Programação Paralela - ELC139

Aplicação do Método de Monte Carlo em OpenMP

Rhauani Fazul e Roger Couto

2018/1

Sumário

- Versão em C++
  - Primeira solução;
  - Segunda solução;
  - Resultados;
  - Extra.
- Versão em C
  - Particularidades;
  - Primeira solução;
  - Segunda solução;
  - Comparação.
- Referências.

Versão em C++

- Código disponível em: https://goo.gl/ugojLw
- Compilação:
  - Makefile:

```
$ make
```

  - Diretamente:

```
$ g++  -std=c++11  -fopenmp  -o firesim  *.cpp  -Wall
```

- Execução:

```
$ ./firesim <nThreads> <forestSize> <nTrials> <nProbs>
```

Versão em C++

- Medição de tempo
  - Header: **<chrono>**
  - Clock: **high_resolution_clock**

```
typedef std::chrono::high_resolution_clock Clock;

auto t_start = Clock::now();
/* Simula incêndio na floresta */
auto t_end = Clock::now();

duration_cast<nanoseconds>(t_end - t_start).count();
                /*  <microseconds>
                    <milliseconds>
                            ....          */
```

Primeira solução

- No contexto do programa **firesim**, para cada valor de probabilidade no intervalo de {0, 1, ... , **nProb**} o percentual de árvores queimadas é calculado diversas vezes (**nTrials)**;

- Nesta solução, a paralelização é feita com base no particionamento **estático** das iterações referentes a **nProb** entre as **nThreads** criadas.

Primeira solução

```
58  #pragma omp parallel private(ip, it, rand) num_threads(n_threads)
59  {
60      Forest* forest = new Forest(forest_size);
61      #pragma omp for schedule(static, chunk_size)
62      for (ip = 0; ip < n_probs; ip++) {
63          prob_spread[ip] = prob_min + (double) ip * prob_step;
64          percent_burned[ip] = 0.0;
65          rand.setSeed(base_seed+ip); // nova seqüência de números aleatórios
66
67          // executa vários experimentos
68          for (it = 0; it < n_trials; it++) {
69              // queima floresta até o fogo apagar
70              forest->burnUntilOut(forest->centralTree(), prob_spread[ip], rand);
71              percent_burned[ip] += forest->getPercentBurned();
72          }
73
74          // calcula média dos percentuais de árvores queimadas
75          percent_burned[ip] /= n_trials;
76          // mostra resultado para esta probabilidade
77          printf("%lf, %lf\n", prob_spread[ip], percent_burned[ip]);
78      }
79  }
```

Primeira solução

- Criando o time de threads:

```
#pragma omp parallel private(ip, it, rand) num_threads(n_threads)
```

- Dividindo as iterações do laço entre o time:

```
#pragma omp for schedule(static, chunk_size)
```

* É necessário alocar uma floresta para cada *thread*:

```
Forest* forest = new Forest(forest_size);
```

Segunda solução

- Nesta solução, a paralelização é feita com base no particionamento **dinâmico** das iterações referentes a **nProb** entre as **nThreads**.

```
#pragma omp for schedule(dynamic)
```

