Relatório PL5 - Exercício 5
Gleizielly Alves (1210133) e Márcia Rocha (1210138)

1. Introdução

Este trabalho tem como objetivo descrever a resolução do exercício 5 da PL5 e analisar a performance dos códigos em suas versões sequenciais e paralelas utilizando as bibliotecas do Rust: Threadpool e Rayon. Além disso, os resultados de performance do Rust em cada uma das implementações foi comparado com os resultados encontrados utilizando o OpenMP.

2. Implementação, resultados e discussões

O exercício 5 propõe a paralelização do cálculo da área de um conjunto de Mandelbrot. Tanto para a versão sequencial disponibilizada, quanto para o desenvolvimento das versões paralelas utilizando as bibliotecas ThreadPool e Rayon, foi definido um número de pontos (NPOINTS) igual a 1000. Para análise de performance foram colhidas 20 amostras de cada implementação. Em seguida, os dados foram plotados em gráficos e foram feitos cálculos estatísticos de média, desvio padrão, máximo e mínimo.

2.1. Versão paralela usando Rayon

Para a versão paralela utilizando o Rayon, foi necessário informar ao arquivo "Cargo.toml" a dependência da biblioteca. Este processo pode ser simplificado utilizando o comando abaixo:

```
$ cargo add rayon
```

Em seguida, foi necessário importar os seguintes módulos dentro do programa principal para usar o *Parallel Iterator* do Rayon:

```
use rayon;
use rayon::prelude::*;
```

Por fim, alterou-se o programa sequencial trocando a função *into_iter()* pela função *into_par_iter()* do Rayon.

```
fn get_num_points_outside_par_rayon() -> i32 {
    (0..NPOINTS).into_par_iter().map(|i|
       (0..NPOINTS).into_par_iter().map(|j| {
          test_point(i,j)
       }).sum::<i32>()
    ).sum::<i32>()
 }
```

Nesta implementação não foi definido explicitamente o número de *threads*. Entretanto, por padrão, o Rayon define o número de *threads* igual ao número de CPUs disponíveis - neste caso, o trabalho foi desenvolvido em um ambiente que possuía 8 CPUs.

2.2. Versão paralela usando ThreadPool

Para a versão paralela utilizando o threadpool, foi necessário adicionar ao arquivo "Cargo.toml" a dependências da biblioteca utilizando o comando abaixo:

```
$ cargo add threadpool
```

Em seguida, foi necessário incluir os seguintes módulos no programa principal:

```
use std::sync::mpsc::channel;
use std::thread;
use threadpool::ThreadPool;
```

Posteriormente, utilizou-se o código abaixo para determinar o número de CPUs disponíveis no sistema e em seguida, criar um *thread pool* com a quantidade de *threads* igual ao número de CPUs.

```
let n_workers = thread::available_parallelism().unwrap().get();    //get number of available CPUs
let pool = ThreadPool::new(n_workers);                             //create thread pool
```

Por fim, foi necessário dividir o trabalho entre as *threads*. Para isto, foi utilizado um ciclo *for* no qual, a cada iteração, as variáveis auxiliares *"min"* e *"max"* definem a linha inicial e final do bloco onde cada *thread* vai executar:

```
let n_jobs = 100;
let (tx, rx) = channel();

for id in 0..n_jobs{
    let tx = tx.clone();
    let min = id * NPOINTS / n_jobs;
    let max = (id + 1) * NPOINTS / n_jobs;
    pool.execute(move || {
      let sum = (min..max).into_iter().map(|i|
        (0..NPOINTS).into_iter().map(|j| {
          test_point(i,j)
        }).sum::<i32>()
      ).sum::<i32>();
      tx.send(sum).expect("channel will be there waiting for the pool");
    });
```

```
  }
  let result = rx.iter().take(n_jobs as usize).fold(0, |a, b| a + b);
  return result;
}
```

Nesta implementação, foi necessário criar um canal utilizando a função *"channel()"*. Esta função retorna duas variáveis que representam as extremidades de transmissão (*tx*) e recepção (*rx*). Como no Rust a variável *tx* não pode ser acessada por diversas *threads* ao mesmo tempo, foi necessário criar uma cópia utilizando a função "*clone()*". As *threads* transmitem o resultado usando "*tx.send(sum)*", enquanto a *thread* principal soma todas as variáveis *tx* através da variável *rx*. Para obter o resultado final (*result*), a *thread* principal soma cada um dos resultados ao iterar sobre a variável *rx* utilizando o método *take()* e *fold()*.

