) / sizeof(data[0]);
  int index = 0;
head:
  if (!(index < length)) {
    goto end;
  }
  const char *name = data[index];
  printf("%i: %s\n", index, name);
  index += 1;
  goto head;
end:
```

Loops 1 - What improved

```
const char *data[] = {
    "Peter Arbeitsloser",
    "Sandra Systemadministratorin",
    "Peter Koch",
};

  const int length = sizeof(data) / sizeof(data[0]);

  for (int index = 0; index < length; ++index) {
    const char *name = data[index];
    printf("%i: %s\n", index, name);
  }
```

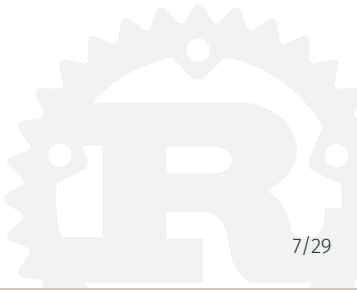
Loops 2 - What happens in rust

For the following slides keep this in mind:

```rust
#[allow(non_upper_case_globals)]
const data: [&str; 3] = [
    "Peter Arbeitsloser",
    "Sandra Systemadministratorin",
    "Peter Koch",
];
```

Loops 3 - While

```
let mut index = 0;
let length = data.len();
while index < length {
    println!("{}: {}", index, data[index]);
    index += 1
}
```



Loops 4 - For each

```
for name in &data {
    println!("{}", name);
}
```

Note the `&` next to `data`.

```
for name in data.iter() {
    println!("{}", name);
}
```

If we prefer a more functional style:

```
let iterator = data.iter();
iterator.for_each(|name| {
    println!("{}", name);
});
```

Iterators

Trait Iterator

```rust
// std::iter::Iterator
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

// For reference std::option::Option
pub enum Option<T> {
    None,
    Some(T),
}
```

```
let iterator = data.iter();
iterator.for_each(|name| {
    println!("{}", name);
});
```

- Why?
- Pros for
  - People programming (filters, maps, maintainability, ...)
  - Compiler (optimizations, early returns, edge cases, ...)

Video (32min): RustFest Rome 2018 - Pascal Hertleif: Declarative programming in Rust

- media.ccc.de/v/rustfest-rome-5-declarative-programming-in-rust
- youtube.com/watch?v=0W20GPEqbcU

Iterators 1 - Parsing without panic

```
struct Person { first_name: String, surname: String, }
let processed = data
        .iter()
        .map(|name| {
            let mut split = name.split(" ");
let (first_name, surname) = (split.next(), split.next());
if first_name.is_none() || surname.is_none() {
    return Err("Unable to parse: to few parts")
}
            Ok(Person {
                first_name: first_name.unwrap().into(),
                surname: surname.unwrap().into(),
            })
        })
        .collect::<Result<Vec<_>, _>>();
```

```
struct Person { first_name: String, surname: String, }
let processed = data.iter()
        .map(|name| {
            let mut split = name.split(" ");
let (first_name, surname) = (split.next(), split.next());
match (first_name, surname) {
    (Some(first_name), Some(surname)) => {
        Ok(Person {
            first_name: first_name.into(), surname: surname.into(
        })
    }
    _ => { Err("Unable to parse: to few parts") }
}
        })
        .collect::<Result<Vec<_>, _>>(); // <- magic happened
```

Iterators 3 - ”processed: {:#?}”

```
processed: Ok(
    [
        Person {
            first_name: "Peter",
            surname: "Arbeitsloser"
        },
        Person {
            first_name: "Sandra",
            surname: "Systemadministratorin"
        },
        Person {
            first_name: "Peter",
            surname: "Koch"
        }
    ]
)
```

Modes of Execution

Programming is ...

... about solving problems

Examples:

- Copy data
- Enhance audio
- Distribute messages
- Store data
- Prepare thumbnails

Key is understanding the problem

Single thread - Linear Execution

How to do more than one thing at the time?

- Linear if tasks are short enough
- Polling
- Event driven (select/epoll or interrupt)
- Hardware SIMD

Simultaneous Multi Threading - SMP

Let's add another level of abstraction

- spawn / join: handle lists of JoinHandles
- pools
  - job queue (threadpool)
  - Work stealing (rayon)
  - futures (tokio or async/await)

New problems: synchronization and communication

Implementation

Send and Sync

Rusts ”pick three” (safety, speed, concurrency)

`Trait std::marker::Send`

Types that can be transferred across thread boundaries.

`Trait std::marker::Sync`

Types for which it is safe to share references between threads.

Crates

Let's reuse that level of abstraction

- std::thread::spawn, join
- pools
  - ThreadPool (Job Queue)
  - FuturesThreadPool (Work stealing)
- rayon (Work stealing)
- timely dataflow (distributed actor model)

New problems: synchronization and communication

Channel example