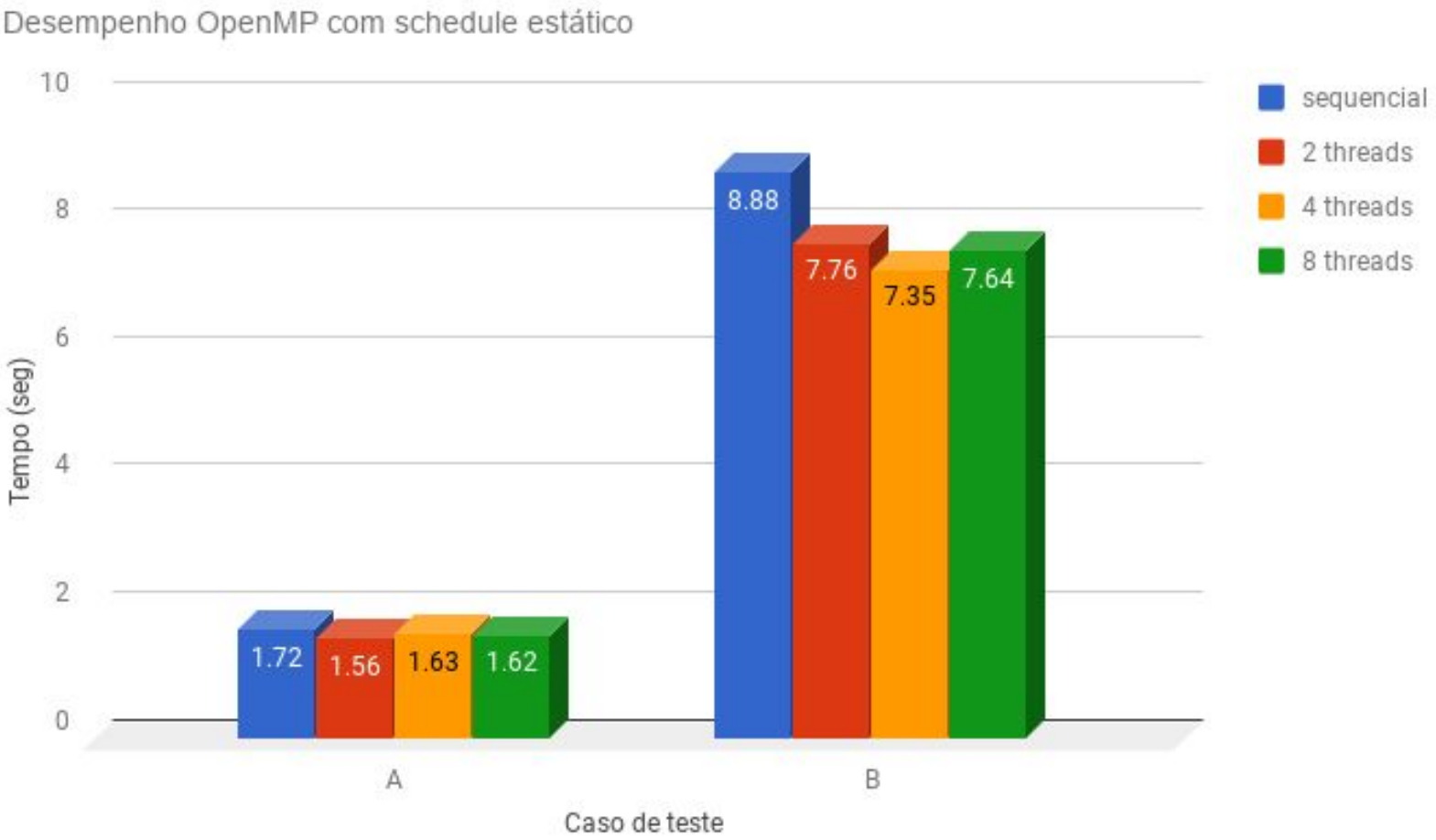
Resultados

- Configurações de entrada utilizadas para os testes:

Caso #	Tamanho do problema	Número de experimentos	Probabilidade máxima
A	10	1000	101
B	15	2000	101
C	20	3000	101
D	25	4000	101
E	30	5000	101

