



**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE**  
**INSTITUTO METRÓPOLE DIGITAL**

**Disciplina:** Tópicos Especiais em Internet das Coisas “B”

**Aluno:** Israel Medeiros Fontes

**Professor:** Kayo Gonçalves e Silva

**Relatório**  
**Quicksort Utilizando OpenMP**

Natal/RN

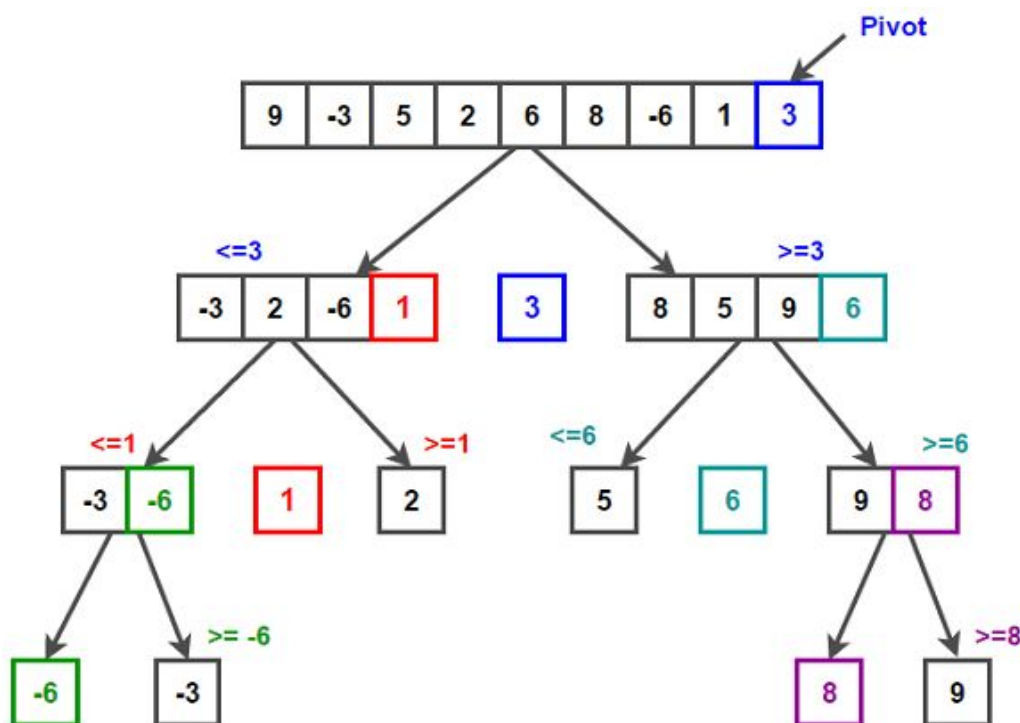
2020

## SUMÁRIO

<b>1.</b>	<b>INTRODUÇÃO</b>	<b>3</b>
<b>2.</b>	<b>METODOLOGIA</b>	<b>5</b>
<b>3.</b>	<b>DESENVOLVIMENTO</b>	<b>6</b>
<b>3.1.</b>	<b>IMPLEMENTAÇÃO</b>	<b>6</b>
<b>3.2.</b>	<b>CORRETUDE</b>	<b>8</b>
<b>3.3.</b>	<b>RESULTADOS</b>	<b>9</b>
<b>3.4.</b>	<b>SPEEDUP, EFICIÊNCIA E ESCALABILIDADE</b>	<b>12</b>
<b>4.</b>	<b>CONCLUSÃO</b>	<b>14</b>
<b>5.</b>	<b>REFERÊNCIAS</b>	<b>15</b>

## 1. INTRODUÇÃO

O Quicksort é um dos algoritmos de ordenação mais famosos na computação. Ele ficou conhecido por sua rapidez, eficiência e simplicidade. Ele adota a estratégia “dividir para conquistar”, de modo que dado um vetor de entrada é escolhido um pivô, o algoritmo deve realocar todos os elementos menores que o pivô a sua esquerda e os maiores que ele a sua direita. Os elementos a esquerda e a direita não necessariamente estarão ordenados entre si, porém o pivô estará ordenado, ou seja, na posição final em que ele deve estar. É feito o mesmo processo nos sub-vetores da esquerda e da direita do pivô inicial recursivamente até que se tenha o vetor completamente ordenado ao final do algoritmo.



