TRABALHO 2 DE PROGRAMAÇÃO PARALELA – MPI[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 Engenharia de Software
Av. Ipiranga, 6681 Prédio 32 Sala 505 – Bairro Partenon – CEP 90619-900 – Porto Alegre – RS

26 de novembro 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 MPI para paralelizar um algoritmo de distância mínima entre dois pontos com divisão e conquista, e analisar os efeitos da paralelização.

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

ABSTRACT

Title: *"Parallel Programming Task 2 – Minimum distance between two points"*

This paper is an academic work in paper format for the Parallel Programming discipline of the Software Engineering course at Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS – Pontifical Catholic University of Rio Grande do Sul). It employs MPI to parallelize a divide-and-conquer minimum distance between two points algorithm, analyzing the effects of parallelization.

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

INTRODUÇÃO

Este trabalho consiste da edição, execução e análise de um algoritmos de cálculo da menor distância entre dois pontos utilizando a estratégia de divisão e conquista. O algoritmo original era sequencial e a tarefa consistia em paralelizá-lo com MPI. Foram aplicados alguns comandos básicos utilizando a API do MPI, na linguagem C++, de modo a paralelizar o processamento dos algoritmos utilizando múltiplos processos, analisando os resultados obtidos.

A execução ocorreu no cluster Grad da PUCRS, utilizando nodos de 16 threads em cada caso (nem sempre os mesmos nodos, porém maquinas com processamento dedicado às tarefas). Foram comparados os tempos de execução com 1, 7, 15, 31 e 63 processos em 1 nodo, 15, 31 e 63 processos em 2 nodos e 63 processos em 4 nodos. 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.

A ESTRATÉGIA DE DIVISÃO E CONQUISTA NO PROBLEMA DA MENOR DISTÂNCIA ENTRE DOIS PONTOS

O problema da menor distância entre dois pontos ocorre em um vetor contendo pontos com coordenadas X,Y, formando um plano de 2 dimensões onde os pontos estão espalhados. Deseja-se encontrar os dois pontos mais próximos entre todo o conjunto. A estratégia da divisão e conquista evita a necessidade de verificar a distância entre todos os pontos do conjunto ($O(n^2)$), tornando o algoritmo $O(n.\log(n))$ usando uma estratégia recursiva.

Na fase de divisão, os pontos são ordenados por sua posição no eixo X e divididos em 2 grupos com o mesmo número de elementos (metade do vetor para cada lado). A partir daí, aplica-se recursivamente a estratégia em cada metade para encontrar a menor distância entre dos pontos de

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

2 Aluno do curso de Engenharia de Software da PUCRS.

3 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.

cada conjunto gerado, ou seja, cada lado é dividido novamente em dois, até chegar-se a um tamanho pequeno suficiente onde vale a pena verificar a menor distância pela estratégia de força bruta.

Na fase de conquista, verifica-se qual a menor distância considerando ambos os lados. Existe ainda a possibilidade da menor distância estar entre pontos que ficaram em grupos diferentes (um de cada lado). Executa-se um pequeno cálculo com os pontos onde existe esta possibilidade, garantindo o funcionamento da estratégia.

SOLUÇÃO IMPLEMENTADA

O enunciado solicitava que fossem implementadas soluções capazes de separar a pipeline de processos em 3, 4 e 5 níveis (além da execução sequencial), utilizando o processo raiz, processos intermediários e processos-folha. Optei por implementar uma solução mais simples, porém bastante eficiente na paralelização, onde todos os processos executam igualitariamente as fases de divisão e conquista, obtendo assim um bom balanceamento de carga. O processo raiz, além de executar a divisão e conquista, também imprime as informações necessárias. Optei também por manter o número de processos solicitado(1, 7, 15, 31, 63), mesmo sobrando uma Thread ociosa em cada execução.

Na fase de divisão, o vetor de pontos é separado em tamanhos iguais, com um número de bags igual ao número de processos. O último processo pode receber um tamanho um pouco maior, caso o total de pontos não seja divisível pelo número de processos.

