UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

KAROLINE KIMIKO FIGUEIREDO SETOUE

PROJETO - MERGE-SORT EM OPENMP

BAURU
2017

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 CONCEITOS INTRODUTÓRIOS

O presente trabalho tem por objetivo apresentar uma análise de tempos de execução em um algoritmo paralelo de Merge-Sort. A implementação escolhida baseia-se no Merge-Sort em sua forma recursiva (CORMEN, 2009). Tal algoritmo utiliza a estratégia de dividir e conquistar, de modo que a primeira fase concentra-se na divisão do vetor de entrada e, a segunda concentra-se na ordenação do vetor. Para atender as exigências da proposta, a etapa de divisão limita-se a 5 níveis de recursão – o que garante que o vetor seja dividido em 32 pedaços. Para aplicação de paralelismo ao Merge Sort, a implementação proposta utilizou a biblioteca OpenMP(OPENMP, 2016) (PACHECO, 2011).

1.1 Objetivos

1.1.1 Objetivos Gerais

Implementar um algoritmo de Merge-Sort utilizando OpenMP.

1.1.2 Objetivos Específicos

a) Particionar o vetor, independente do tamanho, em 32 partes;

b) Utilizar *threads* para realizar a ordenação;

c) Realizar testes com tamanhos de vetores de 200, 400, 800 e 1600 elementos;

d) Realizar testes com 1, 2, 4, 8, 16 e 32 *threads*;

e) Analisar os tempos de execução medidos na fase de ordenação;

2 *MERGE-SORT* T PARALELO

Neste capítulo estão descritos a implementação do algoritmo e os testes realizados.

2.1 Implementação

A implementação do algoritmo de *Merge-Sort* paralelo consiste numa adaptação do algoritmo serial recursivo. O algoritmo se divide em duas etapas: a primeira responsável pelas divisões; e a segunda responsável pela etapa de *merge*.

A primeira etapa do algoritmo contém uma peculiaridade: o vetor deve ser dividido em 32 partes, independente de seu tamanho. Para isso, considerando que cada nível da etapa recursiva de divisão implica na divisão do vetor em duas partes, cada nível dessa árvore possui o dobro de partes do nível anterior. Portanto, ao chegar ao quinto nível, o vetor terá 32 pedaços. Os níveis de recursão estão exemplificados na figura 1.

Esta etapa possui chamadas recursivas atribuídas ao conjunto de *threads* da região paralela. Assim, a tarefa de divisão e ordenação é dividida entre o grupo de *threads* disponíveis. A figura 2 apresenta a implementação paralela do *Merge Sort*.

A segunda etapa do algoritmo é a etapa de *Merge*, na qual cada par de partes de vetor é ordenado e passa pela operação de junção. Nessa etapa, as regiões de ***for*** também são paralelizadas. A implementação é apresentada na figura 3.

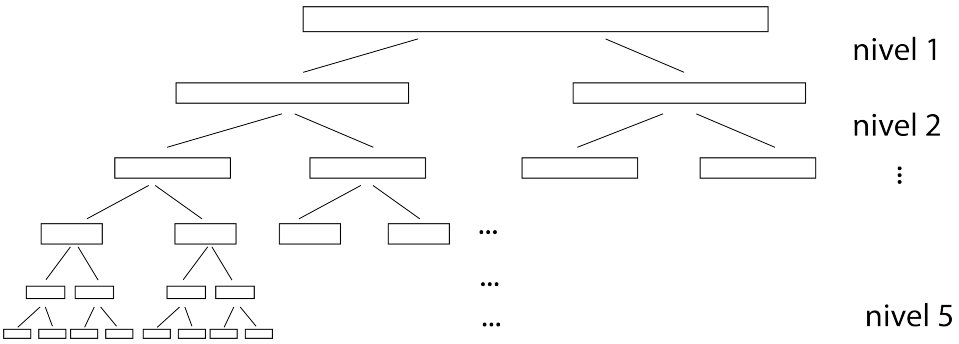
A implementação sugerida cria cada vetor atribuindo valores aleatórios para o mesmo. Em seguida, efetuam-se os testes onde, para cada valor de vetor e quantidade de threads são realizadas 10 execuções do experimento. A primeira chamada do **mergesortP** (*merge sort* paralelo) é feita pela *thread master*, onde ocorrerá o *join* das demais *threads* após o fim das tarefas de ordenação.

