DISTÂNCIA MÍNIMA ENTRE DOIS PONTOS DE UM CONJUNTO USANDO OPENMP - UM COMPARATIVO DE APLICAÇÕES SEQUENCIAIS E PARALELAS[1]

Carlos Rodrigues[2] <carlos.rodrigues@edu.pucrs.br>
Vinícius Branco[3] <vinicius.branco@edu.pucrs.br>
Roland Teodorowitsch[4] <roland.teodorowitsch@pucrs.br> – Professor

Pontifícia Universidade Católica do Rio Grande do Sul – Escola Politécnica – Curso de Engenharia de Software
Av. Ipiranga, 6681 Prédio 32 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

10 de outubro de 2020

RESUMO

Este artigo descreve uma análise do resultado da paralelização de um programa responsável por calcular a distância mínima entre pontos de um conjunto, com a utilização da API OpenMP. Este artigo compõe os artefatos entregáveis do primeiro trabalho proposto na disciplina de Programação Paralela do curso de Engenharia de Software da Escola Politécnica da Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Palavras-chave: OpenMP; Programação Paralela; Desempenho de Aplicações.

ABSTRACT

Title: *"Minimum distance between two points of a set using OpenMP - a comparative of sequential and parallel applications"*

This article describes an analysis of the result of parallelizing a program responsible for calculating the minimum distance between points in a set, using the OpenMP API. This article comprises the deliverable artifacts from the first work proposed in the Parallel Programming discipline of the Engenharia de Software course at Escola Politécnica of Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Keywords: *OpenMP; Parallel Programming; Application Performance.*

1 INTRODUÇÃO

Um desafio bastante conhecido na computação é o cálculo da distância mínima entre pares de pontos em um plano bidimensional, do qual podem se utilizar diversas estratégias e ferramentas para se chegar no resultado. Este desafio oferece a possibilidade de exercer certo controle em casos em que é preciso simular determinadas condições para medição de desempenho de aplicações em diferentes configurações. Dito isso, dois programas foram disponibilizados em diferentes estratégias, ambos essencialmente configurados para execução sequencial utilizando as linguagens C e C++, mas que por meio da API do OpenMP foi possível paralelizar sua execução a fim de comparar os resultados.

Dentre as duas estratégias, considerou-se o algoritmo de força bruta, que nesse contexto é aplicado em um array não ordenado com os pontos do plano bidimensional do início ao fim, aplicando a fórmula de cálculo de distância entre dois pontos, que é a raiz quadrada da soma do quadrado da diferença entre abscissas e quadrado da diferença entre ordenadas.

Porém, devido à complexidade $O(n^2)$, esse algoritmo se torna inviável à medida que o tamanho do conjunto de pontos aumenta. Isso nos permite o uso de uma segunda abordagem, que é o uso de um algoritmo utilizando a estratégia de divisão e conquista, que representa uma complexidade O(n log n), permitindo um conjunto maior de pontos em relação à estratégia de força bruta.

Porém, ambos os programas podem ser modificados para utilização de recursos de paralelismo para melhorar drasticamente seus desempenhos, o que será apresentado a seguir.

---

[1] Artigo elaborado para dissertar sobre o desafio proposto no trabalho 1 da disciplina de Programação Paralela da Escola Politécnica da PUCRS.

[2] Aluno da disciplina de Programação Paralela do curso de Engenharia de Software, da Escola Politécnica da PUCRS.

[3] Aluno da disciplina de Programação Paralela do curso de Engenharia de Software, da Escola Politécnica da PUCRS.

[4] Professor da disciplina de Programação Paralela do curso de Engenharia de Software, da Escola Politécnica da PUCRS.

