Apresentação final

Enferrujados: Luan Mark, João Antônio Soares e Gabriel Bessa

Conteúdos

01

Kernels desenvolvidos

Kerneis desenvolvidos

Nosso grupo conseguiu desenvolver com sucesso 4 dos 5 kernels disponibilizados pela NAS utilizando a linguagem Rust, entretanto foi feito a paralelização de apenas um deles.

Entre os kernels desenvolvidos estão: EP, CG, MG e FT.

Sendo paralelizados os seguintes: EP.

02

Testes de desempenho

Máquina de teste

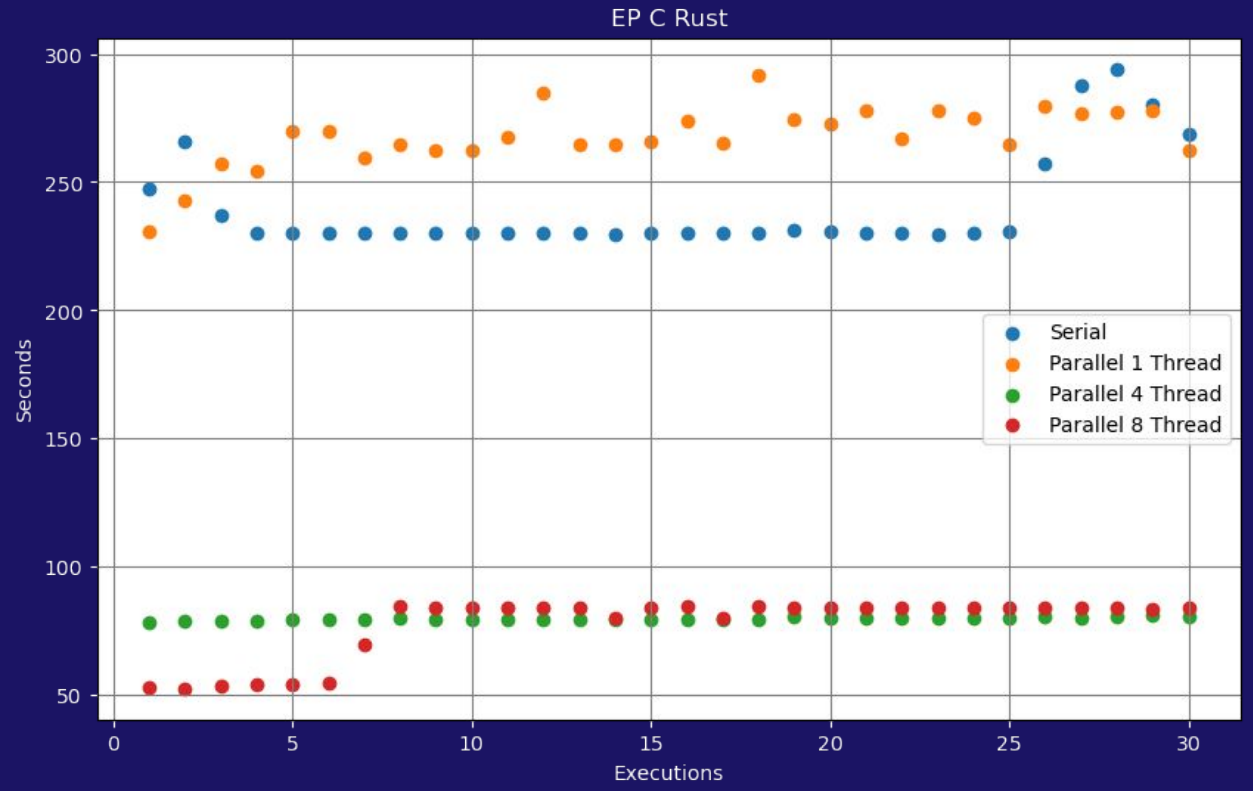
Intel Core i5 10210U (4 cores físicos / 8 cores lógicos)

