Fibonacci usando Multithreading

Paulo Júnior
Fernando Rodrigues

Problema proposto:

O projeto a ser desenvolvido, consiste em gerar a sequência de números da serie de Fibonacci até o número 1.000.000. Utilize threads para cada faixa de 1000 valores, crie uma thread e dispare o processo para cada uma delas.

Como funciona o Fibonacci:

A sequência de Fibonacci é composta por uma sucessão de números descrita pelo famoso matemático italiano Leonardo de Pisa (1170-1250), mais conhecido como Fibonacci, no final do século 12. O matemático percebeu uma regularidade matemática a partir de um problema criado por ele mesmo. Assim, a fórmula a seguir define a sequência: Fn = Fn - 1 + Fn – 2.

Rust

Rust é uma linguagem de programação moderna e segura, desenvolvida pela Mozilla. Ela é conhecida por sua eficiência e capacidade de lidar com concorrência de forma segura.

Utilizamos Rust neste projeto devido à sua capacidade de gerenciar multithreads de forma segura e eficiente, o que é crucial para o cálculo de Fibonacci em larga escala.

Multithreading

Multithreading é uma técnica de programação que permite que vários threads executem simultaneamente dentro de um mesmo processo. Isso pode aumentar significativamente a eficiência e reduzir o tempo de execução de algoritmos complexos.

Solução

A implementação em Rust que fizemos tentou permitir a segurança e estabilidade do programa, evitando problemas comuns associados à concorrência, como condições de corrida e deadlocks.

Para gerar a sequência de Fibonacci até 1.000.000, foram utilizados blocos de processamento chamados de "chunks", cada um contendo 10 threads que eram responsáveis por calcular os termos da sequência.

Foi criada uma função para gerar a sequência, dividir em blocos, criar uma thread para cada bloco e garantir a consistência dos dados compartilhados pelas threads. No final, foram combinados todos os blocos gerados em uma única sequência.

Teste 1

```rust
use std::thread;

fn main() {
    let mut fib_nums: Vec<u64> = vec![0, 1]; // Inicia a sequência com os números 0 e 1
    let mut threads = Vec::new();

    // Define uma função para calcular a sequência de Fibonacci em uma faixa de valores
    fn calculate_fib_range(start: u64, end: u64, fib_nums: &mut Vec<u64>) {
        for i in start..end {
            let next_fib = fib_nums[i as usize - 1] + fib_nums[i as usize - 2];
            fib_nums.push(next_fib);
        }
    }

    // Dispara uma thread para cada faixa de 1000 valores
    for i in (2..1000).step_by(1000) {
        let start = i;
        let end = i + 1000;
        let fib_nums_ref = &mut fib_nums;
        let handle = thread::spawn(move || calculate_fib_range(start, end, fib_nums_ref));
        threads.push(handle);
    }

    // Aguarda todas as threads terminarem antes de continuar
    for handle in threads {
        handle.join().unwrap();
    }

    // Imprime os números da sequência de Fibonacci até o número 1.000.000
    for i in 0..fib_nums.len() {
        if fib_nums[i] > 1_000_000 {
            break;
        }
        println!("{}", fib_nums[i]);
    }
```

Teste 2

```rust
use std::thread;

fn main() {
    let num_threads = 4;                          // Número de threads que serão criadas
    let mut handles = vec![];                     // Vetor para armazenar as referências das threads
    let fib_nums = vec![0; 50];                   // Vetor para armazenar os números de Fibonacci, inicializado com 50 zeros
    let chunk_size = fib_nums.len() / num_threads; // Tamanho de cada "chunk" de números de Fibonacci

    for i in 0..num_threads {
        let start = i * chunk_size;               // Índice de início do chunk atual
        let end = if i == num_threads - 1 {
            fib_nums.len()                        // Se for a última thread, o índice final é o último elemento do vetor
        } else {
            (i + 1) * chunk_size                  // Caso contrário, o índice final é o início do próximo chunk
        };

        let mut fib_nums_thread = vec![0; 50];    // Vetor para armazenar os números de Fibonacci calculados pela thread atual
        let handle = thread::spawn(move || {
            calculate_fib_range(start, end, &mut fib_nums_thread);  // Calcula os números de Fibonacci do chunk atual e armazena no vetor da thread
            fib_nums_thread                       // Retorna o vetor com os números de Fibonacci calculados pela thread
        });

        handles.push(handle);                     // Armazena a referência da thread no vetor handles
    }

    let mut result = vec![];                      // Vetor para armazenar os números de Fibonacci calculados por todas as threads
    for handle in handles {
        let fib_nums_thread = handle.join().unwrap(); // Espera a thread terminar e obtém o vetor de números de Fibonacci calculados
        result.extend_from_slice(&fib_nums_thread); // Adiciona os números de Fibonacci calculados ao vetor result
    }

    for i in 0..result.len() {
        if result[i] > 1_000_000 {
            println!("fib({}) = {}", i, result[i]);  // Imprime o índice e o número de Fibonacci se for maior do que 1_000_000
        }
    }
    println!("{:?}", result);                     // Imprime o vetor completo com os números de Fibonacci
}

fn calculate_fib_range(start: usize, end: usize, fib_nums: &mut [u64]) {
    fib_nums[start] = 0;                          // Define o primeiro número de Fibonacci como zero
    fib_nums[start + 1] = 1;                      // Define o segundo número de Fibonacci como um
    for i in (start + 2)..end {
        fib_nums[i] = fib_nums[i - 1] + fib_nums[i - 2]; // Calcula os demais números de Fibonacci do chunk atual
    }
}
```

Teste 3

```rust
use std::thread;                          // importa o módulo "thread" da biblioteca padrão do Rust

const MAX_IDX: u64 =                      // define uma constante para o maior índice a ser calculado
const CHUNK_SIZE: u64 = 10;               // define uma constante para o tamanho do "chunk" (pedaço) de cálculos a serem realizados em paral

fn main() {
    let mut handles = vec![];             // cria um vetor vazio que irá armazenar os identificadores das threads criadas

// para cada "chunk" de índices a serem calculados...
    for chunk in (0..=MAX_IDX).step_by(CHUNK_SIZE as usize) {
        let end_idx = (chunk + CHUNK_SIZE).min(MAX_IDX) - 1;   // define o índice final do "chunk"
        let handle = thread::spawn(move || {                   // cria uma nova thread para calcular os números de Fibonacci no "chunk"
            for i in chunk..=end_idx {                         // para cada índice no "chunk"...
                let result = fibonacci(i);                     // calcula o número de Fibonacci correspondente
                println!("fibonacci({}) = {}", i, result);     // imprime o resultado
            }
        });
        handles.push(handle);             // armazena o identificador da thread no vetor
    }

    for handle in handles {               // para cada identificador de thread...
        handle.join().unwrap();           // aguarda a thread finalizar
    }
}

// função recursiva que calcula o número de Fibonacci de um dado índice
fn fibonacci(n: u64) -> u64 {
    match n { // verifica o valor de n
        0 => 0, // se n = 0, retorna 0
        1 => 1, // se n = 1, retorna 1
        _ => fibonacci(n - 1) + fibonacci(n - 2),       // se n > 1, retorna a soma dos números de Fibonacci correspondentes aos dois índic
    }
}
```

Como dizia minha ex: Terminamos!

Obrigado pela atenção