Fatorial usando multithreading

Componentes:
Álefe Alves
Leonardo Castro

Objetivo

Realizar o cálculo do fatorial de todos os números até 1.000.000, utilizando threads para cada faixa de 1000 valores, aplicando conceitos de multithreading na linguagem Rust, mostrando as dificuldades e soluções apresentadas.

Fórmula geral do fatorial:

$$n! = n \cdot (n - 1). (n - 2). (n - 3) \ldots 2,1.$$

Solução parcial 1

Código Simples

Divisão de calculo das threads:
Ex.: Thread 1: 1-1000, Thread 2: 1001-2000

A implementação foi mais fácil, porém ouve uma demora na execução do código devido um desbalanceamento no trabalho pois algumas threads terminam antes e outras sobrecarregadas pelo calculo de números gigantescos.

Tempo de execução: 753.0330679s

N!

```rust
use num::BigUint;
use num::One;
use std::time::Instant;
use std::thread;

fn fatorial_parcial(inicio: u64, fim: u64) -> BigUint {
    let mut result = BigUint::one();
    for i in inicio..=fim {
        result *= i;
    }
    result
}

fn main() {
    let fat = 1000000;
    let num_threads = 1000;
    let mut handles = Vec::new();
    let range_fat = fat/num_threads;
    let start = Instant::now();

    for i in 0..num_threads{
        let inicio = i * range_fat + 1;
        let fim = if i == num_threads - 1{
            fat
        }else{
            (i + 1) * range_fat
        };

        println!("Thread {} calcula de {} até {}", i + 1, inicio, fim);

        let handle = thread::spawn(move || {
            fatorial_parcial(inicio, fim)
        });

        handles.push(handle);
    }

    let mut resultado_final = BigUint::one();
    for handle in handles {
        resultado_final *= handle.join().unwrap();
    }

    println!("Fatorial de {} é {}", fat, resultado_final);
    println!("Tempo de execução: {:?}", start.elapsed());

}

```

Solução parcial 2

Código Intercalado

Divisão de calculo das threads:
Ex.: Thread 1: 1, 1001, 2001...
Thread 2: 2, 1002, 2002...

Neste ouve um balanceamento dos números usando o números intercaladoss, porém mesmo assim ouve uma demora dos calculos.

Tempo de execução: 743.5509215s

```rust
extern crate num;
use num::BigUint;
use num::One;
use std::time::Instant;
use std::thread;

fn fatorial_parcial(thread_id: u64, total_threads: u64, max_num: u64) -> BigUint {
    let mut result = BigUint::one();
    let mut num = thread_id;
    while num <= max_num {
        result *= num;
        num += total_threads;
    }
    result
}

fn main() {
    let fat = 1000000;
    let num_threads = 1000;
    let mut handles = Vec::new();
    let start = Instant::now();

    for thread_id in 1..=num_threads {
        let handle = thread::spawn(move || {
            fatorial_parcial(thread_id, num_threads, fat)
        });
        handles.push(handle);
    }

    let mut resultado_final = BigUint::one();
    for handle in handles {
        resultado_final *= handle.join().unwrap();
    }

    println!("{}! = {}",fat, resultado_final);
    println!("Tempo de execução: {:?}", start.elapsed());
}

```

Problemas Encontrados

- **Gerenciamento de threads (1000 threads concorrentes)**
  - **Overhead de criação e thread switching causa perda de desempenho e alto uso de memória.**
- **Overflow numérico**
  - **Alocação dinâmica de BigUint é lenta e fragmenta a memória.**
- **Custo de concatenação dos resultados parciais**
  - **Multiplicação sequencial dos resultados parciais vira gargalo.**
- **Limitações de hardware**
  - **Uso excessivo de RAM e contenção no barramento de memória.**