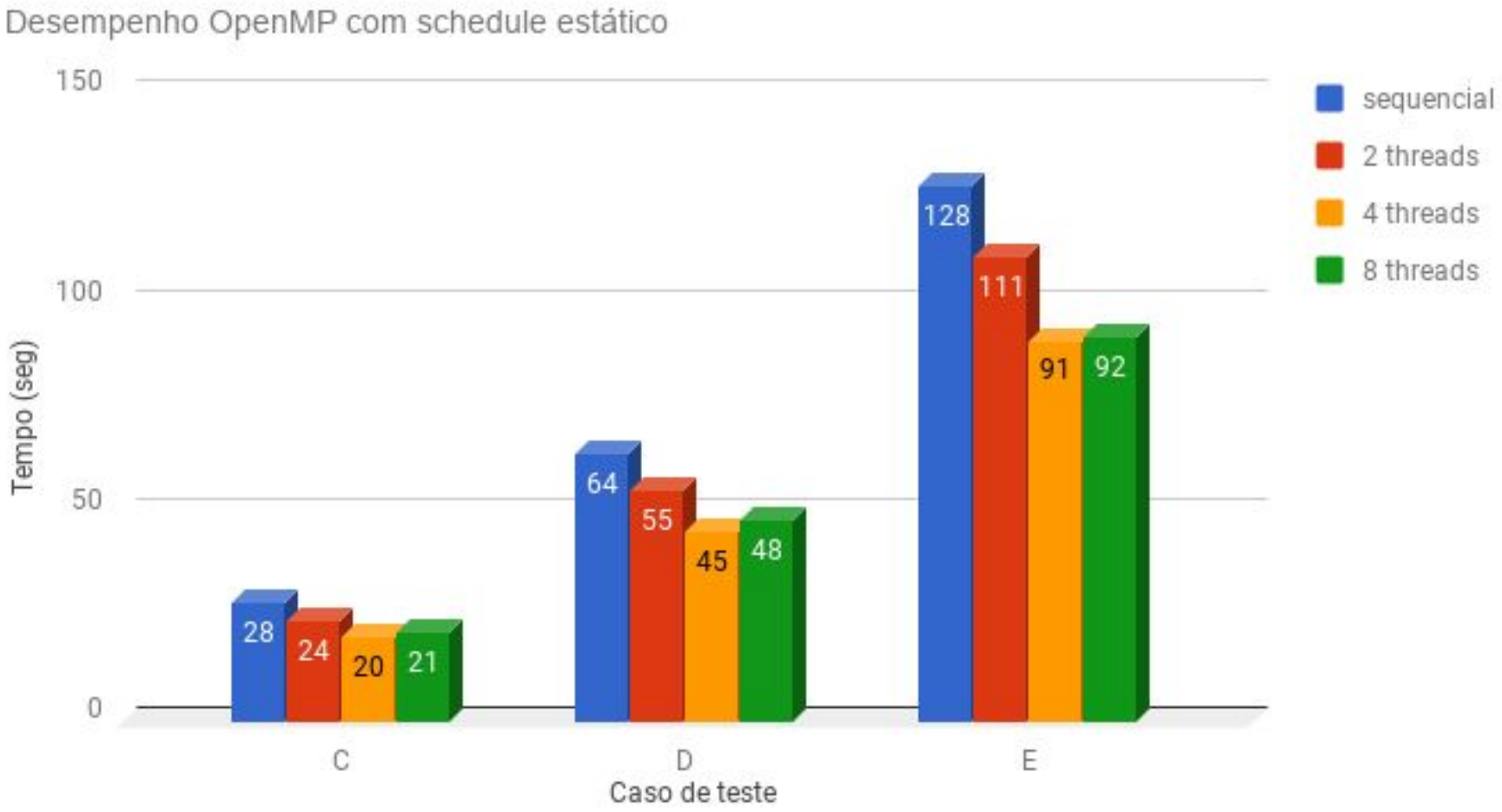
- Para obter uma média de desempenho confiável, cada caso de teste foi executado 30 vezes, variando o número de *threads*:
  - sequencial;
  - 2 , 4 e 8 *threads* (com escalonamento estático e dinâmico);