2 SIMULAÇÕES

2.1 DESCRIÇÃO DO PROCESSADOR E ANÁLISE DE HORAS MÁQUINA

Para a execução das simulações, foram utilizados cerca de 2 horas de um nodo de processamento do cluster Grad do Laboratório de Alto Desempenho (LAD) da PUCRS. A informações pertinentes ao nodo de processamento podem ser observadas a seguir na Tabela 1:

TABELA 1 - CONFIGURAÇÃO DE MÁQUINA

GRAD08	INTEL(R) XEON(R) CPU E5520 @ 2.27GHZ)
ARCHITECTURE:	x86_64
CPU OP-MODE(S):	32-BIT, 64-BIT
BYTE ORDER:	LITTLE ENDIAN
CPU(S):	16
ON-LINE CPU(S) LIST:	0-15
THREAD(S) PER CORE:	2
CORE(S) PER SOCKET:	4
SOCKET(S):	2
NUMA NODE(S):	2
VENDOR ID:	GENUINEINTEL
CPU FAMILY:	6
MODEL:	26
STEPPING:	5
CPU MHZ:	2261048
BOGOMIPS:	4521.84
VIRTUALIZATION:	VT-X
L1D CACHE:	32K
L1I CACHE:	32K
L2 CACHE:	256K
L3 CACHE:	8192K
NUMA NODE 0 CPU(S):	0,2,4,6,8,10,12,14
NUMA NODE 1 CPU(S):	1,3,5,7,9,11,13,15

2.2 ESTRATÉGIA FORÇA BRUTA

Para a paralelização da solução baseada em força bruta, foi utilizada uma diretiva da API do OpenMp, para a paralelização de um laço de repetição responsável pela varredura do array de pontos, como se segue:

```
double points_min_distance_bf(point_t *points, int size) { /* bf = brute-force */
    int i, j;
    double min_d, d;
    min_d = DBL_MAX;
    #pragma omp parallel for private (j, d)
    for (i=0; i< size-1; ++i) {
        for (j=i+1; j<size; ++j) {
            d = points_distance_sqr(points+i,points+j);
            if (d < min_d)
                min_d = d;
        }
    }
    return sqrt(min_d);
}
```

Figura 1 – Paralelização da estratégia de Força Bruta (programa em C)

A linha destacada em negrito corresponde à diretiva de compilação do OpenMP *#pragma omp parallel for* a qual permite que as iterações de um laço de repetição seja dividido pela quantidade de núcleos de processamento disponíveis, criando uma thread para cada um, diminuindo drasticamente o tempo de processamento ao se aproveitar de mais recursos disponíveis. Um ponto importante de se destacar é a cláusula *private (j, d)*. Isso serve para limitar o escopo da variável *j* e *d* à cada thread, ou seja, para fazer uma cópia de cada variável para cada thread.

Os resultados numéricos da simulação podem ser observados na tabela a seguir:

TABELA 2 - RESULTADOS DA ESTRATÉGIA DE FORÇA BRUTA

BF	SEQ	2T	SPD2T	EF2T	4T	SPD4T	EF4T	8T	SPD8T	EF8T	16T	SPD16T	EF16T
10000	0,6673	0,5528	1,2071	0,6036	0,3221	2,0716	0,5179	0,1780	3,7492	0,4686	0,1294	5,1548	0,3222
20000	2,6560	2,2045	1,2048	0,6024	1,2881	2,0620	0,5155	0,7089	3,7467	0,4683	0,5075	5,2339	0,3271
30000	5,9800	4,9640	1,2047	0,6023	2,9012	2,0612	0,5153	1,5916	3,7572	0,4697	1,1113	5,3809	0,3363
40000	10,6326	8,8269	1,2046	0,6023	5,1579	2,0614	0,5154	2,8284	3,7592	0,4699	1,9171	5,5462	0,3466
50000	16,6157	13,7929	1,2047	0,6023	8,0598	2,0615	0,5154	4,4194	3,7597	0,4700	2,9370	5,6574	0,3536
60000	23,9309	19,8633	1,2048	0,6024	11,6056	2,0620	0,5155	6,3632	3,7608	0,4701	4,4032	5,4349	0,3397
70000	32,5691	27,0341	1,2047	0,6024	15,7984	2,0615	0,5154	8,6599	3,7609	0,4701	5,9910	5,4363	0,3398
80000	42,5453	35,3139	1,2048	0,6024	20,6314	2,0622	0,5155	11,3350	3,7535	0,4692	7,8249	5,4372	0,3398
90000	53,8470	44,6936	1,2048	0,6024	26,1261	2,0610	0,5153	14,3233	3,7594	0,4699	10,2754	5,2404	0,3275
100000	66,4726	55,2049	1,2041	0,6021	32,2675	2,0600	0,5150	17,6813	3,7595	0,4699	12,2877	5,4097	0,3381

