Algoritmo de Fibonacci Com Multithreading

Uma jornada técnica pela otimização da sequência de Fibonacci utilizando o poder do paralelismo.

Feito por Luciano dos Santos Nascimento e Wesley Silva Araújo.

O Que é a Sequência de Fibonacci?

Definição Matemática

Cada número é a soma dos dois anteriores.

Começa com 0 e 1.

Aplicações Prática

Presente na natureza e arte.

Base para algoritmos e estruturas de dados.

Introdução ao Multithreading

Definição

Execução simultânea de partes de um programa.

Threads compartilham recursos.

Benefício

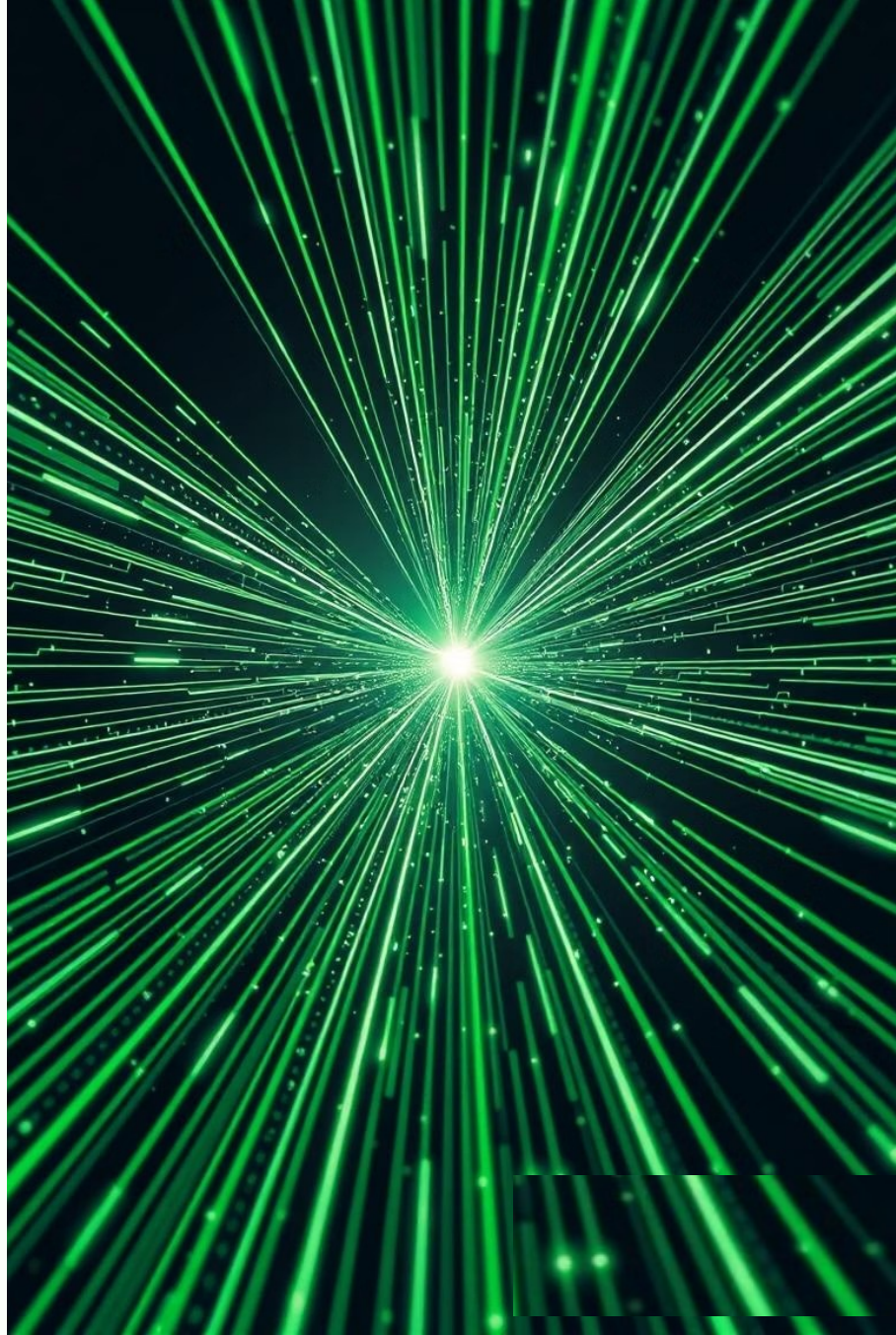
Melhora a responsividade da aplicação.

Aproveita múltiplos núcleos do processador.

Desafio

Complexidade na sincronização.

Potenciais condições de corrida.