2.3. Comparação entre as versões sequencial, ThreadPool e Rayon

Utilizando o módulo *std::time*, o grupo comparou o tempo decorrido das implementação sequencial, com *Threadpool* e com *Rayon*. A Figura 1 mostra as vinte amostras recolhidas para fazer a análise estatística apresentada na Tabela 1.

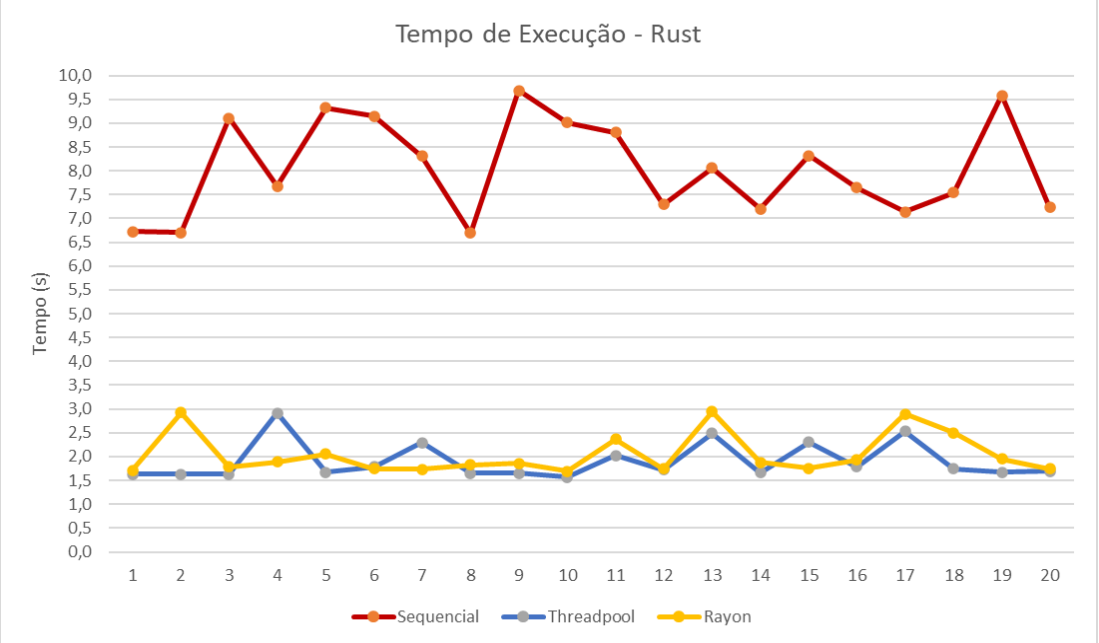

Figura 1 - Comparação entre os tempos de execução com Rust.

A partir da Tabela 1 é possível observar que a implementação paralela utilizando o ThreadPool e Rayon foram semelhantes, mas em média a primeira obteve melhor performance e sofreu o menor desvio padrão.