Resultados



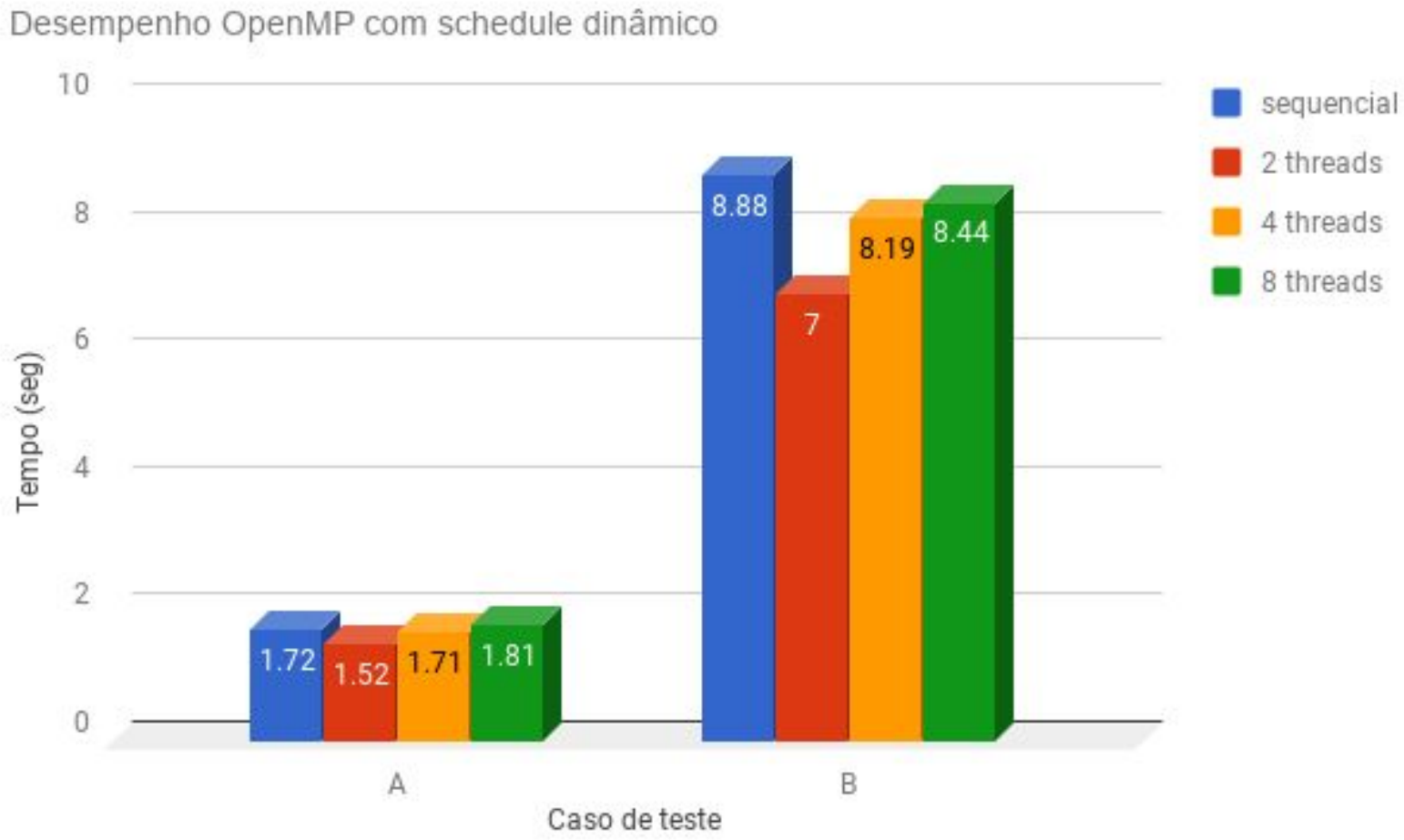
A { 10, 1000, 101 }
B { 15, 2000, 101 }

Resultados



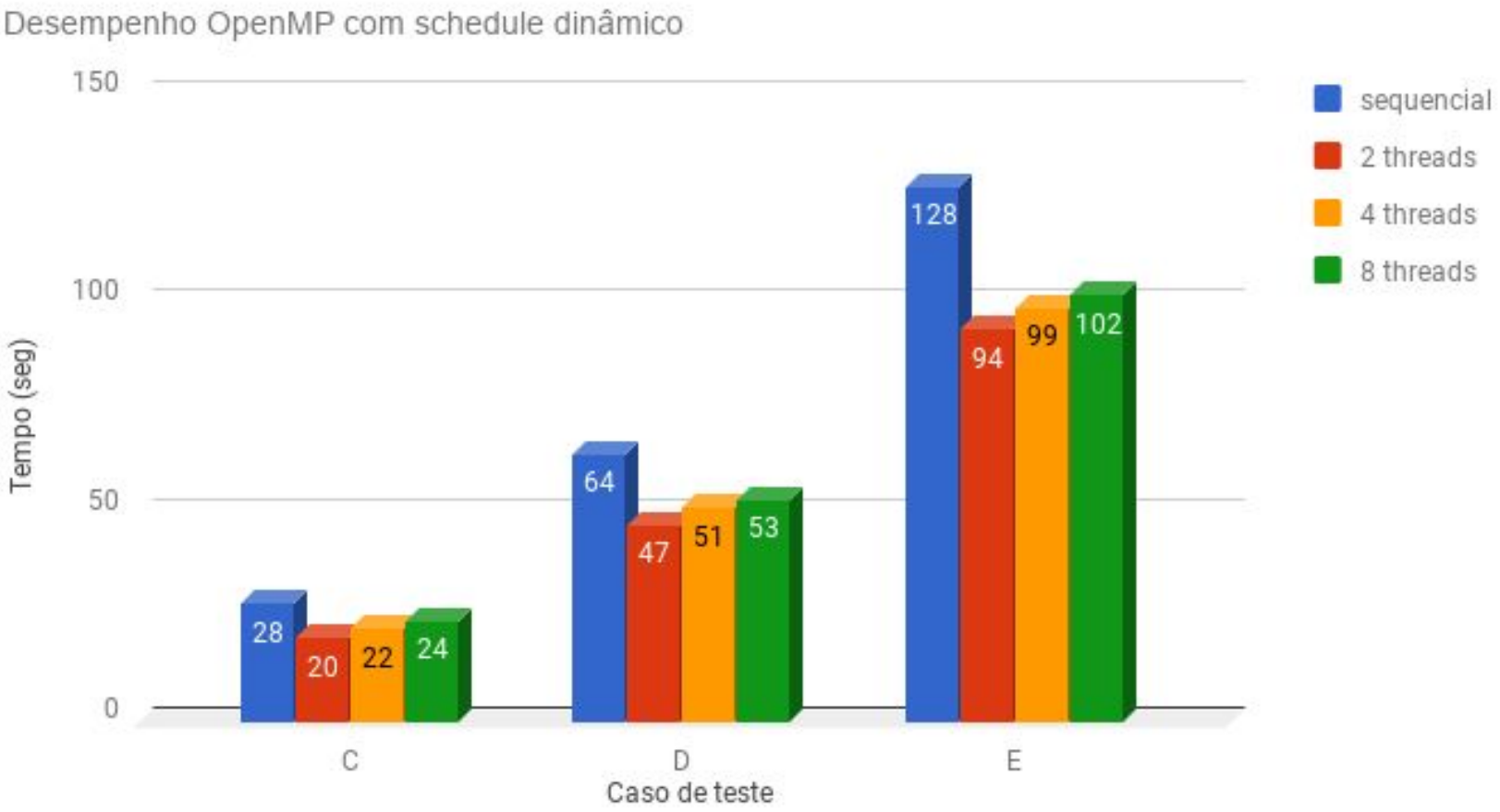
C { 20, 3000, 101 }
D { 25, 4000, 101 }
E { 30, 3000, 101 }