2 x 4 GB RAM DDR4 2667 MHz

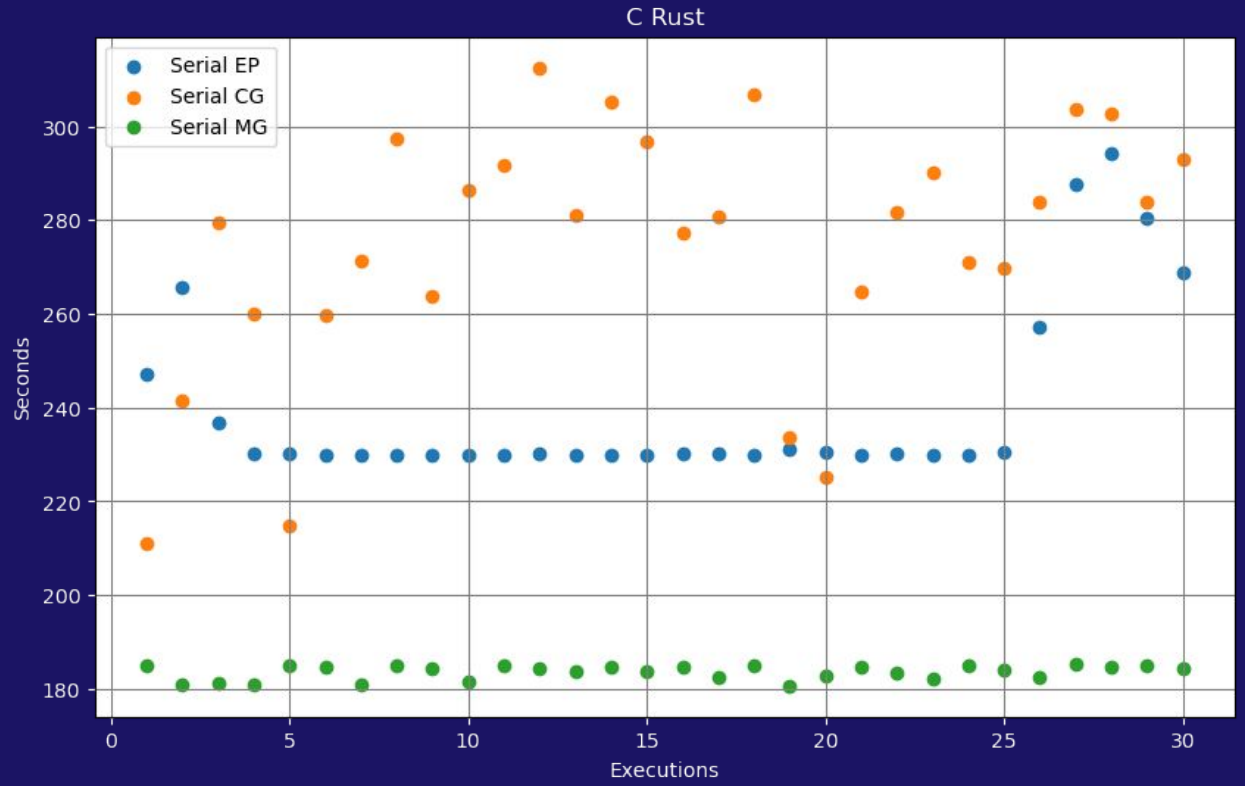
Um pouco sobre os testes

As avaliações de desempenho foram todas baseadas nas execuções dos *kernels* em Fortran, C++ e Rust com a classe C.
Foram feitas 31 execuções, com a primeira dessas execuções tendo seu resultado descartado servindo apenas para inicialização da bateria de testes.
A partir dessas 30 execuções foram retirados os resultados e foi feita uma avaliação e comparação entre cada um deles.

