Trabalho: Paralelização de algoritmos

Mateus Façanha Lima de Souza[1] **Gustavo Xavier Saldanha**[2]

[1]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca - CEFET/RJ

mateus.souza@aluno.cefet-rj.br, gustavo.saldanha@aluno.cefet-rj.br

Abstract. *This article will demonstrate a parallel implementation of known algorithms, which are explained throughout the article, and will display the performance of the parallel implementation, explaining the factors that led to the performance of each.*

Resumo. *Este artigo demonstrará uma implementação paralela de algoritmos conhecidos, sendo estes explicados ao longo do artigo, e fará uma exibição do desempenho da implementação paralela, fazendo explicações sobre os fatores que levaram ao desempenho de cada um.*

1. Algoritmo de Multiplicação Matricial

Definição: A multiplicação matricial é uma operação matemática que recebe duas matrizes como entrada e produz uma terceira matriz, cujo elemento na posição (i, j) é o resultado do produto escalar da i-ésima linha da primeira matriz pela j-ésima coluna da segunda matriz. Para que a multiplicação seja possível, o número de colunas da primeira matriz deve ser igual ao número de linhas da segunda matriz.

Passo a Passo do Algoritmo:

1. **Inicialização:**
   - Dada uma matriz A de dimensões $m \times n$ e uma matriz B de dimensões $n \times p$, o produto $C = A \times B$ resultará em uma matriz C de dimensões $m \times p$.
   - Inicialize uma matriz C de dimensões $m \times p$ com zeros.
2. **Iteração sobre Linhas e Colunas:**
   - Para cada linha i da matriz A (onde i varia de 1 até m):
   - Para cada coluna j da matriz B (onde j varia de 1 até p):
3. **Cálculo do Elemento $C[i][j]$:**
   - Calcule o elemento $C[i][j]$ como a soma dos produtos dos elementos correspondentes da linha i de A e da coluna j de B.
   - Formalmente, $C[i][j] = \sum_{k=1}^{n} A[i][k] \times B[k][j]$, onde k varia de 1 até n.
4. **Conclusão:**
   - Após calcular todos os elementos $C[i][j]$, a matriz C contém o resultado da multiplicação matricial.

Exemplo:

Vamos considerar a multiplicação de duas matrizes:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

A matriz $C = A \times B$ será de dimensão 2×2, e seus elementos são calculados da seguinte forma:

- **Elemento** $C[1][1]$**:**

$$C[1][1] = (1 \times 5) + (2 \times 7) = 5 + 14 = 19$$

- **Elemento** $C[1][2]$**:**

$$C[1][2] = (1 \times 6) + (2 \times 8) = 6 + 16 = 22$$

- **Elemento** $C[2][1]$**:**

$$C[2][1] = (3 \times 5) + (4 \times 7) = 15 + 28 = 43$$

- **Elemento** $C[2][2]$**:**

$$C[2][2] = (3 \times 6) + (4 \times 8) = 18 + 32 = 50$$

Portanto, a matriz resultante C é:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

1.1. Rodadas

Para realizar as rodadas foram usadas respectivamente as entradas de 500 X 500, 1000 X 1000 e 2000 X 2000, como será apresentada na tabela abaixo pelo campo SIZE:

1.2. Tempo de execução do agoritmo

Tempo de execução			
SIZE	1	2	4
500 X 500	0,311	0,175	0,088
1000 X 1000	2,990	1,657	0,796
2000 X 2000	45,233	24,299	12,411

Figure 1. Tempo de execução - Multiplicação Matricial

1.3. Gráficos de colunas Speedup e Eficiência

Speedup			
SIZE	1	2	4
500 X 500	1,0000000	1,7771429	3,5340909
1000 X 1000	1,0000000	1,8044659	3,7562814
2000 X 2000	1,0000000	1,8615169	3,6445895

Figure 2. SpeedUp - Multiplicação Matricial

Eficiência			
SIZE	1	2	4
500 X 500	1	0,89	0,88
1000 X 1000	1	0,90	0,94
2000 X 2000	1	0,93	0,91

Figure 3. Eficiência - Multiplicação Matricial

1.4. Analise sobre os gráficos

Observamos na Multiplicação Matricial que a nossa implementação paralela é muito boa, que resulta em uma ótima eficiência na utilização dos recusos do hardware em relação a demanda, em suma nossa implementação mostra que ao dobrarmos o nosso número de threads dobramos a velocidade de execução e assim sucessivamente.