Resultados



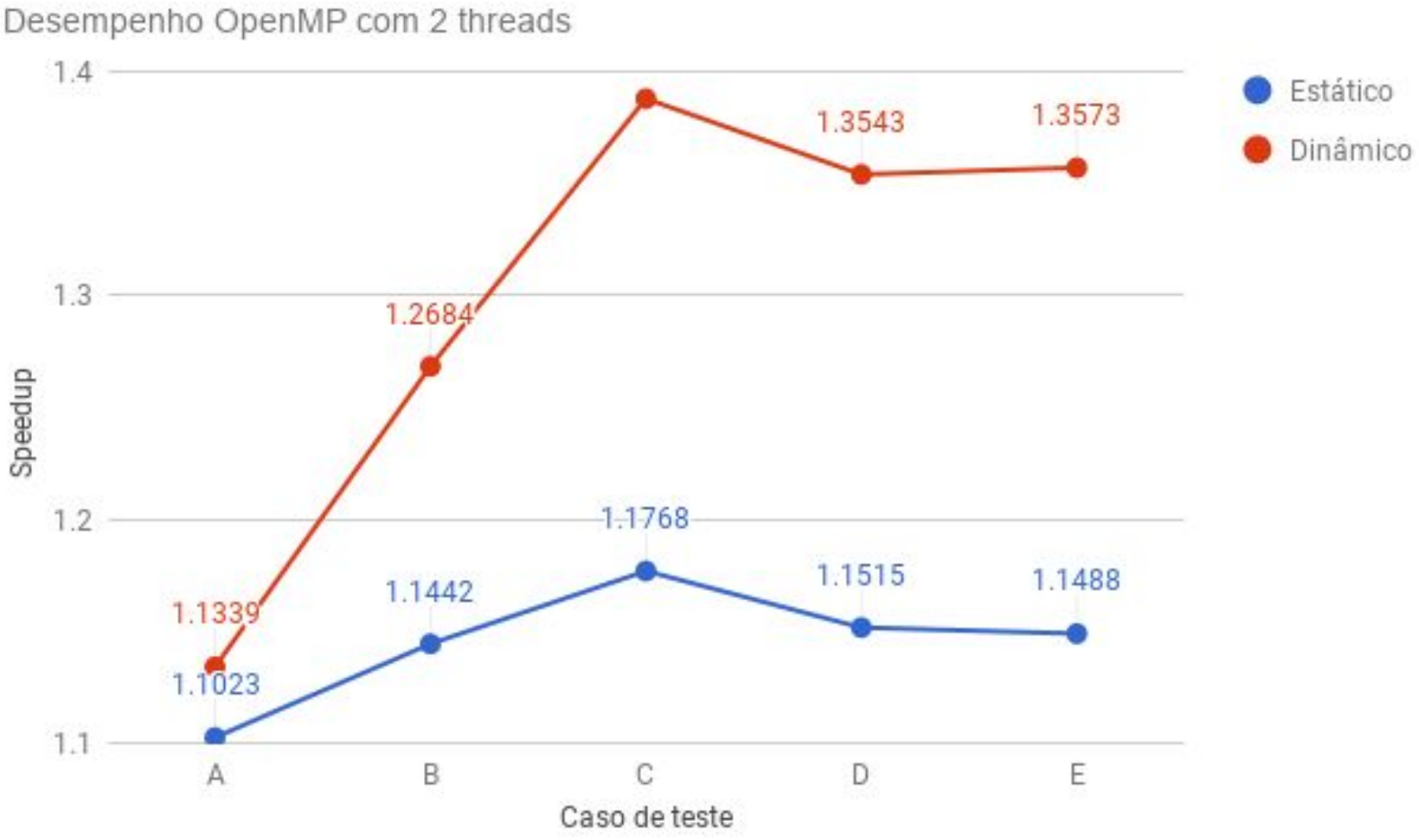
A { 10, 1000, 101 }
B { 15, 2000, 101 }

Resultados

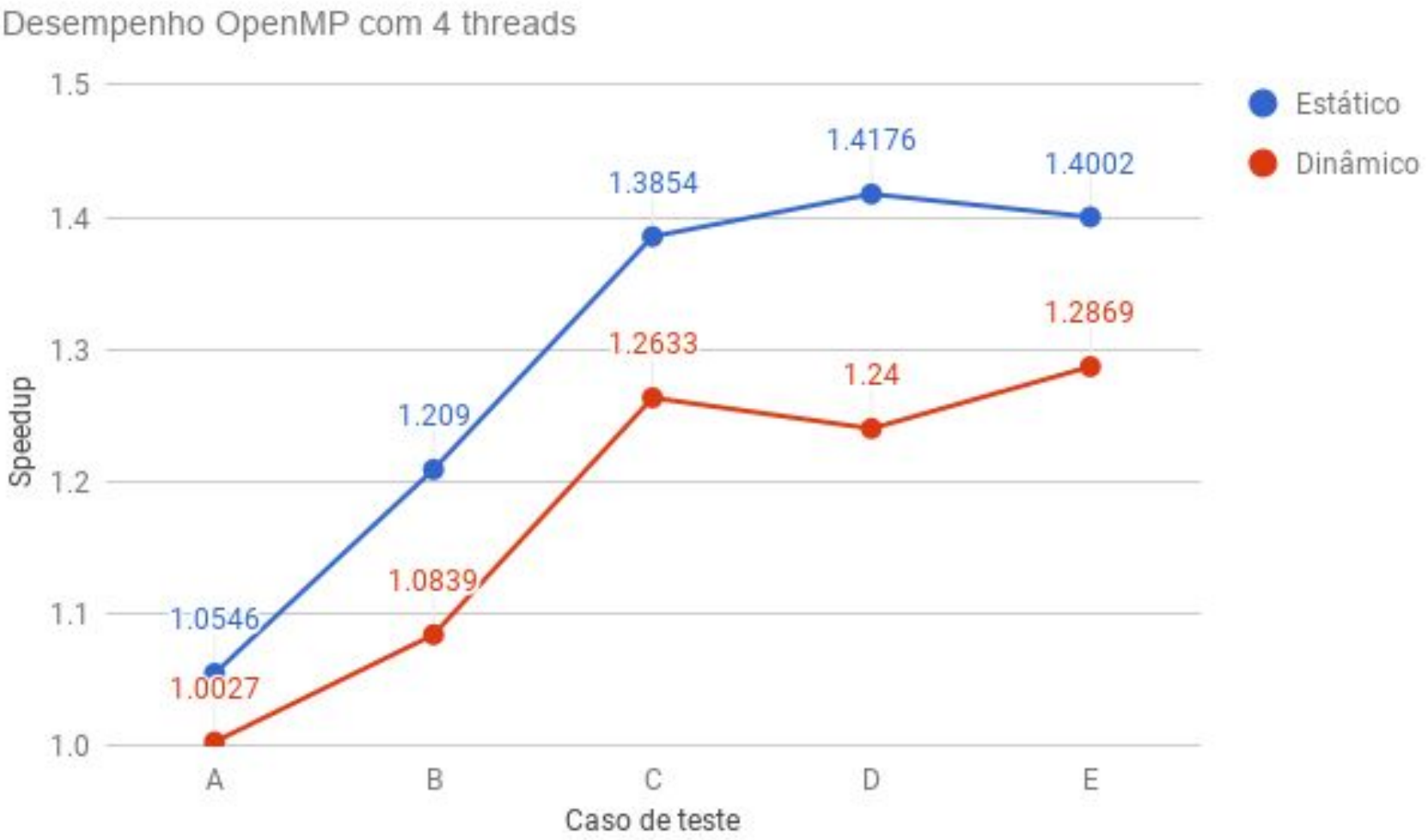


C { 20, 3000, 101 }
D { 25, 4000, 101 }
E { 30, 3000, 101 }

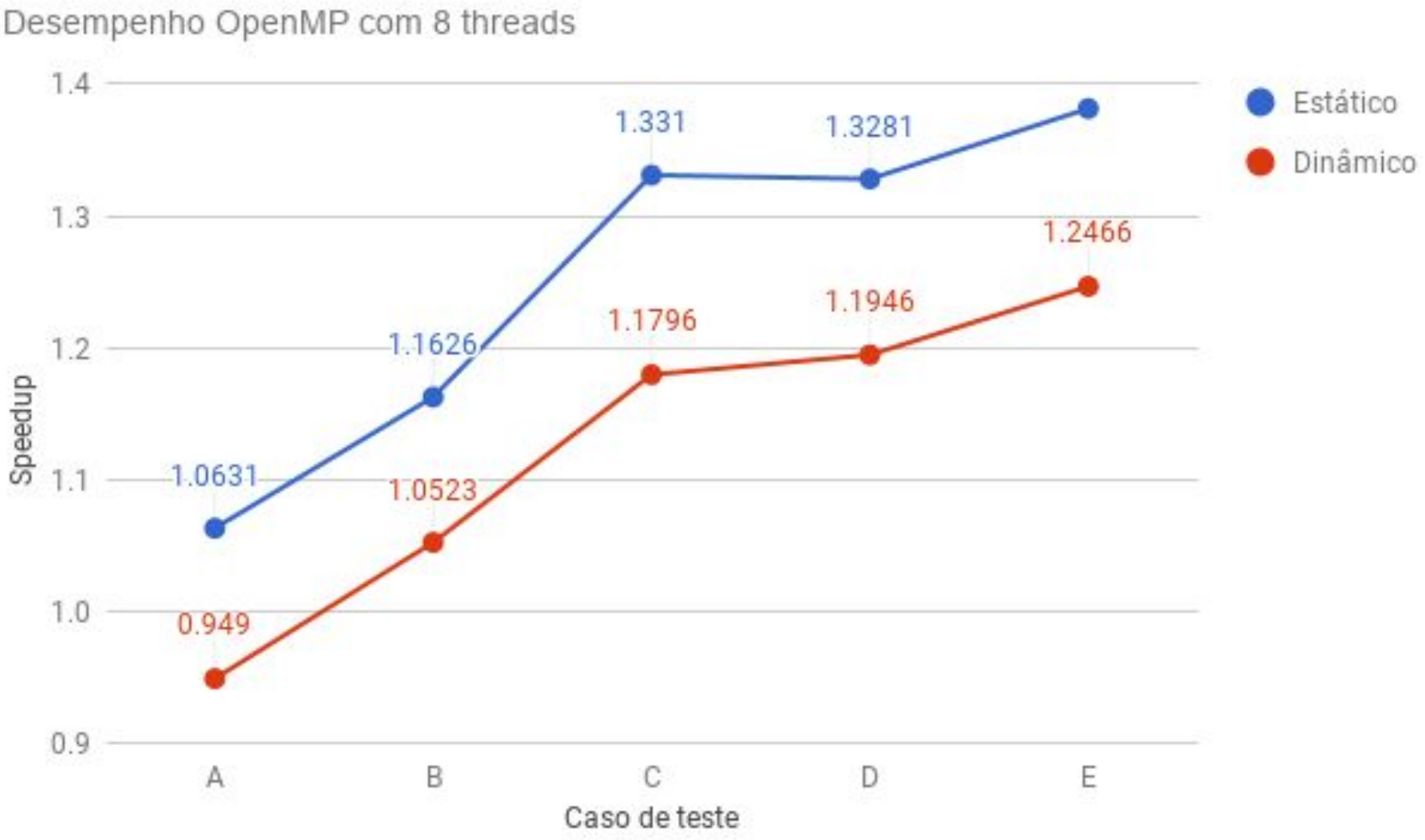
Resultados


Desempenho OpenMP com 2 threads
1.4
1.3
1.2
1.1
Speedup
1.3543
1.3573
1.2684
1.1768
1.1515
1.1488
1.1442
1.1339
1.1023
A
B
C
D
E
Caso de teste
Estático
Dinâmico

Resultados


Desempenho OpenMP com 4 threads
1.5
1.4
1.3
1.2
1.1
1.0
Speedup
1.0546
1.209
1.3854
1.4176
1.4002
1.0027
1.0839
1.2633
1.24
1.2869
A
B
C
D
E
Caso de teste
Estático
Dinâmico

Resultados


Desempenho OpenMP com 8 threads
Speedup
1.4
1.3
1.2
1.1
1.0
0.9
1.0631
1.1626
1.331
1.3281
0.949
1.0523
1.1796
1.1946
1.2466
A
B
C
D
E
Caso de teste
Estático
Dinâmico

Extra

- Tentativas frustradas para tentar aumentar o desempenho com o OpenMP :(

```
#pragma omp parallel for collapse(2)
for (int ip = 0; ip < n_probs; ip++)
    for (int it = 0; it < n_trials; it++)
```

```
// utilizar apenas um loop
for (int n = 0; n < n_probs * n_trials; n++)
    ip = n / n_trials;
    It = n % n_trials;
```

Versão em C

- Também foram implementadas duas versões do programa **firesim** utilizando a linguagem C. As abordagens foram:
    - Uma matriz normal com um **método recursivo** para propagação do fogo;
    - Duas **matrizes esparsas** contendo somente as posições onde o fogo estava se propagando e, outra onde as árvores já tinham sido queimadas.

Versão em C

- Código disponível em: https://goo.gl/LXFB3c
- Compilação:
  - Diretamente:

```
Primeira versão:
   $ gcc burn_rec.c -o burn -fopenmp
Segunda versão:
   $ gcc burn_me.c -o burn -fopenmp
```

- Execução:

```
   $ ./burn <forestSize> <nTrials> <nProbs>
```

Particularidades

- Não havia possibilidade para paralelização na função responsável por propagar o fogo, pois uma etapa da recursão possuia **dependência** com a etapa anterior;
- Desta forma, nas duas próximas abordagens, o que foi paralelizado foi o *loop* que repete as iterações para realizar os testes novamente.

