Instituto Tecnológico de Aeronáutica

CE-265: Processamento Paralelo

Exercício 04: Paralelizar Merge Sort OpenMP

Bruno Benjamim Bertucci - Turma 23.2

1 Algoritmo Merge Sort

O algoritmo Merge Sort é um método popular para a ordenação de um vetor de números. Seu conceito básico envolve partir do vetor desordenado, e dividir ele pela metade, obtendo-se duas divisões, cada qual com metade do tamanho do vetor original. Cada uma dessas duas divisões é novamente dividida pela metade, e esse processo se repete até que atinja-se uma divisão com um único elemento. Em seguida, esses dois elementos são concatenados de forma ordenada, formando um par de elementos ordenados que, por sua vez, é concatenado com outro par ordenado de elementos, de forma ordenada para que se obtenha uma sequência de quatro elementos ordenados. Esse processo segue até que o vetor seja completo de forma completamente ordenada.

2 Execução de Merge Sort sequencial

Inicialmente, foi feita uma implementação sequencial do algoritmo Merge Sort, usando a linguagem C. A implementação foi feita usando uma função recursiva, denominada MergeSort, que recebe um trecho do vetor completo, e o tamanho da divisão sobre a qual cada recursão deve operar. Dessa forma, para cada recursão dessa função, caso o tamanho da divisão seja maior do que 1, o trecho atribuído a ela é dividido em duas metades, sendo cada metade atribuída a mais uma chamada da função. Após essas duas chamadas serem retornadas, uma função Merge é executada, realizando a concatenação dos trechos ordenados que foram retornados.

Essa implementação foi testada com tamanhos de vetores seguindo potências de 2, no caso, as potências sendo 27 e 28. Os tempos de execução estão na Tabela 1.

Potência	Tempo (s)
27	35.614455
28	73.941537

Table 1: Tempos de execução da implementação sequencial.

3 Execução de Merge Sort paralelizado em OpenMP

A partir da implementação recursiva inicial de Merge Sort, foi feita a paralelização do trecho de realização da ordenação do vetor. A paralelização foi feita usando paralelismo aninhado, por isso, logo antes da primeira chamada da função MergeSort, o comando omp_set_nested(1) foi executado para habilitar tais procedimento.

A função MergeSort foi adaptada para receber um argumento adicional, contendo o número de *threads* disponível a cada ocorrência da recusão. Sempre que esse valor for maior do que 1, uma *thread* será alocada para cada uma das duas novas chamadas da função. Caso não haja mais *threads* disponíveis, as novas chamadas serão feitas de forma sequencial. Esse procedimento é feito de tal forma a utilizar todas as *threads* disponíveis, mas sem introuzir latência ou mesmo

erros no programa por excesso de abertura de novas regiões paralelas. O código da função MergeSort paralelizada encontra-se no Anexo A.

A Tabela 2 mostra os tempos de execução de testes realizados com essa implementação paralelizada, para diferentes quantidades de *threads*. Foram testados tamanhos de vetores de 2^{27} e 2^{28}. A Tabela 3 mostra também os *speed-ups* observados, sendo essa métrica dada pela Equação 1, onde p é o número de processadores, e $T(p)$ é o tempo de execução para p processadores. Finalmente, a Figura 1 ilustra graficamente a evolução dos tempos de execução conforma variou-se a quantidade de *threads* para cada tamanho de vetor testado.

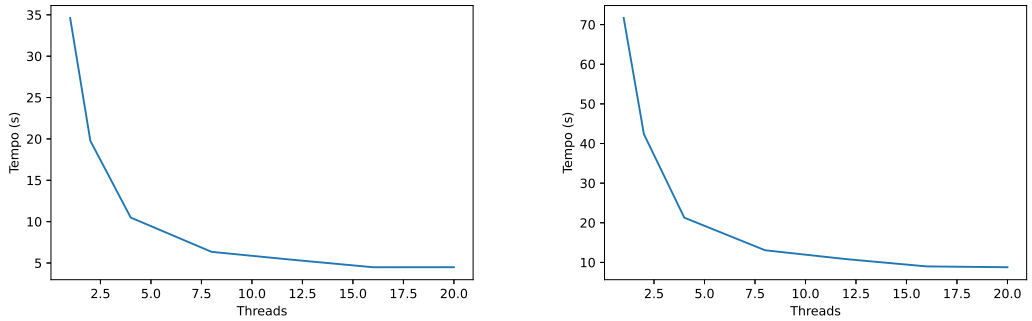
$$S(p) = \frac{T(1)}{T(p)} \tag{1}$$

Threads	27	28
1	34.616630	71.669686
2	19.767123	42.378215
4	10.490489	21.290209
8	6.349246	13.069301
12	5.391437	10.840181
16	4.488550	8.992164
20	4.492456	8.784662

Table 2: Tempos de execução da implementação paralelizada.

Threads	27	28
2	1.751222	1.691192
4	3.299811	3.366321
8	5.452085	5.483819
12	6.420669	6.611484
16	7.712208	7.970238
20	7.705502	8.158502

Table 3: Valores de Speed-up da implementação paralelizada.



(a) Gráfico de tempos de execução para potência 27.

(b) Gráfico de tempos de execução para potência 28.

Figure 1: Gráficos de tempo de execução da implementação paralelizada.

Observando as tabelas e os gráficos, pode-se observar que a implementação mostra ganhos significativos com o uso de paralelização, porém, os valores de *speed-up* são consideravelmente distantes de valores que seriam observados em um programa completamente paralelizável, mesmo para poucas *threads*. Além disso, após 16 *threads*, o aumento de *threads* não leva mais a ganhos significativos de desempenho. Parte desse gargalo é causado pelo fato de que a função Merge não

pode ser paralelizada, pois a reordenação dos trechos envolve a dependência entre um elemento e seus vizinhos e, portanto, cada iteração do procedimento não é independente.

Anexo A – Função recursiva MergeSort paralelizada

```
void MergeSort(int* data, int size, int numThreads){
  int half=size/2;
  if (size > 1) {
    if(numThreads > 1) {
      #pragma omp parallel
      {
        #pragma omp sections
        {
          #pragma omp section
          MergeSort(data, half, (numThreads + 1)/2);
          #pragma omp section
          MergeSort(data+half, half, (numThreads + 1)/2);
        }
      }
    } else {
      MergeSort(data, half, 1);
      MergeSort(data+half, half, 1);
    }

    Merge(data, size);
  }
}
```