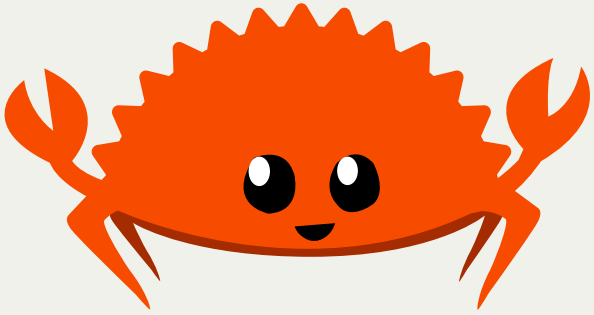
Programação Concorrente

Uma visão geral em Rust



Fearless Concurrency

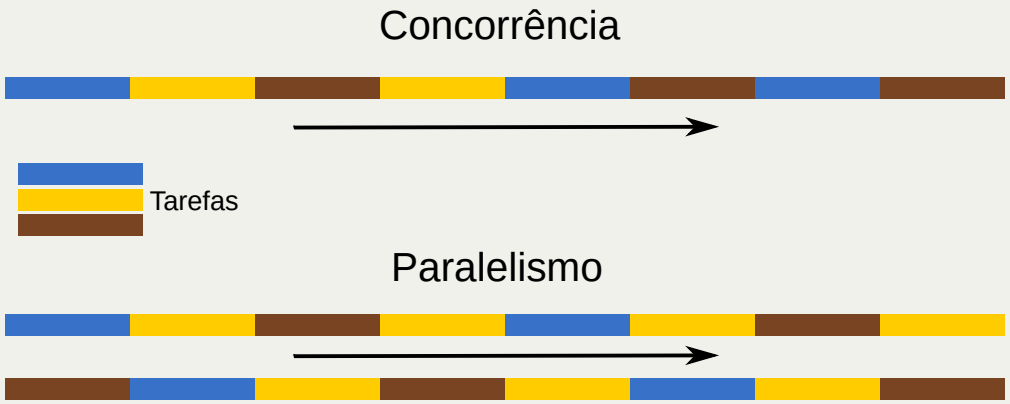
Motivação

Nossos programas são lentos e nossos processadores são subutilizados.

- Lei de Moore: processadores dobram de velocidade em 2 anos.
- Desde a década passada, a velocidade saturou.
- Processadores modernos ganham desempenho com núcleos.
- Mas, nossos programas são sequenciais.

- Processadores gastam a maior parte do seu tempo esperando.
- Muitas vezes quando ainda há trabalho a se fazer.

Paradigmas



Paralelismo é uma forma de concorrência

- Concorrência: corrotinas, programação assíncrona.
- Paralelismo: *threads*, *SIMD*, programação assíncrona.

Concorrência

Fluxo de controle **alternante**

- Corrotinas
- Programação assíncrona

Corrotinas

Assim como *async*, é uma abstração de uma máquina de estados.

↳ Que pode ser **pausada** e **resumida**!

```
def generate_squares():
  i = 1
  while True:
    yield (i * i)
    i += 1

generator = generate_squares()
print(next(generator)) # 1
print(next(generator)) # 4
print(next(generator)) # 9
```

- Escrita de geradores em estilo sequencial.
- Modularização de processos intercalados.

Não temos corrotinas nativas, mas *async* é semelhante.

Rust adota uma abordagem funcional, utilizando iteradores:

```
fn generate_squares() -> impl Iterator<Item = u32> {
  (1..)
    .map(|i| i * i)
}
```

Mas há *crates* que permitem a implementação de corrotinas:

- corona
- coroutine

Programação assíncrona

Eficiente para tarefas limitadas por entrada/saída (IO).

- Abstração de uma máquina de estados
- Código em estilo sequencial
- Permite execução de forma concorrente ou paralela
- *Overhead* inferior ao de *threads*

Futures

Um *future* representa uma computação que está em processo, e eventualmente produzirá um valor.

```
pub trait Future {
  type Output;

  fn poll(self: Pin<&mut Self>, cx: &mut Context) -> Poll<Self::Output>;
}


pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Para escrever código assíncrono, não precisamos utilizar o método `poll`, mas é importante ter uma noção de como funciona por baixo dos panos.

Async/await

A definição de uma função assíncrona se dá da seguinte forma:

```
async fn foo() -> i32;
```

Que equivalente a:

```
fn foo() -> impl Future<Output = i32>;
```

Mas nos permite utilizar `await` na sua implementação.