A primeira coluna representa a quantidade de pontos no array, a segunda coluna (SEQ) representa o tempo em segundos da execução do código sequencial do algoritmo de força bruta, a terceira coluna (2T)

representa os tempos para a execução paralela com duas threads, a quarta coluna (SPD2T) representa o SpeedUp em relação à execução sequencial, e a quinta coluna (EF2T) representa a eficiência, que leva em consideração o SpeedUp e o número de threads. As colunas subsequentes representam os mesmos dados para a execução utilizando 4, 8, e 16 threads.

A seguir é possível ter uma visão mais prática para comparar o tempo de execução para cada configuração da relação threads x quantidade de pontos:

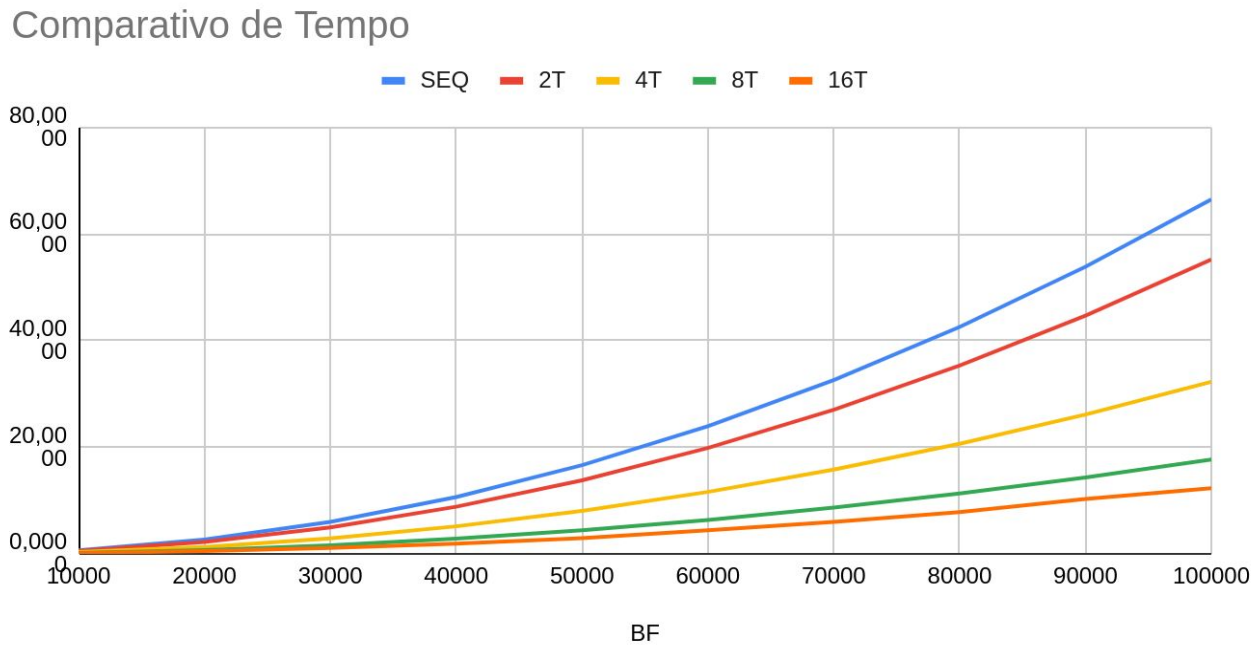

Figura 2 – Tempos para paralelização da estratégia de Força Bruta

É possível observar houve uma redução significativa no tempo conforme a quantidade de pontos e de threads aumentou. A melhora fica mais evidente quando comparamos o SpeedUp, que quase dobrou a cada nova configuração:

