TRABALHO 1 DE PROGRAMAÇÃO PARALELA – OPENMP[1]

Gabriel Ferreira Kurtz[2] <gabriel.kurtz@acad.pucrs.br>
Prof. Roland Teodorowitsch[3] <roland.teodorowitsch@pucrs.br> – Orientador

Pontifícia Universidade Católica do Rio Grande do Sul – Faculdade de Informática – Curso de Ciência da Computação
Av. Ipiranga, 6681 Prédio 32 Sala 505 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

15 de outubro de 2020

RESUMO

Este é o primeiro trabalho da disciplina de Programação Paralela do curso de Engenharia de Software da PUCRS, realizado durante o segundo semestre letivo de 2020. Busca utilizar o OpenMP para paralelizar um algoritmo de distância mínima entre dois pontos, e analisar os efeitos da paralelização.

Palavras-chave: Programação Paralela; OpenMP; Trabalho Acadêmico.

ABSTRACT

Title: *"Parallel Programming Task 1 – Minimum distance between two points"*

This paper is an academic works in paper format for the Parallel Programming discipline of the Software Engineering course at Pontificia Universidade Católica do Rio Grande do Sul (PUCRS – Pontifical Catholic University of Rio Grande do Sul). It employs OpenMP to parallelize a simple minimum distance algorithm, analyzing the effects of the parallelization.

Key-words: *Parallel Programming; OpenMP; Academic Work..*

1 INTRODUÇÃO

Este trabalho consiste da edição, execução e análise de dois algoritmos de cálculo da menor distância entre dois pontos, sendo um no paradigma da força bruta e outro utilizando divisão e conquista. Os algoritmos originais eram sequenciais. Foram aplicados alguns comandos básicos utilizando a API do OpenMP, na linguagem C, de modo a paralelizar o processamento dos algoritmos utilizando múltiplas threads, analisando os resultados como tempo de execução, speedup e eficiência.

A execução ocorreu no cluster Grad da PUCRS, utilizando um nodo de 16 threads em cada caso (nem sempre o mesmo nodo, porém maquinas com processamento dedicado a isto). Para cada algoritmo, foram contabilizados os tempos das execuções sequenciais (1 thread) e paralelas (2, 4, 8 e 16 threads). Os resultados das execuções foram comparados para demonstrar que o algoritmo ainda funciona corretamente após as modificações.

A partir dos resultados dos tempos de execução, foram gerados gráficos e uma breve análise.

2 ALGORITMO DE FORÇA BRUTA

O algoritmo de força bruta faz um cálculo da distância entre todos os pontos, buscando encontrar a distância mínima. Por esta razão, sua complexidade é $O(n^2)$.

---

[1]Trabalho realizado para a disciplina de Programação Paralela do curso de Engenharia de Software da PUCRS.

[2]Professor das disciplinas de Introdução à Ciência da Computação e Programação Distribuída do curso de Ciência da Computação, e da disciplina de Sistemas Distribuídos do curso de Sistemas de Informação, da Faculdade de Informática da PUCRS.

[3]Aluno do curso de Engenharia de Software da PUCRS.

2.1 Código

Foram acrescentados alguns comandos do OpenMP ao código original. Optei por utilizar o ‘parallel for’ no laço principal do programa. Como a variável ‘i’ já era privada por ser o índice do laço externo, foi necessário proteger as variáveis ‘j’, índice do laço interno, e ‘d’, da distância calculada no laço interno de modo a evitar seu compartilhamento entre as threads. Após alguns testes locais, o scheduler que apresentou melhores resultados foi o modo ‘dynamic’, com chunk size de 200, embora estas alternativas tenham feito pouca diferença.