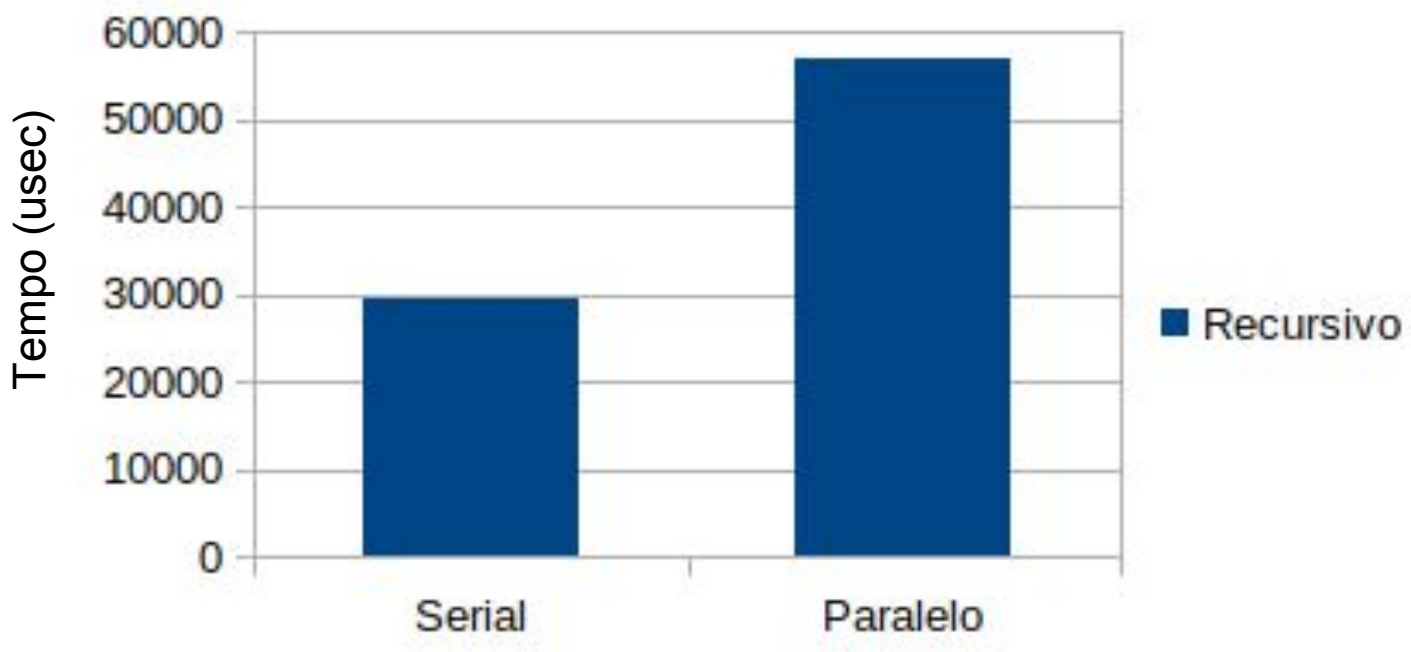
Particularidades

- Nos testes a seguir, foi padronizado uma configuração de teste considerada satisfatória para ilustrar as particularidades das implementações:
  - Tamanho do problema: 30;
  - Repetições: 1000;
  - Probabilidades: de 1 a 100%;
  - Número de *threads*: 4;

- Os dados mostrados são equivalente aos tempo médio de execução (*usec)*.

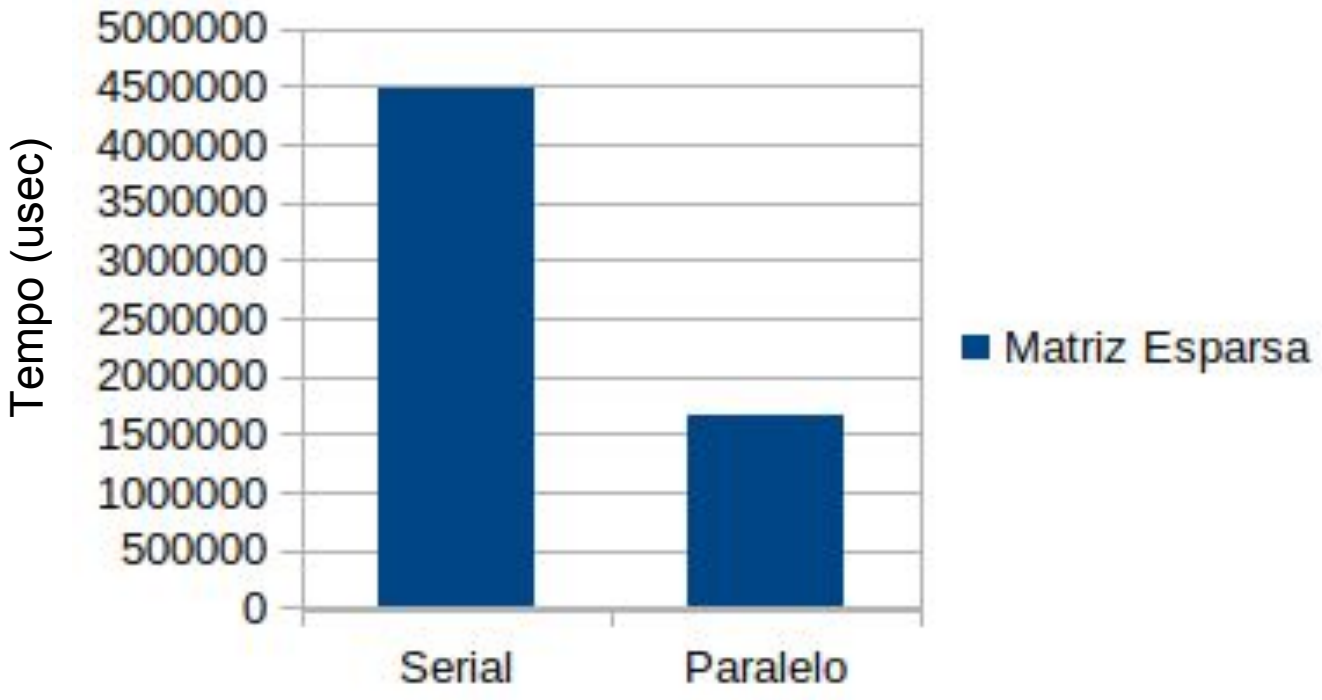
Primeira Solução

- Mesmo paralelizando as repetições, o programa teve foi uma queda de desempenho. Foram testadas diversas opções de *scheduling*.