Tabela 1 - Análise Estatística utilizando o Rust

	Sequencial	Threadpool	Rayon
Average (s)	8,059505	1,902919	2,045882
Std. Deviation (s)	0,984528	0,376310	0,418979
Maximum (s)	9,687156	2,913110	2,942210
Minimum (s)	6,688845	1,573498	1,693482

2.4. Versões paralelas com OpenMP

O desenvolvimento e a paralelização da função que calcula a área de Mandelbrot também foi feito utilizando o OpenMP. Neste caso, utilizou-se da versão com a diretiva *parallel for*, que apresentou melhor performance em relação a utilização de *tasks*.

```
void calc_parallel(struct d_complex c){
  double area, error, eps  = 1.0e-5;
  int i, j;
  #pragma omp parallel num_threads(8) private(c,i,j)
  {
    #pragma omp for collapse(2) //schedule(dynamic, 100)
    for (i=0; i<NPOINTS; i++) {
      for (j=0; j<NPOINTS; j++) {
        c.r = -2.0+2.5*(double)(i)/(double)(NPOINTS)+eps;
        c.i = 1.125*(double)(j)/(double)(NPOINTS)+eps;
        testpoint(c);
      }
    }
  }
  // Calculate area of set and error estimate and output the results
  area=2.0*2.5*1.125*(double)(NPOINTS*NPOINTS-numoutside)/(double)(NPOINTS*NPOINTS);
  error=area/(double)NPOINTS;
}
```

Além disso, também foi feita uma análise da versão paralela utilizando o escalonamento dinâmico de bloco de tamanho 100. Neste escalonamento, o OpenMP designa uma iteração para cada thread. Quando esta thread acabar, será designada para a próxima iteração que ainda não tiver sido executada.

A Figura 2 apresenta um comparativo entre o tempo de execução de 20 amostras para os três casos citados.

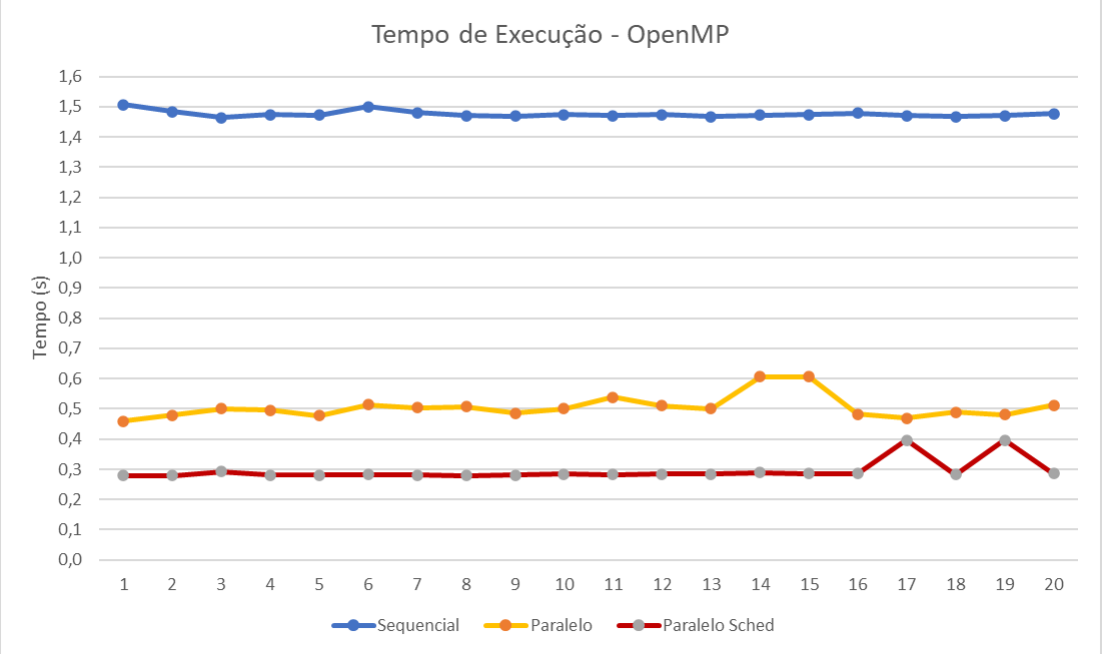

Figura 2 - Comparação entre os tempos de execução com OpenMP.