**Figura 1:** Exemplo de vetor em processo de ordenação utilizando o Quicksort.

Referência: <https://deepai.org/machine-learning-glossary-and-terms/quicksort-algorithm>

Na Figura 1 podemos observar um vetor de nove elementos sendo ordenado pelo algoritmo Quicksort. Num primeiro momento é escolhido o último elemento do vetor como pivô, o número 3, em seguida todos os elementos menores do que ele são transferidos para sua esquerda e os maiores para sua direita. Ao final desse processo, o 3 estará na posição em que deve estar no final da ordenação. Já na segunda fase, o algoritmo realiza o mesmo processo só que agora com o sub-vetor da esquerda e da direita. Essa recursão acontece até que o tamanho do sub-vetor seja um, pois assim já teremos um vetor ordenado por si só.

Este trabalho tem como objetivo a implementação do algoritmo de ordenação Quicksort utilizando métodos de computação paralela com o intuito de tornar tal ordenação mais rápida. Para isso iremos explorar técnicas de computação paralela de memória compartilhada, mais precisamente utilizando a biblioteca *Open Multi-Processing* (OpenMP). Após a implementação será feita análises comparativas entre o paradigma serial e o paralelo de tal algoritmo.

## 2. METODOLOGIA

Foram implementados dois algoritmos, um no paradigma serial e outro no paradigma paralelo com memória compartilhada. Em tal implementação, utilizei a linguagem de programação C que possui suporte a biblioteca OpenMP. Após a implementação dos algoritmos, foi realizada uma bateria de testes coordenados por scripts escritos em *Shell Script* para analisar o tempo de execução dos dois algoritmos.

Os testes nos paradigmas serial e paralelo foram realizados no Supercomputador da UFRN mantido pelo *Núcleo de Processamento de Alto Desempenho* (NPAD) e se deram da seguinte maneira: primeiro, foram realizadas 4 execuções do código serial de modo que cada execução referiu-se a um tamanho de problema diferente, ou seja, a uma quantidade de números a serem gerados pseudo-aleatoriamente e ordenados pelo método Quicksort. Da mesma forma será feito com o algoritmo paralelo, só que agora, para cada tamanho de problema, o código foi executado com 4, 8, 16 e 32 threads e em cada execução foi contabilizado o tempo. Para que houvesse uma precisão maior, essa bateria de testes foi realizada 10 vezes e foi feita uma média aritmética dos tempos contabilizados.

Ao final será feita as análises de corretude, speedup, eficiência e escalabilidade do algoritmo paralelo em comparação com o serial.

### 3. DESENVOLVIMENTO

#### 3.1 IMPLEMENTAÇÃO

Na implementação do algoritmo serial, foi utilizado o pseudocódigo a seguir.

```
//Troca os valores de posição em um vetor
swap( array[], a, b ):
    tmp ← array[a]
    array[a] ← array[b]
    array[b] ← tmp

//Particiona o vetor com o pivô ordenado
partition( array[], start_array, end_array ):
    pivot ← array[end_array]
    small_pivot ← start_array
    more_pivot ← start_array

    for more_pivot < end_array do:
        if array[more_pivot] <= pivot then:
            swap( array, more_pivot, small_pivot )
            small_pivot++
        more_pivot++
    return small_pivot

quicksort_sequential( array[], start_array, end_array ):
    if start_array < end_array then:
        pivot ← partition( array, start_array, end_array )
        quicksort_sequential( array, start_array, pivot-1 )
        quicksort_sequential( array, pivot+1, end_array )

main( number_elements ):
    array ← generate_random_numbers( number_elements )
    quicksort_sequential( array, array.start, array.end )
```