```
use threadpool::ThreadPool; use std::sync::mpsc::channel;
let n_workers = 4; let n_jobs = 8;

let pool = ThreadPool::new(n_workers);
let (tx, rx) = channel();
for _ in 0..n_jobs {
    let tx = tx.clone();
    pool.execute(move || {
        tx.send(1).expect("channel will be there");
    });
}
drop(tx); // <- Why?

assert_eq!(rx.iter() /*.take(n_jobs)*/ .sum()
    , /* n_jobs = */ 8);
```

Channel cascade example

```
let (tx, mut rx) = channel();
tx.send( (0, 0) ).is_ok();
for _ in 0..TEST_TASKS {
    let rx_pre = rx;
    let (tx_chain, rx_chain) = channel();
    rx = rx_chain;

    pool.execute(move || {
        let r = pi_approx_random(TRIES as u64
                                 , rand::random::<f64>);
        let b = rx_pre.recv().unwrap();
        tx_chain.send( (b.0 + r.0, b.1 + r.1) ).is_ok();
    });
}
println!("chain.pi: {}", format_pi_approx(rx.recv().unwrap()));
```

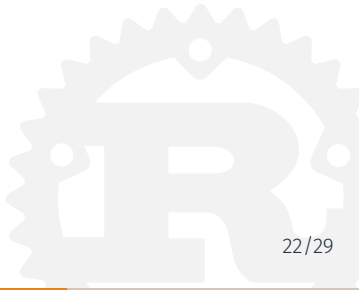
Receipt: from Loops to Iterators

Collect from Channel - 0

`v_len` holds the number of elements we expect

```
let mut pictures = vec![];

for _ in 0..v_len {
  if let Some(pi) = rx.recv().unwrap() {
      pictures.push( pi );
  } else {
      // Abort because of error
      return;
  }
}
```

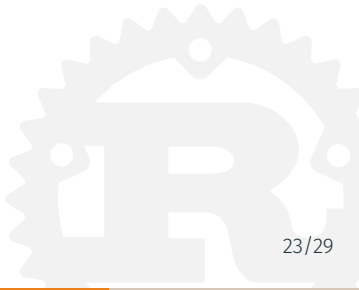


Collect from Channel - 1

With `iter()` we don't need to know the length anymore

```
let mut pictures = vec![];

for pi in rx.iter() {
  if let Some(pi) = pi {
      pictures.push( pi );
  } else {
      // Abort because of error
      return;
  }
}
```



Collect from Channel - 2

With `for_each(...)` we don't need to know the length anymore

```
let mut pictures = vec![];

rx.iter().for_each(|pi| {
  if let Some(pi) = pi {
      pictures.push( pi );
  } else {
      // Abort because of error
      return;
  }
});
```

Collect from Channel - 3

Use `map` and `collect`

```
let pictures = rx.iter().map(|pi| {
        if let Some(pi) = pi {
            Ok( pi )
        } else {
            // Abort because of error
            println("our custom error message");
            Err( () )
        }
    })
    .collect::<Result<Vec<PictureInfo>, ()>>()
    .unwrap();
```

Collect from Channel - 4

Move the error message out

```
let pictures = rx.iter().map(|pi| {
      if let Some(pi) = pi {
          Ok( pi )
      } else {
          // Abort because of error
          Err("our custom error message")
      }
  })
  .collect::<Result<Vec<PictureInfo>, ()>>()
  .expect("unable to iterate trough pictures");
```

Collect from Channel - 5

Parallelize with `rayon`

```
let pictures = rx.par_iter().map(|pi| {
      if let Some(pi) = pi {
          Ok( pi )
      } else {
          // Abort because of error
          Err("our custom error message")
      }
  })
  .collect::<Result<Vec<PictureInfo>, ()>>()
  .expect("unable to iterate trough pictures");
```

Questions

Thank you for your attention!

Stefan Schindler @dns2utf8

Happy hacking! Please ask questions!

slides & Examples: `https://github.com/dns2utf8/thread-pools-and-iterators`

Why another language? - 0

- It is hard to write safe and correct code.
- Even harder to write correct parallel code.

```
char *pi = "3.1415926f32";
while(1) {
    printf("Nth number? ");  err = scanf("%d", &nth);

    if (err == 0 || errno != 0) {
      printf("invalid entry\n");    while (getchar() != '\n');
      continue;
    }

    printf("Input: %d\n", nth);
    printf("Gewünschte Stelle: '%c'\n", pi[nth]);
}
```

Why another language? - 1

```
let pi = "3.1415926f32";
loop {
    print!("Nth number? ");
    io::stdout().flush().unwrap(); // force display on terminal
    let mut input = String::new();
    match io::stdin().read_line(&mut input) {
        Ok(_bytes_read) => {
            let nth: usize = input.trim().parse()
                                    .expect("invalid selection");
            println!("{}-th: '{:?}'", nth, pi.chars().nth(nth));
        }
        Err(error) => println!("error: {}", error),
    }
}
```