A partir da tabela 2 é possível observar que em média a implementação utilizando a diretiva *parallel for* obteve melhor performance. Entretanto, ao adicionar a *clause* "*schedule"* especificando escalonamento dinâmico e um bloco de tamanho 100, foi possível obter uma melhora na performance de aproximadamente 0,21s.

Tabela 2 - Análise Estatística utilizando o OpenMP

	Sequencial	**Parallel**	**Parallel with schedule**
Average (s)	1,475721	0,506129	0,294585
Std. Deviation (s)	0,010311	0,037879	0,034058
Maximum (s)	1,506659	0,60701	0,396413
Minimum (s)	1,464231	0,45931	0,278706

3. Conclusão

No exercício 5, calculou-se a área de Mandelbrot com diferentes abordagens. A Figuras 3 apresenta a comparação entre as médias, valores máximos e mínimos das versões paralelas utilizando ThreadPool e Rayon, e a versão sequencial. Enquanto a Figura 4 mostra a mesma análise para a implementação em C utilizando o OpenMP.

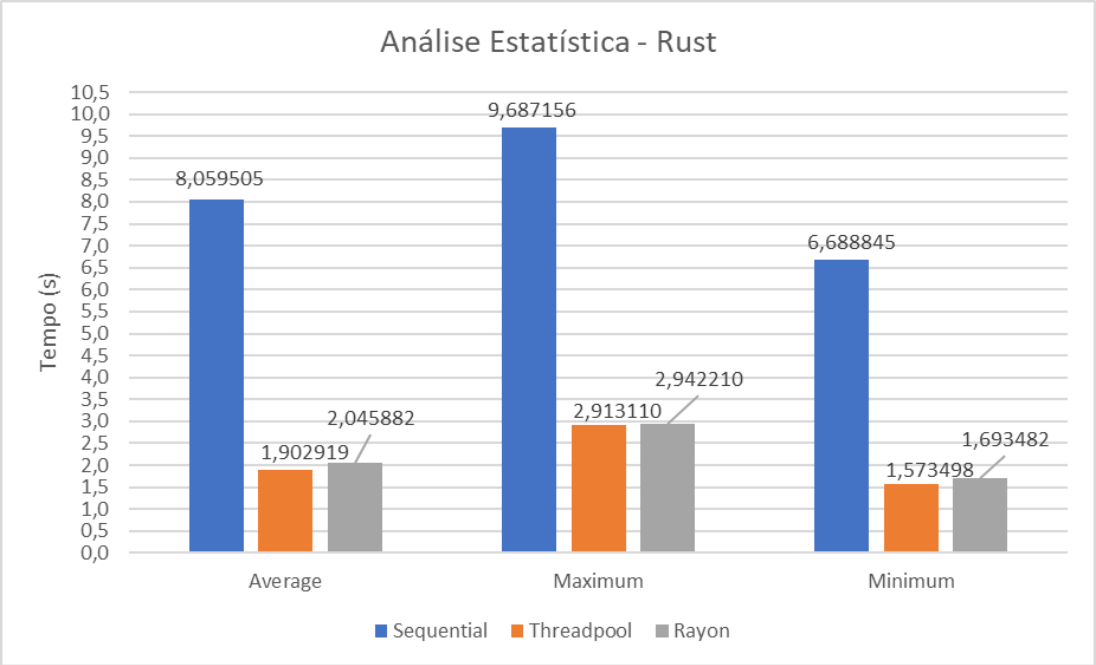

Figura 3 - Análise Estatística utilizando o Rust.