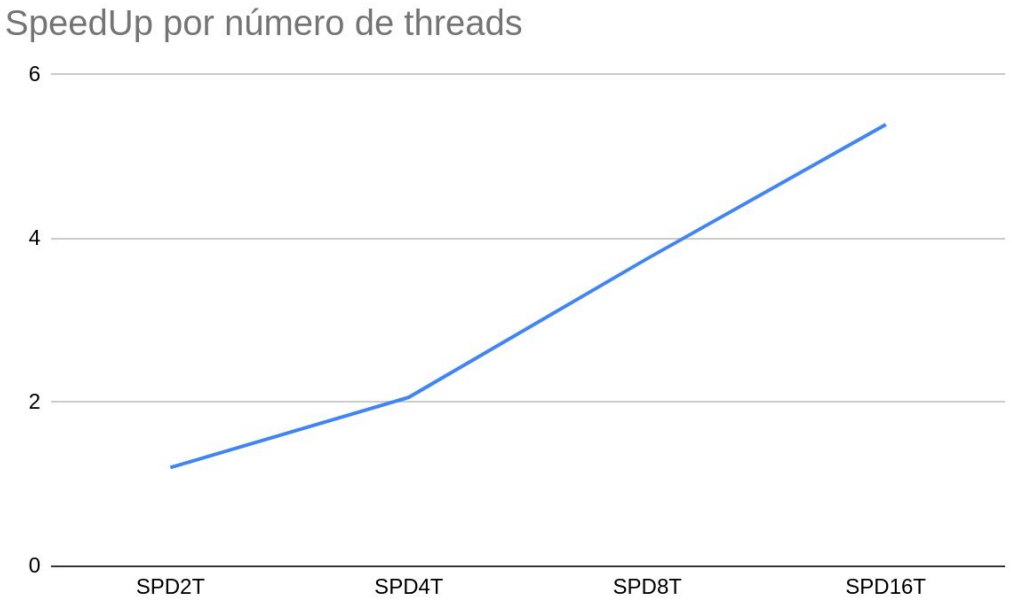

Figura 3 – SpeedUp para paralelização da estratégia de Força Bruta

Outro ponto interessante foi a queda da eficiência conforme aumentou-se o número de threads, ou seja, apesar de progressivo, a melhora no desempenho se torna cada vez menos significativa:

