Exercício Programa 2 – Disciplina de Arquitetura de Computadores

Prof. Clodoaldo A. M. Lima

Bruno Rogacheski NUSP 7651479

Pedro Beraldo NUSP 6777363

Renan Rogério Boni NUSP 8921238

Relatório - EP 2

1. Introdução

O presente trabalho realizou uma análise dos resultados da paralelização de dois algoritmos de ordenação: *QuickSort e MergeSort*. A criação e o gerenciamento de cada thread se deu através de diretivas de compilação, com o uso da biblioteca OpenMP e linguagem C.

O objetivo é verificar o impacto do processo no tempo de execução total do programa fazendo um comparativo entre o algoritmo sendo executado em uma única thread (programa sequencial) e uma outra implementação em que a execução se deu em 1, 2, 4, 8, 16 e 100 threads.

Cada implementação foi executada 10000 vezes, calculamos as médias de cada uma e confeccionamos os gráficos do presente relatório.

O tempo foi calculado fazendo a diferença entre o tempo de início do algoritmo e o tempo de término, ou seja, não foram consideradas chamadas de método, condicionais nem outros métodos auxiliares, como a remoção de números iguais e mensagens de saída em tela.

2. QuickSort e MergeSort

O Quicksort é o algoritmo onde escolhemos um elemento o qual chamamos de pivô, a partir disto é organizada a lista para que todos os números anteriores a ele sejam menores que ele, e todos os números posteriores a ele sejam maiores que ele. Ao final desse processo pivô já está em sua posição final. Executamos este processo recursivamente nas duas listas (subsequente e consequente ao pivô) até que o vetor esteja organizado. A complexidade desse algoritmo é O (n *log* n) para o melhor caso e caso médio, e O (n^2) para o pior caso

A ideia básica do Mergesort é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo Mergesort divide a sequência original em pares de dados, agrupa estes pares na ordem desejada e agrupa as sequências de pares já ordenados, formando uma nova sequência ordenada de quatro elementos, e assim por diante, até ter toda a sequência ordenada[1]. A complexidade desse algoritmo para o pior, melhor e caso médio é de O (n *log* n).

3. OpenMP

Para realizar o processo de paralelização utilizamos a biblioteca OpenMP que consiste em uma API que facilita o processo de geração e gerenciamento de threads em um programa através de diretivas de compilação.

Utilizamos 4 métodos desta biblioteca:

- ***#pragma omp paralel:*** Define uma região paralela, que é o código que será executado por vários threads em paralelo.

- **#pragma omp section:** Identifica as seções de código seja dividido entre todos os threads
- ***#pragma omp sections nowait:*** Idem ao anterior, porem com a clausula *nowait* que informa que as threads não precisam esperar o término das demais para entrar na *section*;
- ***#pragma omp parallel sections shared(pos) shared(list)***

