Um panorama sobre Rust

Monografia na matéria MAC 5742:
Introdução à Computação Paralela e Distribuída
Professor Alfredo Goldman Vel Lejbman
IME USP

Thilo Koch, 872 8802

28 de Junho de 2015

Conteúdo

1 Introdução

Rust é uma linguagem de programação bem recente. A versão 1.0 que promete a estabilidade sintática e semântica da linguagem no futuro[TM14] foi liberada agora em maio de 2015 [Tea15]. Começado como projeto particular do funcionário da Mozilla Graydon Hoare e depois apoiado pela Mozilla para ajudar no desenvolvimento de uma nova *browser engine* "Servo"[ser] Rust foi anunciada pela primeira vez em 2010.

A ideia principal que levou a criação de Rust é a ideia de uma linguagem de programação para o desenvolvimento de sistemas (de baixo nível) que tenha um modelo de memória seguro e que inclua também elementos modernos como por exemplo *pattern matching*, tipos algébricos, funções de ordem maior (*closures*) e genéricos[HAA12]. Os princípios gerais que guiam o desenvolvimento da linguagem são: não deixar a segurança de memória ser comprometida, abstrações não devem custar nada no tempo de execução, praticidade é a chave[des].

A linguagem foi desenvolvida visando substituir linguagens de programação de baixo nível como por exemplo as linguagens C e C++ nas respectivas áreas de aplicação. Um dos principais problemas de programação com C/C++ é a da segurança de memória. Os erros mais comuns são relacionados a ponteiros que apontam para regiões inválidas de memória (*use after free*, *index out of bounds*), vazamento de memória (*memory leaking* - memória não mais usada mas erradamente não liberada).

Por isso uma das mais importantes caraterísticas de Rust são as promessas de segurança de memória que ela dá. Essa caraterística é especialmente importante para a programação concorrente e paralela porque ajuda a evitar problemas típicos e intrínsecos desses princípios de programação. O conceito fundamental de *ownership* que torna isso possível será descrito na seção 3.

Além disso a linguagem oferece vários recursos normalmente não encontrados em linguagens para a programação de baixo nível que serão brevemente apresentados na seção 2. Na seção 4 serão explorados os recursos da linguagem quanto a problemas referentes à programação paralela, especialmente aqueles discutidos durante o semestre.

2 Introdução à linguagem Rust

2.1 Rust *Toolchain*

As ferramentas incluídas na distribuição de Rust:

- **rustc**: O compilador de Rust é um *frontend* do `llvm` e por isso tem as opções de *backend* do `llvm`. O próprio compilador compila ele mesmo desde 2011 [Hoa].
- **rustdoc**: Uma ferramenta para gerar documentação a partir do código fonte que pode conter descrições em *MarkDown* que ajuda na consistência da documentação em relação ao código fonte.
- **rust-gdb**: Uma versão do `GNU debugger` para Rust.
- **cargo**: O `Cargo` é um gerenciador de projetos para Rust. `Cargo` baixa e compila as dependências e compila o código fonte do projeto. A ferramenta usa `rustdoc` para a geração de documentação, executa testes e *benchmarks*. A ferramenta ainda é nova mas espera-se mais opções [car].

- **IDE**: Apesar de ser um projeto bastante novo já existem vários plugins para as ferramentas comuns que apoiam o desenvolvimento em Rust. Além disso há uma versão do ambiente de trabalho Eclipse: RustDT [rusd].

2.2 A linguagem

A linguagem Rust é uma linguagem para a programação de baixo nível e a primeira vista semelhante a C/C++, porém se distinguem muito. Nessa seção vamos apresentar as diferenças mais importantes entre Rust e C numa visão geral, uma vez que os construtos mais básicos não devem apresentar dificuldades de aprendizagem para desenvolvedores experientes (veja como ponto de partida a documentação em [rusb]).

Além da diferença mais fundamental que é o princípio de *ownership* (explicado na seção 3), Rust oferece vários recursos normalmente não encontrados nesse tipo de linguagem de programação que foram inspirados por linguagens de alto nível (por exemplo linguagens de script, Python, Ruby etc.).

