Universidade Federal Fluminense

Laboratório de Programação Paralela
2022.2

Instituto de Computação

Implementação do Odd-Even Sort Paralelizado Utilizando a Biblioteca OpenMPI

Tiago Maia de Lacerda Brasil
619031065

Odd-Even Sort Sequencial

```
void odd_even_sort(double *array, int length)
{
    for (int i = 0; i < length; i++)
    {
        if (i % 2 == 0)
        {
            for (int j = 0; j < length - 1; j += 2)
            {
                compare_exchange(&array[j], &array[j + 1]);
            }
        }
        else
        {
            for (int j = 1; j < length - 1; j += 2)
            {
                compare_exchange(&array[j], &array[j + 1]);
            }
        }
    }
}
```

Compare-Exchange

```
void compare_exchange(double *a, double *b)
{
    if (double *b < double *a)
    {
        double temp = *a;
        *a = *b;
        *b = temp;
    }
}
```

Odd-Even Sort Sequencial - Exemplo

1 2 3 4 5 6 7 8 9 0

Odd-Even Sort Sequencial - Exemplo

`i=0`

Odd-Even Sort Sequencial - Exemplo

`i=0`

Odd-Even Sort Sequencial - Exemplo

`i=1`

Odd-Even Sort Sequencial - Exemplo

`i=1`

Odd-Even Sort Sequencial - Exemplo

`i=2`

Odd-Even Sort Sequencial - Exemplo

i=2

Odd-Even Sort Sequencial - Exemplo

`i=3`

Odd-Even Sort Sequencial - Exemplo

i=3

Odd-Even Sort Sequencial - Exemplo

i=4

Odd-Even Sort Sequencial - Exemplo

i=4

Odd-Even Sort Sequencial - Exemplo

`i=5`

Odd-Even Sort Sequencial - Exemplo

i=5

Odd-Even Sort Sequencial - Exemplo

`i=6`

Odd-Even Sort Sequencial - Exemplo

i=6

Odd-Even Sort Sequencial - Exemplo

`i=7`

Odd-Even Sort Sequencial - Exemplo

i=7

Odd-Even Sort Sequencial - Exemplo

i=8

Odd-Even Sort Sequencial - Exemplo

i=8

Odd-Even Sort Sequencial - Exemplo

0 1 2 3 4 5 6 7 8 9

Odd-Even Sort Paralelo

```
void odd_even_sort_parallel(double **array, int array_length, int size, int rank)
{
    int chunk_length;
    double *chunk;

    if (rank == 0)
    {
        chunk_length = array_length / size + (array_length % size != 0);

        *array = (double *)realloc(*array, chunk_length * size * sizeof(double));

        for (int i = array_length; i < chunk_length * size; i++)
        {
            (*array)[i] = __INT_MAX__;
        }
    }
    ...
}
```

Odd-Even Sort Paralelo

```
void odd_even_sort_parallel(double **array, int array_length, int size, int rank)
{
    ...
    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Bcast(&chunk_length, 1, MPI_INT, 0, MPI_COMM_WORLD);

    chunk = (double *)malloc(chunk_length * sizeof(double));

    MPI_Scatter(*array, chunk_length, MPI_DOUBLE, chunk, chunk_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    if (rank == 0)
    {
        free(*array);
    }

    quicksort(chunk, chunk_length);
    ...
}
```

Odd-Even Sort Paralelo

```
void odd_even_sort_parallel(double **array, int array_length, int size, int rank)
{
    ...
    for (int i = 0; i < size; i++)
    {
        if (i % 2 == 0)
        {
            if ((rank % 2 == 0) && (rank < size - 1))
            {
                compare_split(chunk, chunk_length, rank, rank, rank + 1);
            }
            else if (rank % 2 == 1)
            {
                compare_split(chunk, chunk_length, rank, rank - 1, rank);
            }
        }
        ...
    }
    ...
}
```