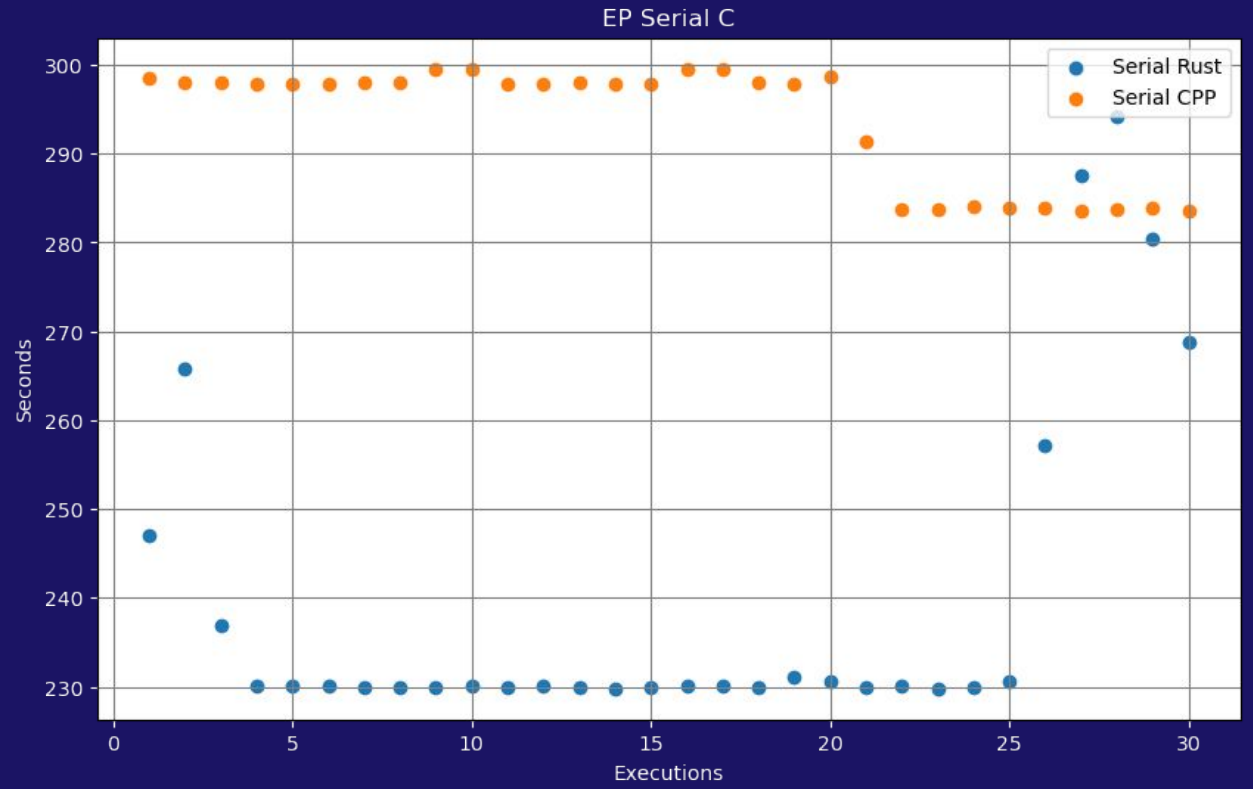
Nuvem de dispersão



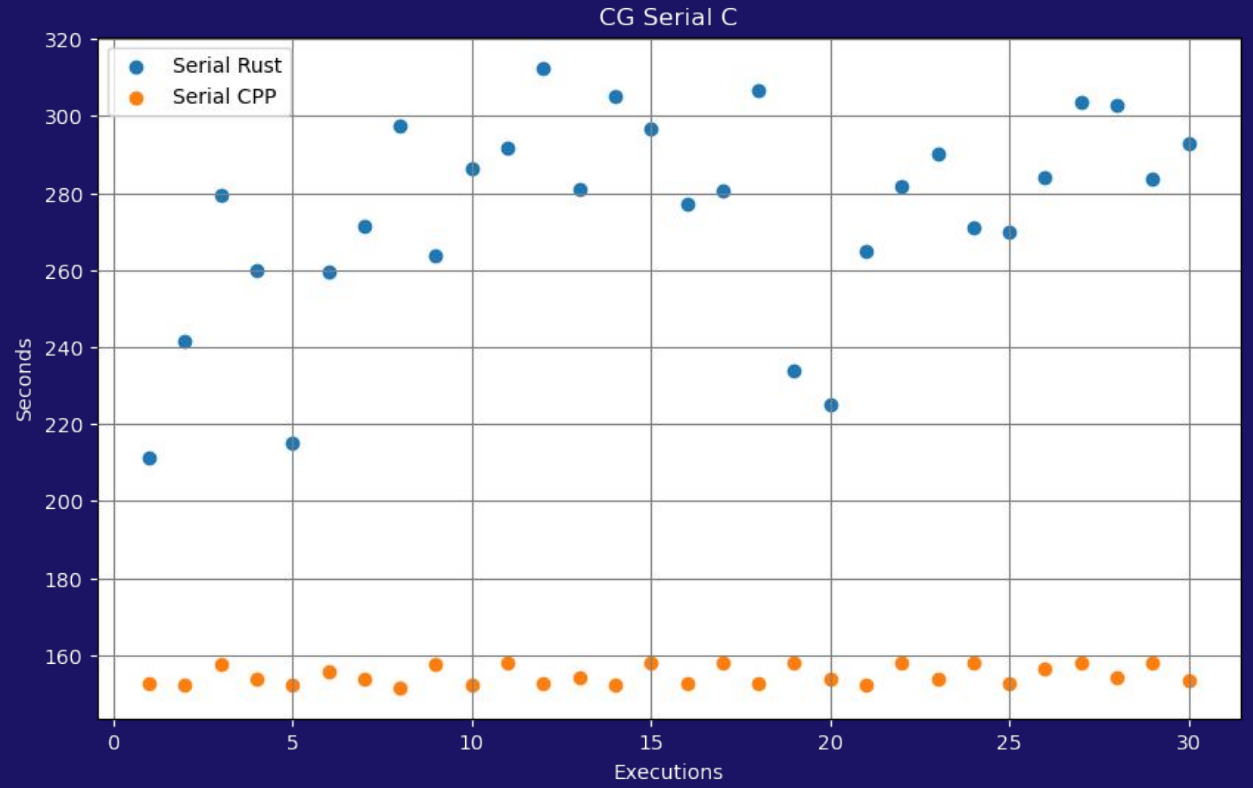
Nuvem de dispersão



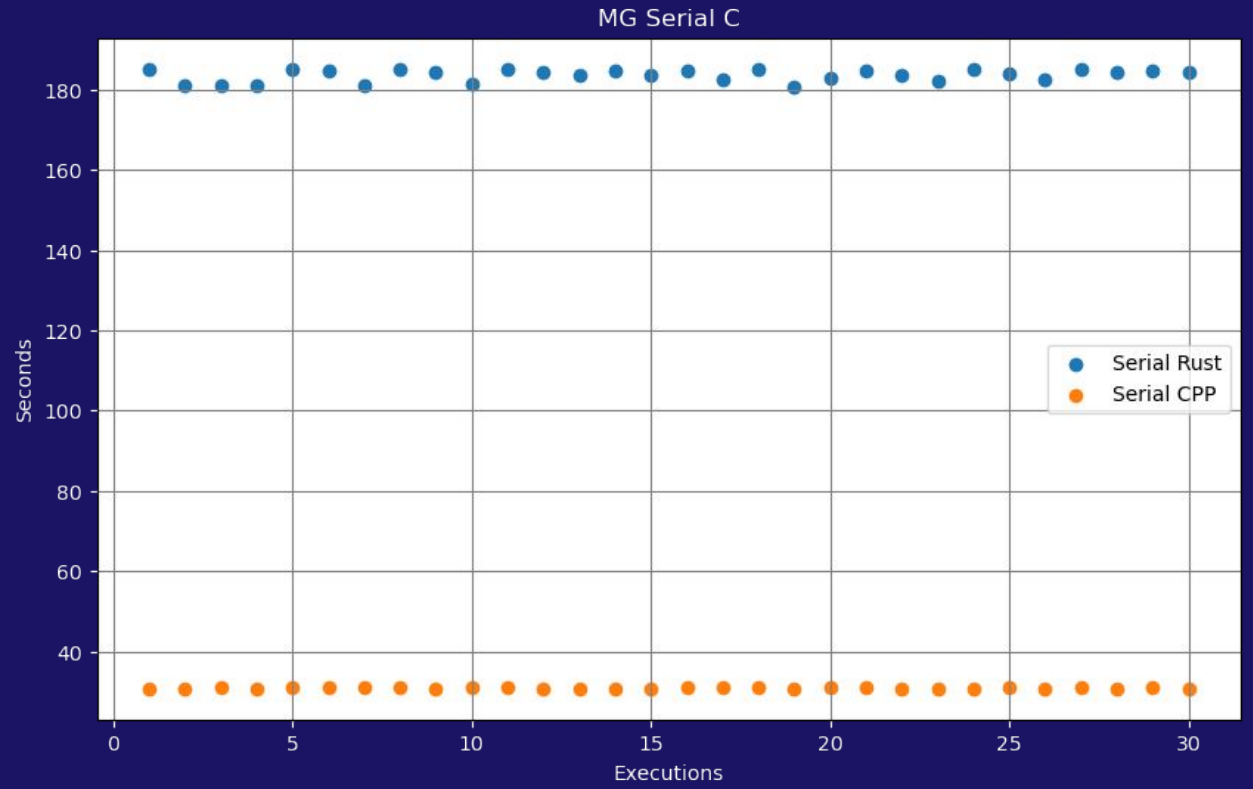
Nuvem de dispersão



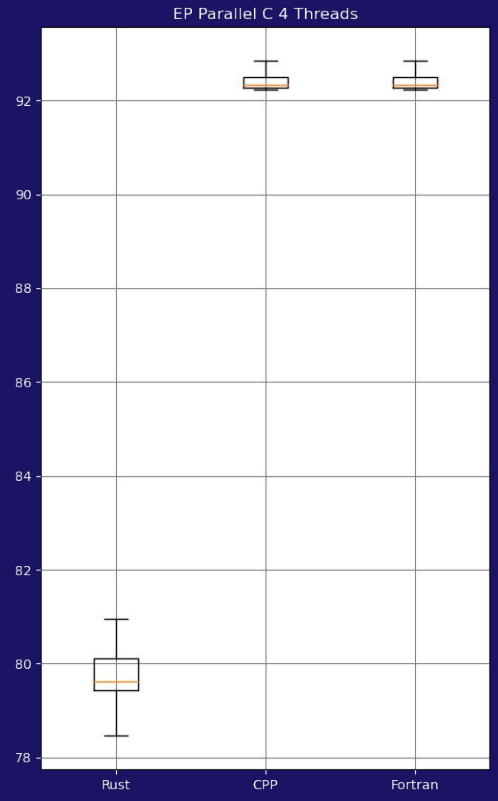
Nuvem de dispersão

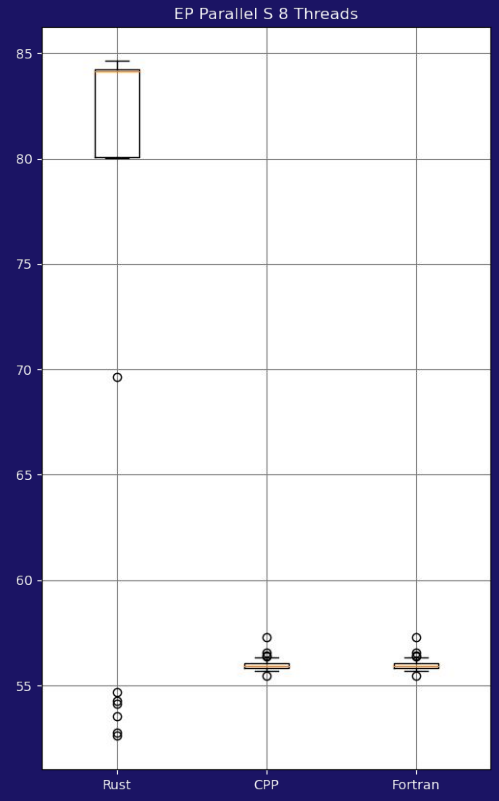


Nuvem de dispersão



Boxplot do EP Paralelo





03

Concorrência em Rust

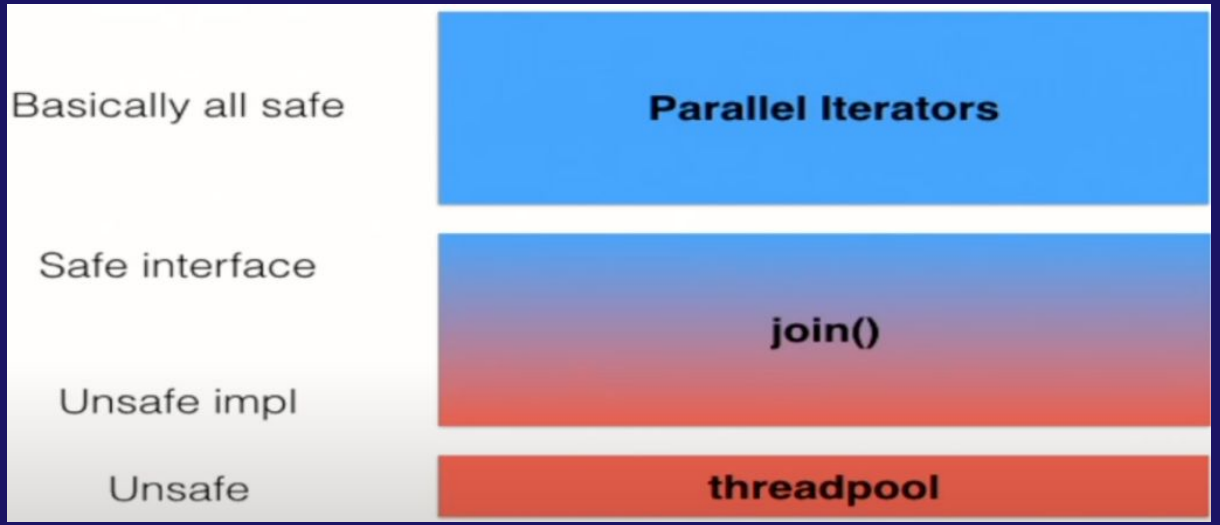
Recursos de concorrência em Rust

Utilizamos a biblioteca Rayon disponível em Rust para fazer a paralelização do kernel EP.

