Comparação entre linguagens

Implementação da Eliminação de Gauss - Uma comparação entre linguagens de programação.

- Nome: Marcelo Petitot Rezende

Introdução

O objetivo deste trabalho é compreender as características de Rust e Golang comparando programas implementados nestas linguagens com a implementação em C. Para isso, começamos com uma implementação em c do método de eliminação de Gauss para a resolução de sistemas de equações lineares.

Quanto as implementações, a implementação em c foi escrita por mim, e as implementações em Rust e Golang foram feitas basicamente traduzindo o código em c para estas linguagens.

O programa começa gerando uma matriz estendida (linhas x (colunas + 1)) e atribuindo para cada posição valores aleatórios entre 1 e 10 em ponto flutuante. Essa matriz gerada é então impressa na tela e depois o método de eliminação de Gauss é executado de fato. Ao fim do algoritmo, a solução do sistema de equações é impresso na tela e o programa é encerrado.

Comparações

1. Tipos de dados
    - Cada linguagem usa diferentes abordagens para lidar com tipos de dados:

Elemento	C	Rust	Go
Número decimal	`float`	`f32`	`float64`
Vetor de soluções	`float solution[N]`	`[f32; N]`	`var solution [N]float64`
Matriz	`float matrix[N] [N+1]`	`[[f32; N + 1]; N]`	`[N][N+1]float64`
Número aleatório	`rand() % 10 + 1`	`rng.gen_range(1..10)`	`rand.Intn(10) + 1`

- Rust exige que os arrays tenham tamanho fixo conhecido em tempo de compilação ([f32; N]), enquanto C e Go permitem variáveis globais ou definidas em tempo de execução.
- Go usa `float64` como tipo padrão de ponto flutuante.
- Rust e Go têm bibliotecas de números aleatórios mais estruturadas, enquanto C usa `rand()` da `stdlib`.

2. Acesso às variáveis e organização de memória:
    - Cada Linguagem trata a organização e manipulação de memória de maneira diferente

Aspecto	C	Rust	Go
Alocação de matriz	Estática (`float matrix[N][N+1]`)	Estática (`[[f32; N+1]; N]`)	Estática (`[N][N+1]float64`)
Troca de linhas	`for (k = i; k < N+1; k++) { swap }`	`matrix.swap(i, max_row)`	`for k := i; k < N+1; k++ { swap }`

- Rust e Go evitam ponteiros diretos, garantindo segurança de memória (Rust, com verificação em tempo de compilação e usando recursos como Ownership e Borrowing, e Go com o Garbage Collector).
- Rust possui a função nativa `swap()` para troca de linhas, tornando o código mais seguro e legível
- C permite alocação direta da matriz na pilha sem verificação de segurança.

3. Chamadas de função
    - As três linguagens seguem um modelo similar para chamada de funções, mas com algumas diferenças importantes:

Função	C	Rust	Go
Definição	`void function(type param) {}`	`fn function(param: type) {}`	`func function(param type) {}`
Retorno de valor	`void` ou tipo específico	Rust sempre especifica o tipo	`func` define o retorno explicitamente
Parâmetro por referência	Passagem de arrays implícita	Sempre por valor (exceto `&mut`)	Sempre por valor (cópia, exceto slices)

Observações:

- C permite a modificação do array original porque os arrays são passados por referência automaticamente.
- Rust passa arrays por valor por padrão, mas pode usar `&mut` para modificar diretamente.
- Go também passa arrays por valor, exigindo `slices` (`[]float64`) para permitir modificações diretas.

4. Controle de fluxo
    - Os comandos de controle de fluxo são semelhantes entre as linguagens:

Estrutura de controle	**C**	**Rust**	**Go**
Laço `for`	`for (i = 0; i < N; i++) {}`	`for i in 0..N {}`	`for i := 0; i < N; i++ {}`
Laço `while`	`while (condicao) {}`	`while condicao {}`	Go não tem laço while
Laço reverso	`for (i = N-1; i >= 0; i--) {}`	`for i in (0..N).rev() {}`	`for i := N-1; i >= 0; i-- {}`
Condicional `if`	`if (condicao) {}`	`if condicao {}`	`if condicao {}`

Observações:

- Rust tem `while`, mas também permite o uso de `loop {}` com `if break` para comportamento equivalente.
- Go Não tem while, mas usa for sem inicialização e incremento como substituto.
- Os loops for funcionam de forma semelhante nas três linguagens.