Odd-Even Sort Paralelo

```
void odd_even_sort_parallel(double **array, int array_length, int size, int rank)
{
    ...
    for (int i = 0; i < size; i++)
    {
        ...
        else
        {
            if ((rank % 2 == 1) && (rank < size - 1))
            {
                compare_split(chunk, chunk_length, rank, rank, rank + 1);
            }
            else if ((rank % 2 == 0) && (rank > 0))
            {
                compare_split(chunk, chunk_length, rank, rank - 1, rank);
            }
        }
    }
    ...
}
```

Odd-Even Sort Paralelo

```
void odd_even_sort_parallel(double **array, int array_length, int size, int rank)
{
    ...
    if (rank == 0)
    {
        *array = (double *)malloc(chunk_length * size * sizeof(double));
    }

    MPI_Gather(chunk, chunk_length, MPI_DOUBLE, *array, chunk_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    free(chunk);
}
```

Compare-Split

```
void compare_split(double *chunk, int chunk_length, int rank, int lo_rank, int hi_rank)
{
    MPI_Status status;
    double other[chunk_length];

    MPI_Sendrecv(chunk, chunk_length, MPI_DOUBLE, (rank == lo_rank) ? hi_rank : lo_rank, 0, other,
chunk_length, MPI_DOUBLE, (rank == lo_rank) ? hi_rank : lo_rank, 0, MPI_COMM_WORLD, &status);

    if (rank == lo_rank)
    {
        quicksort_split(chunk, other, chunk_length, chunk_length);
    }
    else
    {
        quicksort_split(other, chunk, chunk_length, chunk_length);
    }
}
```

Odd-Even Sort Paralelo - Exemplo com 3 processos

1 2 3 4 5 6 7 8 9 0

Odd-Even Sort Paralelo - Exemplo com 3 processos

1 2 3 4 5 6 7 8 9 0 ∞ ∞

∞ = `__INT_MAX__`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=0`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=0`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=1`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=1`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=2`

Odd-Even Sort Paralelo - Exemplo com 3 processos

`i=2`

Odd-Even Sort Paralelo - Exemplo com 3 processos

0 1 2 3 4 5 6 7 8 9 ∞ ∞

Odd-Even Sort Paralelo - Exemplo com 3 processos

0 1 2 3 4 5 6 7 8 9

Cálculo do Tempo de Execução

O tempo de execução dos algoritmos foi calculado utilizando a função **clock_gettime** da biblioteca **time**, com resolução de nanosegundos. Foi levado em consideração o tempo real (*wall-clock time*) decorrido durante a execução dos algoritmos.

```
double odd-even(...)
{
    struct timespec begin, end;
    clock_gettime(CLOCK_REALTIME, &begin);
    ...
    clock_gettime(CLOCK_REALTIME, &end);
    long seconds = end.tv_sec - begin.tv_sec;
    long nanoseconds = end.tv_nsec - begin.tv_nsec;
    double elapsed = seconds + nanoseconds * 1e-9;
    return elapsed;
}
```

Método de Comparação

Foram gerados vetores de tamanhos variando entre **1000** e **40000** a cada **1000**, com elementos do tipo double variando no intervalo **[0.0, 1.0)**.