O objetivo do Rayon é facilitar a adição de paralelismo ao código sequencial - basicamente ele converte execução serial de *for loops* e iteradores em execução paralela. Também garante que o uso da API do Rayon não introduzirá condições de corrida no código.

Recursos de concorrência em Rust

Estrutura do Rayon dividida em camadas



Recursos de concorrência em Rust

Rayon's core primitive: join

O uso do "join" é muito simples. Você o invoca com duas closures e ele potencialmente as executará em paralelo. Uma vez que ambas tenham terminado, ele retornará.
A decisão de usar ou não threads paralelos é feita dinamicamente, com base na disponibilidade de núcleos ociosos.

Rayon's Join operation

```
// `do_something` and `do_something_else` *may* run in parallel
join(|| do_something(), || do_something_else())
```

Recursos de concorrência em Rust

How join is implemented: work-stealing

Em cada chamada para join(a, b), identificamos duas tarefas a e b que podem ser executadas com segurança em paralelo, mas ainda não sabemos se há threads ociosas. Tudo o que a thread atual faz é adicionar b em uma fila local de "trabalhos pendentes" e, em seguida, imediatamente começar a executar a.

Sempre que está ociosa, cada thread parte para examinar as filas de "trabalhos pendentes" de outras threads: se encontrarem um item lá, eles roubam e o executam eles mesmos.

O que torna o *work-stealing* elegante é que ele se adapta à carga da CPU. Ou seja, se todos os *workers* estiverem ocupados, então join(a, b) basicamente se reduz a executar cada *closure* sequencialmente (ou seja, a(); b();). Isso não é pior do que o código sequencial. Mas se houver threads ociosas disponíveis, então obtemos o paralelismo.

Concorrência no NPB

Parallel Patterns: Map, Map Reduce

- Map consiste na replicação de uma função que é aplicada a todos os elementos de um conjunto indexado. Isso pode ser usado para paralelizar laços "*for*" quando as iterações são independentes.