**Código 1:** Pseudocódigo do algoritmo Quicksort sequencial.

O Código 1 apresenta uma implementação simples do algoritmo Quicksort sequencial. A função *quicksort\_sequential* recebe como parâmetros o array a ser ordenado, a posição inicial e a final desse vetor. Em seguida é verificada a condição de parada da recursão, que acontece quando a posição inicial do array é igual a final, ou seja, quando o sub-vetor possuir tamanho unitário. Caso ainda existam sub-vetores a serem ordenados, a função *partition* recebe o sub-vetor, seleciona seu pivô e realiza o processo de particionamento transferindo

todos os valores menores que o pivô para esquerda e os maiores para a direita. Ao final, a função *partition* retorna a posição final a qual do pivô.

```
//Troca os valores de posição em um vetor
swap( array[], a, b ):
    tmp ← array[a]
    array[a] ← array[b]
    array[b] ← tmp

//Particiona o vetor com o pivô ordenado
partition( array[], start_array, end_array ):
    pivot ← array[end_array]
    small_pivot ← start_array
    more_pivot ← start_array

    for more_pivot < end_array do:
        if array[more_pivot] <= pivot then:
            swap( array, more_pivot, small_pivot )
            small_pivot++
            more_pivot++
    return small_pivot

quicksort_parallel( array[], start_array, end_array ):
    if start_array < end_array then:
        pivot ← partition( array, start_array, end_array )

        #pragma omp task shared(array) firstprivate(start_array, end_array)
        {
            quicksort_sequential( array, start_array, pivot-1 )
        }
        #pragma omp task shared(array) firstprivate(start_array, end_array)
        {
            quicksort_sequential( array, pivot+1, end_array )
        }
        #pragma omp taskwait

main( n_elements, n_threads ):
    array ← read_numbers( n_elements )
    #pragma omp parallel default(none) shared(array) num_threads(n_threads)
    {
        #pragma omp single nowait
        {
            quicksort_parallel( array, array.start, array.end )
        }
    }
```

**Código 2:** Pseudocódigo do algoritmo Quicksort paralelo com OpenMP.

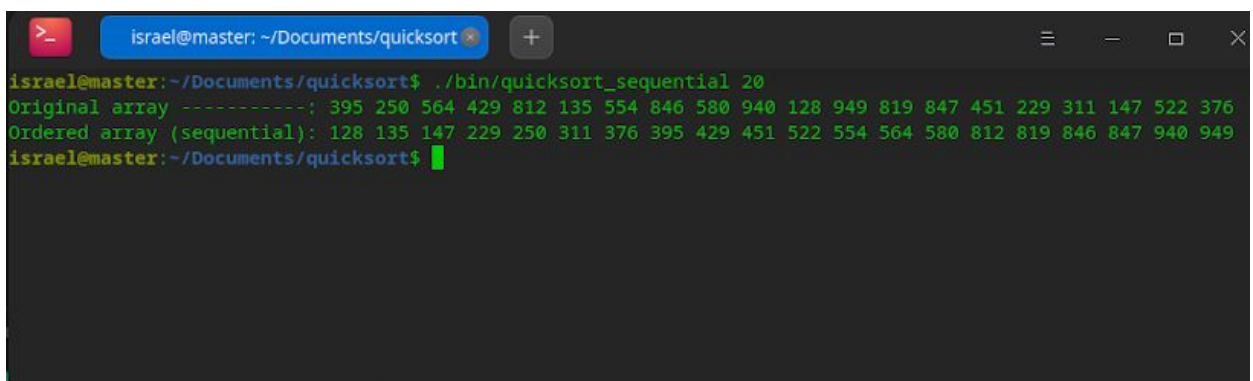
Já o Código 2 nos traz um esboço do código paralelo com memória compartilhada escrito fazendo uso da biblioteca OpenMP. O paradigma é diferente apesar do código se assemelhar ao paradigma sequencial. Na resolução desse problema, foi explorado o princípio de *tasks*, ou no português literal, tarefas. Com as *tasks* é possível gerar tarefas a serem executadas por várias *threads* simultaneamente, possibilitando o paralelismo que necessitamos para o objetivo deste trabalho.

Analisando um pouco melhor o Código 2 percebemos, na função *main()* que dá início ao algoritmo, a leitura do vetor que foi gerado pelo algoritmo sequencial e em seguida dá-se início a região paralela, compartilhando entre todas as *threads* o vetor a ser ordenado. Apenas uma *thread* chama a função *quicksort\_parallel()*, por consequência do uso da tag *single*, e todas as outras *threads* seguem disponíveis para execução das próximas instruções dado o uso da tag *nowait*.