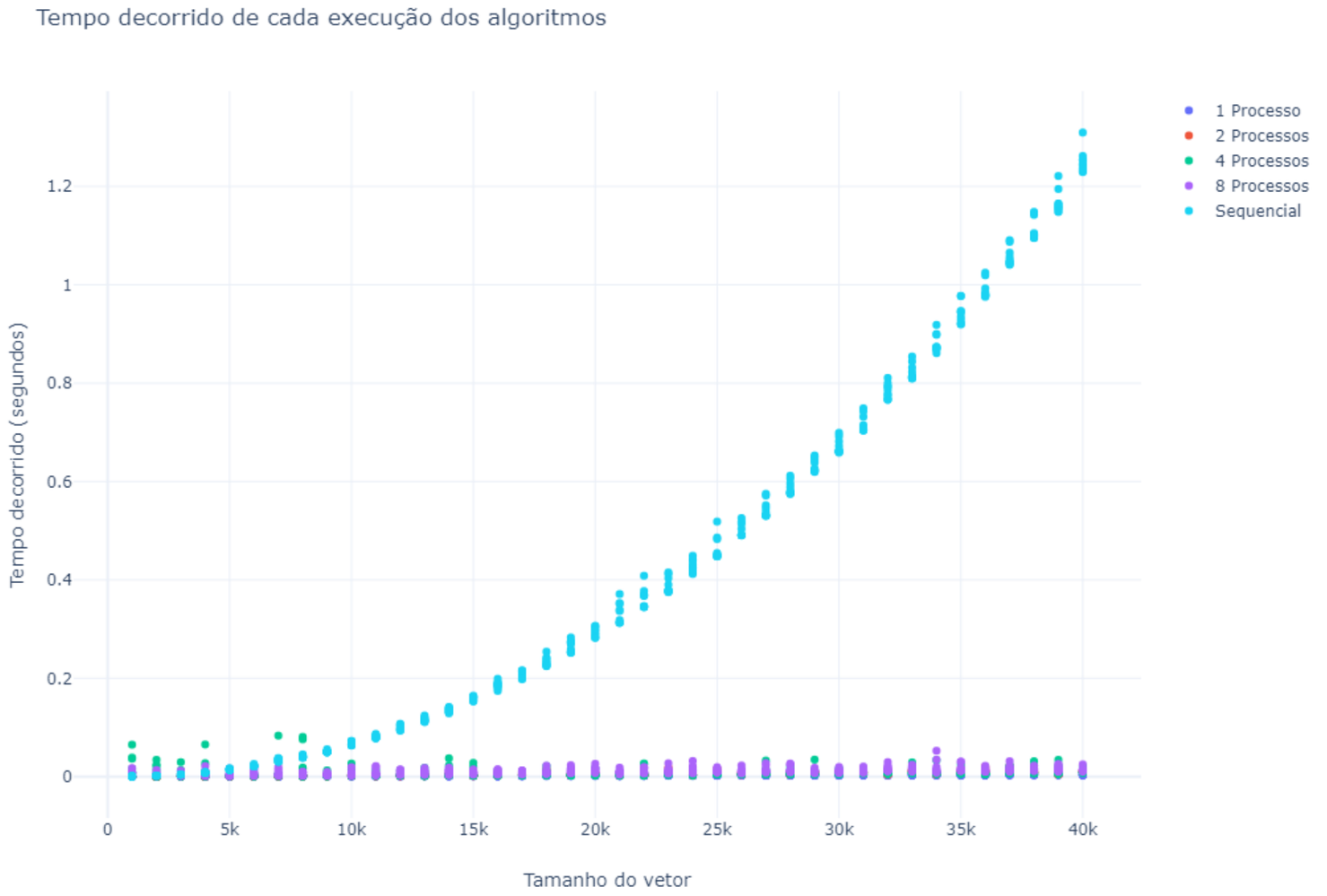
O algoritmo sequencial foi executado **10** vezes para cada vetor.

O algoritmo paralelo foi executado **10** vezes para cada vetor, com **1**, **2**, **4** e **8** processos.

Resultados

Da forma como foi implementado, o algoritmo sequencial tem complexidade **O(n²)** em qualquer caso, o que resulta no comportamento observado.

