UNIVERSIDADE FEDERAL DE LAVRAS
Departamento de Computação Aplicada

Atividade de análise de desempenho

Código e Nome: GCC177 – Programação Paralela e Concorrente
Nome do Aluno: Israel Santos Vieira

Ambiente de teste

Processador: AMD Ryzen 9 3900X @ 3.8Ghz
Número de núcleos de processamento: 12
Número de threads: 24

Introdução

Os testes que serão apresentados a seguir tem o propósito de monitorar e obter os resultados relativos à performance de um sistema, com o uso do algoritmo **Crivo de Erathostenes** – implementado em Java.

Resultados obtidos

A priori, foram feitos os seguintes testes de performance: **Teste de Speedup**, **de eficiência** e **da fração sequencial determinada experimentalmente** (métrica de Karp-Flatt). Cada um dos testes foram executados **10 vezes**, com as mesmas entradas, visando obter uma média precisa dos resultados – visto que fatores diversos, como o próprio sistema operacional e programas em execução, podem alterar os valores obtidos. Nesse contexto, foram realizadas as execuções com quantidades de threads diferentes **(T=1, T=4, T=8, T=12)**, em que cada thread foi executada com os seguintes valores de entrada: **n = 200k, n = 500k, n = 1 milhão e n = 2 milhões**. Após isso, para contabilizar e comparar os valores, a cada entrada diferente foram calculadas a média do tempo de execução e o desvio padrão x quantidade de threads, como apresentado nos gráficos a seguir.