Na função *quicksort\_parallel()* temos algo muito semelhante ao algoritmo sequencial, entretanto é nela que dá-se início ao paralelismo de fato. Podemos observar, na função *quicksort\_parallel()*, a criação de *tasks* recursivamente que vão sendo executadas pelas *threads* disponíveis ao finalizarem as *tasks* anteriores. Desse modo, cada *thread* será utilizada para ordenar um conjunto de sub-vetores.

### 3.2 CORRETUDE

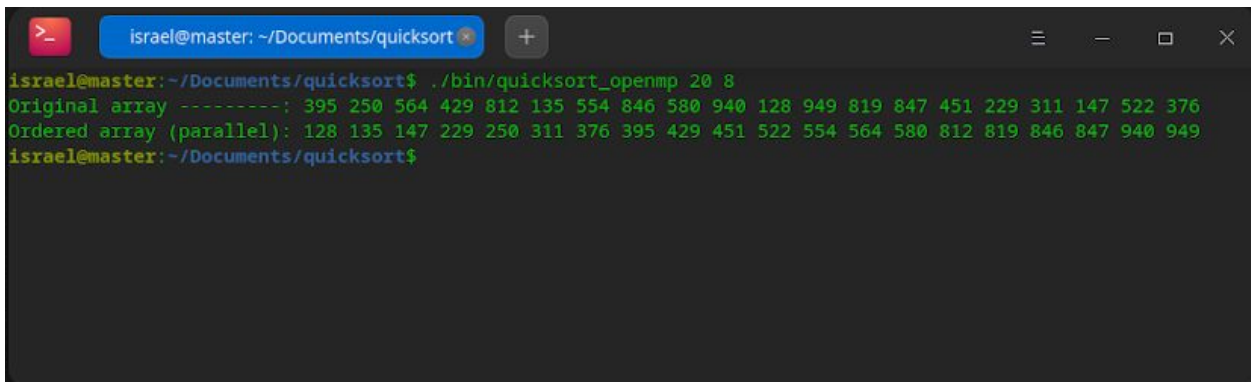
Dada a implementação dos dois algoritmos, foi analisada a corretude dos mesmos com o tamanho de problema reduzido a fim de descobrir se as duas implementações entregam o resultado esperado, ou seja, ordenam os vetores de fato.



```
israel@master: ~/Documents/quicksort
israel@master:~/Documents/quicksort$ ./bin/quicksort_sequential 20
Original array -----: 395 250 564 429 812 135 554 846 580 940 128 949 819 847 451 229 311 147 522 376
Ordered array (sequential): 128 135 147 229 250 311 376 395 429 451 522 554 564 580 812 819 846 847 940 949
israel@master:~/Documents/quicksort$
```

**Figura 2:** Corretude do algoritmo sequencial.





```
israel@master: ~/Documents/quicksort
israel@master:~/Documents/quicksort$ ./bin/quicksort_openmp 20 8
Original array -----: 395 250 564 429 812 135 554 846 580 940 128 949 819 847 451 229 311 147 522 376
Ordered array (parallel): 128 135 147 229 250 311 376 395 429 451 522 554 564 580 812 819 846 847 940 949
israel@master:~/Documents/quicksort$
```

**Figura 3:** Corretude do algoritmo paralelo.

As Figuras 2 e 3 apresentam a execução dos dois algoritmos com tamanho de problema 20. A Figura 2 traz o resultado da execução do algoritmo sequencial, o resultado é o vetor ordenado, como se era esperado. Já a Figura 3 apresenta o resultado da execução do algoritmo paralelo dado o mesmo vetor criado pelo sequencial, utilizando 8 *threads*. Aqui também temos o vetor ordenado como era o esperado.

### 3.3 RESULTADOS

Depois de sabido que os algoritmos entregam o resultado esperado, foram realizados os testes de comparação de tempo de execução descritos na metodologia. Para cada paradigma foi realizada uma bateria de testes.