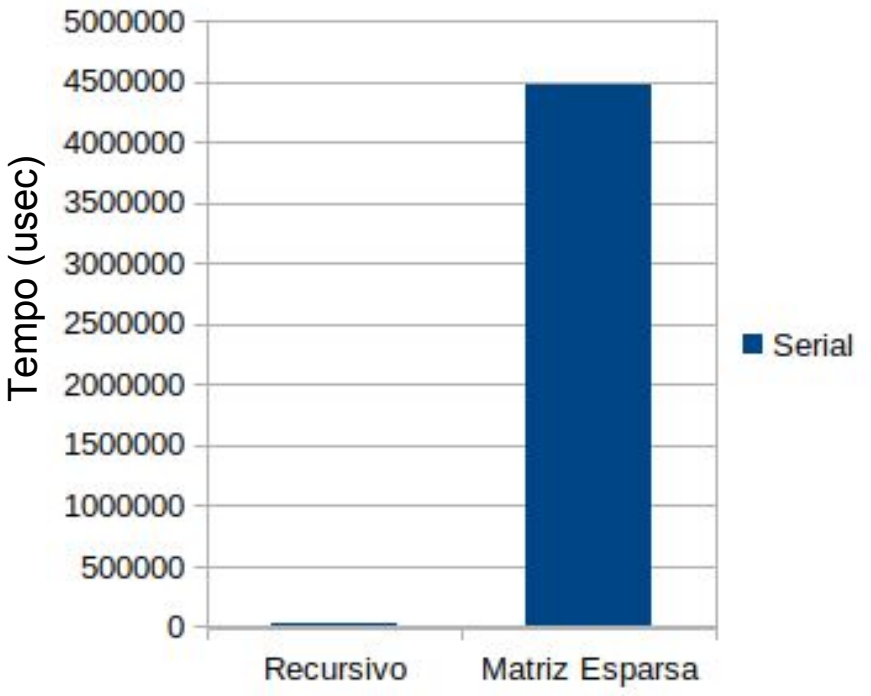
Segunda Solução

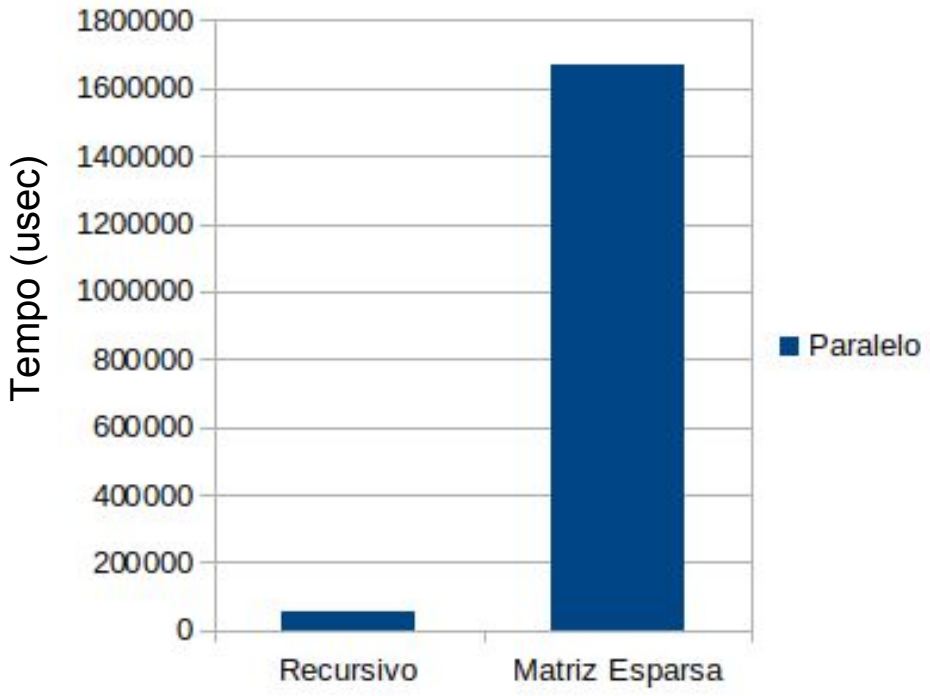
- Nessa abordagem, o paralelismo apresentou grande aumento de desempenho, porém o desempenho comparado a versão recursiva é consideravelmente inferior.



Comparação

- Pode se notar a grande diferença desempenho entre as soluções:

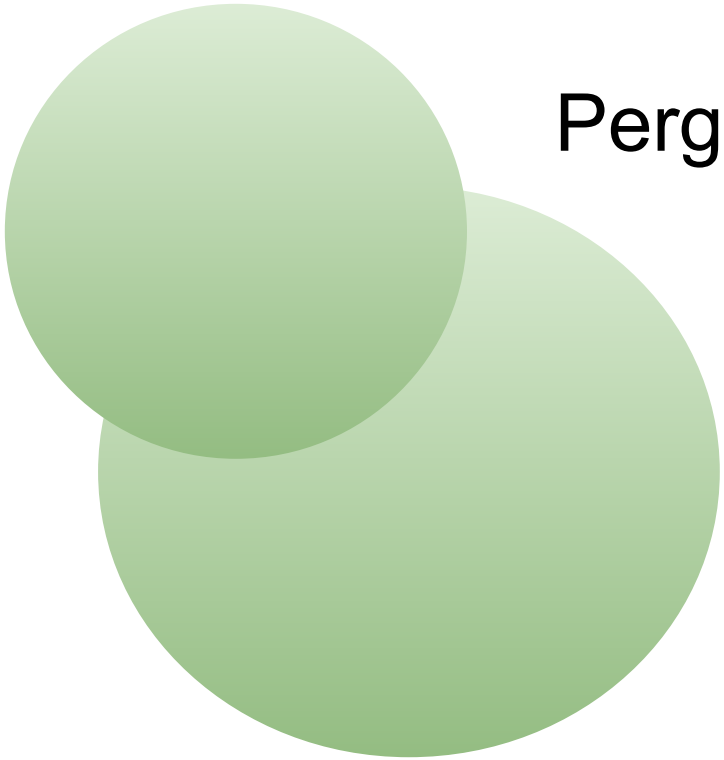




Referências

- FSU Department of Science Computing. ***C++ Examples of Parallel Programming with OpenMP***. https://goo.gl/sqmujr
- Imperial College London. ***Monte Carlo Simulation & Parallel Computing***. https://goo.gl/N35vdu
- Lawrence Livermore National Laboratory. ***OpenMP***. https://goo.gl/o2wTxR
- Mark Bull. ***OpenMP Tips, Tricks and Gotchas***. https://goo.gl/L9Xhyp
- OpenMP. ***OpenMP C and C++ Application Program Interface***. https://goo.gl/wPbQCn
- OpenMP. ***Summary of OpenMP 3.0 C/C++ Syntax***. https://goo.gl/VdvSpi
- Shodor. I***nteractivate: Fire!!***. https://goo.gl/gL9ft6
- Wikipedia. ***Monte Carlo method***. https://goo.gl/pM5nGs

Obrigado!



Perguntas?

{ rwfazul, recouto }@inf.ufsm.br