A seguir, apresentaremos algumas caraterísticas da linguagem que o autor julgou inovativas e interessantes no contexto da linguagem (de baixo nível, alto desempenho, etc.)

- A linguagem é baseada em expressões que significa que os construtos que nós normalmente entendemos como comandos em C, são expressões em Rust e consequentemente produzem valores.

  ```
  let y = if x == 5 { 10 } else { 15 }
  ```

  → y == 10 ou 15

  ```
  let z = { x += 1; 2 * x };
  ```

  → z == 2 * x

- Um *switch-case* mais poderoso que em C. A`match` é expressão também. E obrigatoriamente tem que englobar todos os valores possíveis.

  ```
  let y = match x {
              1 | 2   => "one or two",
              3       => "three",
              4...9   => "[4,9]",
              _       => "anything",
          };
  ```

- Os laços tipo `for` não existem na forma de C mas trabalham sempre com iteradores - evitando muitos problemas de *index out of bound*.

```
let a = [0, 1, 2, 3, 4];
for e in a.iter() {
    println!("{}", e); // Prints 1, 2, 3
}
```

- Macros confortáveis como parte da linguagem.

```
macro_rules! foo { ($v:ident) => (let $v = 3); }

fn main() {
    foo!(x);
    // x is 3 here
}
```

- *Closures*

```
let plus_one = |x| {
    let mut result: i32 = x;
    result += 1;

    result          // no semicolon here
                    // closures are expressions
};
```

- *Unit tests* e *benchmarking* apoiados na linguagem e executados com `cargo`.

Essa seleção não pode ser completa. Também destacável para desenvolvedores devem ser: o tipo `enum` sofisticado, o tipo `struct` que pode associar funções e o princípio que os valores de retorno de funções são `struct`. Uma apresentação detalhada de todos essas caraterísticas não cabe nessa monografia, que também não quer simplesmente replicar informações da excelente documentação que existe online [rusb].

Implementação da multiplicação de matrizes

Para testar o compilador de Rust escrevemos versões simples e equivalentes da multiplicação de matrizes sequencial em C e em Rust. O resultado pode ser visto na Figura 1 que mostra que o compilador de Rust produz código rápido com um desempenho comparável com o código produzido pelas ferramentas mais usadas para C.

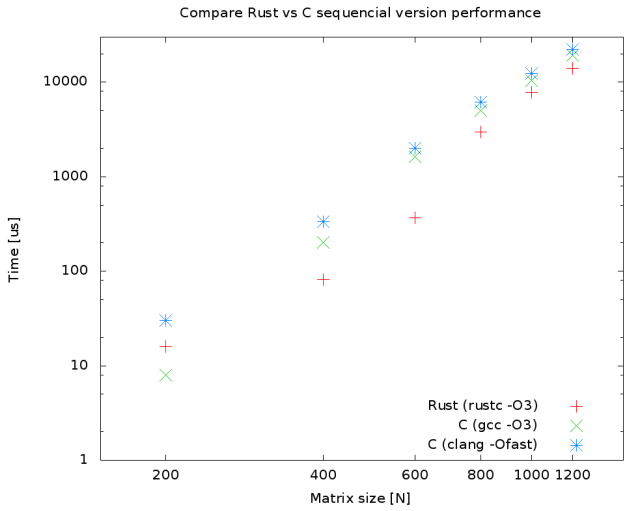

Figura 1: Desempenho da multiplicação de matrizes sequencial em C e Rust

3 O princípio de *Ownership*

O princípio de *ownership* é a especifidade de Rust, e forma a base do modelo de memória da linguagem [1]. A regra básica define que cada objeto tem um único proprietário (*owner*) em cada instante da execução do programa (um conceito semelhante a o de *unique pointers* em C++). O escopo de um objeto é o bloco (delimitado com { }) em que o objeto foi criado. Quanto a execução sai desse bloco, o objeto é destruído automaticamente sem necessidade de um destrutor. A *ownership* de um objeto pode ser transferida para um outro escopo, nesse caso o objeto não é mais válido no escopo original (veja os exemplos na figura 2).

```
1  fn take(v: Vec<i32>) {
2      // qualquer codigo
3  }
4
5  // em algum lugar no main
6  let v = vec![1, 2, 3];
7  take(v);
8  println!("v[0] is: {}", v[0]);
9
10 ->
11 Error: use of moved value 'v'
```