- MapReduce é a união dos padrões Map e Reduce. O padrão Reduce combina todos os elementos de uma coleção e produz um único elemento usando um operador binário associativo. Portanto, no MapReduce, cada elemento do Map é combinado em um único elemento. Esse padrão pode ser usado para paralelizar laços "for" quando as iterações apresentam dependências de dados específicas e é necessária uma sincronização.

Table 2
Structure of each NPB benchmark in terms of parallel patterns in FastFlow(FF), OpenMP (OMP), and Intel TBB (TBB).

Benchmark	Map			MapReduce			Barriers		
	FF	OMP	TBB	FF	OMP	TBB	FF	OMP	TBB
EP	1	1	1	1	1	1	1	1	1
MG	11	11	10	1	1	1	11	11	10
CG	7	18	7	4	6	4	7	11	7
FT	8	8	8	1	1	1	8	10	8
IS	6	7	6	1	1	1	6	7	6
BT	23	23	23	–	–	–	23	23	23
SP	19	19	19	–	–	–	19	19	19
LU	22	23	22	1	1	1	22	19	22

Concorrência no NPB

EP – PARALLEL IMPLEMENTATION – TBB VS Rayon

- Implementação TBB [Löff / Dalvan] - foi paralelizado com MapReduce usando static scheduling. Adicionalmente foi necessário uma sincronização "especial", como a implementação padrão do MapReduce aceita apenas tipos padrão e há uma redução em um array, eles implementaram manualmente a sincronização dos dados no TBB, Fastflow e OpenMP
- Nossa implementação em Rust-Rayon – Também utiliza MapReduce, usando a função de alta ordem ***Fold*** a qual itera sobre os elementos de uma coleção ou sob um intervalo e realiza a operação de redução em um acumulador. E por último a operação ***Reduce*** para sincronizar os valores do acumulador de cada *chunk* em um valor final.

EP TBB MapReduce pattern

```
tbb::parallel_for(tbb::blocked_range<size_t>(1,np+1),[&](const
tbb::blocked_range<size_t>& r){
  for(k=r.begin(); k != r.end(); k++){
    //SOME COMPUTATION
  }
  critical_section.lock();
        // Q[i], Sx, Sy Synchronization
        critical_section.unlock();
  });
```

EP Rust-Rayon MapReduce Pattern

```
let result:(f64,f64,Vec<u32>) = (1..np+1).into_par_iter().fold(||
  (0.0,0.0,vec![0;(NQ)as usize]),|mut tupl, k| {
  for i in 0..NK {
  // SOME COMPUTATION
  }
  tupl
  }).reduce_with(|mut tupl1, tup|{
              //SOME REDUCTION
          }).unwrap();
```

04

Desafios encontrados

Desafios encontrados

- Reshaping de vetores
- Tratativa de parâmetros globais
- Mutabilidade interior de variáveis

```
makea(naa,
      nzz,
      a,
      colidx,
      rowstr,
      firstrow,
      lastrow,
      firstcol,
      lastcol,
      arow,
      (int(*)[NONZER+1])(void*)acol,
      (double(*)[NONZER+1])(void*)aelt,
      iv);
```

```
fft(dir: -1, x1: &u1, x2: &u1, &u);
```

```
let mut u1: RefCell<Vec<Dcomplex>> = RefCell::new(vec![dcomplex_create(0.0, 0.0); NTOTAL]);
```

```
makea(&mut naa, &mut nzz, &mut a, &mut colidx, &mut rowstr,
    &firstrow, &lastrow, &firstcol, &lastcol, &mut arow, &mut acol, &mut aelt, &mut iv,
    &NONZER, &RCOND, &SHIFT, &mut tran, &amult);
```

05

Conclusões

Conclusões

Segundo os resultados pudemos verificar que ainda é necessário uma melhoria na otimização dos *kerneis* desenvolvidos, ainda assim obtivemos bons resultados em alguns testes, se aproximando dos benchmarks escritos em outras linguagens.
É preciso também seguir com a implementação dos *kerneis* com suas respectivas versões paralelas.

Obrigado pela atenção!