Dentro do algoritmo de ordenação, as chamadas das próximas tarefas são feitas dentro das instâncias de **omp parallel sections**. Após dividir o vetor em 32 partes, cada parte da última camada da recursão é ordenada com o merge sort serial, retornando para a camada anterior. Na operação de Merge, como descrito acima, cada *for* é realizado de maneira paralela, atribuindo as tarefas para as *threads* do grupo que estão disponíveis.

2.2 Resultados e discussão

A tabela 1 apresenta os resultados de tempo de execução médio em 10 execuções de teste, de acordo com o número de *threads* e tamanhos de vetor. Os valores apresentam

Figura 1 – Níveis de recursão



Fonte: elaborada pela autora

o tempo médio medido em *ticks* por segundo, calculados utilizando a função *clock()* da biblioteca **time.h**. Em negrito, os menores tempos de execução.

Tabela 1 – Resultados

Tamanho do vetor	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads
200	9.200000	9.000000	6.800000	**5.900000**	6.500000	7.500000
400	13.300000	**11.200000**	28.300000	13.600000	11.500000	12.600000
800	19.000000	16.900000	16.800000	**16.000000**	16.300000	16.300000
1600	31.500000	**31.000000**	31.400000	31.300000	31.400000	35.300000

Elaborado pela autora.

No teste realizado é possível notar que os processos com 2 e 8 *threads* obtiveram os menores valores de tempo de execução. No entanto, é possível notar que o paralelismo com mais de *threads* apresenta melhor desempenho em todos os casos, exceto no vetor de 1600 posições, onde o tempo de execução com 32 threads foi o maior. Este fato pode ser explicado pela máquina onde o teste foi executado (processador i5 de 4 núcleos) e o número de threads que excede a quantidade de núcleos pode ter aumentado o tempo de execução, devido a necessidade de compartilhar recursos de processamento.

O paralelismo aplicado a etapa de merge 3 também garante melhor desempenho nos casos com maior número de threads, para todos os tamanhos de vetor.

O gráfico da figura 4 apresenta as curvas de tempo de execução para cada tamanho

Figura 2 – Implementação do *MergeSort*

```
void mergesortP(int *vector, int start, int end, int nivel) {
    int middle;

    if (start < end) {
        if( nivel > 5 ) {
            mergesort(vector, start, end);
            return;
        }


        #pragma omp parallel sections
        {
            middle = (end + start) / 2;

            #pragma omp section
            mergesortP(vector, start, middle, nivel+1);

            #pragma omp section
            mergesortP(vector, middle+1, end, nivel+1);

        }
        #pragma omp parallel sections
        {
            merge(vector, start, middle, end);
        }
    }
}
```