Figura 2: Transferência de *ownership*

Objetos podem ser acessados também por referências: referências mutáveis e não mutáveis. Em cada momento de execução do programa podem existir várias referências de leitura (não mutáveis) para um único objeto ou uma única referência de escrita e leitura (referência mutável - qualificador `mut`). Assim Rust evita *race conditions* em que várias referências acessam o objeto de maneira não sincronizada (por exemplo: invalidação de iteradores). Na semântica de Rust

[1] Esta seção usa várias fontes da internet [rusa] e [Mor], vários exemplos de código fonte foram copiados desses sites

a declaração de referências é chamada *borrowing*. O tempo de existência de objetos (*lifetime*) é intrinsecamente associado ao *borrowing*. Para que erros como *use after free* não aconteçam, uma referência não pode ter um tempo de existência maior que o próprio objeto. *Lifetimes* podem ser declarados explicitamente em certas circunstâncias. O compilador de Rust verifica os *lifetimes* de todos os objetos e termina com erro quando as regras são violadas (como pode ser visto na figura 3).

```
let mut x = 5;
let y = &mut x;     // -+ &mut borrow
                    //  |
*y += 1;            //  |
                    //  |
println!("{}", x); // -+ try to borrow again
                    // but y is still valid!
->
error!
```

```
let mut x = 5;
let y = &mut x;     // -+ &mut borrow
{                   //  |
    *y += 1;        //  |
}                   // -+ borrow ends
println!("{}", x); // -+ try to borrow again

->
correct!
```

Figura 3: *Borrowing*

4 Programação Paralela

Além de ser uma linguagem de programação que garante a segurança de memória para muitos casos, a linguagem foi cogitada também com um foco no apoio à programação paralela. Neste capítulo vamos examinar alguns recursos que a linguagem oferece para esse fins. Para isso serão usadas implementações-exemplo da multiplicação de matrizes porque seu algoritmo é relativamente simples, porém possui boas propriedades para explorar vários aspectos relevantes da programação paralela (concorrência, paralelizabilidade, necessidade de comunicação, mistura de *memory bound* e *cpu-bound*).

4.1 Threads, canais e memória compartilhada

Um programa em execução em Rust é uma coleção de threads, cada um com o próprio estado e a própria pilha. A comunicação entre threads pode ser feita por canais ou por memória compartilhada. Para isso, Rust oferece na biblioteca-padrão um módulo para a criação e controle de threads que define o tipo `Thread`. Até agora existe um único método para a criação de threads (`spawn`); outras funcionalidades para `Thread` são planejadas, mas ainda não liberadas na versão estável de Rust (1.0.0). Desenvolvemos duas variantes da multiplicação de matrizes para exemplificar o uso de threads, canais e memória compartilhada.

Implementação de multiplicação de matrizes com threads e canais

Esta variante da multiplicação de matrizes usa memória compartilhada (somente leitura) para ler os valores das matrizes de entrada[2] (veja figura 4). Com esses valores cada thread calcula

[2]Como Rust ainda não tem arrays de duas dimensões de tamanho variável no tempo de compilação, usamos arrays (vetores) de uma dimensão.

um elemento e envia o resultado pelo canal criado (linha 7) como tupla: índice do elemento e o valor calculado. O thread principal recebe as mensagens e preenche a matriz de resultado.

```
1  [...]
2
3  let a_sh = Arc::new(a_mut);                    // threadsafe reference (read only)
4  let b_sh = Arc::new(b_mut);
5
6  // create channel
7  let (tx, rx): (Sender<(usize,i32)>, Receiver<(usize,i32)>) = mpsc::channel();
8
9  for i in 0..n {
10     for j in 0..n {
11
12         let a = a_sh.clone();                  // a copy of the threadsafe reference
13         let b = b_sh.clone();
14         let thread_tx = tx.clone();
15
16         thread::spawn(move || {                // spwan thread asynchronously
17
18             let mut sum = 0;
19             for k in 0..n {
20                 sum += a[i*n+k] * b[k*n+j];    // read-only access a and b
21             }
22             thread_tx.send((i*n+j, sum)).unwrap();          // send tuple (index, cij) to main thread
23         }
24     }
25 }
26
27 for _ in 0..n*n {
28     let res = rx.recv();                       // wait to receive result
29     match res {
30         Ok(return_val) => {
31             let (index, val) = return_val;
32             c_mut[index] = val;                // save result at index
33         }
34        Err(e) =>  { println!("error communicating: {}", e); }
35     }
36 }
37
38 [...]
```