Optei por criar uma seção crítica dentro do trecho que verifica se a distância é a nova mínima. Coloquei os comandos OpenMP dentro do ‘if (d < min_d)’ pois, fora dele, impediria o acesso simultâneo das threads a verificar se seu ‘d’ é o novo mínimo, de modo que ficou bastante lento. Dentro da seção crítica, adicionei um novo ‘if (d < min_d)’, para não correr o risco de haver um acesso simultâneo onde duas threads tem um ‘d’ menor que o mínimo, e o valor maior acaba setando após o primeiro tendo os dois passado o ‘if’.

Por exemplo, se o mínimo atual é 10, a thread A calcula o ‘d’ com valor 5 e a thread B calcula ‘d’ com valor 6. A meu ver, se o ‘if (d < min_d)’ está fora da seção crítica, seria possível ambas verificarem concorrentemente que são o novo mínimo, de modo que B poderia sobrescrever o valor de A.

```c
/* min-dist-bf.c (Roland Teodorowitsch; 17 Sep. 2020)
 * Compilation: gcc min-dist-bf.c -o min-dist-bf -fopenmp -lm
 * Note: Includes some code from the sequential solution of the
 *       "Closest Pair of Points" problem from the
 *       14th Marathon of Parallel Programming avaiable at
 *       http://lspd.mackenzie.br/marathon/19/points.zip
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <omp.h>

#define SIZE 100000
#define START 10000
#define STEP  10000

#define EPS 0.00000000001

typedef struct {
    double x;
    double y;
} point_t;

point_t points[SIZE];

unsigned long long llrand() {
    unsigned long long r = 0;
    for (int i = 0; i < 5; ++i)
        r = (r << 15) | (rand() & 0x7FFF);
    return r & 0xFFFFFFFFFFFFFFFFULL;
```

```
}

void points_generate(point_t *points, int size, int seed) {
    int p, i, found;
    double x, y;
    srand(seed);
    p = 0;
    while (p<size) {
        x = ((double)(llrand() % 20000000000) - 10000000000) / 1000.0;
        y = ((double)(llrand() % 20000000000) - 10000000000) / 1000.0;
        if (x >= -10000000.0 && x <= 10000000.0 && y >= -
10000000.0 && y <= 10000000.0) {
            points[p].x = x;
            points[p].y = y;
            p++;
        }
    }
}

double points_distance_sqr(point_t *p1, point_t *p2) {
    double dx, dy;
    dx = p1->x - p2->x;
    dy = p1->y - p2->y;
    return dx*dx + dy*dy;
}

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) schedule(dynamic, 200)
    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) {
                #pragma omp critical
                if (d < min_d) min_d = d;
            }

        }
    }
    return sqrt(min_d);
}

int main() {
    int i;
    double start, finish;
```

```
    points_generate(points,SIZE,0);
    for (int i=START; i<=SIZE; i+=STEP) {
        start = omp_get_wtime();
        printf("%.6lf\n", points_min_distance_bf(points,i));
        finish = omp_get_wtime();
        fprintf(stderr,"%d %lf\n",i,finish-start);
    }
    return 0;
}
```

Figura 1 – Código-fonte do algoritmo de força bruta

2.2 Resultados

A seguir estão os resultados obtidos com a paralelização do algoritmo de força bruta.