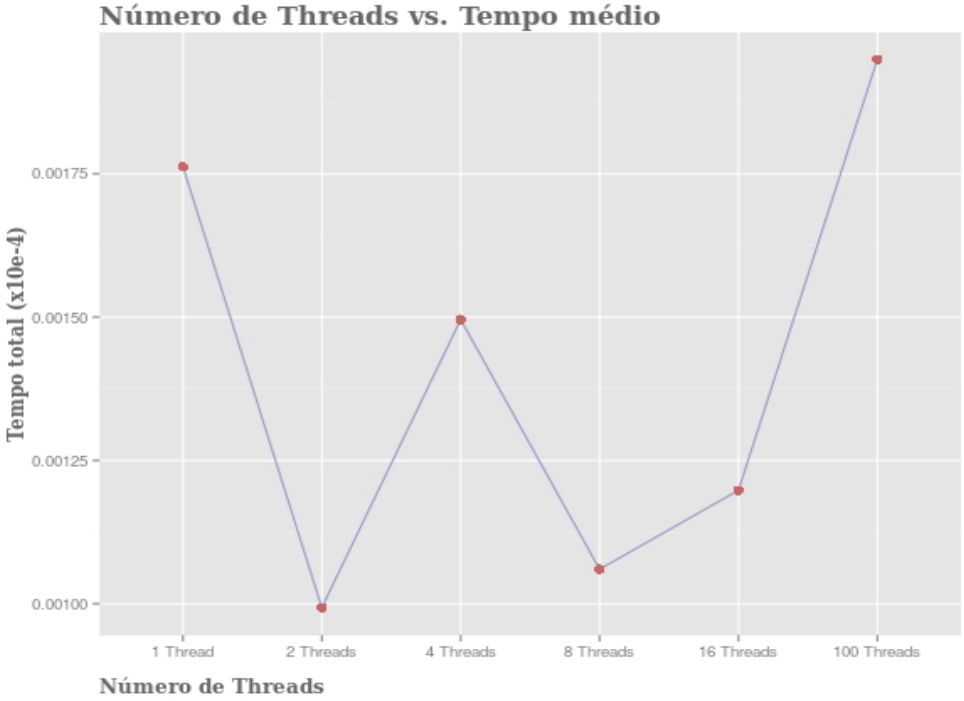
4. Processo de Paralelização

4.1 MergeSort

O trecho paralelizado foi a região onde há a efetiva ordenação e não há uma região crítica. Cada thread é responsável pela ordenação de uma pequena parte do vetor. Tomamos o devido cuidado na escolha da região para evitar a cópia de todo o array para cada thread, isso aumentaria muito o tempo de execução, ocupando um espaço desnecessário e inviabilizando o processo e a razão da paralelização.

```
void sort(int low, int high,int *vetorR, int *lastResult) {
    int mid;
    if(low < high) {
        mid = (low + high) / 2;
        #pragma omp parallel
        {
            #pragma omp sections nowait
            {
                #pragma omp section
                    sort(low, mid, vetorR, lastResult);
                #pragma omp section
                    sort(mid+1, high, vetorR, lastResult);
            }
        }
        merging(low, mid, high, vetorR, lastResult);
```

O teste foi realizado em listas de 10 mil entradas aleatórias, e repetido 10 mil vezes. Após esse processo confeccionamos o seguinte gráfico:



As quantidades de threads foram alteradas utilizando a variável de ambiente *export OMP NUM TREADS = N*.

Como esperado, um dos piores desempenhos foi com apenas 1 thread, ou seja, a execução sequencial do algoritmo, perdendo apenas para o teste com 100 threads, onde se gasta muito tempo com leitura e escrita de registradores para cada thread, inviabilizando a execução e obliterando a razão da paralelização

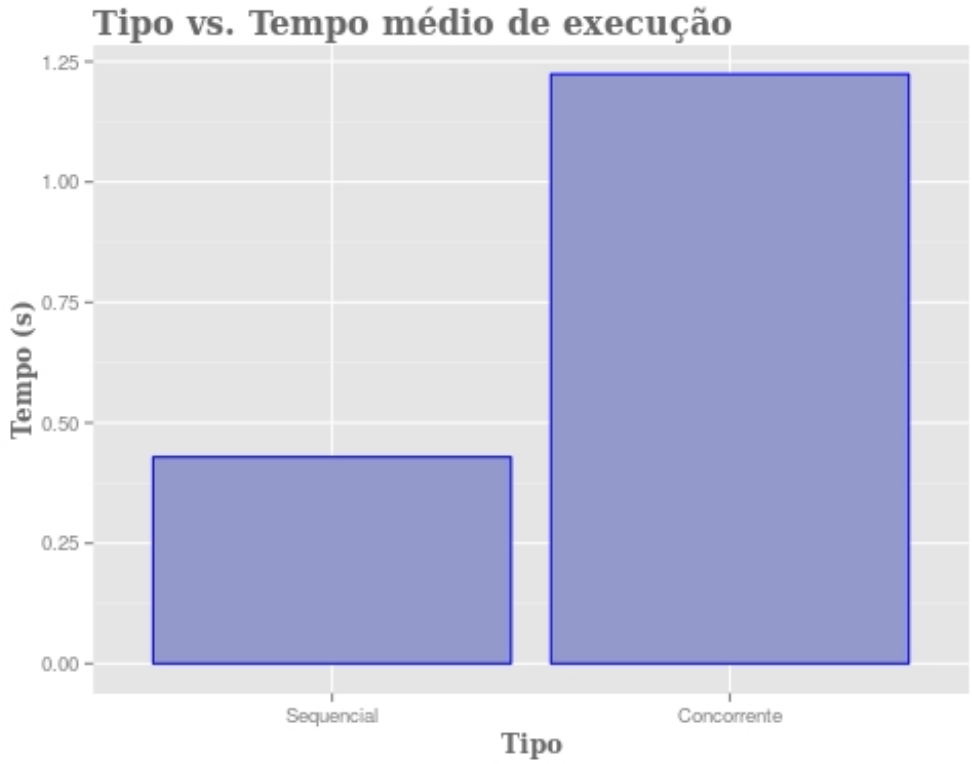
Verificamos que, com a nossa implementação, o menor tempo médio de execução foram com 2 threads, seguido por 8 e 16 threads, como o esperado, já que há um limite na eficiência da quantidade de paralelizações, como podemos ver no caso em que N = 100, se tornando um *outlier*.

Houve um pico no tempo de execução com N = 4 que creditamos ao hardware testado e as condições do teste, já que o esperado é que se mantivesse nas proximidades dos resultados obtidos entre 2 e 8 threads.