Figura 4: Multiplicação de matrizes com com threads e canais

Implementação de multiplicação de matrizes com threads e memória compartilhada

Esta variante da multiplicação de matrizes usa memória compartilhada (somente leitura) para ler os valores das matrizes de entrada (veja figura 5). Com esses valores cada thread calcula um elemento, abre o *lock* acima da matriz e escreve o resultado diretamente na matriz de resultado. Apesar de não ser necessário para esse algoritmo, temos que usar um mecanismo de *locking* para o compilador ver a garantia de acesso sincronizado. Na realidade não precisamos disso porque cada thread calcula um elemento distinto. Em Rust existe o construto `slices` que pode quebrar arrays em pedaços que podem ser emprestados (*borrowed*) para os threads, garantindo que cada thread tem apenas acesso a um elemento. Assim a condição de Rust que pode existir somente uma referência de escrita-leitura para cada dado sem usar *locks*. Infelizmente o construtor para `slices` só trabalha com arrays, cujo tamanho tem que ser conhecido no tempo de compilação, que não se aplica no nosso caso. Esperamos que com o amadurecimento da linguagem aparecem mais recursos que podem ser aplicados no nosso caso.

```
[...]
let a_sh = Arc::new(a_mut);                    // threadsafe reference (read only)
let b_sh = Arc::new(b_mut);

let mut tids = Vec::new();                     // vector to save thread IDs

for i in 0..n {
    for j in 0..n {

        let a = a_sh.clone();                  // a copy of the threadsafe reference
        let b = b_sh.clone();
        let c_shared = c_mut.clone();          // a copy of the reference to the result matrix

        let tid = thread::spawn(move || {      // spwan thread asynchronously

            let mut sum:i32 = 0;
            for k in 0..n/4 {
                sum += a[i*n+k] * b[k*n+j];    // read-only access a and b
            }
            let mut c_shared = c_shared.lock().unwrap();    // get lock
            c_shared[i*n+j] = sum;                          // set result element
        });                                                 // lock goes out of scope and is freed
        tids.push(tid);
    }
}

for tid in tids {
    let res = tid.join();                                   // wait for threads to join
    match res {                                             // evaluate thread result
        Ok(v)  => { }
        Err(e) => { println!("error: {:?}", e); }
    }
}
[...]
```

Figura 5: Multiplicação de matrizes com com threads e memória compartilhada trancada

Não rodamos testes de desempenho com esses algoritmos porque ficou claro, em testes preliminares, que o desempenho deles é pior que o desempenho da variante sequencial por magnitudes. Além disso travou a variante com *locking* a partir de um certo tamanho de matrizes.

O princípio de *ownership* e suas implicações para referências complicam a programação com threads in Rust. As ferramentas que apoiam esse tipo de programação ainda estão em desenvolvimento com mais aspectos planejados (por exemplo *futures*[3], `thread::scoped`[4])

4.2 *Streaming SIMD Extensions* - SSE

Um recurso interessante também presente em outras linguagens de programação é a possibilidade de integrar facilmente técnicas de programação paralela oferecidas pelas *streaming SIMD extensions* (SSE). SSE é um conjunto de registros e instruções adicionais à arquitetura x86/x64 dos processadores da Intel e de outras empresas.