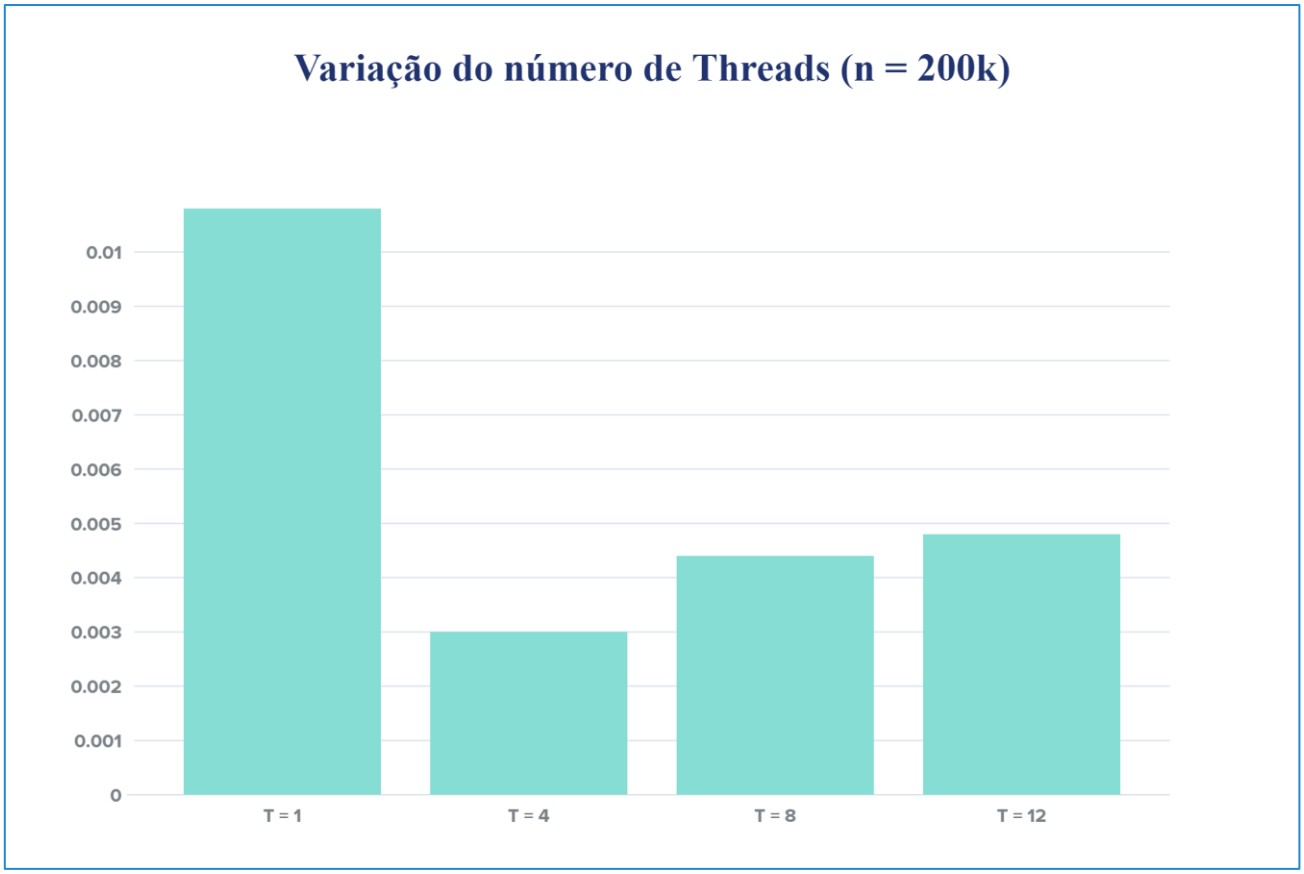

Figura 1 *- Média do tempo de execução com 200 mil valores.*

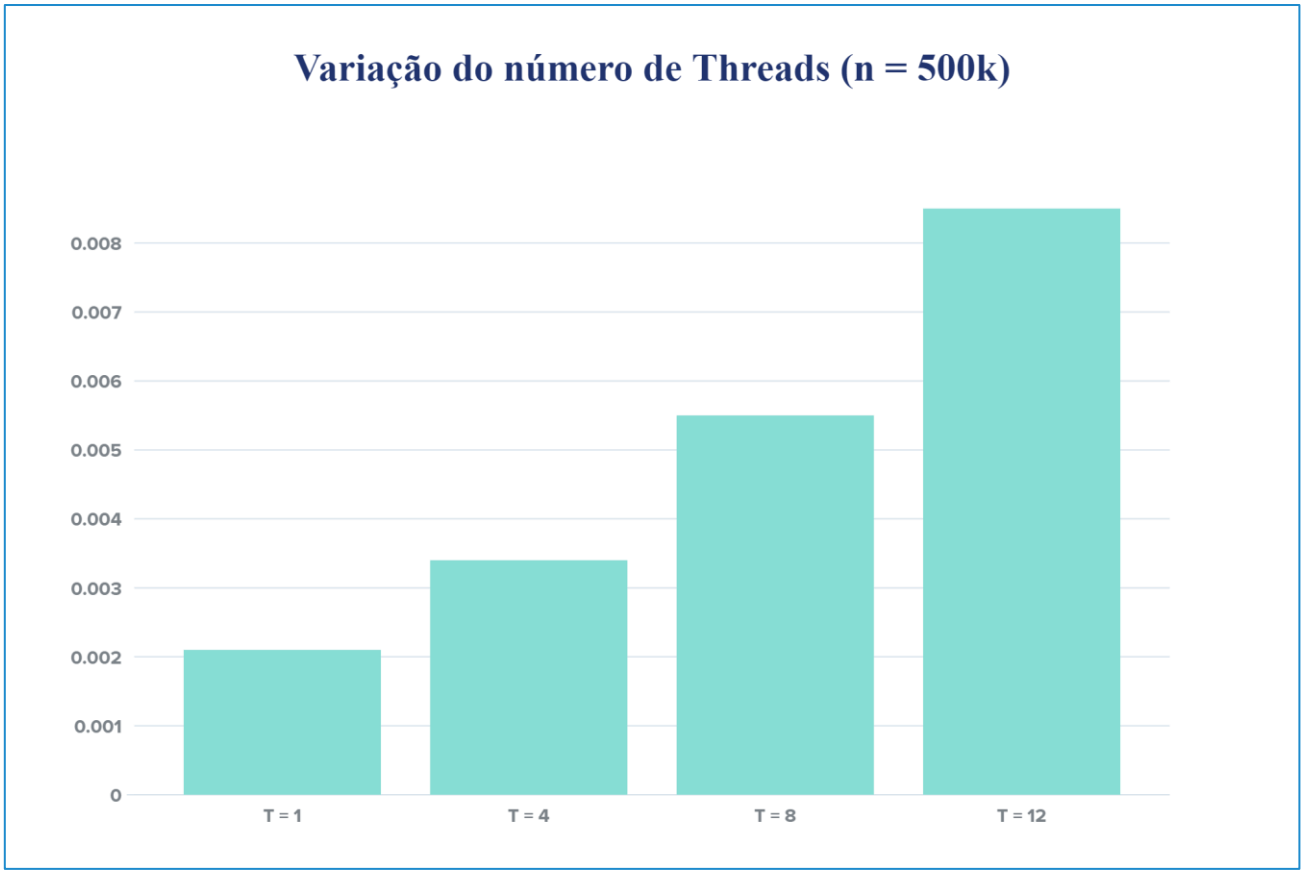

Figura 1.2 *- Média do tempo de execução com 500 mil valores.*

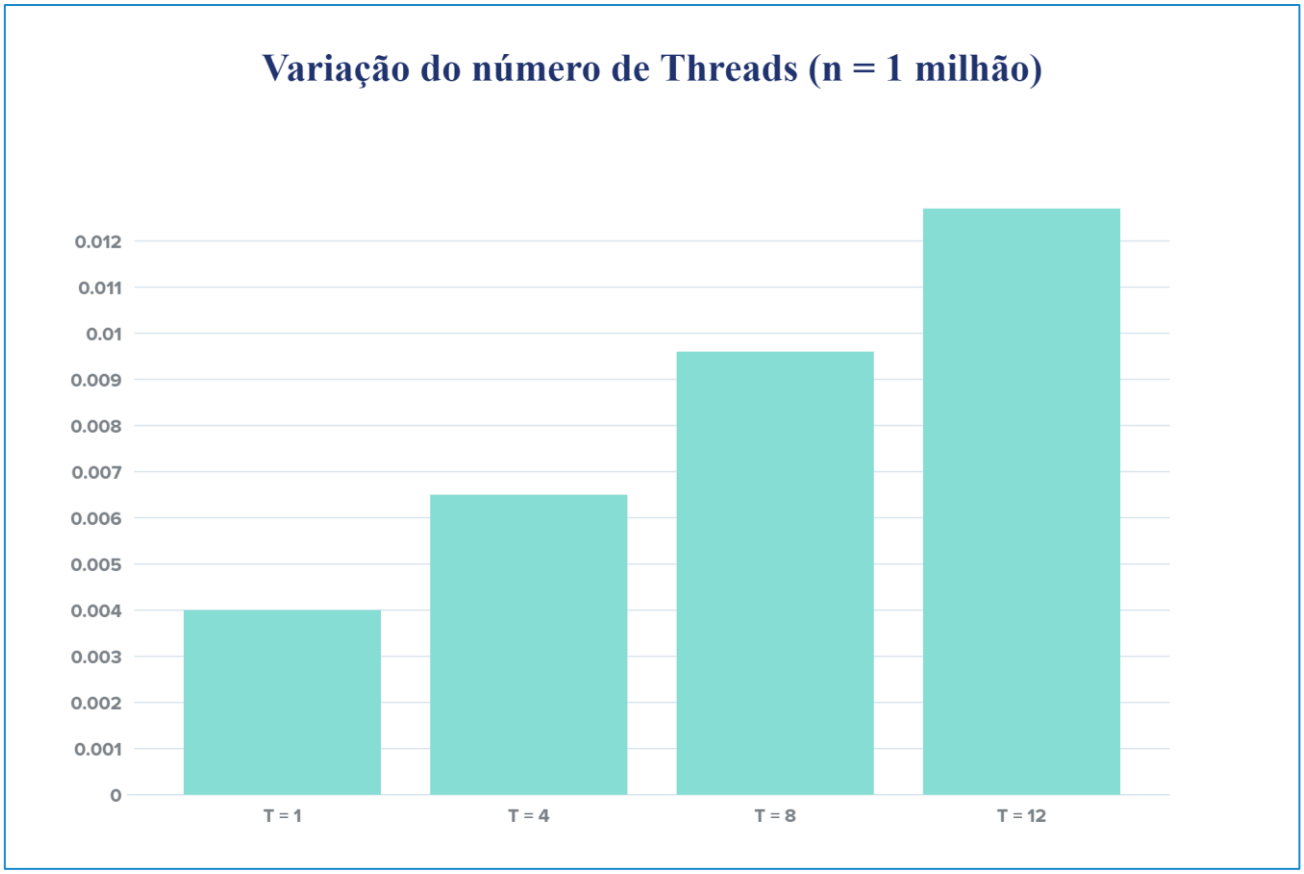

Figura 1.3 *- Média do tempo de execução com 1 milhão de valores.*

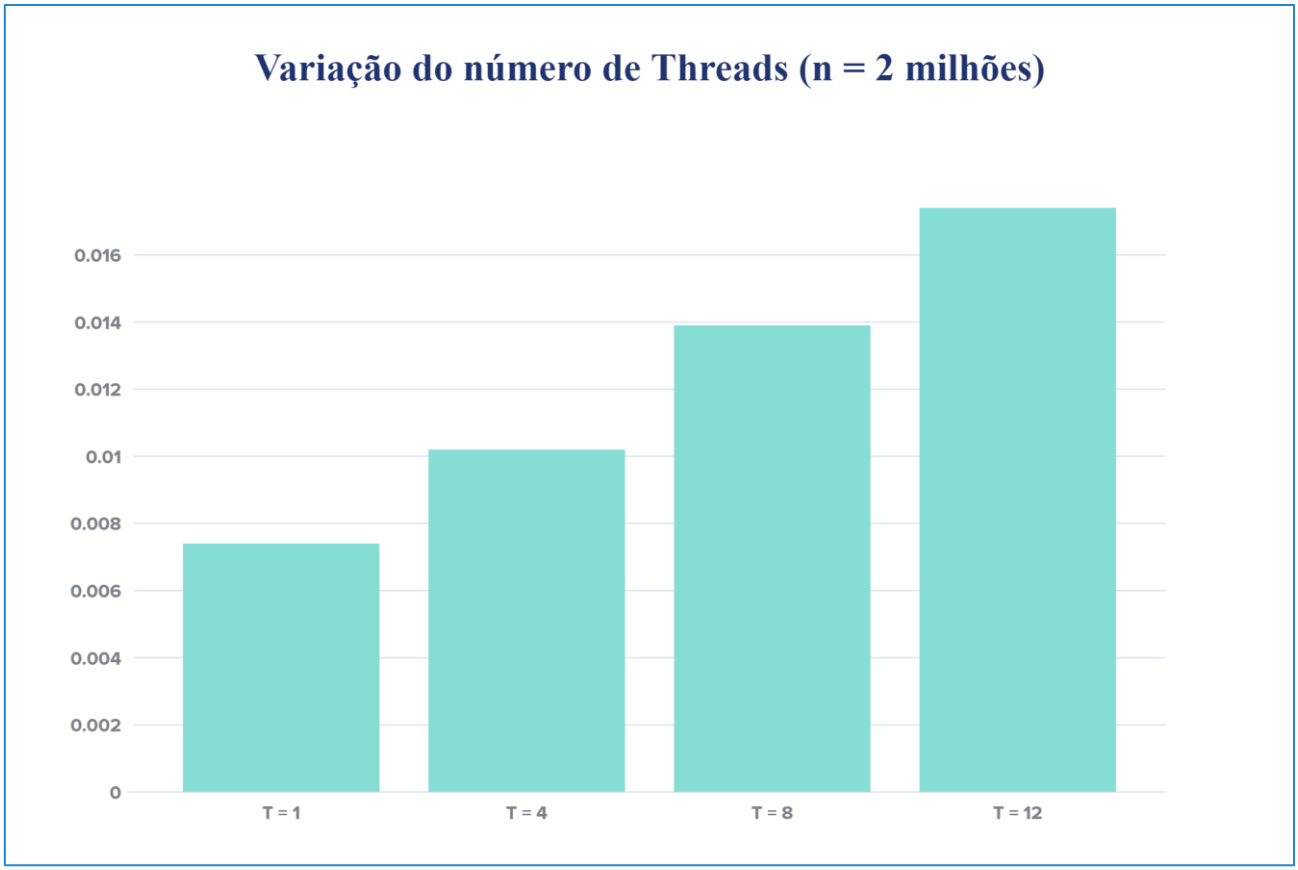

Figura 1.4 *- Média do tempo de execução com 2 milhões de valores.*

T x N	200 mil	500 mil	1 milhão	2 milhões