4.2 QuickSort

A paralelização deste algoritmo possui um grau de complexidade um pouco maior pois requer uma atenção extra na região a ser paralelizada bem como as diretivas do OpenMP a serem utilizadas.

Em nossa primeira implementação houve um aumento considerável no tempo de execução da versão concorrente, em relação a sequencial, o que demonstra uma implementação errada das diretivas, já que um tempo menor era esperado.



Acreditamos que tal erra se deva as seguintes diretrizes:

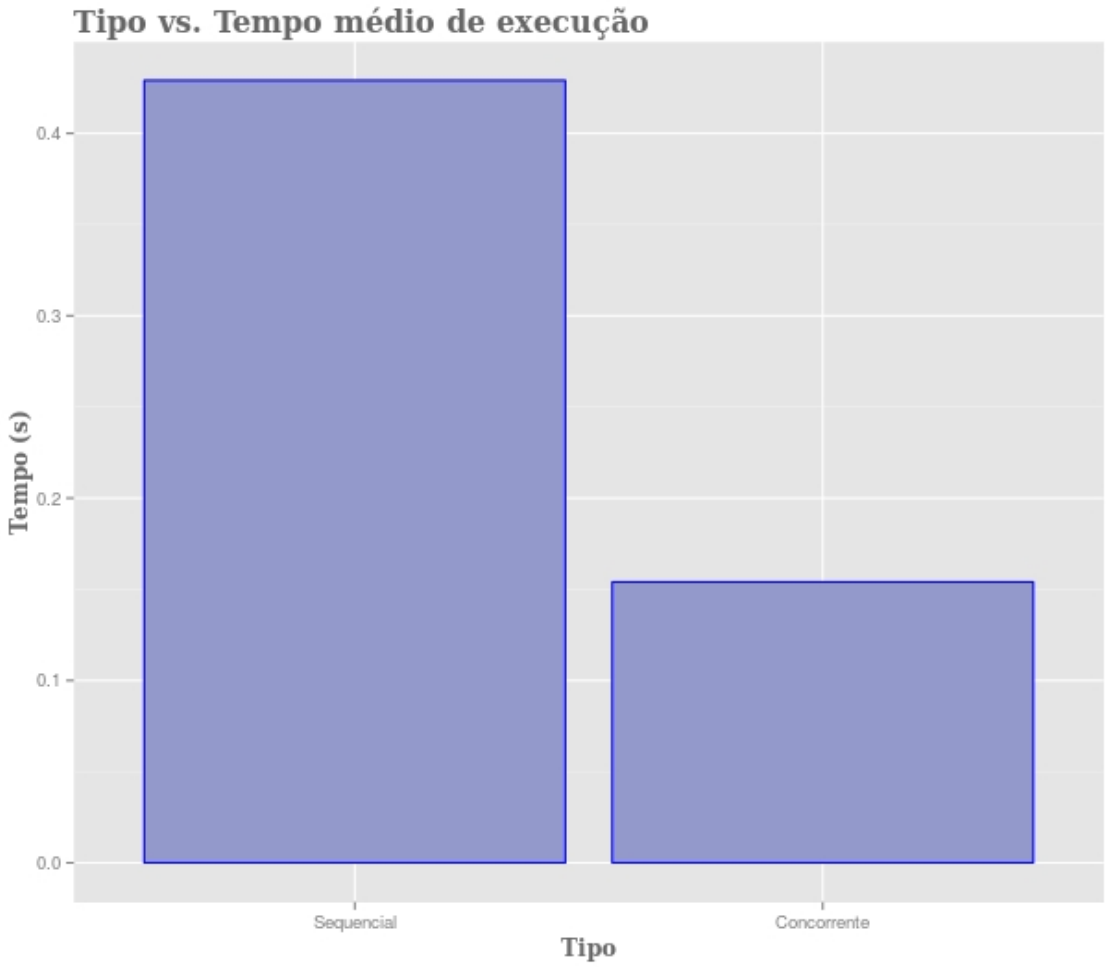
```
#pragma omp parallel sections
{
        #pragma omp section
        {
            quick_sort(l, pos-1, list);
        }

        #pragma omp section
        {
            quick_sort(pos+1, r, list);
        }

}
```

Nessa implementação, as threads têm eficiência reduzida, e impede de serem criadas recursivamente, sobrecarregando as existentes.

