CONGRESSO CATARINENSE DE CIÊNCIA DA COMPUTAÇÃO
2017. Ano 4, Vol 4. ISSN: 2319-0655

COMPARAÇÃO DE DESEMPENHO NO ALGORITMO DE ORDENAÇÃO TIMSORT UTILIZANDO PARALELIZAÇÃO OPENMP

Daniel Rodrigo Marconatto[1]
[1]Instituto Federal Catarinense Campus Rio do Sul

danielmarconatto@gmail.com

Resumo. Este artigo busca conceituar, implementar e analisar os resultados de desempenho do algoritmo de ordenação TimSort, realizando uma comparação entre a versão paralelizada e a versão sequencial.
Palavras-chave: TimSort; OpenMP; Paralelização; Threads;

Abstract. This article seeks to conceptualize, implement and analyze the performance results of the temporal ordering algorithm, making a comparison between a parallel version and a sequential version.
Key-words: TimSort; OpenMP; Parallelization; Threads;

1. Introdução

Com o passar dos anos, houve um avanço tecnológico imenso na área de tecnologia da informação, e os processadores não ficaram para trás nesse quesito, a frequência base dos processadores aumentou consideravelmente, permitindo maior agilidade no processamento de dados.

Porém chegamos à um ponto onde somente aumentar a frequência base não resolvia mais os problemas de demanda que o mercado enfrentava, então surgiu a idéia de colocar mais núcleos de processamento em um único processador, permitindo assim, na teoria, aumentar pelo menos duas vezes mais a capacidade de processamento.

Aqui estudaremos a aplicação e implementação dessa tecnologia no algoritmo de ordenação TimSort, e analisaremos os resultados comparando a implementação deste algoritmo em sua versão sequencial e na versão paralela.

2. Funcionamento

Segundo LIMA(2013), TimSort é um algoritmo de ordenação híbrido, ou seja, é uma junção de outros algoritmos de ordenação com o objetivo de utilizar as suas vantagens individuais no melhor caso possível. Por exemplo, um algoritmo que é bastante eficaz com uma lista pequena de números independente do melhor ou pior caso, pode ser combinado com um algoritmo que tenha um bom desempenho em ordenar uma lista com muitos números porém apenas no melhor caso, assim é possível conciliar estes dois para que eles possam

ordenar uma lista grande de números com um desempenho melhor do que se fosse utilizar um algoritmo isolado.

No caso do TimSort, ele é derivado dos algoritmos MergeSort e InsertionSort (para melhor entendimento, chamarei aqui, apenas Merge e Insertion, respectivamente) com o objetivo de fornecer uma boa performance em vários tipos de dados do mundo real.

O algoritmo Insertion possui um funcionamento básico e simples, onde a lista é percorrida até encontrar o menor valor dela que esteja fora de ordem, então este elemento é inserido na posição correta.

```
1.  void insertionSort(int arr[], int left, int right)
2.  {
3.      for (int i = left + 1; i <= right; i++)
4.      {
5.          int temp = arr[i];
6.          int j = i - 1;
7.          while (arr[j] > temp && j >= left)
8.          {
9.              arr[j+1] = arr[j];
10.             j--;
11.         }
12.         arr[j+1] = temp;
13.     }
14. }
```

Tabela 1: Implementação do InsertionSort na linguagem C++

Já o Merge é um pouco mais complicado, pois utiliza o método chamado de “dividir para conquistar”, onde a lista é dividida pela metade através da recursividade e será reordenada na hora de desconstrução da “árvore binária” causada pela recursão.

```
1.  void merge(int arr[], int l, int m, int r)
2.  {
3.      int len1 = m - l + 1, len2 = r - m;
4.      int left[len1], right[len2];
5.      for (int i = 0; i < len1; i++)
6.          left[i] = arr[l + i];
7.      for (int i = 0; i < len2; i++)
8.          right[i] = arr[m + 1 + i];
9.      int i = 0;
10.     int j = 0;
11.     int k = l;
12.     while (i < len1 && j < len2)
13.     {
14.         if (left[i] <= right[j])
15.         {
16.             arr[k] = left[i];
17.             i++;
18.         }
19.         else
20.         {
21.             arr[k] = right[j];
```

```
22.             j++;
23.         }
24.         k++;
25.     }
26.     while (i < len1)
27.     {
28.         arr[k] = left[i];
29.         k++;
30.         i++;
31.     }
32.     while (j < len2)
33.     {
34.         arr[k] = right[j];
35.         k++;
36.         j++;
37.     }
38. }
```