Solução Final

Código com Mutex

Divisão de calculo das threads:
Ex.: Thread 1: 1-1000, Thread 2: 1001-2000

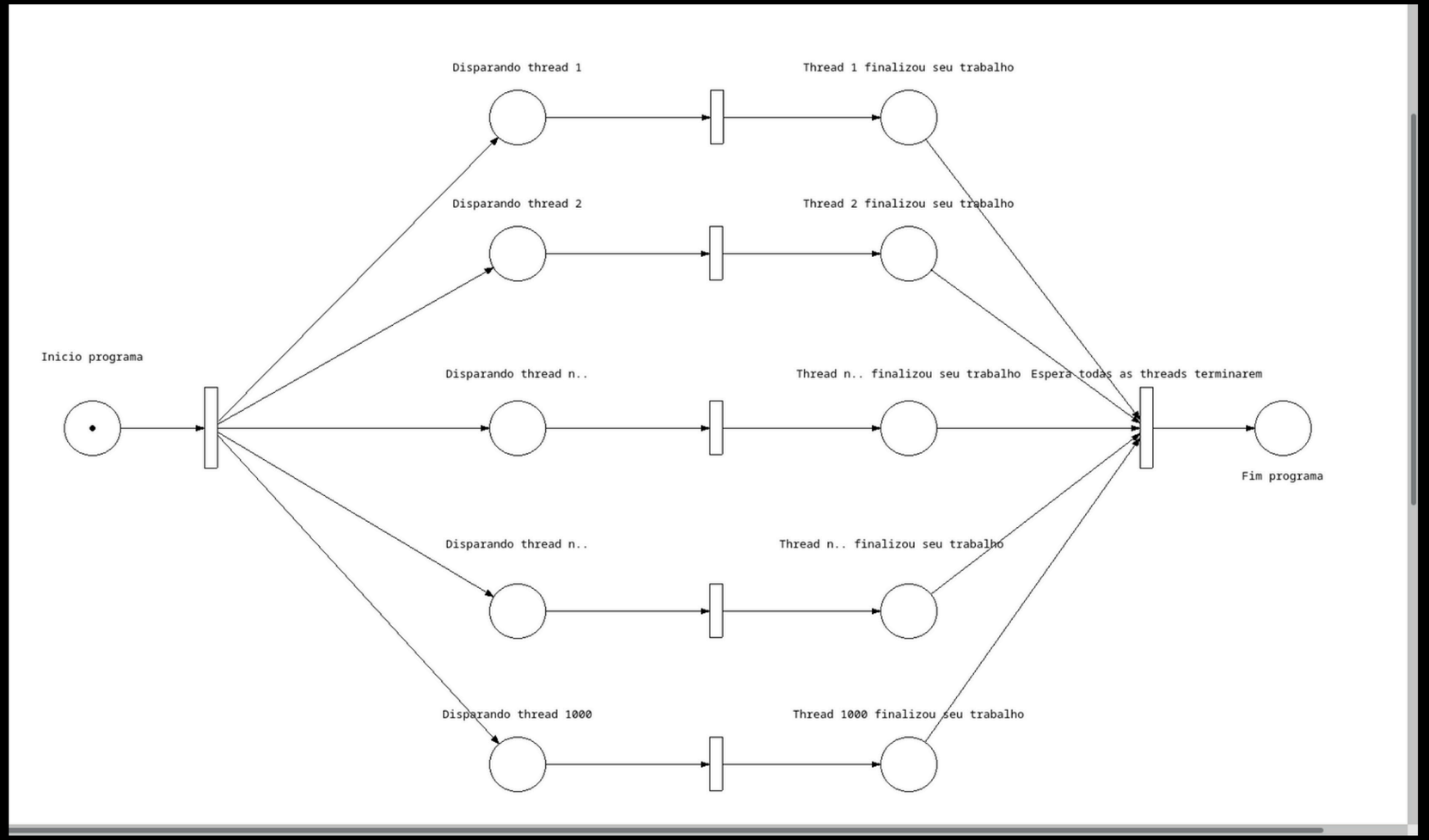
Neste solução reaproveitamos cálculos realizados por outras threads, possibilitando assim um menor custa para o cálculo de números maiores.

```rust
fn dividir_trabalho(num: u32) {
    let num_threads = 1000;//Números de threads
    let mut handles = Vec::new();//Cria um vetor para salvar ponteiros para as threads
    let mapa: Arc<Mutex<HashMap<u32, BigUint>>> = Arc::new(Mutex::new(HashMap::new()));//Estrutura hash, p
    let range_fat = num / num_threads; //Define o range em que cada thread irá trabalhar
    for i in 0..num_threads{//For para inicializar cada thread
        let inicio = i * range_fat + 1;//Cálcula o início do trabalho para a thread
        let fim = if i == num_threads - 1{//Cálculo o fim do trabalho para a thread
            num
        }else{
            (i + 1) * range_fat
        };
        println!("Thread {} calcula de {} até {}", i + 1, inicio, fim);
        let mapa_clone = Arc::clone(&mapa);//Cria um clone da tabela hash para usarmos dentro da thread
        let handle = thread::spawn(move || {//Incializa thread
            for j in inicio..=fim{//For para calcular todos os fatoriais do intervalo
                let mut ind: u32 = 1;
                let mut val = BigUint::one();
                //bloco para dar lock e realizar a pesquisa na hash e depois desbloquear.
                {
                    //Habilita o lock na tabela hash evitando acesso simultaneo
                    let hash = mapa_clone.lock().unwrap();

                    for k in (1..j).rev() {//For reverso que procura na hash se os valores de menores que
                        if let Some(valor) = hash.get(&k) {
                            val = valor.clone();
                            ind = k;
                            break;
                        }
                    }
                }
                //Habilita o lock na tabela hash evitando acesso simultaneo
                let mut hash = mapa_clone.lock().unwrap();
                let resultado : BigUint = fatorial(j, ind, val);

                hash.insert(j, resultado.clone());//Salva o calculo na hash
                println!("O fatorial de {} calculado pela thread {} é {}.", j, i+1, resultado);
            }
        });
        handles.push(handle);
    }
    //let mut resultado_final = BigUint::one();
    for handle in handles {
        handle.join().unwrap();
    }
    //println!("Fatorial de {} é {}", num, resultado_final);
}
```

Rede de Petri



N! Referências

- PROGRAMMING IDIOMS. Recursive factorial (simple). Disponível em: https://programming-idioms.org/idiom/31/recursive-factorial-simple/450/rust. Acesso em: 31 maio 2025.
- USERS RUST-LANG. Parallel product for factorial – surprised by the results. 14 maio 2019. Disponível em: https://users.rust-lang.org/t/parallel-product-for-factorial-surprised-by-the-results/30776/14. Acesso em: 31 maio 2025.
- DANDYVICA. Using threads on Rust – Part 3. Dev.to, 17 nov. 2020. Disponível em: https://dev.to/dandyvica/using-threads-on-rust-part-3-2bpf. Acesso em: 31 maio 2025.