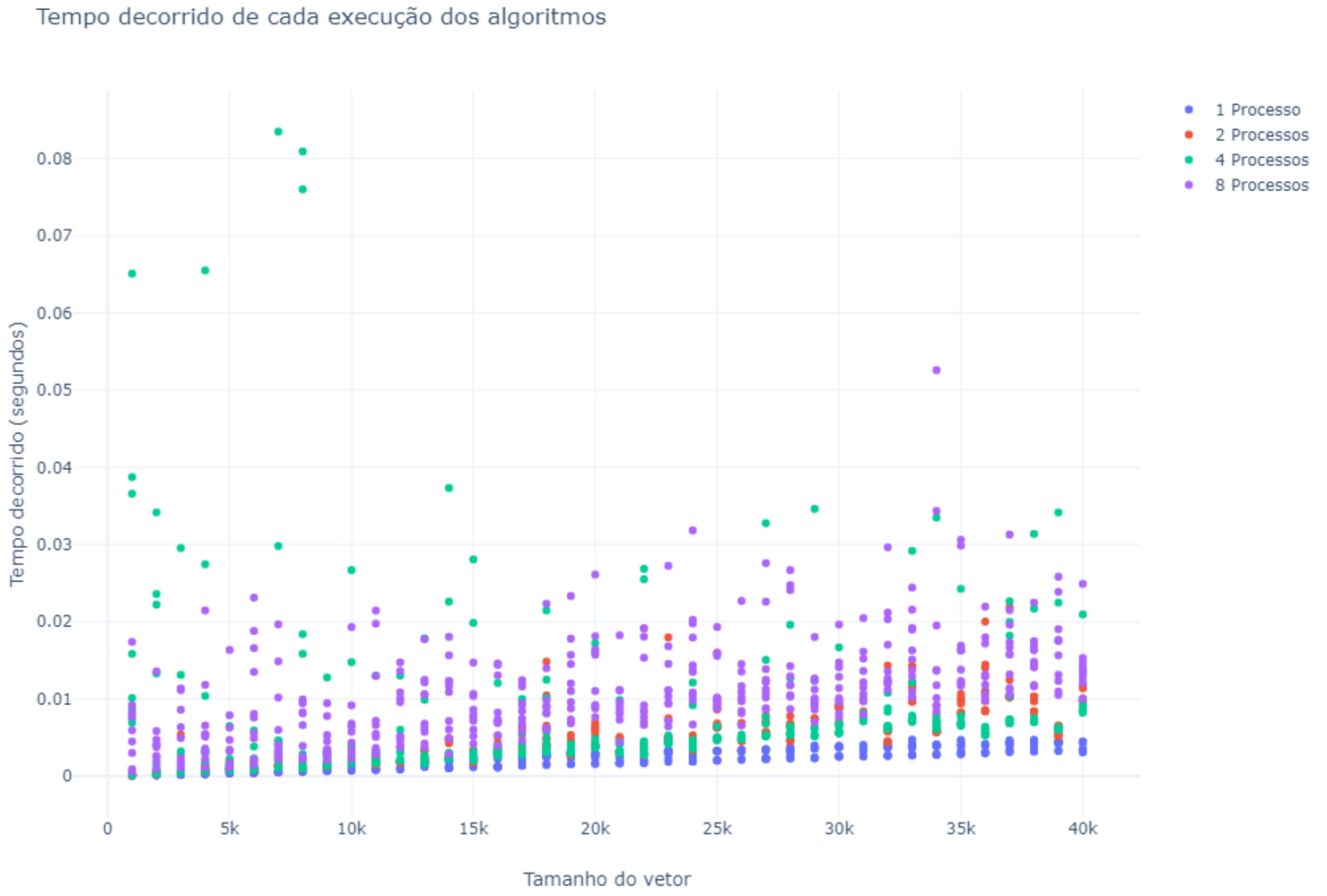
Há como modificar o algoritmo para encerrar a execução prematuramente ao identificar que o vetor está ordenado, o que levaria a uma complexidade em melhor caso de **O(n)**.



Resultados

O algoritmo com apenas **1** processo se comporta como o quicksort. Com mais processos, o *overhead* de comunicação apenas piora o desempenho.