Tabela 2: Implementação do MergeSort na linguagem C++

No TimSort, basicamente se o vetor ou o subvetor (chamado de Run) a ser ordenado possuir menos elementos que o definido pelo programador (é recomendável utilizar um valor de 32 a 64), será utilizado apenas o algoritmo Insertion por sua eficácia em listas com poucos elementos, caso contrário será feita a mesclagem com o Merge.

```
1.  using namespace std;
2.  const int RUN = 32;
3.  void timSort(int arr[], int n)
4.  {
5.      for (int i = 0; i < n; i+=RUN)
6.          insertionSort(arr, i, min((i+31), (n-1)));
7.      for (int size = RUN; size < n; size = 2*size)
8.      {
9.          for (int left = 0; left < n; left += 2*size)
10.         {
11.             int mid = left + size - 1;
12.             int right = min((left + 2*size - 1), (n-1));
13.             merge(arr, left, mid, right);
14.         }
15.     }
16. }
17. int main()
18. {
19.     int arr[] = {5, 21, 7, 23, 19};
20.     int n = sizeof(arr)/sizeof(arr[0]);
21.     timSort(arr, n);
22.     return 0;
23. }
```

Tabela 3: Implementação do TimSort na linguagem C++

Todas essas implementações estão disponíveis em através do repositório gratuito GitHub[1].

1 Disponível em: https://github.com/DanielMarconatto/TimSort.

3. SpeedUp

Segundo MARCONATTO(2017), a principal métrica para avaliar o ganho de desempenho obtido na paralelização de um algoritmo é o SpeedUp.

De acordo com Orellana (2011) o SpeedUp é definido como a razão entre o tempo de execução do algoritmo sequencial T S = (n), que depende apenas do tamanho n do problema, e o tempo de execução do algoritmo paralelo T p (p, n), que depende do tamanho do problema e da quantidade p de processadores utilizados.

A eficiência da paralelização é comprovada caso o valor do SpeedUp for igual ou próximo ao número de threads utilizadas na execução daquele algoritmo. Por exemplo, se um algoritmo leva 10 milissegundos para ordenar um vetor na versão sequencial, e 5 milissegundos para ordenar um vetor na versão paralelizada utilizando 2 threads, obtendo um SpeedUp de valor próximo a 2, isso significa que o algoritmo foi bem paralelizado e comprova a sua eficácia.

4. Plataforma de Testes

Os testes foram realizados em uma plataforma de hardware composta por um processador Intel Core i7 8550U Quad Core 1.8 GHz (4.00 GHz em Max Turbo) com 8 Threads, totalizando 12 elementos de processamento, e 8 GB de RAM DDR4 a 2133 MHz, rodando com sistema operacional Ubuntu 18.04.

Os códigos na linguagem C++ foram compilados utilizando o compilador GCC versão 7.3.0.

Cada algoritmo foi executado dez vezes na versão sequencial e na paralela com 2, 4, 8, 16 e 32 threads, onde foram utilizados vetores de 1000 posições inversamente ordenados e aleatoriamente populados imutáveis, foi considerado apenas a média de tempo de execução das mesmas. É importante frisar que como padrão foi definido o Run com valor padrão de 50 posições.

As dez repetições foram executadas em sequência logo ao iniciar a plataforma de testes, para cada grupo threads, por exemplo, a máquina foi ligada e logo executada dez repetições do algoritmo de ordenação TimSort com vetor inversamente populado na sua versão sequencial, após isso, a máquina era reiniciada para o próximo teste.

5. Resultados

Os seguintes resultados foram obtidos utilizando vetor inversamente ordenado:

	1	2	4	8	16	32
Sequencial	0,000336					
Paralelo		0,000362	0,0003685	0,003392	0,0015885	0,002573

SpeedUp		0,9281767956	0,9118046133	0,09905660377	0,2115203022	0,1305868636

Tabela 4: Cálculo de SpeedUp com vetor inversamente populado.

Analisando a Tabela 4, percebemos que nenhuma paralelização obteve o desempenho que se esperava, a paralelização com 2 Threads chegou mais próximo do desejado, porém mesmo assim não se tornou mais eficiente que a versão sequencial do algoritmo.