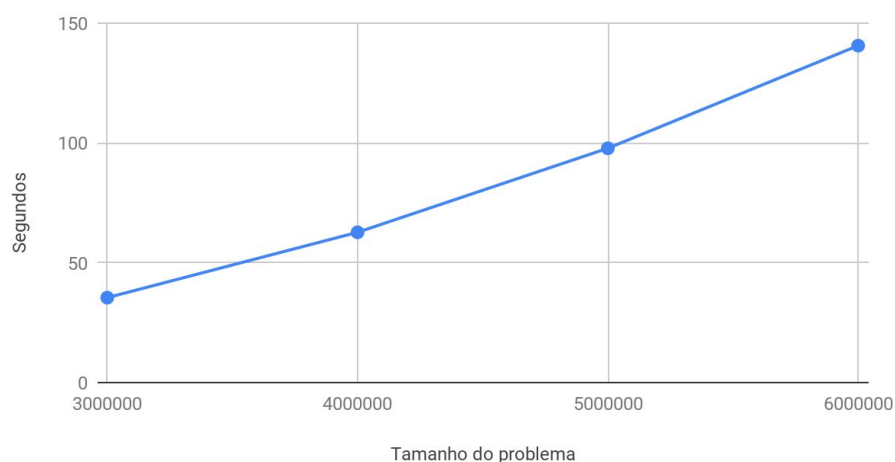
Todos os testes descritos a partir de agora foram executados no supercomputador do NPAD.

Tamanho do problema	Tempo de execução (s)
3000000	35.43
4000000	62.70
5000000	97.77
6000000	140.60

**Tabela 1:** Tempo de execução da implementação sequencial.

Dado tamanhos de problemas diferentes, ou seja, tamanhos de vetores diferentes, foi contabilizado o tempo de execução conforme a Tabela 1.

### Tempo de execução serial



**Gráfico 1:** Tempo de execução da implementação sequencial.

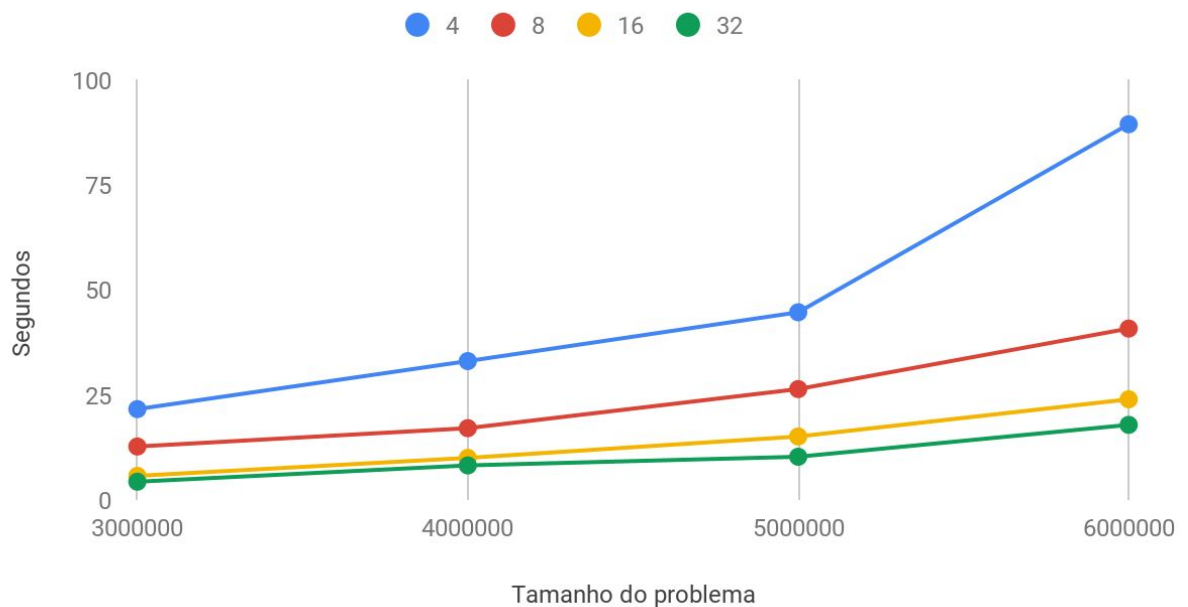
Com o Gráfico 1, conseguimos perceber que o algoritmo sequencial tem um tempo de execução próximo do linear, de acordo com a mudança do tamanho do problema.