SSE possui registros adicionais de 128 bits e instruções que operam sobre esses registros. Na lógica de SSE os registros adicionais são usados para armazenar vários valores ao mesmo tempo em um registro de 128 bits (por exemplo 4 valores de 32 bits) e executar operações que então

[3]Futures as-is have yet to be deeply reevaluated: https://github.com/rust-lang/rust/blob/master/src/libstd/sync/future.rs

[4]std::thread::JoinGuard (and scoped) are unsound because of reference cycles: https://github.com/rust-lang/rust/issues/24292

operam paralelamente sobre o registro (por exemplo 4 adições de 32 bits). SSE oferece operações aritméticas, lógicas, de comparação e outras para números inteiros e números de ponto flutuante para tamanhos diferentes (8, 16, 32, 64 bits).

Implementação da multiplicação de matrizes

```
1  #![feature(core_simd)]
2  use std::simd::i32x4;
3
4  [...]
5
6  for i in 0..n {
7      for j in 0..n {
8          let mut sum: i32 = 0;
9          for k in 0..n/4 {
10             let rk = k*4;
11             let y =
12                 i32x4(a[i*n+rk], a[i*n+rk+1],   a[i*n+rk+2],   a[i*n+rk+3])  *
13                 i32x4(b[rk*n+j], b[(rk+1)*n+j], b[(rk+2)*n+j], b[(rk+3)*n+j]);
14             sum += y.0 + y.1 + y.2 + y.3;
15         }
16         c_mut[i*n+j] = sum;
17     }
18 }
19
20 [...]
```

Figura 6: Multiplicação de matrizes com SSE

Uma vez que a opção SIMD é ligada (linha 1) e a biblioteca é importada (linha 2) a programação fica muito fácil. Com os construtores de SIMD (por exemplo `i32x4()`) os valores são carregados nos registros SSE. Operações sobre esses valores são executadas com operadores comuns (no nosso exemplo: multiplicação *).

Pela experiência adquirida na matéria testamos também uma versão que acessa os valores da matriz `b` de forma alinhada (a matriz `a` já é acessada dessa forma naturalmente). Para isso a matriz `b` é transposta primeiro e no laço, os valores da matriz `b` são acessos em blocos de 4 valores seguidos, como visto na figura 7. Os custos computacionais da transposição (um acesso escrita-leitura por elemento da matriz) são muito menores que os custos de acesso no laço (2 * N acessos de leitura por elemento da matriz) e por isso o desempenho depende principalmente da operação no laço.

```
[...]
let y =
        i32x4(a[i*n+rk], a[i*n+rk+1], a[i*n+rk+2], a[i*n+rk+3]) *
        i32x4(b[j*n+rk], b[j*n+rk+1], b[j*n+rk+2], b[j*n+rk+3]);

        sum += y.0 + y.1 + y.2 + y.3;
[...]
```

Figura 7: Multiplicação de matrizes com SIMD acesso alinhado

Na figura 8 podemos ver que o desempenho da implementação com SSE sem alinhamento de acesso tem o pior desempenho de todos as variantes. Isso pode ser explicado com os custos comparativamente altos para carregar os registros SSE. A variante sequencial tem um desempenho comparável com a primeira variante porque os três laços representam uma estrutura facilmente otimizável pelo compilador. O desempenho da implementação com SSE e acesso alinhado tem um *speedup* maior que 4 para matrizes maiores. Supomos que esse efeito do alinhamento do acesso é mais importante para matrizes maiores porque para eles o efeito de *caching* de memória é menor devido ao fato que eles não cabem mais inteiramente no cache do processador, que tem normalmente apenas um tamanho de poucos megabytes. Para matrizes menores o mecanismo de *caching* melhora o tempo de acesso repetitivo e equaliza o prejuízo do acesso não alinhado.