Em cada um destes conjuntos de pontos, é executado o algoritmo recursivo sequencial da divisão e conquista, no processo responsável por aquele pedaço, encontrando o mínimo entre aqueles pontos. Posteriormente, é efetuada uma redução nos mínimos encontrados por cada processo para encontrar o mínimo total da fase de divisão.

A fase de conquista inicia buscando menores distâncias entre pontos que ficaram em processos adjacentes. São redefinidos os limites de início e fim (L/R) de cada processo para considerar apenas os pontos cuja distância no eixo X entre o ponto e a fronteira é menor do que a distância mínima da fase de divisão. Cada processo executa isto na sua fronteira da esqueda, de modo que o processo 0 não executa esta fase (o número de processos é 1 menor do que o número de fronteiras).

Com os novos limites L/R, é executado novamente o algoritmo recursivo sequencial em cada um dos processos. É feita uma nova redução com os mínimos encontrados e, finalmente, compara-se a distância mínima da fase de conquista com a mínima da fase de divisão, mantendo-se a menor como o resultado do algoritmo (menor distância entre dois pontos do vetor inteiro original).

CÓDIGO

O código a seguir reflete a explicação anterior do problema. Comentei alguns trechos onde imprimia algumas informações para ajudar a debugar o programa, de modo a manter as saídas anexadas como foram solicitadas, porém permitindo ativar os comentários novamente. Há alguns comentários indicando também qual parte do código faz cada parte da solução descrita.

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


#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, int id) {
    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 = points_min_distance_dc(point,border,l,m, id);
    double dR = points_min_distance_dc(point,border,m,r, id);
    // printf("%d) dL - dR: %.2lf %.2lf\n", id, dL, dR);
    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 argc, char *argv[]) {
        int i, p, id;
        double elapsed_time;

        int raiz=0;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        // if(id==raiz) printf("\n%d) Comm size: %d\n", id, p);
        MPI_Barrier(MPI_COMM_WORLD);
        // printf("%d) Comm rank: %d\n", id, id);

        points_generate(points,SIZE,11);
        sort(&points[0], &points[SIZE], compX);

        for (i=START; i<=SIZE; i+=STEP) {
            elapsed_time = -MPI_Wtime();
            // if(id==raiz) printf("%d) ------ Size: %d ------\n", id, i);
            MPI_Barrier(MPI_COMM_WORLD);

            // DIVISÃO
            // Divide vetor de pontos em setores(bags) de tamanhos "iguais" para cada thread
            int sector_size = i/p;
            // printf("%d) Tamanho setor: %d\n", id, sector_size);
            int start = id*sector_size;
            int end = (i - (p-id-1)*sector_size - 1);
            // printf("%d) Start - End: %d %d\n", id, start, end);

            // Executa algoritmo recursivo de divisão e conquista no setor alocado
            // (Resultado nao considera mínimos entre dois pontos de setores diferentes)
            double minDist_divisao = sqrt(points_min_distance_dc(points,border,start,end, id));
            double minDist_divisao_reduzido;

            // printf("%d) Minimo Divisao: %.6lf\n", id, minDist_divisao);
            // Reduz valor do mínimo, informando todas as threads do mínimo encontrado
              MPI_Allreduce(&minDist_divisao, &minDist_divisao_reduzido, 1, MPI_DOUBLE, MPI_MIN,
MPI_COMM_WORLD);
            // printf("%d) Minimo divisao reduzido: %.6lf\n", id, minDist_divisao_reduzido);


            // CONQUISTA
```

```
                double minDist_conquista = minDist_divisao;
                double minDist_conquista_reduzido = minDist_divisao_reduzido;
                // Fronteira da esquerda é verificada, por isso não verifica id=0
                // (Seria fronteira com o início do vetor)
                if(id != 0) {
                    // Atribui novo limite entre bag da thread e bag da "esquerda"
                    end = start;
                    // Verifica pontos cuja distancia no eixo X sao menores que minimo atual
                    // (Podem gerar novo mínimo com 2 pontos que estavam em bags diferentes)
                    double limite_x_l = points[start].x - minDist_divisao_reduzido;
                    double limite_x_r = points[end].x + minDist_divisao_reduzido;
                    while(points[start].x > limite_x_l && start > 0) start--;
                    while(points[end].x < limite_x_l && end < SIZE-1) end++;

                    minDist_conquista = sqrt(points_min_distance_dc(points,border,start,end, id));
                    // printf("%d) Minimo Conquista: %.6lf\n", id, minDist_conquista);
                }