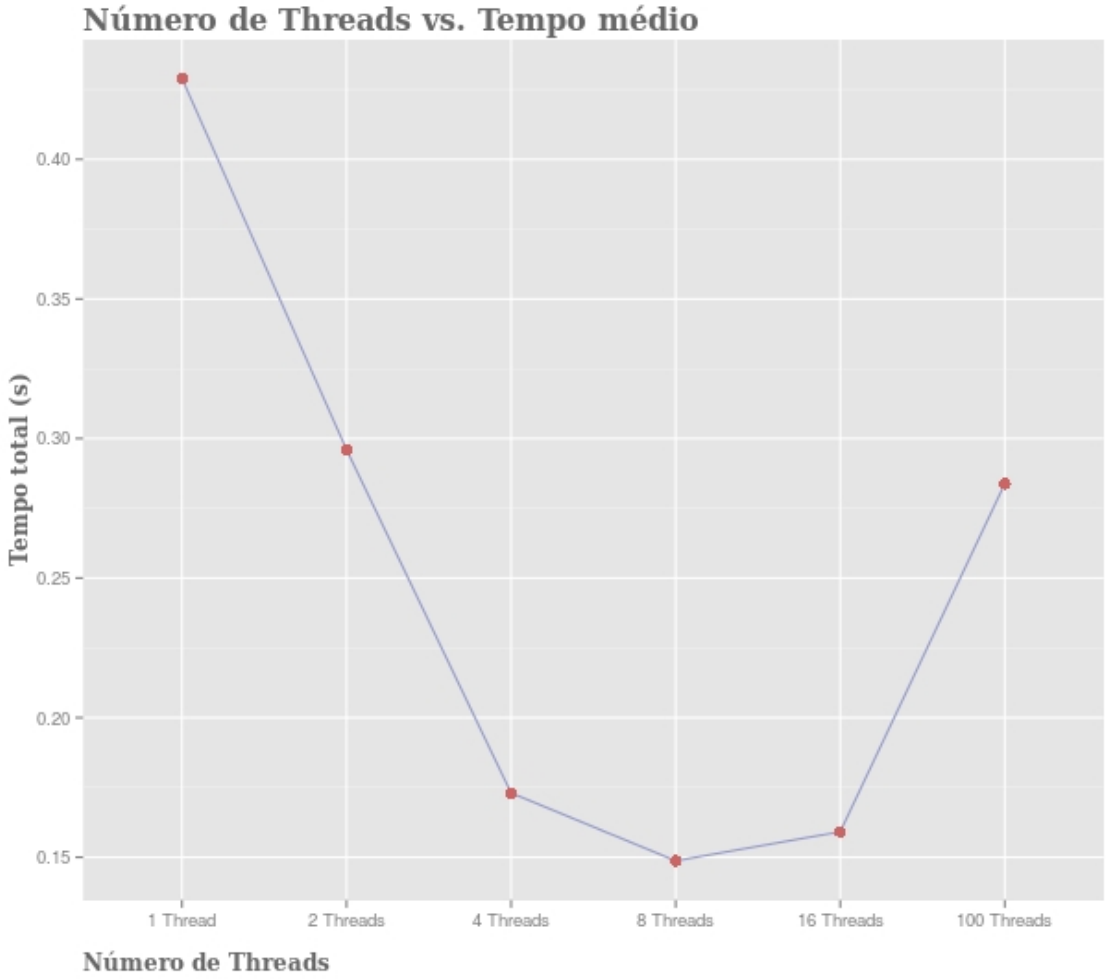
Após profunda análise e estudo do algoritmo e da biblioteca OpenMP chegamos a uma implementação mais robusta, onde algumas variáveis são compartilhadas entre as threads e outras são privadas, através da diretiva ***#pragma omp task firstprivate***. Com o indicativo *firstprivate* indicamos que cada thread terá sua instância das variáveis de terminadas, otimizando o uso das mesmas e, consequentemente, diminuindo o tempo total de execução, como pode ser visto no gráfico abaixo



Cerne do código paralelizado:

```
#pragma omp task firstprivate (list, l, r, pos)
    quick_sort(l, pos-1, list);
#pragma omp task firstprivate (list, l, r, pos)
    quick_sort(pos+1, r, list);
```

Em nossos testes obtivemos a melhor performance com 8 threads.



Conclusão

O presente trabalho possibilitou evidenciar a eficiência do processo de paralelização, bem como os cuidados que são necessários nesse processo como a determinação da área crítica, da quantidade (média) de threads a serem utilizadas e o local onde se deve paralelizar visando uma otimização no tempo total de execução do código.

Encontramos problemas na hora de portar o código para outros SO s, devido ao fato da forma em que esses sistemas lidam com o gerenciamento das threads, através do escalonador.

Outro ponto importante que ficou evidente foi a importância no controle da criação de threads durante a recursão, fato evidenciado pela maior morosidade de nossa primeira implementação do *QuickSort*.

Constatamos também que a criação de um limite superior para a quantidade de threads criadas é mais eficiente que setar uma quantidade fixa de threads.

Bibliografia

[1] http://www.cos.ufrj.br/~rfarias/cos121/aula_07.html