Agora veremos os resultados com vetor aleatoriamente populado:

	1	2	4	8	16	32
Sequencial	0,000328					
Paralelo		0,0003755	0,000394	0,0021555	0,0015725	0,0024005
SpeedUp		0,8735019973	0,8324873096	0,1521688703	0,2085850556	0,1366382004

Tabela 5: Cálculo de SpeedUp com vetor aleatoriamente populado.

Ao observarmos a Tabela 5, podemos notar que nenhum grupo de threads chegou à ordenar com um desempenho próximo do esperado, assim como nos testes com vetor invertido apenas o algoritmo com duas Threads obteve um desempenho melhor que os outros, porém, ainda ficando atrás do algoritmo sequencial.

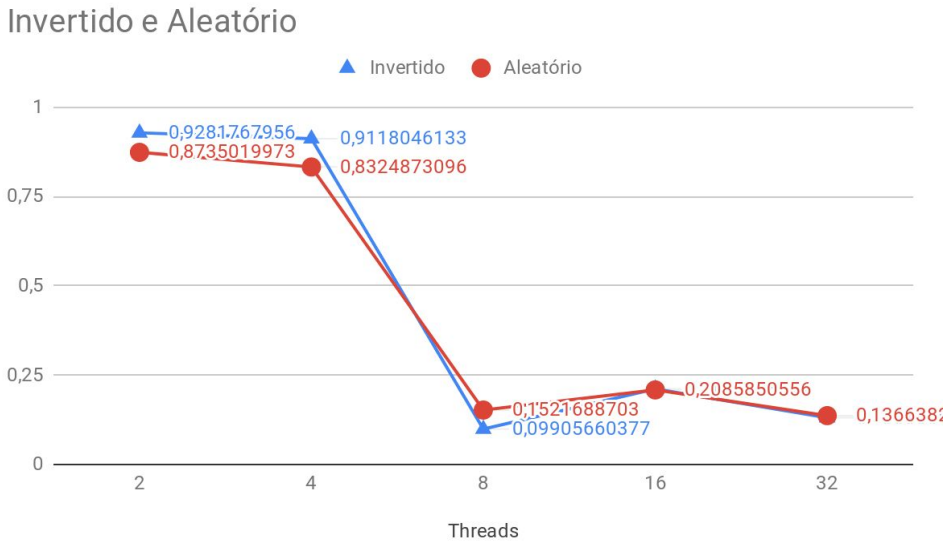

Gráfico 1: Comparação entre vetor aleatoriamente populado e vetor invertido.

6. Conclusão

Ao observarmos os resultados, podemos perceber que a paralelização do TimSort não foi muito eficiente, já que nenhum grupo de threads chegou a apresentar um desempenho superior ao algoritmo TimSort em sua versão sequencial.

Fica aqui como sugestão para objeto de estudo a execução deste mesmo algoritmo com vetores maiores, sugerível talvez com 10.000 posições ou mais, fazendo com que o algoritmo utilize de melhor forma o paralelismo na etapa de junção dos subvetores (Merge Sort).

7. Referências

Lima, Alan S. A. Union Sort: Estudo e Criação de um Rápido Algoritmo Híbrido de Ordenação. Trabalho de Conclusão de Curso - Ciência da Computação, Universidade de Fortaleza. Fortaleza, p. 18. 2013. Disponível em:<http://fbuni.edu.br/sites/default/files/tcc_-_2013_2_-_alan_siqueira_athayde_lima.pdf>. Acesso em: 17 Junho 2019.

Marconatto, Marco A. Sumarização da especificação OpenMP em Linguagens de Programação: Desempenho e Maturidade. Trabalho de Conclusão de Curso - Ciência da Computação, Instituto Federal Catarinense Campus Rio do Sul. Rio do Sul, p. 25. 2017.

ORELLANA, E. T. V. Minicurso: Introdução ao Processamento de Alto Desempenho. In: Escola Regional de Alto Desempenho (ERAD) Região Nordeste, 2011, Ilheus. Anais da Escola Regional de Alto Desempenho (ERAD) Região Nordeste, 2011.

Skerritt, Brendon. TimSort - the fastest sorting algorithm you've never heard of. Disponível em: <https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399>. Acesso em: 21 Maio 2019.

Wikipedia. Tim Peters. Disponível em: <https://en.wikipedia.org/w/index.php?title=Tim_Peters_(software_engineer)&oldid=886078303>. Acesso em: 21 Maio 2019.

Wikipedia. Timsort. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Timsort&oldid=51469720>. Acesso em: 21. Maio 2019.