```
async fn bar() {
  let value = foo().await;
  println!("{}", value);
}
```

```rust
use futures::future;

async fn download(url: &str) -> Result<(), std::io::Error>;

async fn download_files(file1: &str, file2: &str) {
  let download1 = download(file1);
  let download2 = download(file2);

  let (result1, result2) = future::join(download1, download2).await;

  if let Err(error) = result1 {
    eprintln!("erro: {}", error);
  }

  if let Err(error) = result2 {
    eprintln!("erro: {}", error);
  }
}
```

Executor

O executor é o responsável por gerenciar a execução das futures.

Na prática, a maioria das crates implementam executores em uma e várias *threads*.

- futures
- tokio
- async-std
- executors

Paralelismo

Múltiplos controles de fluxo **simultâneos**

- *Threading*
- *SIMD*

Threading

Controle de fluxo paralelo no mesmo espaço de endereçamento.

`std::thread`

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Eficiente para tarefas limitadas pelo processador.

```rust
static slice: &[u8] = &[ 1, 2, 3 ];

let join_handle = std::thread::spawn(
    || {
        // some work here
        println!("{}", slice[0]);
    }
);

println!("{}", slice[1]);

// some work here

let result = join_handle.join();
```

Memória compartilhada

Múltiplas *threads* acessando os mesmos dados.

- Um tipo `T` é *Send* se ele pode ser **enviado** para outra *thread*.
- Um tipo `T` é *Sync* se ele pode ser **compartilhado** entre *threads*.
  ↳ Ou seja: `&T` é *Send*.

A maioria dos tipos é *Send*, exceto `std::rc::Rc` e similares.

A maioria dos tipos também é *Sync*, com exceção daqueles que permitem **mutabilidade interior não sincronizada**.

Rust inibe corrida de dados, mas não todas condições de corrida.

Atômicos

Processadores modernos possuem instruções atômicas.

- Em Rust, temos tipos de dados atômicos.
- Tais tipos permitem **mutabilidade interior sincronizada**.

Na biblioteca padrão:

- std::sync::atomic
  - AtomicUsize
  - AtomicI32
  - AtomicBool
  - …

Operações:

```
impl AtomicUsize {
  pub fn load(&self, order: Ordering) -> usize;
  pub fn store(&self, val: usize, order: Ordering);
  ...
}

impl Send for AtomicUsize;
impl Sync for AtomicUsize;
```

Apesar da aparente simplicidade, o modelo de memória é bastante complicado:

```
pub enum Ordering {
    Relaxed,
    Release,
    Acquire,
    AcqRel,
    SeqCst,
}
```

Arc

Atomically Reference Counted permite **posse compartilhada**: Gerência de **posse** entre *threads* independentes.

```
impl<T> Arc<T> {
  pub fn new(data: T) -> Arc<T>;
}

impl<T> Deref for Arc<T> {
  type Target = T
}

impl<T> Send for Arc<T> where T: Send + Sync;

impl<T> Sync for Arc<T> where T: Send + Sync;
```

```rust
use std::sync::Arc;

let vec = Arc::new(vec![ 1, 2, 3 ]);

let my_vec = vec.clone();
let join_handle = std::thread::spawn(
    move || {
        // some work here
        println!("{}", my_vec[0]);
    }
);

println!("{}", vec[1]);

// some work here

let result = join_handle.join();
```

Mutex e RwLock

Primitivas de **exclusão mútua**, providas pelo sistema operacional.

Em Rust:

> *Lock data, not code.*

Garantem que o acessos concorrentes aos dados não aconteçam.

Mutex

```
impl<T> Mutex<T> {
  pub fn new(t: T) -> Mutex<T>;

  pub fn lock(&self) -> LockResult<MutexGuard<T>>;
}

impl<T> Send for Mutex<T> where T: Send;
impl<T> Sync for Mutex<T> where T: Send;
```

RwLock

```
impl<T> RwLock<T> {
  pub fn new(t: T) -> RwLock<T>;

  pub fn read(&self)  -> LockResult<RwLockReadGuard<T>>;
  pub fn write(&self) -> LockResult<RwLockWriteGuard<T>>;
}

impl<T> Send for RwLock<T> where T: Send;
impl<T> Sync for RwLock<T> where T: Send + Sync;
```

RwLock