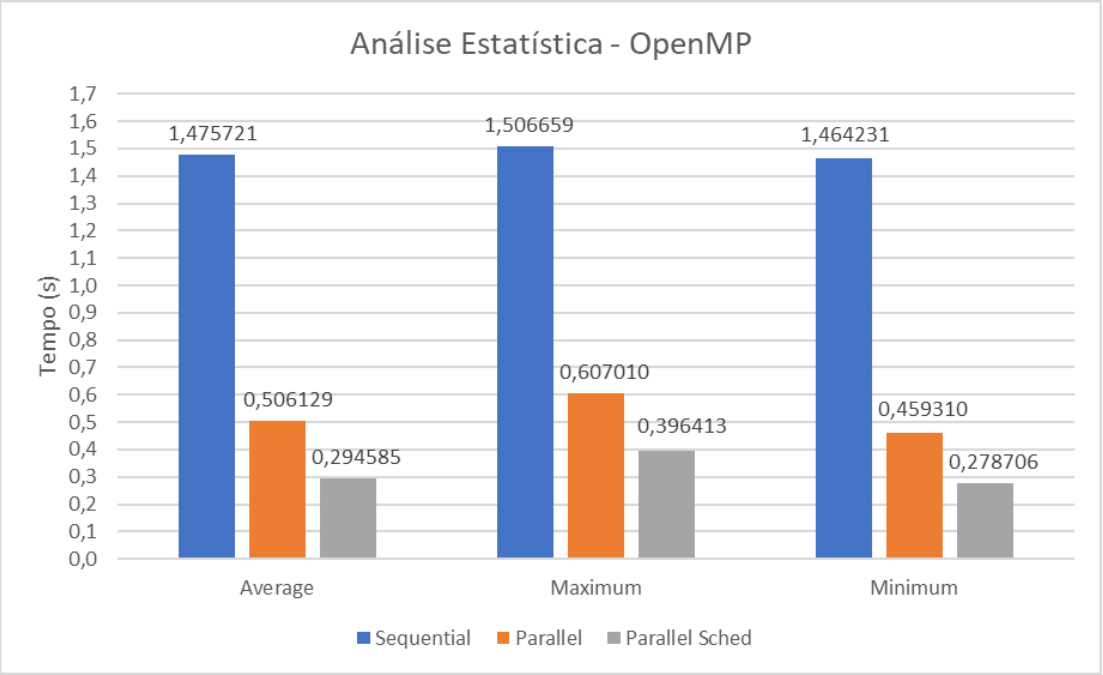

Figura 4 -Análise Estatística utilizando o OpenMP

Ao comparar a média dos tempos de execução da versão sequencial em C com o OpenMP e do Rust, observou-se que o programa utilizando OpenMP obteve melhor performance. Este fato manteve-se ao comparar as versões paralelas do Rust com a versão paralela utilizando o OpenMP. Dessa forma, é possível concluir que a paralelização utilizando a linguagem C junto a biblioteca do OpenMP, em termos de análise temporal, possui melhor performance. Entretanto, o Rust oferece algumas vantagens em relação ao C e ao compilador GCC que devem ser consideradas.

A primeira vantagem consiste no fato do Rust ser uma linguagem fortemente tipada. Isso significa que o compilador garante que nenhuma operação seja aplicada a uma variável de um tipo incorreto. Além disso, a propriedade *"borrowing"* do Rust garante um acesso *thread-safe*. Esta propriedade assegura o compartilhamento de dados, fazendo com que diferentes threads não consigam modificar o valor de uma variável ao mesmo tempo.

Por fim, as bibliotecas de paralelização do Rust como o Rayon, simplificam o paralelismo ao modo a ser possível converter as iterações sequenciais do código em iterações paralelas. Todas essas características, em conjunto, tornam o código Rust mais escalável e seguro do que o código em C.

4. Referências

[1] Crates. "rayon - crates.io: Rust Package Registry". Disponível em: <https://crates.io/crates/rayon>. Acesso em dezembro de 2022.

[1] Microsoft. "Recursos exclusivos do Rust". Disponível em: <https://learn.microsoft.com/pt-br/training/modules/rust-introduction/3-rust-features> . Acesso em dezembro de 2022.

[3] Rust Documentation. "Threadpool - Rust". Disponível em: <https://docs.rs/threadpool/latest/threadpool/>. Acesso em dezembro de 2022.