1.5. Escalabilidade

O algoritmo de multiplicação matricial se mostrou altamente escalável. Conforme fomos aumentando o número de Threads a eficiência continuou muito boa o que significa que pode ser dividido em várias tarefas menores que podem ser executadas simultaneamente e ainda assim ter uma eficiência elevada.

2. Merge Sort

Definição: O Merge Sort é um algoritmo de ordenação eficiente que segue o paradigma ”dividir e conquistar”. Ele divide a lista de elementos em sublistas menores até que cada sublista contenha um único elemento ou nenhum. Em seguida, as sublistas são repetidamente combinadas (merge) de forma ordenada para produzir listas ordenadas maiores, até que toda a lista seja recombinada em uma lista ordenada final.

Passo a Passo do Algoritmo:

1. **Divisão Recursiva:**
   - Se a lista contém apenas um elemento, ela já está ordenada.
   - Se a lista contém mais de um elemento, divida-a em duas sublistas de tamanhos aproximadamente iguais.
2. **Ordenação Recursiva das Sublists:**
   - Recursivamente, aplique o Merge Sort às duas sublistas para ordená-las.
3. **Intercalação (Merge):**
   - Combine (merge) as duas sublistas ordenadas em uma única lista ordenada. Para isso:
     - Compare o primeiro elemento de cada sublista.
     - Adicione o menor elemento à lista resultante e remova-o da sublista original.
     - Repita o processo até que todos os elementos de ambas as sublistas tenham sido adicionados à lista resultante.
   - Após a intercalação, a lista resultante estará ordenada.
4. **Conclusão:**
   - O processo de divisão e intercalação continua recursivamente até que a lista completa esteja ordenada.

Exemplo:

Vamos considerar a ordenação de uma lista utilizando Merge Sort:

Lista inicial: $[38, 27, 43, 3, 9, 82, 10]$

- **Passo 1: Divisão Recursiva**

$$[38, 27, 43, 3] \quad \text{e} \quad [9, 82, 10]$$

- **Passo 2: Divisão Contínua até sublistas unitárias**

$$[38, 27] \quad \text{e} \quad [43, 3]$$

$$[38] \quad [27] \quad [43] \quad [3]$$

- **Passo 3: Intercalação**

$$[27, 38] \quad \text{(a partir de [38] e [27])}$$

$$[3, 43] \quad \text{(a partir de [43] e [3])}$$

$$[3, 27, 38, 43] \quad \text{(a partir de [27, 38] e [3, 43])}$$

- **Passo 4: Repita o processo para a outra metade**

$$[9] \quad [82, 10] \rightarrow [9] \quad [10, 82]$$

$$[9, 10, 82] \quad \text{(a partir de [9] e [10, 82])}$$

- **Passo 5: Intercalação final**

$$[3, 9, 10, 27, 38, 43, 82] \quad \text{(a partir de [3, 27, 38, 43] e [9, 10, 82])}$$

A lista ordenada final é: [3, 9, 10, 27, 38, 43, 82]

2.1. Rodadas

Para realizar as rodadas foram usadas respectivamente as entradas de 100000, 200000, 400000, como será apresentada na tabela abaixo pelo campo SIZE, e 1000 execuções para cada entrada afim de computador um tempo que seja visivel o resultado da paralelização:

2.2. Tempo de execução do agoritmo

Tempo de execução			
SIZE	1	2	4
1000000	5,801	2,953	1,562
2000000	12,104	6,132	3,218
4000000	25,407	12,932	6,689

Figure 4. Tempo de execução - Merge Sort

2.3. Gráficos de colunas Speedup e Eficiência

Speedup			
SIZE	1	2	4
1000000	1,00	1,96	3,71
2000000	1,00	1,97	3,76
4000000	1,00	1,96	3,80

Figure 5. SpeedUp - Merge Sort

Eficiência			
SIZE	1	2	4
1000000	1,00	0,98	0,93
2000000	1,00	0,99	0,94
4000000	1,00	0,98	0,95

Figure 6. Eficiência - Merge Sort

2.4. Analise sobre os gráficos

Observamos na Merge Sort que a nossa implementação paralela é muito boa, que resulta em uma ótima eficiência na utilização dos recusos do hardware em relação a demanda, em suma nossa implementação mostra que ao dobrarmos o nosso número de threads dobramos a velocidade de execução e assim sucessivamente.

2.5. Escalabilidade

A escalabilidade do Merge Sort segue a ideia de ”dividir e conquistar”, onde o problema é dividido em subproblemas menores, resolvidos independentemente e depois combinados. Ao paralelizar o algorítmo, tivemos um resultado de eficiência muito positivo. Isso faz com que ele seja altamente escalável.

3. Algoritmo Selection Sort

Definição: O Selection Sort é um algoritmo de ordenação que funciona dividindo a lista de entrada em duas partes: a sublista ordenada, que é construída da esquerda para a direita, e a sublista não ordenada, que contém os elementos restantes. A cada iteração, o algoritmo encontra o menor elemento da sublista não ordenada e o troca com o primeiro elemento da sublista não ordenada, expandindo assim a sublista ordenada e diminuindo a sublista não ordenada.

Passo a Passo do Algoritmo:

1. **Inicialização:**
   - Comece com a lista completa como a sublista não ordenada e a sublista ordenada vazia.
2. **Iteração sobre a Lista:**

- Para cada posição i na lista (variando de 1 até $n-1$, onde n é o tamanho da lista):
  - Encontre o menor elemento na sublista não ordenada (a partir da posição i até o final da lista).
  - Troque o menor elemento encontrado com o elemento na posição i.
  - O elemento trocado agora faz parte da sublista ordenada.

3. **Conclusão:**
   - Repita o processo até que a sublista não ordenada esteja vazia, o que significa que toda a lista foi ordenada e o algorítmo chegou ao fim.

Exemplo:

Lista inicial: $[64, 25, 12, 22, 11]$

- **Passo 1:**
  - Encontre o menor elemento na lista completa, que é 11.
  - Troque 11 com o primeiro elemento 64.
  - Resultado do 1° passo: $[11, 25, 12, 22, 64]$.
- **Passo 2:**
  - Encontre o menor elemento na sublista não ordenada $[25, 12, 22, 64]$, que é 12.
  - Troque 12 com o segundo elemento 25.
  - Resultado do 2° passo: $[11, 12, 25, 22, 64]$.
- **Passo 3:**
  - Encontre o menor elemento na sublista não ordenada $[25, 22, 64]$, que é 22.
  - Troque 22 com o terceiro elemento 25.
  - Resultado do 3° passo: $[11, 12, 22, 25, 64]$.
- **Passo 4:**
  - Encontre o menor elemento na sublista não ordenada $[25, 64]$, que é 25.
  - Como o menor elemento já está na posição correta, não é necessária a troca.
  - Resultado do 4° passo: $[11, 12, 22, 25, 64]$.
- **Passo 5:**
  - A sublista não ordenada agora tem apenas um elemento $[64]$, que é o maior e já está na posição correta.

Lista final ordenada: $[11, 12, 22, 25, 64]$.