Threads	Tamanho do problema	Tempo de execução (s)
4	3000000	21.63
4	4000000	33.06
4	5000000	44.68
4	6000000	89.43
8	3000000	12.73
8	4000000	17.09
8	5000000	26.40
8	6000000	40.80
16	3000000	5.74
16	4000000	10.01
16	5000000	15.11
16	6000000	23.96
32	3000000	4.35
32	4000000	8.23
32	5000000	10.29
32	6000000	17.88

**Tabela 2:** Tempo de execução da implementação paralela.

Da mesma maneira foi analisado o tempo de execução na implementação paralela, com os mesmos vetores da implementação sequencial, mas agora é verificada com 4, 8, 16 e 32 *threads* simultâneas conforme a Tabela 2.

### Tempo de execução OpenMP



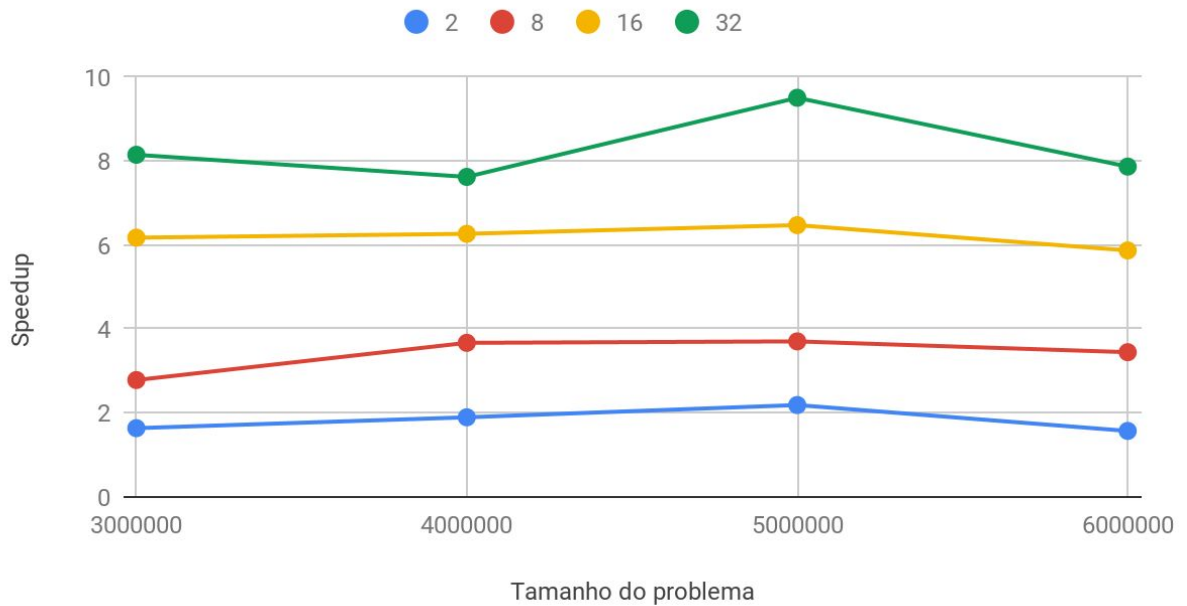
**Gráfico 2:** Tempo de execução da implementação paralela.

Como podemos perceber no Gráfico 2, o tempo de execução para a implementação em paralelo utilizando OpenMP também segue próximo a linear para as quatro quantidades de *threads* diferentes. No entanto, é perceptível um aumento mais acentuado na execução com 4 *threads* com tamanho do vetor igual a 6000000 se comparado com o tamanho de problema igual a 5000000 com a mesma quantidade de *threads*. Esse comportamento atípico não será objeto de estudo neste trabalho.

As execuções com 8, 16 e 32 *threads* se comportaram de forma semelhante, mas logo fica perceptível que ao duplicar a quantidade de *threads* o tempo não diminui pela metade.