5. Manejo de segurança e erros:
   A linguagem Rust se destaca pela segurança da memória:

Aspecto	**C**	**Rust**	**Go**
Gerenciamento de memória	Manual (`malloc`, `free`) ou pilha	Seguro (verificação de empréstimo e Rc/Arc para referências)	Coletor de lixo automático
Acesso a memória	Permite ponteiros inseguros	Usa borrow checker para evitar acessos inválidos	Sem acesso direto a ponteiros, usa coletor de lixo
Manipulação de overflow	Nenhuma proteção por padrão	Protegido contra overflow	Verificação de overflow em tempo de execução

- C pode levar a erros de segmentação (segmentation faults).
- Rust impede acesso inválido à memória por meio do sistema de empréstimos e do borrow checker.
- Go usa um coletor de lixo para liberar memória automaticamente e verifica overflows em tempo de execução.

6. Métricas quantitativas:
   Embora os programas tenham algumas diferenças em termos de estrutura de código, o uso das três linguagens resultou em programas escritos de forma similar.

Métrica	C	Rust	Go
Número de linhas	~70	~60	~70
Número de comandos	~50	~45	~48
Número de funções	4	3	4

- O programa escrito em Rust tem menos linhas devido à funcionalidade do `swap()` e a sua sintaxe mais expressiva e segura que reduz a necessidade de verificações manuais.
- Go fica no meio, sendo mais compacto, principalmente por causa do coletor de lixo, que elimina algumas etapas que o programador precisaria cuidar manualmente.
- C tem mais código boilerplate, especialmente no gerenciamento manual de memória (alocação, liberação, etc.) e em trocas de valores (o swap realizado no código)

Comparação de desempenho

- Os testes foram realizados em uma máquina com as seguintes especificações:

Parte	Modelo
Processador	AMD Ryzen 5 5500U
Memória RAM	8 GB DDR4-3200MHz
Sistema operacional	Arch Linux

Foram realizados dois tipos de testes para medir a performance entre os programas:

- Para cada programa, foi medido o tempo de execução da função `GaussianElimination()` e impresso no final da execução do programa.
  - Os comandos de compilação foram, para cada linguagem:
  - C: `gcc -o c_x gauss_elim.c -lm`

    - Rust: `cargo run`
    - Go: `go build -o go_x gauss_elim.c`
- Um programa para a realização de benchmarks de comandos, hyperfine foi utilizado. O software funciona rodando o comando especificado pelo menos 10 vezes, e guardando informações relacionados aos tempos que esses programas levaram para rodar, como a média de tempo, o tempo máximo e mínimo. Este programa permite gerar tabelas estilo markdown que permitem comparar os tempos em segundos.
    - Para estes testes, os programas foram compilados com argumentos de otimização:
    - C: `gcc -O3 -o c_x gauss_elim.c -lm`
    - Rust: `cargo build --release`
    - Go: `go build -ldflags="-s -w" -o go_x gauss_elim.go`

Observações:

- Para as três linguagens, foram feitos testes nos programas utilizando os valores 50, 500, 1000 e 2000 como tamanhos de matrizes. A unica exceção é Rust, que não foi capaz de compilar o código para a matriz de tamanho 2000, por problemas de stack overflow.