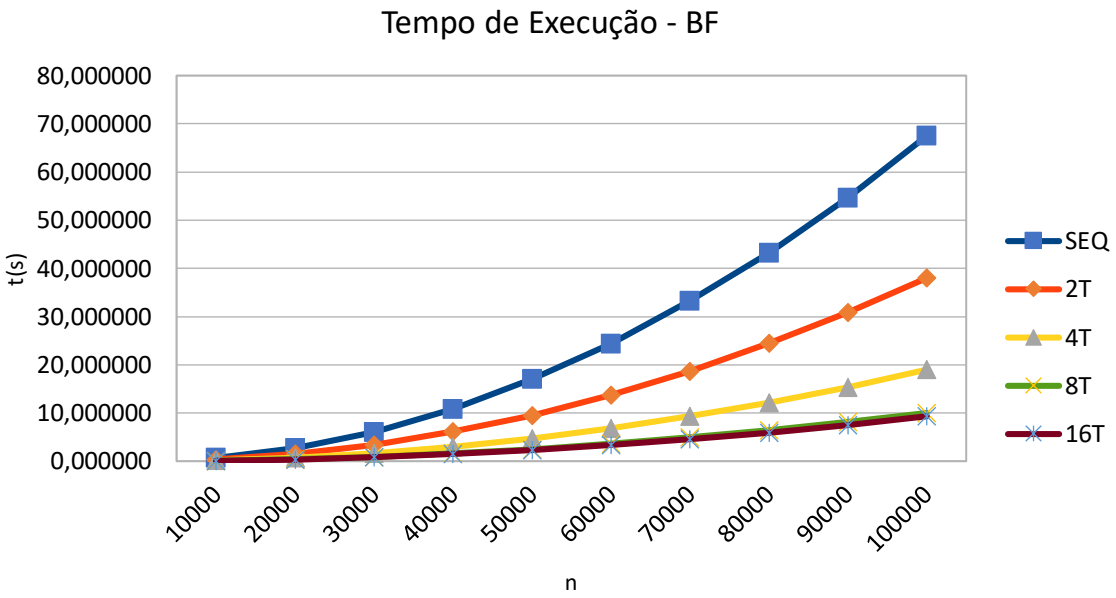

Figura 2 – Tempos de execução no algoritmo de força bruta

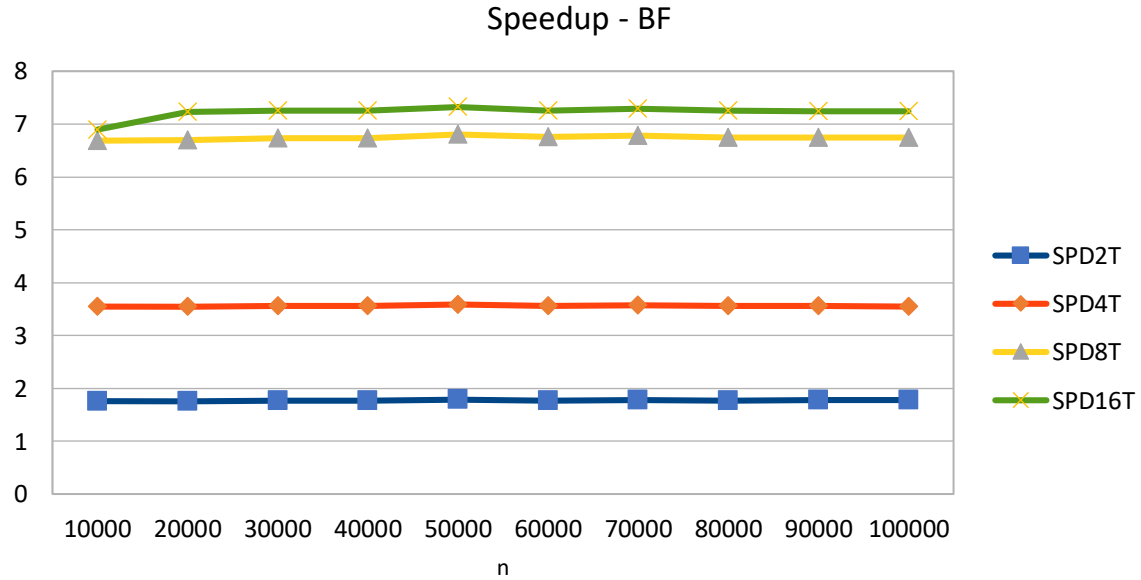

Figura 3 – Speedups no algoritmo de força bruta

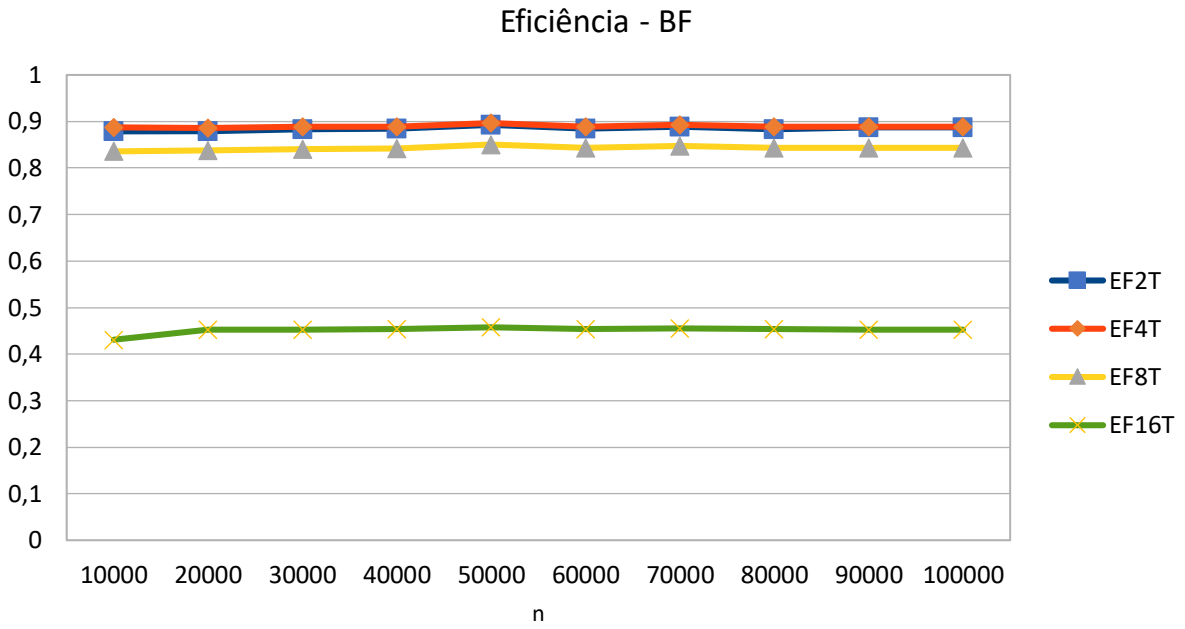

Figura 4 – Eficiências no algoritmo de força bruta

2.3 Análise

Podemos verificar que houve uma eficiência muito grande (próxima a 100%) com 2, 4 e 8 threads, de modo que o tempo de processamento melhorou bastante com o acréscimo de threads até este ponto. Na execução com 16 threads, a melhora do speedup foi bem pouca em relação a 8, e a taxa de eficiência caiu bastante refletindo isto, provavelmente pelo fato do overhead para paralelizar as tarefas ser maior que o ganho a partir de determinado ponto.

3 Algoritmo de Divisão e Conquista

O algoritmo de divisão e conquista utiliza algumas chamadas recursivas para reduzir progressivamente a área a ser calculada, de acordo com a distância mínima atual, de modo a otimizar a busca pela distância mínima. Assim, é capaz de lidar com tamanhos maiores de 'n' (total de pontos) com tempos de processamentos similares.

3.1 Código

Nesta paralelização, optei por utilizar os comandos 'parallel'/'sections'/'section' para alocar novas threads a cada chamada recursiva. Seria possível criar a mesma lógica usando 'parallel'/'single'/'task' com algumas mudanças, porém não houve mudança significativa de tempos de execução, portanto optei pelo que considerei mais simples.