### 3.4 SPEEDUP, EFICIÊNCIA E ESCALABILIDADE

#### Speedup



**Gráfico 3:** Speedup da implementação paralela.

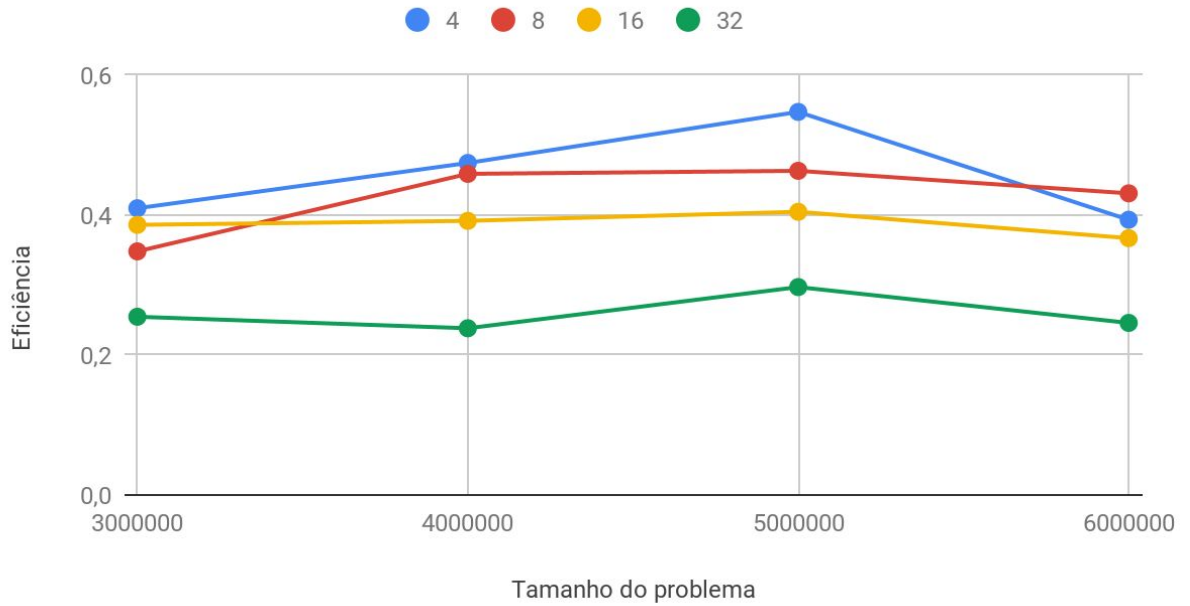
O Speedup é um índice que traduz quantas vezes um algoritmo paralelo é mais rápido do que sua implementação sequencial. Ele é calculado da seguinte maneira:

$$S = \frac{T_s}{T_p}$$

onde  $S$  é o Speedup,  $T_s$  é o tempo da execução serial e  $T_p$  é o tempo da execução em paralelo.

O Gráfico 3 apresenta os Speedups para cada execução realizada referente ao tamanho do problema e a quantidade de *threads*. É perceptível que o Speedup é alto, logo o código paralelo consegue ser até 9x mais rápido do que o serial.

## Eficiência



**Gráfico 4:** Eficiência da implementação em paralelo.

O Gráfico 4 traz a eficiência da implementação paralela que é calculada da seguinte forma:

$$E_f = \frac{S}{m}$$

onde  $E_f$  é a eficiência  $S$  é o Speedup e  $m$  é a quantidade de núcleos de processamento.

Percebemos que o algoritmo em paralelo é fracamente escalável, dado que ao aumentar o número de *threads* para um mesmo tamanho de problema, a eficiência diminui.

#### 4. CONCLUSÃO

Foi implementado os paradigmas sequencial e paralelo com a biblioteca OpenMP do algoritmo Quicksort. Os testes foram realizados com êxito utilizando o supercomputador da UFRN. Os resultados da comparação entre os dois paradigmas trouxeram uma certa dualidade, pois apesar do algoritmo paralelo se mostrar mais rápido do que o sequencial a eficiência cai quando o número de *threads* aumenta.

Em um próximo trabalho será analisado o motivo que levou ao comportamento atípico na execução do algoritmo paralelo com 4 *threads* e tamanho de problema 6000000. Também poderá ser estudado formas de explorar melhor o paralelismo neste algoritmo.

## 5. REFERÊNCIAS

WIKIPEDIA. **Quicksort**. Disponível em: <https://pt.wikipedia.org/wiki/Quicksort>. Acesso em: 14 dez. 2020;

WIKIPEDIA. **Lei de Amdahl**. Disponível em: [https://pt.wikipedia.org/wiki/Lei\\_de\\_Amdahl](https://pt.wikipedia.org/wiki/Lei_de_Amdahl). Acesso em: 14 dez. 2020.