Fonte: elaborada pela autora

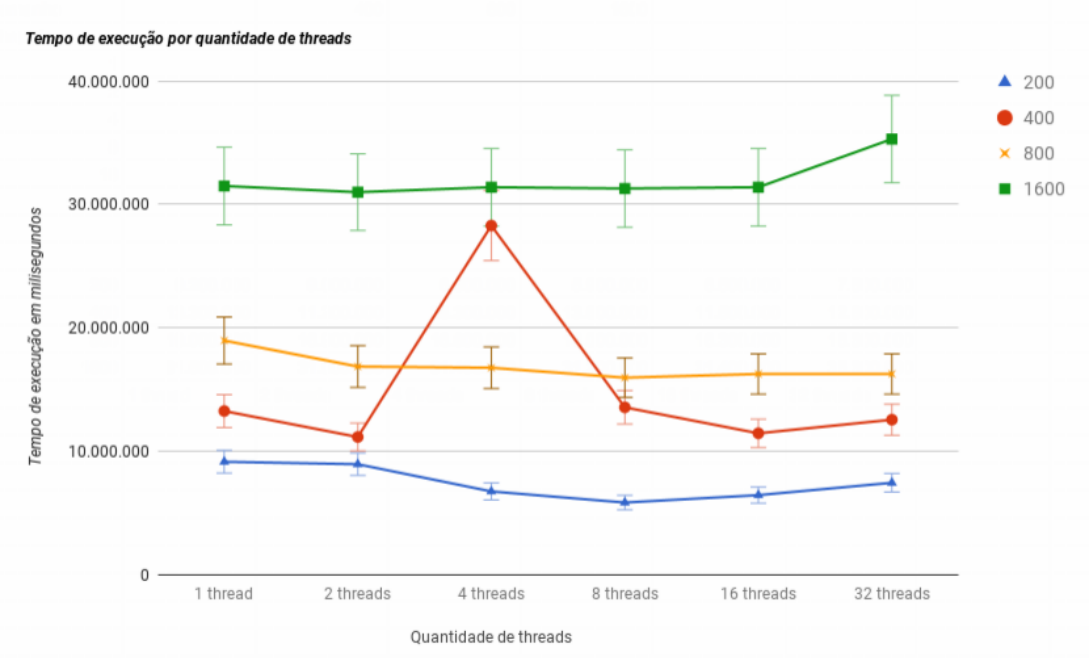
de vetor. É possível notar que na ordenação com 4 threads do vetor de tamanho 400 apresentou um tempo de execução maior que as demais que pode ser resultado do uso do argumento *dynamic*, visto que ao finalizar a execução do for uma ou mais *threads* poderiam estar ocupadas para realizar outra tarefa.

Figura 3 – Implementação da etapa de *merge*

```
void merge(int *vector, int start, int middle, int end) {
    int *left, *right;
    int nleft, nright;
    int counter, l, r, i;

    nleft = middle - start + 2;
    nright = end - middle + 1;

    left = (int *) malloc(sizeof(int) * nleft);
    right = (int *) malloc(sizeof(int) * nright);

    counter = 0;
    #pragma omp parallel shared(left, counter, vector)
    {
        #pragma omp for schedule(dynamic)
        for (i = start; i <= middle; i++) left[counter++] = vector[i];
        left[counter] = INT_MAX;

        counter = 0;
        #pragma omp for schedule(dynamic)
        for (i = middle+1; i <= end; i++) right[counter++] = vector[i];
        right[counter] = INT_MAX;
        // Intercalacao
        l = r = 0;
        #pragma omp for schedule(dynamic)
        for (i = start; i <= end; i++) {
            if (left[l] <= right[r]) {
                vector[i] = left[l++];
            } else {
                vector[i] = right[r++];
            }
        }
    }
}
```

Fonte: elaborada pela autora

Figura 4 – Tempos de execução



Fonte: elaborada pela autora

3 CONCLUSÃO

Após a análise dos tempos de execução do experimento, é possível notar que o paralelismo tornou a execução mais rápida na maior parte dos casos, de modo que o uso de mais threads apresentou melhora no tempo de execução em relação ao processamento com 1 thread (com exceção do caso do vetor de 1600 posições e 32 threads). Em geral, as execuções com 2 e 8 threads obtiveram os menores tempos em comparação com as demais.

REFERÊNCIAS

CORMEN, T. H. *Introduction to algorithms*. [S.l.]: MIT press, 2009. 30-37 p.

OPENMP. *The OpenMP API specification for parallel programming*. 2016. Disponível em: <http://www.openmp.org/>. Acesso em: 11 junho 2017.

PACHECO, P. *An introduction to parallel programming*. [S.l.]: Elsevier, 2011. 209 p.