3.1. Rodadas

Para realizar as rodadas foram usadas respectivamente as entradas de 100000, 200000, 400000, como será apresentada na tabela abaixo pelo campo SIZE:

3.2. Tempo de execução do agoritmo

Speedup			
SIZE	1	2	4
1000000	1,00	1,36	2,31
2000000	1,00	1,35	2,27
4000000	1,00	1,34	2,45

Figure 7. Tempo de execução - Selection Sort

3.3. Gráficos de colunas Speedup e Eficiência

Speedup			
SIZE	1	2	4
1000000	1,00	1,36	2,31
2000000	1,00	1,35	2,27
4000000	1,00	1,34	2,45

Figure 8. SpeedUp - Selection Sort

Eficiência			
SIZE	1	2	4
1000000	1,00	0,68	0,58
2000000	1,00	0,67	0,57
4000000	1,00	0,67	0,61

Figure 9. Eficiência - Selection Sort

3.4. Analise sobre os gráficos

Observamos na Selection Sort que a nossa implementação paralela é mais fraca, pois a medida que utilizamos mais recursos para a execução paralela a nossa eficiência cai, logo mesmo que utilizemos mais recursos para o processamento, o nosso ganho não será significativo.

3.5. Escalabilidade

Este algoritmo não é ideal para listas grandes devido à necessidade de comparar cada elemento com todos os outros restantes em cada iteração. Sua escalabilidade é limitada por essa natureza quadrática. A paralelização do Selection Sort é limitada, pois cada iteração depende dos resultados das iterações anteriores. Isso foi visto nos resultados de eficiência, quanto maior o valor de entrada, pior a eficiência do algoritmo.

4. Closest Pair of Points Problem

Definição: O problema do *Closest Pair of Points* (par de pontos mais próximos) consiste em encontrar dois pontos em um conjunto de pontos no plano euclidiano que estão mais próximos um do outro. A abordagem mais eficiente para resolver este problema utiliza a técnica de ”dividir e conquistar”.

Passo a Passo do Algoritmo:

1. **Ordenação Inicial:**
   - Ordene todos os pontos por suas coordenadas x. Esta ordenação inicial permitirá dividir o conjunto de pontos de maneira eficiente.
2. **Divisão Recursiva:**
   - Divida o conjunto de pontos ordenados em dois subconjuntos de tamanhos aproximadamente iguais pela linha vertical que passa pelo ponto mediano (meio) da ordenação x.
3. **Resolução Recursiva para Subproblemas:**
   - Recursivamente, resolva o problema do par de pontos mais próximos para os dois subconjuntos. Isso retorna o par de pontos mais próximos em cada subconjunto, bem como a distância mínima d entre qualquer par de pontos em cada subconjunto.
4. **Intercalação e Verificação da Faixa Central:**
   - Depois de encontrar o par de pontos mais próximos em cada subconjunto, crie uma faixa vertical (de largura $2d$) centrada na linha que divide os subconjuntos.
   - Apenas os pontos dentro dessa faixa são candidatos a serem mais próximos do que d.
   - Para cada ponto na faixa, compare-o com os próximos 7 pontos em ordem de coordenada y (ordenados por y). Isso é baseado na propriedade de que, na faixa central, apenas um número limitado de pontos precisa ser comparado para encontrar o par mais próximo.
5. **Combinação dos Resultados:**
   - Compare as distâncias entre os pares de pontos encontrados nos subconjuntos e na faixa central para encontrar a distância mínima global.
6. **Conclusão:**
   - O par de pontos com a menor distância encontrada é o par mais próximo de todo o conjunto.