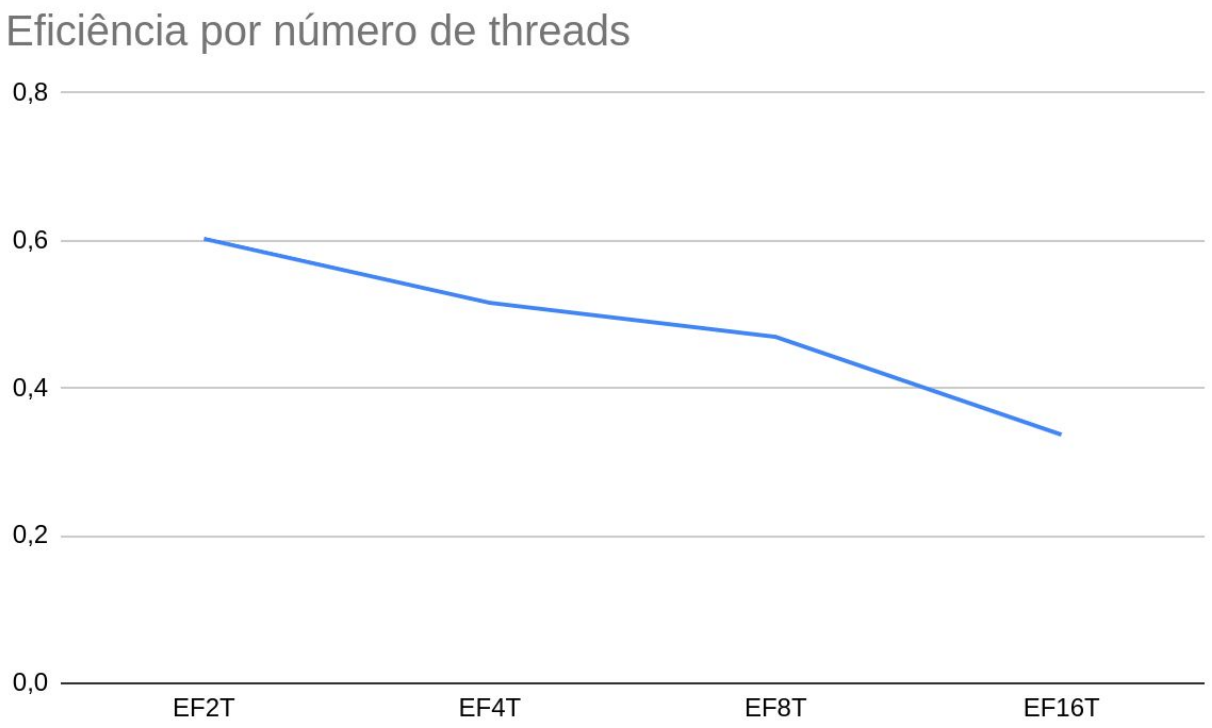

Figura 4 – Eficiência para paralelização da estratégia de Força Bruta

2.3 ESTRATÉGIA DIVISÃO E CONQUISTA

Para a paralelização da solução baseada em divisão e conquista, foi usada uma abordagem orientada a sections através da primitiva do OpenMp para paralelização de blocos de código que podem ser executados em paralelo. A seguir é possível observar o código sequencial e o código paralelo:

```
/*versão sequencial*/
    int m = (l+r)/2;
    double dL = points_min_distance_dc(point,border,l,m);
    double dR = points_min_distance_dc(point,border,m,r);
    minDist = (dL < dR ? dL : dR);


/*versão paralela*/
    int m = (l+r)/2;
    double dL;
    double dR;
    #pragma omp parallel sections
     {
        #pragma omp section
        dL = points_min_distance_dc(point,border,l,m);

        #pragma omp section
        dR = points_min_distance_dc(point,border,m,r);
     }
    minDist = (dL < dR ? dL : dR);
```

Figura 5 – Paralelização da estratégia de Divisão e Conquista (programa em C)

A diretiva *#pragma omp parallel sections* define que o bloco de código entre as chaves poderá ser executado em paralelo, enquanto que *#pragma omp section* define um individualmente um dos blocos que deve ser executado em paralelo. Note que foi necessário declarar as variáveis *dL* e *dR* fora do bloco de sections para que os mesmos pudessem ser utilizados fora do escopo da mesma para o posterior cálculo de distância mínima presente na última linha.

Os resultados numéricos da simulação podem ser observados na tabela a seguir:

Tabela 2 - Resultados da estratégia de força bruta

DC	SEQ	2T	SPD2T	EF2T	4T	SPD4T	EF4T	8T	SPD8T	EF8T	16T	SPD16T	EF16T
10000	2,2021	1,1438	1,9252	0,9626	1,1432	1,9263	0,4816	1,1445	1,9240	0,2405	1,1488	1,9168	0,1198
20000	4,3337	2,2468	1,9288	0,9644	2,2443	1,9309	0,4827	2,2792	1,9014	0,2377	2,2988	1,8852	0,1178
30000	7,1595	3,6683	1,9517	0,9759	3,6697	1,9510	0,4877	3,7113	1,9291	0,2411	3,6798	1,9456	0,1216
40000	8,6139	4,4317	1,9437	0,9718	4,4324	1,9434	0,4859	4,4366	1,9416	0,2427	4,4220	1,9480	0,1217
50000	11,2523	5,7704	1,9500	0,9750	5,7705	1,9500	0,4875	5,7581	1,9542	0,2443	5,8108	1,9364	0,1210
60000	14,2848	7,2907	1,9593	0,9797	7,2892	1,9597	0,4899	7,3718	1,9378	0,2422	7,3791	1,9358	0,1210
70000	14,8917	7,6823	1,9384	0,9692	7,6163	1,9552	0,4888	7,7333	1,9256	0,2407	7,7114	1,9311	0,1207
80000	17,1726	8,8539	1,9396	0,9698	8,8023	1,9509	0,4877	8,8264	1,9456	0,2432	8,8798	1,9339	0,1209
90000	19,7432	10,1711	1,9411	0,9706	10,1129	1,9523	0,4881	10,2153	1,9327	0,2416	10,2009	1,9354	0,1210
100000	22,4661	11,5614	1,9432	0,9716	11,5847	1,9393	0,4848	11,5864	1,9390	0,2424	11,5691	1,9419	0,1214

Igualmente ao resultado numérico da simulação da estratégia de força bruta, a primeira coluna representa a quantidade de pontos no array, a segunda coluna (SEQ) representa o tempo em segundos da execução do código sequencial do algoritmo de divisão e conquista, a terceira coluna (2T) representa os tempos para a execução paralela com duas threads, a quarta coluna (SPD2T) representa o SpeedUp em relação à execução sequencial, e a quinta coluna (EF2T) representa a eficiência, que leva em consideração o SpeedUp e o número de threads. As colunas subsequentes representam os mesmos dados para a execução utilizando 4, 8, e 16 threads.

A seguir é possível ter uma visão mais prática para comparar o tempo de execução para cada configuração da relação threads x quantidade de pontos:

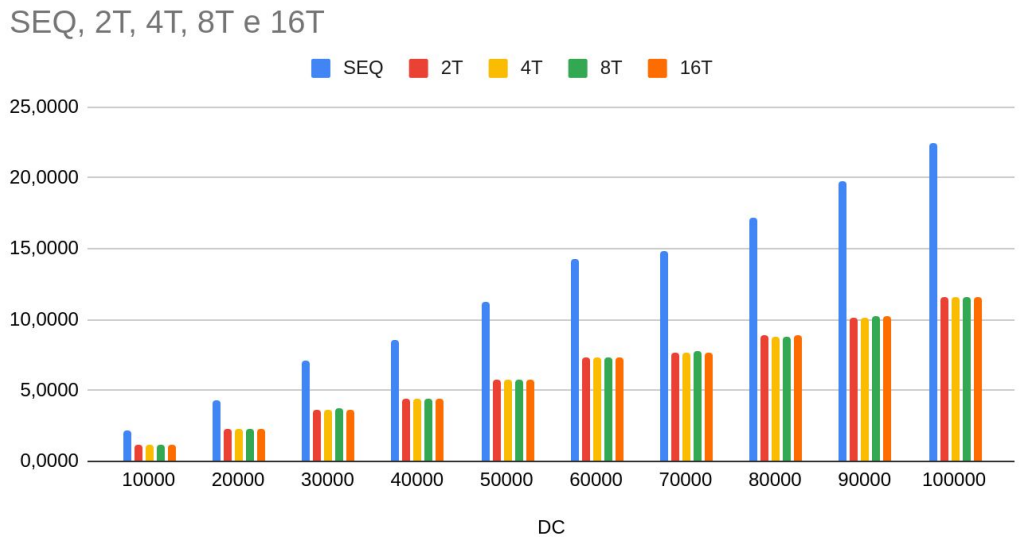

Figura 6 – Tempos para paralelização da estratégia de Divisão e Conquista

O mais importante a se destacar, é que o tempo diminuiu ao paralelizar a aplicação, mas essa diferença é notável somente ao habilitar o processamento com 2 threads, sendo que ao aumentar o número de threads após esse estágio, o tempo se mantém inalterado em relação ao número de threads, comportando-se apenas em função da quantidade de pontos. Esse fato fica mais claro ao se comparar o SpeedUp da versão paralela:

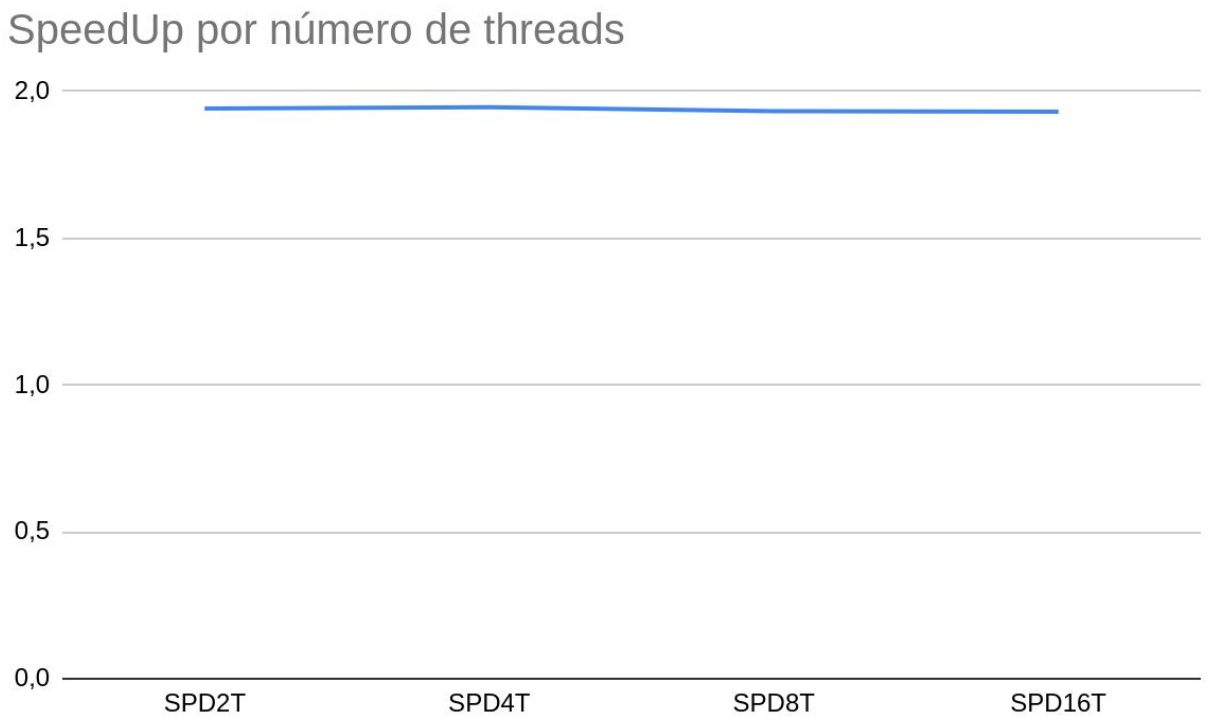

Figura 7 – SpeedUp para paralelização da estratégia de Divisão e Conquista

Claramente é possível observar que o SpeedUp em relação à versão sequencial se manteve estável conforme o número de threads aumentou, o que impacta diretamente na eficiência, que cai drasticamente:

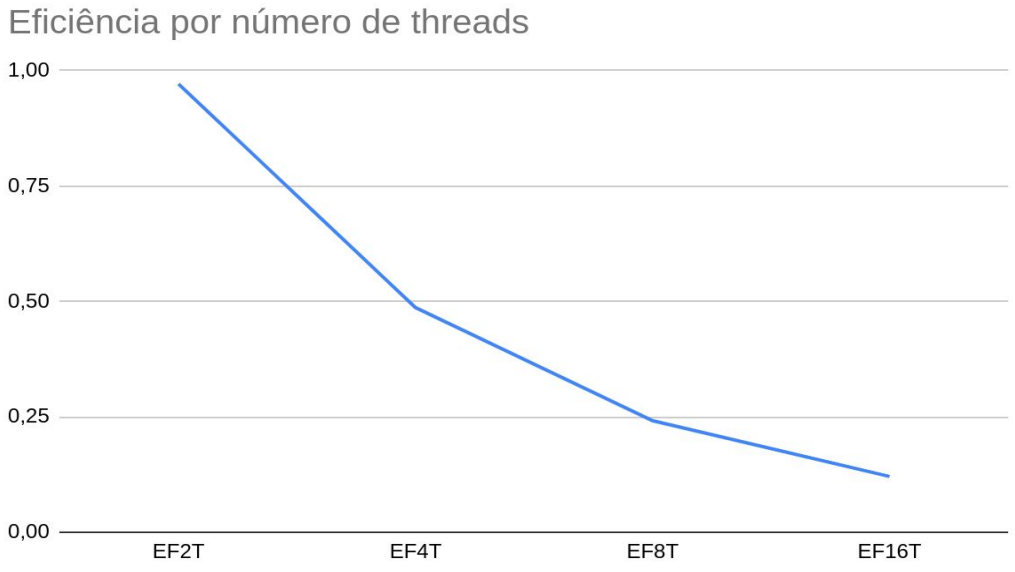

Figura 8 – Eficiência para paralelização da estratégia de Divisão e Conquista

Esse resultado se deve principalmente ao fato de que a paralelização da aplicação foi feita através da divisão da execução de um pequeno trecho de código em duas *sections* através das diretivas do OpenMP, o que significa que a paralelização pôde se apropriar de no máximo duas threads, por mais que esteja habilitado o uso de até 16 threads na aplicação.

2.4 BALANCEAMENTO DE CARGA

Segundo DE LIMA MARTINS (2002), balanceamento de carga é dividir o trabalho igualmente entre as tarefas fazendo com que estejam ocupadas o tempo todo. Em programação paralela aumenta-se a complexidade para balancear a carga durante o processamento, dividir o trabalho igualmente para aumentar o desempenho de um sistema, tirando proveito de todo o poder computacional disponível para a execução de processos.

Na estratégia de Força Bruta descrita no tópico 2.2 o balanceamento de carga é feito de uma maneira mais simples, pois cada interação de um laço de repetição é dividido pela quantidade de núcleos de processamento disponíveis. Sendo assim o núcleos sempre estão ocupados com alguma tarefa, melhorando o balanceamento.

A estratégia de Divisão e Conquista descrita no tópico 2.3, sendo uma estrutura baseada em árvores e recursão, dividindo uma tarefa em tarefas menores dificulta obter um balanceamento adequado durante a divisão de tarefas (HWANG, 1998). Grande parte das estratégias de divisão e conquista utilizam técnicas de balanceamento.

CONCLUSÃO

O objetivo do presente artigo se cumpriu mediante ao fato de que foi possível analisar em detalhes a diferença entre duas estratégias para a resolução do problema proposto, através de medições de desempenho para as versões sequenciais e paralelas, e exposição dos resultados numéricos e montagem de gráficos de desempenho. Também foi notável a diferença entre as estratégias de paralelização adotadas utilizando sections e paralelismo de laço de repetição, destacando-se a limitação apresentada pela estratégia de sections em relação ao número de threads, que está diretamente relacionado a quantidade de sections paralelos.

AGRADECIMENTO(S)

Um agradecimento especial ao professor orientador da disciplina, que produziu e disponibilizou diversos materiais em texto e vídeo para guiar e dar praticidade no desenvolvimento do presente trabalho de forma totalmente remota e assíncrona.

REFERÊNCIAS

DE LIMA MARTINS, Simone. Programação Paralela. 2002.

HWANG, Kai. Scalable Parallel Computing.1998.

ANDREWS, G. R. Foundations of Multithreaded, Parallel, and Distributed Programming. 2. ed. Addison-Wesley, 2006. 361 p.

QUINN, M. J. Parallel Programming in C with MPI and OpenMP. McGraw Hill, 2004. 529 p.