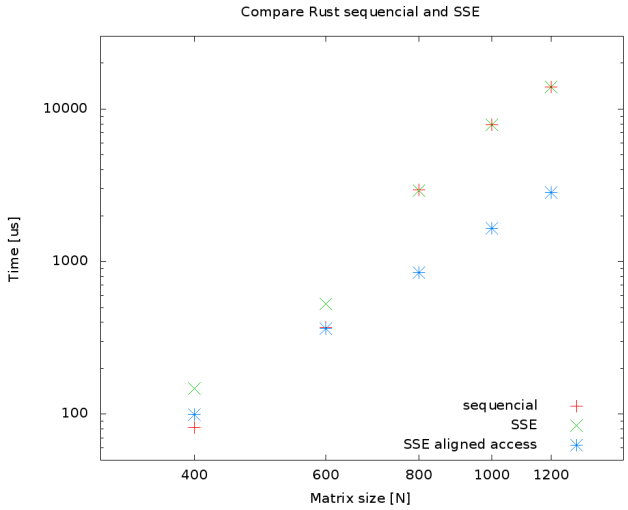

Figura 8: Desempenho multiplicação de matrizes sequencial e com SSE

A biblioteca padrão do Rust dá acesso à programação com SSE de forma simples e transparente e oferece bons potenciais para melhor desempenho de algoritmos SIMD.

4.3 CUDA, OpenMP, MPI

Devido à falta de madureza da linguagem especialmente na área de computação paralela ainda não existem ferramentas para a programação com OpenMP e MPI. Porém existem projetos em estado inicial que pretendem implementar essas ferramentas para Rust (por exemplo: MPI

[rusc] e discussão geral sobre o futuro desenvolvimento: [Mat]). Útil nesse contexto deve ser o *Foreign Function Interface* (FFI) que permite chamadas de código C a partir de código Rust; outras linguagens são também planejadas.

Para a programação com arquiteturas CUDA prova-se útil que o compilador de Rust é baseado no `llvm`, que é capaz de produzir código em PTX. Assim se torna possível gerar o código necessário para os GPUs a partir de código em Rust. Em [HPC+13] essa proposta é desenvolvida com sucesso, porém não está "pronto para produção".

5 Conclusões

A linguagem Rust une vários conceitos modernos com o conceito de linguagens de programação em baixo nível. A especificidade e a vantagem são as garantias de segurança de memória viabilizadas pelo princípio de propriedade (*ownership*).

Por ser uma linguagem muito nova ainda faltam algumas caraterísticas nas bibliotecas básicas sobretudo na área de paralelização. Supomos que essa falta será suprida brevemente porque já existe uma comunidade significativa bem como uma empresa claramente interessada no desenvolvimento de Rust.

Referências

[car] Cargo guide. `http://doc.crates.io/guide.html`. Último acesso: 06/2015.

[des] Rust design faq. `http://doc.rust-lang.org/complement-design-faq.html`. Último acesso: 06/2015.

[HAA12] G. Hoare e A. Abel Avram. Interview on rust. `http://www.infoq.com/news/2012/08/Interview-Rust`, 2012. Último acesso: 06/2015.

[Hoa] G. Hoare. stage1/rustc builds. `https://mail.mozilla.org/pipermail/rust-dev/2011-April/000330.html`. Último acesso: 06/2015.

[HPC+13] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine e N.D. Matsakis. Gpu programming in rust: Implementing high-level abstractions in a systems-level language. Em *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, páginas 315–324, May 2013.

[Mat] N. Matsakis. Data parallelism in rust. `http://smallcultfollowing.com/babysteps/blog/2013/06/11/data-parallelism-in-rust/`. Último acesso: 06/2015.

[Mor] C. Morgan. Rust ownership, the hard way. `http://chrismorgan.info/blog/rust-ownership-the-hard-way.html`. Último acesso: 06/2015.

[rusa] Rust book, capítulo ownership. https://doc.rust-lang.org/book/ownership.html. Último acesso: 06/2015.

[rusb] Rust documentation. https://doc.rust-lang.org/stable/. Último acesso: 06/2015.

[rusc] Rust mpi. https://github.com/gordon1992/rust_mpi. Último acesso: 06/2015.

[rusd] Rustdt. http://rustdt.github.io/. Último acesso: 06/2015.

[ser] The servo browser engine. https://github.com/servo/servo. Último acesso: 27/06/2015.

[Tea15] The Rust Core Team. Announcing rust 1.0. http://blog.rust-lang.org/2015/05/15/Rust-1.0.html, 2015. Último acesso: 06/2015.

[TM14] A. Turon e N. Matsakis. Stability as a deliverable. http://blog.rust-lang.org/2014/10/30/Stability.html, 2014. Último acesso: 06/2015.