Estratégia de Multithreadig para Fibonacci

Divião de Tarefa

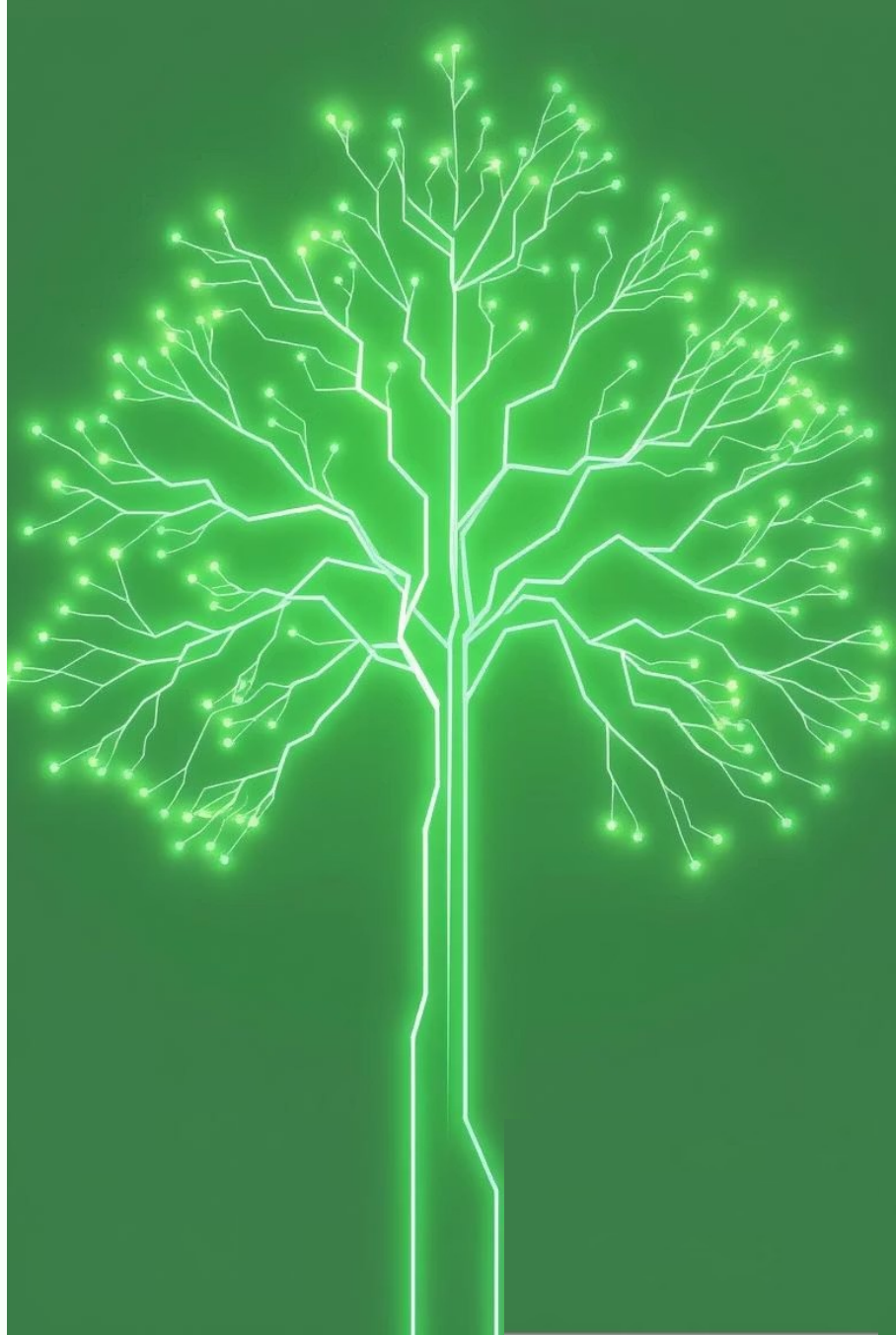
Cada thread calcula uma parte da sequência.