1	0,0094	0,00005	0,00009	0,0001
4	0,0003	0,0001	0,00009	0,0001
8	0,0013	0,0002	0,00009	0,0002
12	0,0003	0,0001	0,0002	0,0001

Tabela 1 *– Desvio padrão considerando threads x número de entradas.*

Logo, é possível perceber que com 200 mil valores de entrada, o tempo de execução para uma quantidade maior de threads foi melhor, uma vez que para uma única thread executar todas as funções há uma sobrecarga, impactando no tempo de processamento. Ao aumentar a quantidade threads e de entradas, as médias de tempo de execução foram maiores, devido ao aumento do número de computações exigidas de cada thread. Embora exista uma divisão de carga entre as threads, após aumentar o número de entradas o tempo de processamento foi maior. Portanto, é possível que o algoritmo não distruibua devidamente a carga de trabalho, sendo notório esse problema nas métricas abordadas a seguir.

Outrossim, foi calculado o speedup (gráfico da Figura 2), a eficiência (gráfico da Figura 3) e a fração sequencial definida experimentalmente (métrica de Karp-Flatt), representando a performance do algoritmo comparando o tempo de processamento sequencial, com o tempo de processamento paralelo. Não foi necessário abordar as métricas para 1 elemento de processamento, pois o valor de Karp-Flatt e de speedup seriam 0 e de eficiência igual a 1, visto que não há tempo de processamento paralelo.