Resultados dos testes simples:

Tamanho da matriz	**50 [sec]**	**500 [sec]**	**1000 [sec]**	**2000 [sec]**
C	0.000199	0.096601	0.777718	6.385371
Rust	0.000448	0.395489	3.167209	stack overflow
Go	0.000037	0.018371	0.377598	3.604881

Resultados do hyperfine:

Comando para rodar o programa compilado	**Média [sec]**	**Mínimo [sec]**	**Máximo [sec**
`./c/c_50`	0.001 ± 0.000	0.000	0.00
`./c/c_500`	0.010 ± 0.001	0.009	0.01
`./c/c_1000`	0.070 ± 0.004	0.063	0.08

Comando para rodar o programa compilado	Média [sec]	Mínimo [sec]	Máximo [sec
`./c/c_2000`	1.733 ± 0.048	1.677	1.83
`./go/go_50`	0.002 ± 0.000	0.001	0.00
`./go/go_500`	0.027 ± 0.001	0.025	0.03
`./go/go_1000`	0.422 ± 0.012	0.412	0.44
`./go/go_2000`	3.679 ± 0.056	3.615	3.78
`./rust/gauss_elim/target/rust_50/release/gauss_elim`	0.001 ± 0.000	0.000	0.00
`./rust/gauss_elim/target/rust_500/release/gauss_elim`	0.030 ± 0.001	0.029	0.03
`./rust/gauss_elim/target/rust_1000/release/gauss_elim`	0.233 ± 0.004	0.229	0.24

Um ponto interessante a ser levantado é a diferença de performance que os compiladores podem causar na execução dos programas. Os compiladores das três linguagens permitem diferentes argumentos que podem mudar a forma que o programa e compilado e podem alterar, as vezes de forma significante, o tempo que um programa leva para executar. Um exemplo disso é em C, onde podemos usar a flag de compilação `-Ox` (x podendo ser 0, 1, 2 ou 3) que indica o nível de otimização utilizado. Rust usa uma nomenclatura similar para seus níveis de otimização, e o Go tem argumentos próprios para ativar ou não otimizações na compilação.

O objetivo com este trabalho foi utilizar o comando mais simples de compilação possível para cada linguagem, mas isso pode resultar em uma situação onde um programa foi compilado com mais otimizações que o outro, e, por isso, teve uma performance melhor. O exemplo mais claro disso é com a linguagem Rust e o comando `cargo run`. Este comando compila o programa no modo de debug, o que significa que a compilação, além de não usar

nenhuma otimização, mantém os símbolos para debugging, o que resulta no programa rodando significantemente mais lento que os das outras linguagens.

Análise geral

As três linguagens estudadas, embora muito similares em diversos pontos, ainda buscam resolver problemas específicos, algo que faz sentido com o conceito de linguagens de programação, afinal, nenhuma linguagem é a ideal para resolver todos os problemas. Essas linguagens de programação diferem principalmente na forma com que lidam com o gerenciamento de memória:

- C oferece total liberdade para o programador, o que abre um leque de possibilidades de programação maior que em outras linguagens, mas como consequência, também abre um leque de possíveis problemas caso o programador não tenha cautela.
- Rust busca o oposto. Total restrição ao acesso de dados, por meio de conceitos como Borrowing e Ownership, mas que em compensação resultam em programas com mínimas, para não dizer nenhuma, falha de segurança de memória.
- Go é uma linguagem que busca a simplicidade, passando a responsabilidade de gerenciamento de memória para o Garbage collector, o que pode resultar em um sacrifício de desempenho. Em compensação, é a linguagem mais fácil de aprender e dominar dentre as três.

Em termos de performance, podemos ver grande diferença de performance entre os programas para tamanhos grandes de matriz, como no caso da matriz de tamanho 2000. Quando utilizando o comando mais simples para compilar os programas em cada linguagem, podemos ver que a linguagem mais rápida foi a Go, devido ao fato de que o comando de compilação `go build` aplica algumas otimizações por padrão. A linguagem rust foi significantemente pior em performance devido ao comando `cargo run`, por padrão, compilar usando o modo de debugging, que carece de qualquer tipo de otimização

Quando aplicamos otimizações nas três linguagens, os resultados mais próximos, com a linguagem C produzindo os tempos mais rápidos e com a linguagem Rust ficando em segundo lugar por pouco.