Sería possível, adicionalmente, alocar threads com o 'parallel for' de forma similar ao que foi feito na força bruta, porém isso não causou nenhuma melhora simultaneamente à paralelização dos setores recursivos. Removendo os setores recursivos, houve uma pequena melhora assim. Optei por manter somente a paralelização da recursão.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <omp.h>
#include <algorithm>

#define SIZE 10000000
#define START 1000000
#define STEP  1000000

#define EPS 0.00000000001
#define BRUTEFORCESSIZE 200

using namespace std;

typedef struct {
    double x;
    double y;
} point_t;

point_t points[SIZE];
point_t border[SIZE];

unsigned long long llrand() {
    unsigned long long r = 0;
    int i;
    for (i = 0; i < 5; ++i)
        r = (r << 15) | (rand() & 0x7FFF);
    return r & 0xFFFFFFFFFFFFFFFFULL;
}

void points_generate(point_t *points, int size, int seed) {
    int p, i, found;
    double x, y;
    srand(seed);
    p = 0;
    while (p<size) {
        x = ((double)(llrand() % 20000000000) - 10000000000) / 1000.0;
        y = ((double)(llrand() % 20000000000) - 10000000000) / 1000.0;
        if (x >= -10000000.0 && x <= 10000000.0 && y >= -
10000000.0 && y <= 10000000.0) {
            points[p].x = x;
            points[p].y = y;
            p++;
        }
    }
}
```

```
bool compX(const point_t &a, const point_t &b) {
    if (a.x == b.x)
        return a.y < b.y;
    return a.x < b.x;
}

bool compY(const point_t &a, const point_t &b) {
    if (a.y == b.y)
        return a.x < b.x;
    return a.y < b.y;
}

double points_distance_sqr(point_t *p1, point_t *p2) {
    double dx, dy;
    dx = p1->x - p2->x;
    dy = p1->y - p2->y;
    return dx*dx + dy*dy;
}

double points_min_distance_dc(point_t *point,point_t *border,int l, int r) {
    double minDist = DBL_MAX;
    double dist;
    int i, j;
    if (r-l+1 <= BRUTEFORCESSIZE) {
        for (i=l; i<r; i++){
            for (j = i+1; j<=r; j++) {
                dist = points_distance_sqr(point+i, point+j);
                if (dist<minDist) {
                    minDist = dist;
                }
            }
        }
        return minDist;
    }

    int m = (l+r)/2;
    double dL, dR;

    #pragma omp parallel
    {
        #pragma omp 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);

    int k = l;
    for(i=m-1; i>=l && fabs(point[i].x-point[m].x)<minDist; i--)
        border[k++] = point[i];
    for(i=m+1; i<=r && fabs(point[i].x-point[m].x)<minDist; i++)
        border[k++] = point[i];

    if (k-l <= 1) return minDist;

    sort(&border[l], &border[l]+(k-l), compY);

    for (i=l; i<k; i++) {
        for (j=i+1; j<k && border[j].y - border[i].y < minDist; j++) {
            dist = points_distance_sqr(border+i, border+j);
            if (dist < minDist)
                minDist = dist;
        }
    }

    return minDist;
}

int main() {
    int i;
    double start, finish;