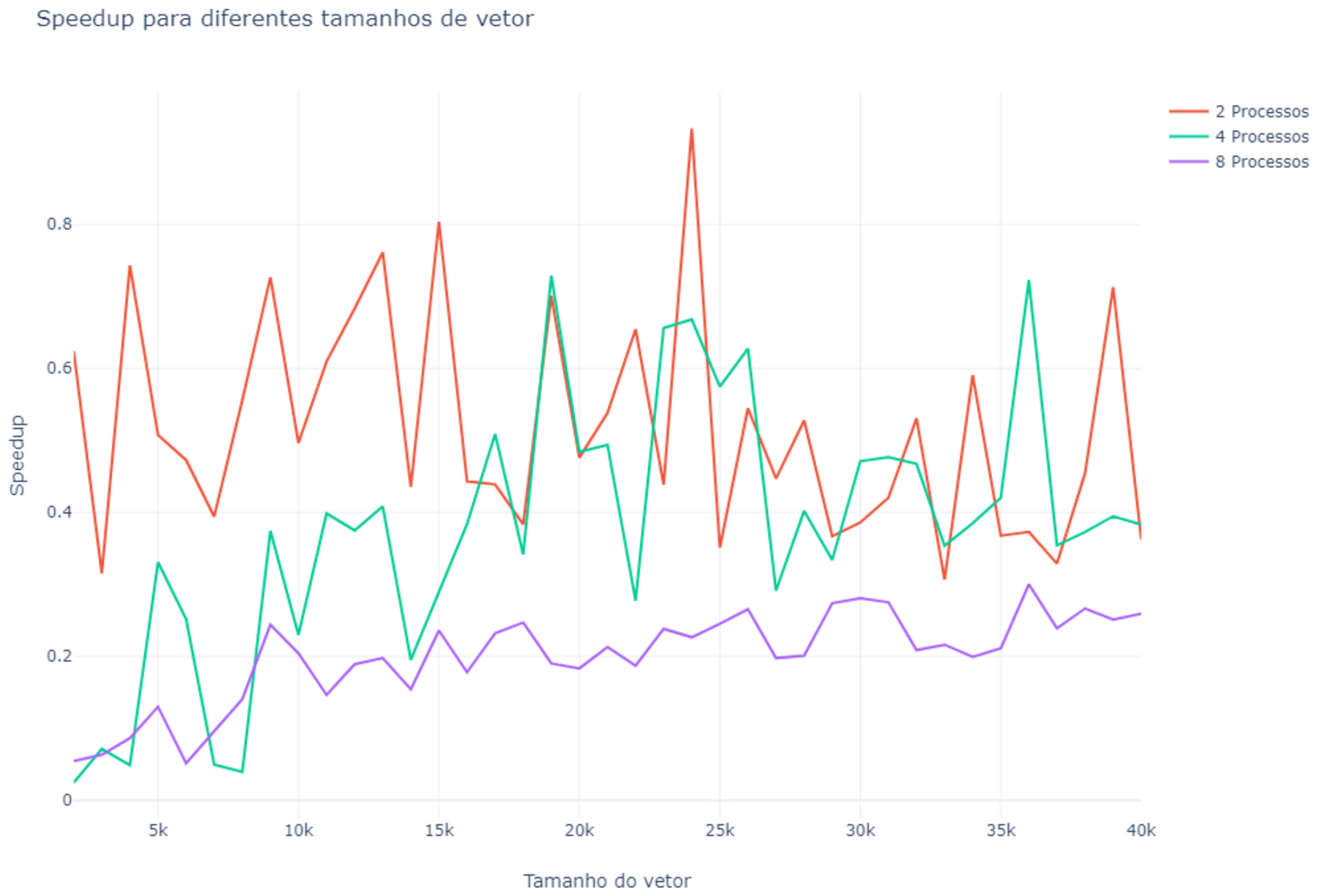
É importante notar que todos estes processos se encontram na mesma máquina física, e que este *overhead* seria maior ainda se fosse utilizado um canal de comunicação por redes, por exemplo.



Resultados

Para cada tamanho de vetor foi calculado o ***speedup*** como a razão entre a média do tempo decorrido do algoritmo **sequencial (1 processo)** pela média do tempo decorrido do algoritmo **paralelo**, ou:

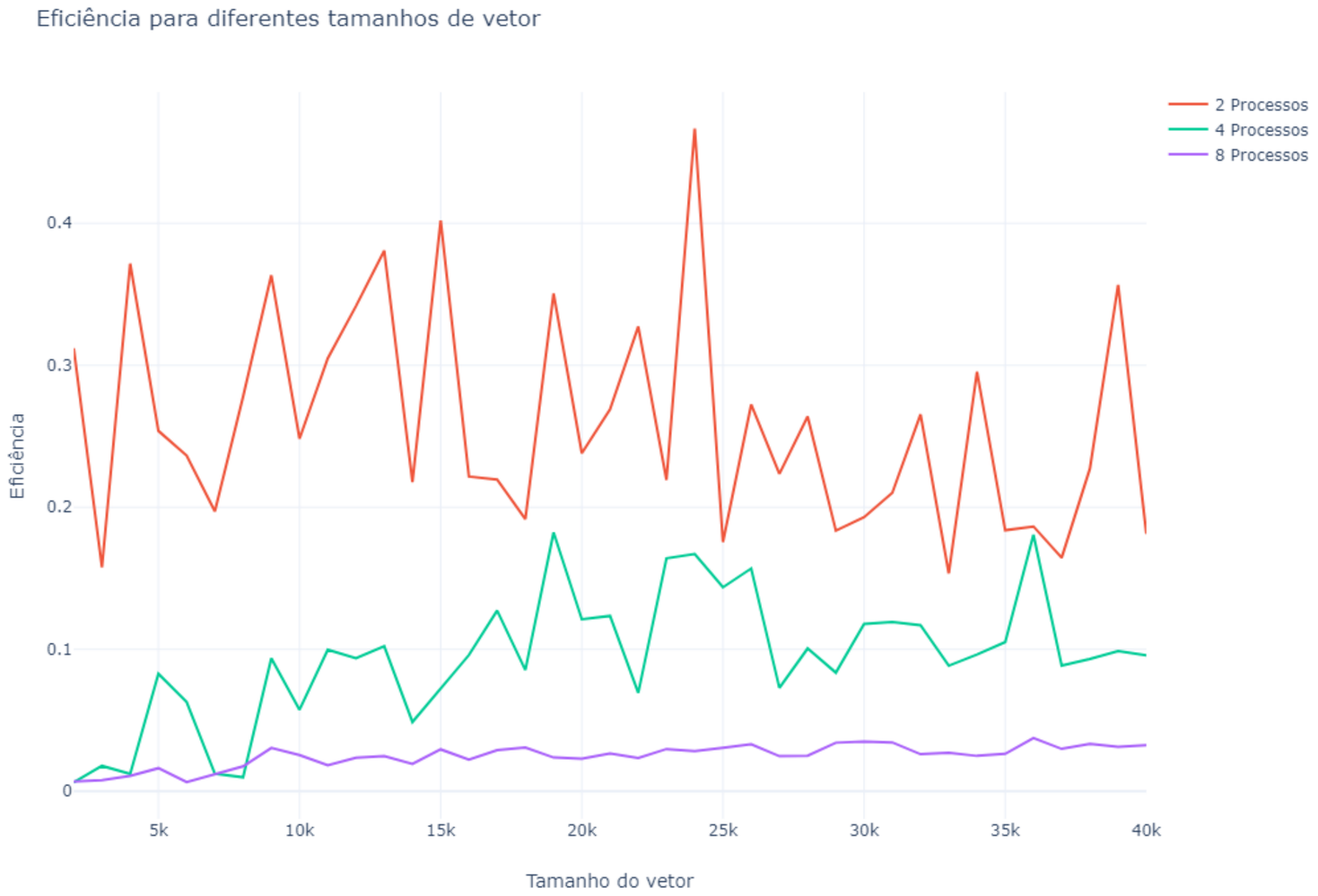
$$\frac{t_s}{t_p}$$



Resultados

Para cada tamanho de vetor foi calculado a **eficiência** como a razão entre a média do tempo decorrido do algoritmo **sequencial (1 processo)** pela média do tempo decorrido do algoritmo **paralelo** vezes o número de **processos**, ou:

$$\frac{t_s}{t_p \times n_p}$$



Código Fonte

GitHub

Referências

1	pp10.pdf
2	en.wikipedia.org/wiki/Odd-even_sort
3	linux.die.net/man/3/clock_gettime