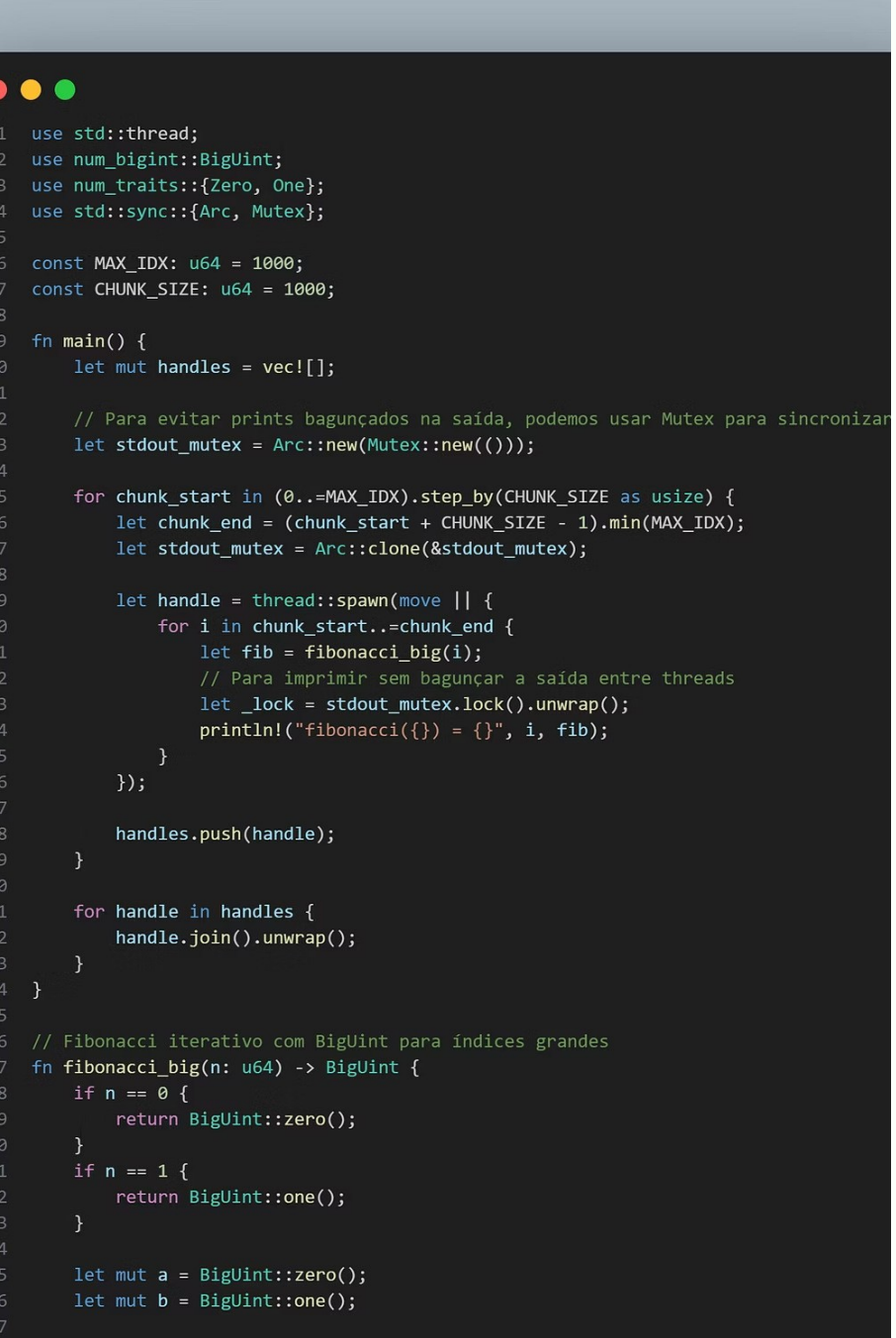
Memorização Ditribuída

Compartilha resultados já calculados entre threads.

Sicronização

Garante consistência e evita conflitos de dados.





```rust
use std::thread;
use num_bigint::BigUint;
use num_traits::{Zero, One};
use std::sync::{Arc, Mutex};

const MAX_IDX: u64 = 1000;
const CHUNK_SIZE: u64 = 1000;

fn main() {
    let mut handles = vec![];

    // Para evitar prints bagunçados na saída, podemos usar Mutex para sincronizar
    let stdout_mutex = Arc::new(Mutex::new(()));

    for chunk_start in (0..=MAX_IDX).step_by(CHUNK_SIZE as usize) {
        let chunk_end = (chunk_start + CHUNK_SIZE - 1).min(MAX_IDX);
        let stdout_mutex = Arc::clone(&stdout_mutex);

        let handle = thread::spawn(move || {
            for i in chunk_start..=chunk_end {
                let fib = fibonacci_big(i);
                // Para imprimir sem bagunçar a saída entre threads
                let _lock = stdout_mutex.lock().unwrap();
                println!("fibonacci({}) = {}", i, fib);
            }
        });

        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
}

// Fibonacci iterativo com BigUint para índices grandes
fn fibonacci_big(n: u64) -> BigUint {
    if n == 0 {
        return BigUint::zero();
    }
    if n == 1 {
        return BigUint::one();
    }

    let mut a = BigUint::zero();
    let mut b = BigUint::one();
```

Implementação

```rust
// Fibonacci iterativo com BigUint para índices grandes
fn fibonacci_big(n: u64) -> BigUint {
    if n == 0 {
        return BigUint::zero();
    }
    if n == 1 {
        return BigUint::one();
    }

    let mut a = BigUint::zero();
    let mut b = BigUint::one();

    for _ in 2..=n {
        let c = &a + &b;
        a = b;
        b = c;
    }

}
```

Benefícios e Desafio da Abordagem Multithreading

Ganho de Performace

Redução significativa do tempo de execução.

Maior escalabilidade para grandes "n".

Aumento da Complexidade

Gerenciamento de threads exige cuidado.

Debugging e otimização são mais difíceis.