Exemplo:

Vamos considerar um conjunto de pontos no plano:

Pontos iniciais: $\{(2,3),(12,30),(40,50),(5,1),(12,10),(3,4)\}$

- **Passo 1: Ordenação Inicial**

$$\{(2,3),(3,4),(5,1),(12,30),(12,10),(40,50)\}$$

- **Passo 2: Divisão Recursiva**

$$L = \{(2,3),(3,4),(5,1)\}, \quad R = \{(12,30),(12,10),(40,50)\}$$

- **Passo 3: Resolução Recursiva para Subproblemas**
  - Resolva o problema para os subconjuntos L e R separadamente.
- **Passo 4: Intercalação e Verificação da Faixa Central**
  - Encontre a menor distância d nos subconjuntos L e R. Suponha que $d = 5$.
  - Crie uma faixa central de largura $2d = 10$ em torno da linha divisória.
  - Identifique os pontos na faixa e compare até 7 pontos próximos para determinar se algum par tem uma distância menor que d.
- **Passo 5: Combinação dos Resultados**
  - Compare as distâncias dos pares de pontos encontrados nos subconjuntos L, R e na faixa central.
- **Passo 6: Conclusão**
  - Suponha que o par mais próximo seja $(2, 3)$ e $(3, 4)$ com distância 1.41.

Portanto, o par de pontos mais próximo é $(2, 3)$ e $(3, 4)$ com uma distância de aproximadamente 1.41.

4.1. Rodadas

Para realizar as rodadas foram usadas respectivamente as entradas de 500, 1000, 2000, como será apresentada na tabela abaixo pelo campo SIZE, e 1000 execuções para cada entrada afim de computador um tempo que seja visivel o resultado da paralelização:

4.2. Tempo de execução do agoritmo

Tempo de execução			
SIZE	1	2	4
500	4,284	2,342	1,24
1000	8,56	4,69	2,509
2000	17,169	9,384	5,204

Figure 10. Tempo de execução - Closest Pair of Points Problem

4.3. Gráficos de colunas Speedup e Eficiência

Speedup			
SIZE	1	2	4
500	1,00	1,83	3,45
1000	1,00	1,83	3,41
2000	1,00	1,83	3,30

Figure 11. speed Up - Closest Pair of Points Problem

Eficiência			
SIZE	1	2	4
500	1,00	0,91	0,86
1000	1,00	0,91	0,85
2000	1,00	0,91	0,82

Figure 12. Eficiência - Closest Pair of Points Problem

4.4. Analise sobre os gráficos

Observamos na Closest Pair Of Points que a nossa implementação paralela é muito boa, que resulta em uma ótima eficiência na utilização dos recusos do hardware em relação a demanda, em suma nossa implementação mostra que ao dobrarmos o nosso número de threads dobramos a velocidade de execução e assim sucessivamente.

4.5. Escalabilidade

O algoritmo utiliza a técnica de ”dividir e conquistar” este algoritmo é escalável e eficiente para entradas grandes devido à sua divisão recursiva e ao processamento em subproblemas menores isso torna o algoritmo paralelizável até certo ponto. Tivemos resultádos positivos quanto a eficiência do algoritmo.

5. Um comparativo da utilização de memória cache e desempenho entre os algoritmos de ordenação na implementação paralela

5.1. Utilização de memória cache

O Merge Sort tende a usar mais cache, mas o Selection Sort usa a cache de forma mais eficiente devido ao seu padrão de acesso sequencial e previsível.

Por esse mesmo motivo, o algorítmo com maior cache miss foi o Merge Sort, o padrão de acesso de memória não linear e a natureza recursiva levam a um aumento nos cache misses em comparação com o Selection Sort, que por ser sequencial, possui um padrão de acesso de memória mais linear e previsível.

5.2. Desempenho entre os algoritmos de ordenação

O comparativo de desempenho, visto nos resultados de eficiência, mostra que o *Merge Sort* é o algoritmo com maior ecalabilidade e maior possibilidade de paralelização. O *Selection Sort* é limitado em sua aplicação paralela devido à sua natureza sequencial e à baixa escalabilidade. Para grandes conjuntos de dados e em sistemas com múltiplos processadores, o Merge Sort oferece um desempenho significativamente melhor em comparação com o Selection Sort, pois ele se mostra mais otimizado para utilizar a arquitetura paralela.