                MPI_Barrier(MPI_COMM_WORLD);
                    MPI_Allreduce(&minDist_conquista, &minDist_conquista_reduzido, 1, MPI_DOUBLE,
MPI_MIN, MPI_COMM_WORLD);

                double resultadoFinal = minDist_conquista_reduzido < minDist_divisao_reduzido ?
minDist_conquista_reduzido : minDist_divisao_reduzido;
                    // if(id == raiz) printf("%d) Minimo Conquista reduzido: %.6lf\n", id,
minDist_conquista_reduzido);
                // if(id == raiz) printf("%d) Resultado Final: %.6lf\n", id, resultadoFinal);
                if(id == raiz) printf("%.6lf\n", resultadoFinal);

                double elapsed_time_reduzido;
                    MPI_Reduce(&elapsed_time, &elapsed_time_reduzido, 1, MPI_DOUBLE, MPI_MAX, raiz,
MPI_COMM_WORLD);
                elapsed_time += MPI_Wtime();
                if(id == raiz) fprintf(stderr,"%d %lf\n",i,elapsed_time);
                MPI_Barrier(MPI_COMM_WORLD);
            }

            MPI_Finalize();
            return 0;
        }
```

RESULTADOS

A seguir estão os resultados obtidos.

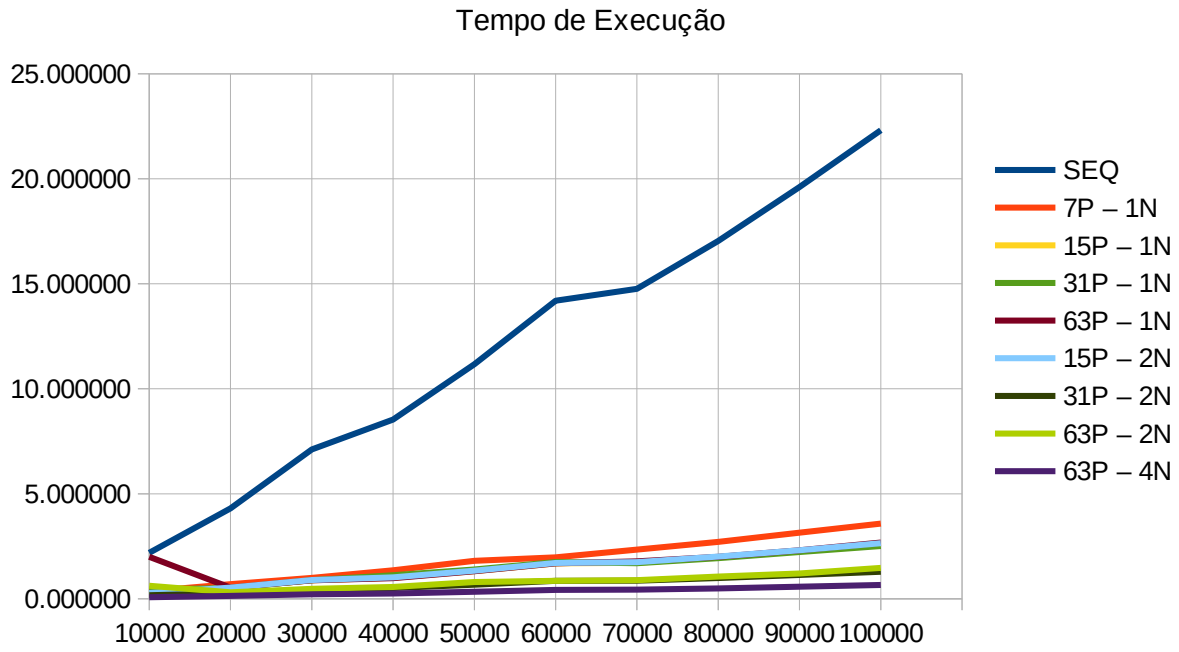

Figura 1 – Tempo de execução

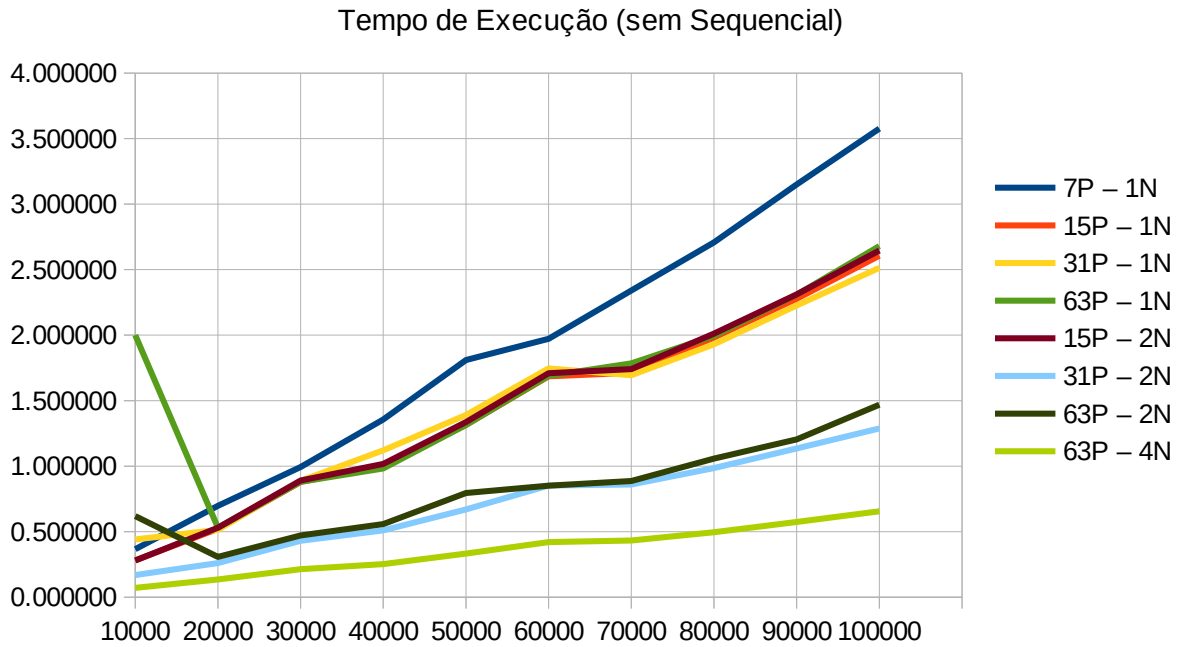

Figura 2 – Tempo de execução sem sequencial

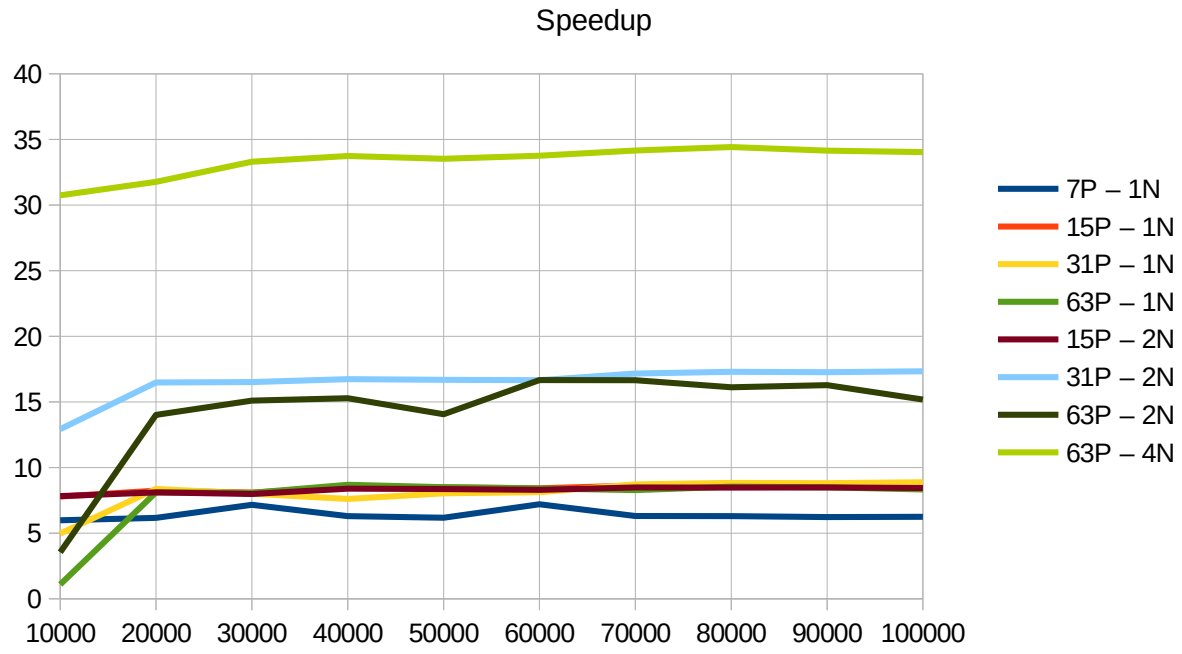

Figura 3 – Speedup

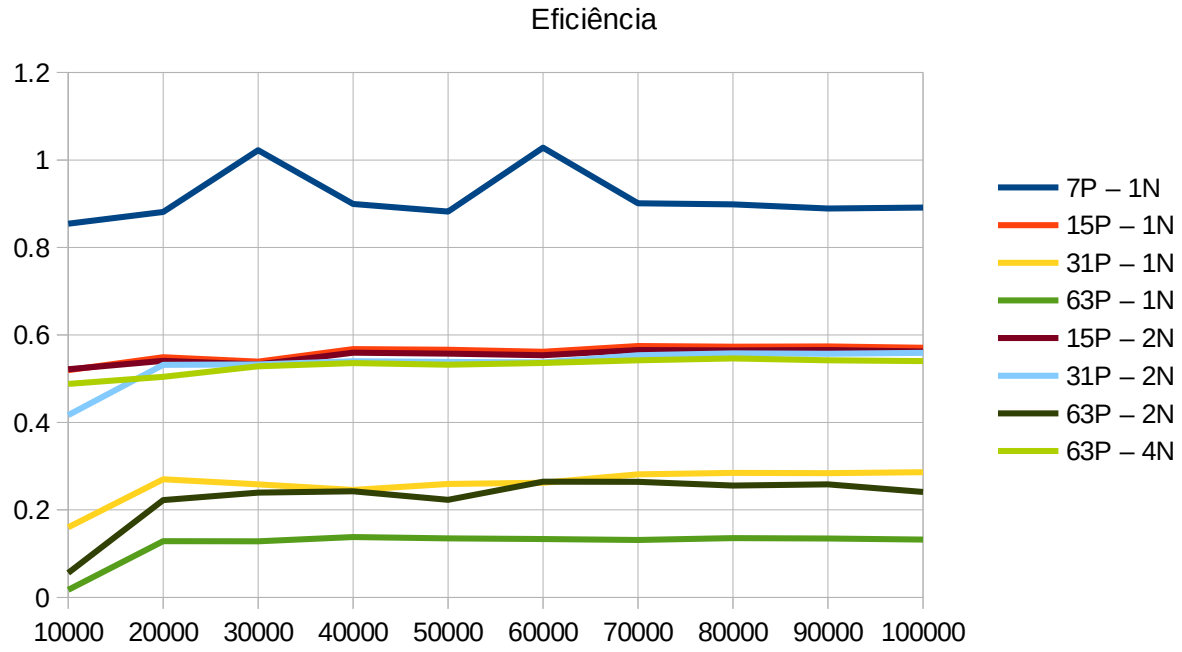

Figura 4 – Eficiência

ANÁLISE

Como cada nodo possui 16 threads, podemos ver que a eficiência ficou limitada nos casos em que havia mais do que 16 processos por nodo. Nos demais casos, a eficiência foi boa. Na execução com 7 processos, ficou próxima de 100% e, nos demais casos em que não houve limitação de threads por processo, ficou entre 55% e 60% mesmo no caso com mais processos (63 processos/4 nodos). Isto significa quase 35 vezes mais veloz do que o sequencial neste caso, conforme o gráfico de speedup. Aparentemente, seria possível paralelizar o algoritmo em ainda

mais processos seguindo esta mesma estratégia sem ser limitado ainda pela perda de eficiência.

Acredito que o balanceamento de carga foi bom. Todos os processos tinham responsabilidades semelhantes e trabalhavam com conjuntos de dados de tamanhos praticamente iguais, sendo que somente o processo raiz tinha pequenas responsabilidades de imprimir as saídas.

EXECUÇÕES E BATCHJOBS

Utilizei um batchjob para cada configuração de nodos, executando em cada um deles todas as configurações de número de processos necessárias para aqueles nodos e imprimindo as saídas de print e stderr no mesmo formato do algoritmo original. A seguir, o batchjob de 1 nodo:

```
#!/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/T2

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

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

echo --- Iniciando Execução com 1 Nodo --- >> saida-dc.txt ; >> tempos-dc.txt

################ Sequencial
echo --- Execução Sequencial em 1 Nodo ---
echo --- Execução Sequencial em 1 Nodo --- >> saida-dc.txt
echo --- Execução Sequencial em 1 Nodo --- >> tempos-dc.txt
./min-dist-sequencial >> saida-dc.txt 2>> tempos-dc.txt

################ Paralela 7 Processos
echo --- Execução com 7 Processos em 1 Nodo ---
echo --- Execução com 7 Processos em 1 Nodo --- >> saida-dc.txt
echo --- Execução com 7 Processos em 1 Nodo --- >> tempos-dc.txt
mpirun -np 7 min-dist-paralelo >> saida-dc.txt 2>> tempos-dc.txt

################ Paralela 15 Processos
echo --- Execução com 15 Processos em 1 Nodo ---
echo --- Execução com 15 Processos em 1 Nodo --- >> saida-dc.txt
echo --- Execução com 15 Processos em 1 Nodo --- >> tempos-dc.txt
mpirun -np 15 min-dist-paralelo >> saida-dc.txt 2>> tempos-dc.txt

################ Paralela 31 Processos
echo --- Execução com 31 Processos em 1 Nodo ---
echo --- Execução com 31 Processos em 1 Nodo --- >> saida-dc.txt
echo --- Execução com 31 Processos em 1 Nodo --- >> tempos-dc.txt
```

```
mpirun -np 31 min-dist-paralelo >> saida-dc.txt 2>> tempos-dc.txt

################ Paralela 63 Processos
echo --- Execução com 63 Processos em 1 Nodo ---
echo --- Execução com 63 Processos em 1 Nodo --- >> saida-dc.txt
echo --- Execução com 63 Processos em 1 Nodo --- >> tempos-dc.txt
mpirun -np 63 min-dist-paralelo >> saida-dc.txt 2>> tempos-dc.txt

echo Final Time is `date`
```

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 Grad, não levei mais de 2 horas no total das execuções, utilizando entre 1 e 4 nodos em modo exclusivo. Na prática, cada execução durou em torno de 4 minutos e, nos casos que houve problema, removi as execuções com o comando “qdel”. Analisei as filas e acredito ter sido respeitoso com os colegas que estavam utilizando.

Cada cenário foi executado no mínimo 4 vezes, considerando sempre o menor tempo obtido, conforme pode ser verificado na tabela que consta no arquivo ZIP enviado.

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

CONCLUSÃO

O algoritmo desenvolvido é mais simples do que foi solicitado no enunciado. Não fui capaz de compreender plenamente como implementar a solução proposta, porém acredito que entreguei um algoritmo eficiente, performático, capaz de encontrar as soluções corretas em todos os casos e que permitiu fazer as análises solicitadas, ajudando no aprendizado da disciplina e dos conceitos básicos de MPI e Programação Paralela.

Ficou demonstrado que o MPI é uma ferramenta bastante poderosa para obter maior performance computacional através da paralelização.

REFERÊNCIAS

Geeks for Geeks - Closest Pair of Points using Divide and Conquer algorithm. Disponível em: <https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/> - Acesso em 25/11/2020.