    points_generate(points,SIZE,11);
    sort(&points[0], &points[SIZE], compX);
    for (i=START; i<=SIZE; i+=STEP) {
        start = omp_get_wtime();
        printf("%.6lf\n", sqrt(points_min_distance_dc(points,border,0,i-1)));
        finish = omp_get_wtime();
        fprintf(stderr,"%d %lf\n",i,finish-start);
    }
    return 0;
}
```

Figura 5 – Código-fonte do algoritmo de divisão e conquista

3.2 Resultados

Seguem os resultados do algoritmo de divisão e conquista:

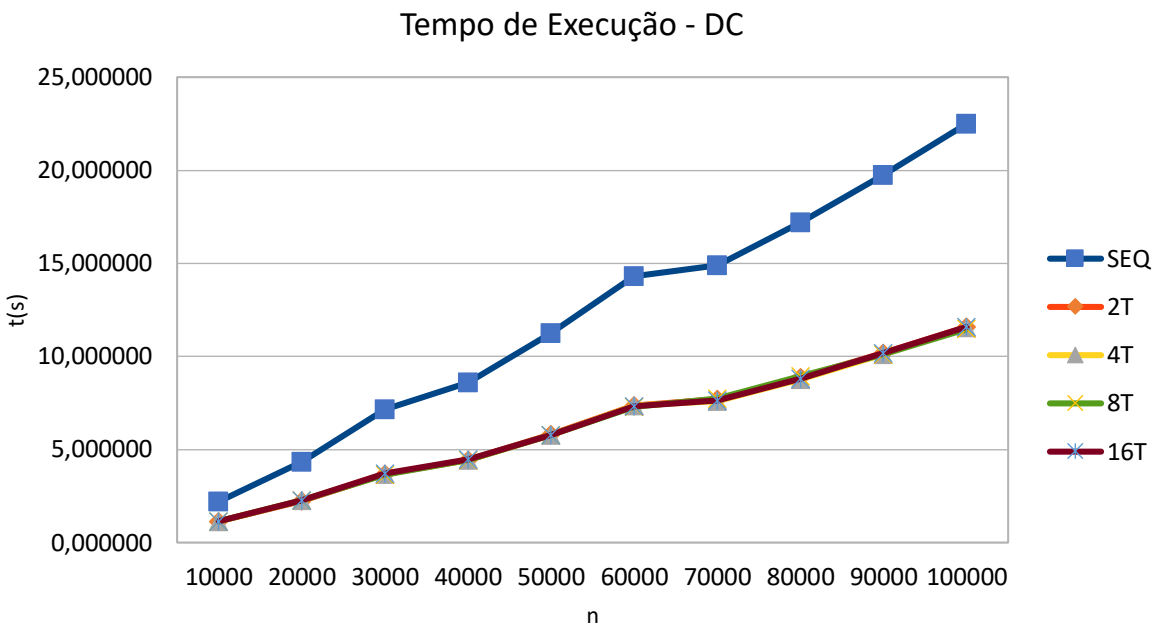

Figura 6 – Tempos de execução no algoritmo de divisão e conquista

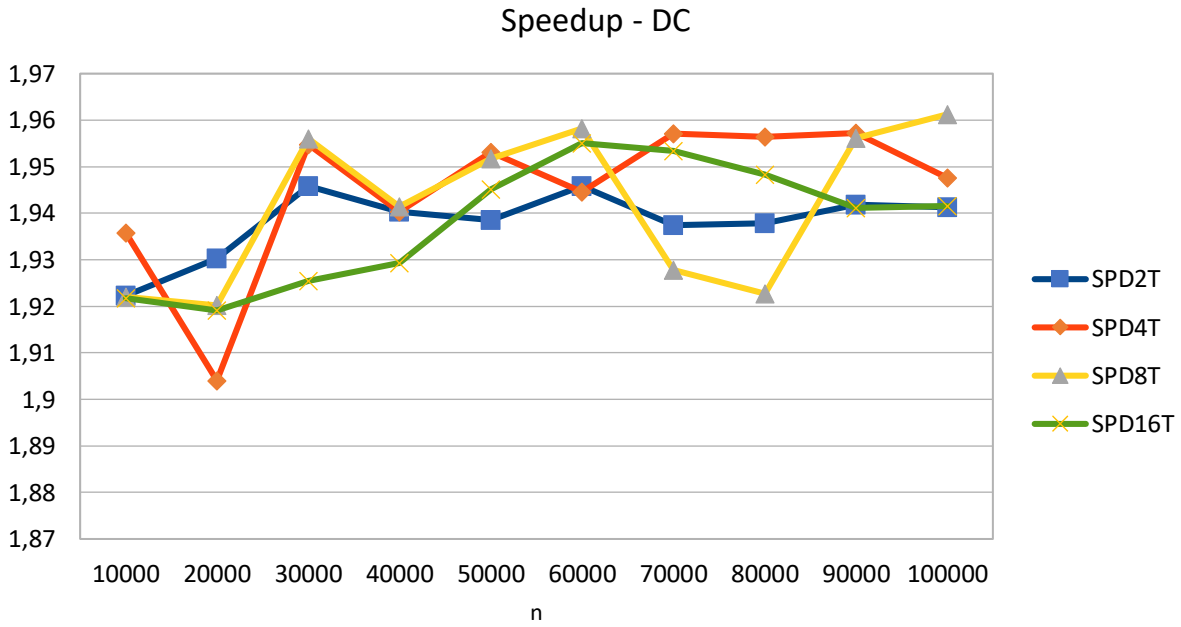

Figura 7 – Speedups no algoritmo de divisão e conquista