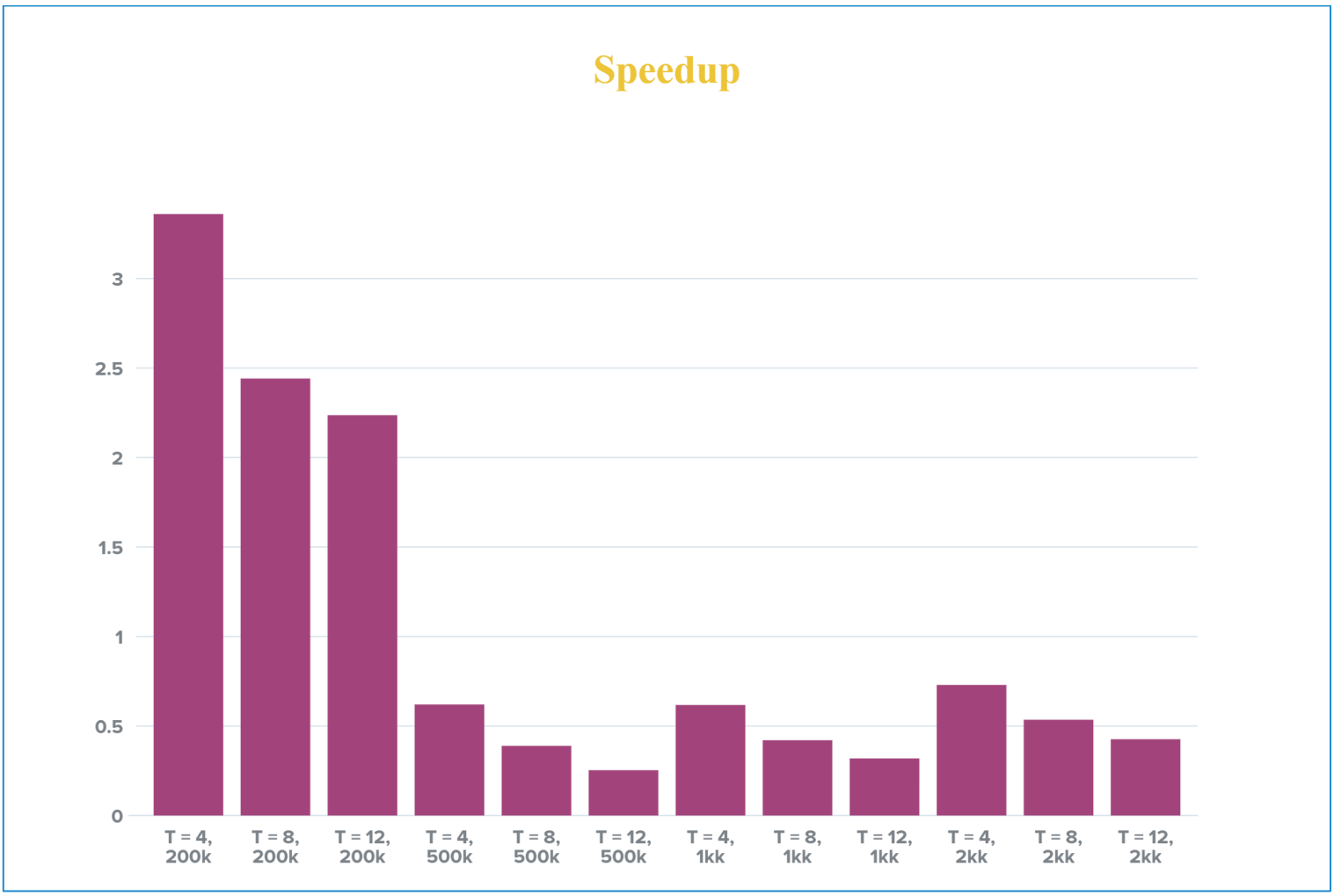

Figura 2 *– Speedup calculado com n de 200k até 2kk de valores.*

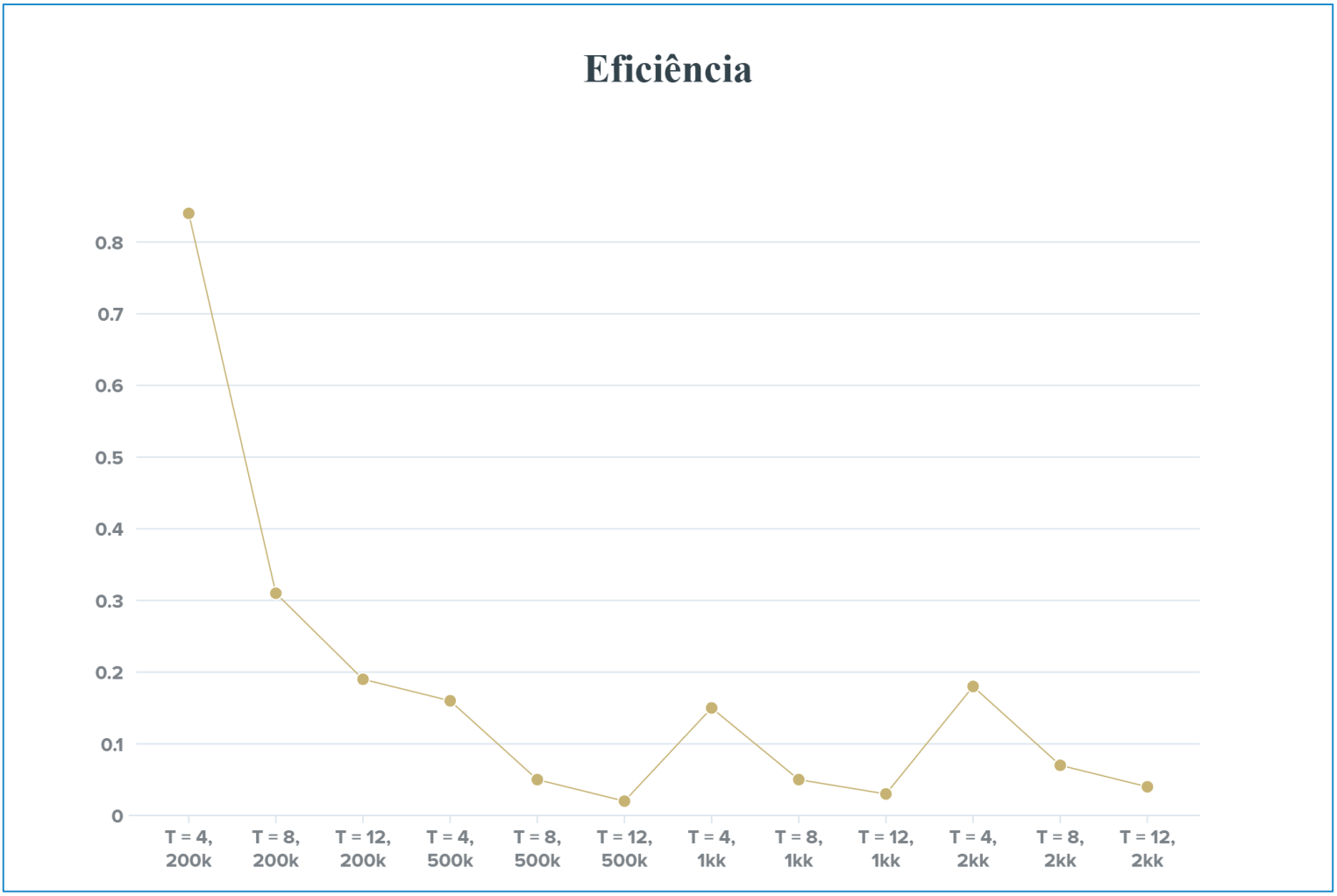

Figura 3 *– Eficiência calculada com n de 200k até 2kk de valores.*

T x N	200 mil	500 mil	1 milhão	2 milhões
4	0,2975	1,6114	1,619	1,3705
8	0,4095	2,5672	2,3763	1,8685
12	0,447	3,9485	3,1346	2,3441

Tabela 2 *– Métrica de Karp-Flatt com threads x quantidade de entradas.*

Conclusão

A partir do gráfico de Speedup, de eficiência e da tabela de Karp-Flatt, é evidente que há uma inconsistência no algoritmo, em que a carga de trabalho não é bem dividida

entre as threads, tornando a paralelização atual do algoritmo inviável, uma vez são obtidos valores ínfimos de eficiência para um número maior de threads. Esse problema é observado quando a entrada de valores é maior, sendo notório que o aumento da quantidade de threads impacta negativamente nos resultados obtidos através da implementação atual do algoritmo. A falha nessa implementação se deve ao fato dela dividir as tarefas repartindo a entrada pelo número de threads – possibilitando resultados não inteiros – e, 'transferindo' a carga do resto da divisão (entrada / quantidade de threads) para a última thread criada. Essa carga final faz com que somente uma thread trabalhe quando existem divisões com restos maiores. É importante ressaltar que, em virtude da limitação da linguagem Java, a qual não permite