```
use std::sync::{Arc, RwLock};

let rwlock = Arc::new(RwLock::new(0));

for _ in 0..5 {
    let data_handle = rwlock.clone();

    std::thread::spawn(
        move || {
            let data = data_handle.read().unwrap();
            print!("{} ", data)
        }
    );

    let data_handle = rwlock.clone();

    std::thread::spawn(
        move || {
            let mut data = data_handle.write().unwrap();
            *data += 1;
        }
    );
}
```

Há uma condição de corrida: (**!**)

Execução	Resultado
1	0 1 1 1 1
2	0 2 2 3 4
3	0 1 1 1 2
4	0 0 1 2 4

Mas não é uma corrida de dados.

Passagem de mensagens

Canais de comunicação, onde se envia e recebe dados.

`std::sync::mpsc`

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>);

pub fn sync_channel<T>(bound: usize) -> (SyncSender<T>, Receiver<T>)

impl<T> Send for Sender<T> where T: Send;

impl<T> Send for SyncSender<T> where T: Send;
impl<T> Sync for SyncSender<T> where T: Send;

impl<T> Send for Receiver<T> where T: Send;
```

O canal abstrai uma fila de mensagens.

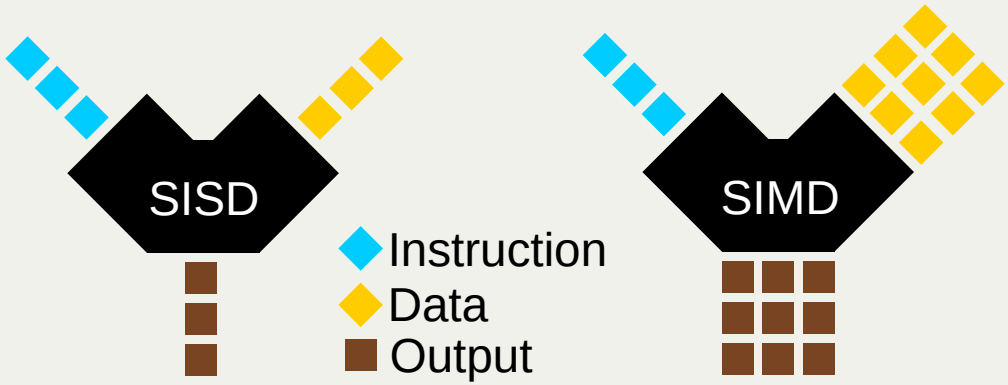
Tipicamente utilizado para comunicação entre threads:

```
let (tx, rx) = std::sync::mpsc::channel();

for i in 0..10 {
  let tx = tx.clone();

  std::thread::spawn(
    move || {
      tx.send(i).unwrap();
    }
  );
}

for _ in 0..10 {
  let j = rx.recv().unwrap();
  println!("{}", j);
}
```

O canal abstrai a gerência de posse dos dados.

SIMD

Processadores modernos possuem instruções para dados agregados:



Single Instruction, Multiple Data

Eficientes para tarefas que operam em blocos de dados compactos e uniformes.

O compilador é capaz de gerar código *SIMD*:

```
fn sum_into(a: &[f32; 64], b: &[f32; 64], c: &mut [f32; 64]) {
    for i in 0..64 {
        c[i] = a[i] + b[i];
    }
}
```

```
...
movups  xmm0, xmmword ptr [rdi]
movups  xmm1, xmmword ptr [rsi]
addps   xmm1, xmm0
movups  xmmword ptr [rdx], xmm1
movups  xmm0, xmmword ptr [rdi + 16]
movups  xmm1, xmmword ptr [rsi + 16]
...
```

Mas nem sempre ele é esperto o suficiente.

Algumas *crates* ajudam a utilizar *SIMD* de forma explícita:

- faster
- simd
- packed-simd

```
let array_of_10s = [-10; 3000]
    .simd_iter(i8s(0))
    .simd_map(|v| v.abs())
    .scalar_collect();
```

Abstrações

Rayon

Paralelismo de dados com *threads*

```
fn sum_of_squares(input: &[i32]) -> i32 {
  input.iter()
    .map(|&i| i * i)
    .sum()
}

use rayon::prelude::*;

fn sum_of_squares_parallel(input: &[i32]) -> i32 {
  input.par_iter() // <-- just change that!
    .map(|&i| i * i)
    .sum()
}
```

Parallel stream

Paralelismo de dados com *async*

Crossbeam

Ferramentas genéricas para programação concorrente:

- `scope`: threads that borrow local variables from the stack.
- `WaitGroup`: synchronizing the begin/end of computations.
- `AtomicCell`: a thread-safe mutable memory location.
- `ShardedLock`: a sharded reader-writer lock.
- `channel`: multi-producer multi-consumer channels.

Fim

https://github.com/gahag/concurrent-programming-talk