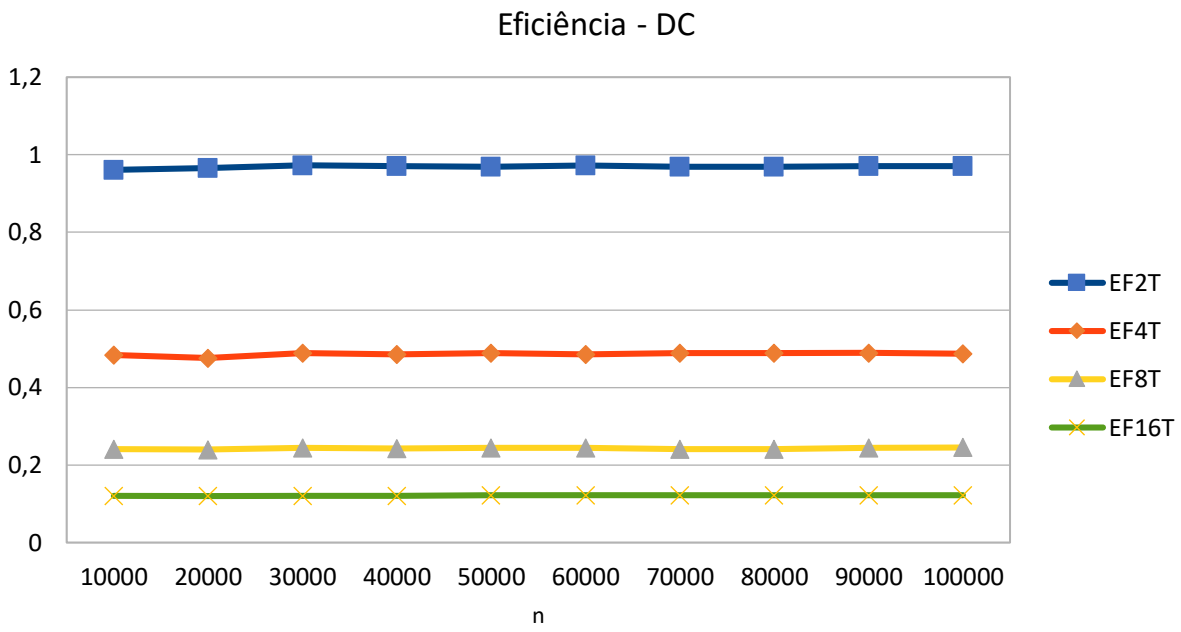

Figura 8 – Eficiências no algoritmo de divisão e conquista

3.3 Análise

No caso do algoritmo de divisão e conquista, somente fui capaz de obter boa eficiência na paralelização com 2 threads. Entre 4 e 16 threads, o ganho de tempo foi bastante baixo, sendo que em algumas execuções seus speedups ficaram abaixo do speedup de 2 threads em alguns valores de n, o que demonstra que talvez a forma de paralelizar que encontrei não seja a ideal, ou que esta solução não seja tao propícia a paralelizações com mais threads.

4 EXECUÇÕES E BATCHJOBS

Ambos Batchjobs que utilizei seguem a mesma estrutura, baseada nos exemplos vistos em aula:

```
#!/bin/bash

################-> are comments
################-> "#PBS" are Batch Script commands

#PBS -m abe

################ Verbose mode

#PBS -V

################

################ Change these parameters according to your requisites

#PBS -l nodes=1:ppn=16:cluster-Grad,walltime=00:20:00

################ Where:
```

```
################ nodes = number of nodes requested
################ ppn = number of cores per node
################ cluster-Atlantica / cluster-Gates = cluster name
################ walltime = max allocation time

################ Please, change this e-mail address to yours

#PBS -M gabriel.kurtz@acad.pucrs.br

################

#PBS -r n

################ Output options

#PBS -j oe

################

################ Please, change this directory to your working dir.

#PBS -d /home/pp12708/Programacao-Paralela/T1

################

################
echo Running on host `hostname`
echo
echo Initial Time is `date`
echo
echo Directory is `pwd`
echo
echo This jobs runs on the following nodes:
echo `cat $PBS_NODEFILE | uniq`
echo
echo JOB_ID:
echo `echo $PBS_JOBID`
echo ################

############# If running a sequential or openMP program

:> saida-bf.txt ; :> tempos-bf.txt

################ Sequencial
echo --- Sequencial ---
echo --- Sequencial --- >> saida-bf.txt
echo --- Sequencial --- >> tempos-bf.txt
./min-dist-bf-sequencial >> saida-bf.txt 2>> tempos-bf.txt

################ 2 Threads
```

```
export OMP_NUM_THREADS=2
echo --- 2 Threads ---
echo --- 2 Threads --- >> saida-bf.txt
echo --- 2 Threads --- >> tempos-bf.txt
./min-dist-bf >> saida-bf.txt 2>> tempos-bf.txt

############### 4 Threads
export OMP_NUM_THREADS=4
echo --- 4 Threads ---
echo --- 4 Threads --- >> saida-bf.txt
echo --- 4 Threads --- >> tempos-bf.txt
./min-dist-bf >> saida-bf.txt 2>> tempos-bf.txt

############### 8 Threads
export OMP_NUM_THREADS=8
echo --- 8 Threads ---
echo --- 8 Threads --- >> saida-bf.txt
echo --- 8 Threads --- >> tempos-bf.txt
./min-dist-bf >> saida-bf.txt 2>> tempos-bf.txt

############### 16 Threads
export OMP_NUM_THREADS=16
echo --- 16 Threads ---
echo --- 16 Threads --- >> saida-bf.txt
echo --- 16 Threads --- >> tempos-bf.txt
./min-dist-bf >> saida-bf.txt 2>> tempos-bf.txt




###############

echo Final Time is `date`
```

Figura 9 – Batchjob

Os resultados são enviados ao ‘saida-bf.txt’ pela saída padrão, para verificar o funcionamento correto dos algoritmos, enquanto os tempos são anotados no ‘tempos-bf.txt’ através do ‘stderr’. Todos algoritmos utilizados apresentaram os mesmos resultados na execução sequencial e em todas execuções paralelas.

4.1 Utilização das máquinas

Procurei utilizar as máquinas com responsabilidade, usando um ‘walltime’ de 20 minutos, de modo que fiz boa parte do trabalho em modo local utilizando as 8 threads do meu PC. Nas execuções realizadas no cluster, não levei mais de 2 horas no total das execuções, sempre utilizando somente 1 cluster, em modo exclusivo, eventualmente com 2 jobs simultâneos (um para cada algoritmo). Analisei as filas e acredito ter sido respeitoso com os colegas que estavam utilizando.

4.2 Repositório

Para a troca de dados entre minha máquina local e o cluster, utilizei um repositório no GitHub que pode ser encontrado em https://github.com/gabrielkurtz/Programacao-Paralela

5 CONCLUSÃO

Foi possível verificar algumas das possibilidades da programação paralela para melhorar a performance de algoritmos, bem como conhecer algumas das dificuldades relacionadas à sua aplicação. Aprendi bastante do sistema Linux, tendo instalado ele recentemente na minha máquina local, e de como fazer scripts, além de ter sido uma experiência interessante com o uso de clusters.