que o método de execução de uma thread seja executo mais de uma vez, cada thread é criada novamente assim que é solicitado um novo trabalho, impactando negativamente nos resultados. A discussão abordada é evidente nas figuras a seguir:

```
38          var k_square = (int)Math.pow(k, 2);
39          int maxInterval = (m_MaxSize + 1) - k_square;
40
41          int usedThreads = maxInterval >= threadsAmount ? threadsAmount : maxInterval;
42          int subInterval = (int)Math.floor(maxInterval / usedThreads);
43          var currentRest = maxInterval % threadsAmount;
```

Figura 4 *– Divisão do intervalo de carga de trabalho de cada thread.*

```
45          for(int i = 0; i < usedThreads; i++) {
46              int startIndex = i * subInterval + k_square;
47              int endIndex = (i + 1) * (int)subInterval + k_square;
48              if(i == usedThreads - 1) {
49                  endIndex += (int) currentRest;
50              }
51
52              m_Threads[i] = new CrivoThread(this, startIndex, endIndex, k);
53          }
```

Figura 5 *– Definição de index inicial e final de cada nova execução de trabalho de uma thread.*

Por conseguinte, é imprescindível que, para uma boa performance – no que tange à paralelização de um sistema – a carga de trabalho entre as threads seja dividida igualmente, evitando que uma ou mais threads fiquem 'ociosas', ou sobrecarreguem outras threads com uma carga maior.