TPP2: MPI Divisão e Conquista (D&C)

Derick P. Garcez (13201878), Vinícius A. dos Santos (13201941)

1. Introdução

Este relatório apresenta o terceiro trabalho da disciplina de Programação Paralela e Distribuida. Todo o código e imagens deste trabalho está presente no repositório *github.com/derickpg/TPP2*.

2. Implementação

O programa se divide em duas partes, raiz quando id é zero ou nodo/filho quando o id é diferente de zero: Podemos ver um exemplo na Figura **??** anexada no final deste relatório.

2.1. Raiz

Na parte do nodo raiz, o sistema inicializa o nível da arvore em 1 e logo em seguida gera um array de números aleatórios, então é realizado uma divisão desse array em duas partes, e cada uma é enviada para um filho juntamente com o nível da arvore (o id do filho é definido por uma fórmula usando o id do nodo pai para manter a árvore equilibrada). Ex:

$$id_{filhoEsq} = id_{pai} * 2 + 1 \quad id_{filhoDir} = id_{pai} * 2 + 2$$

Após isso o processo raiz aguarda o retorno de ambos os filhos com esses dois arrays ordenados e por fim juntar em uma única saída.

2.2. Nodo / Filho

Em um laço de repetições este fica recebendo mensagens de qualquer origem e qualquer tag. No fluxo padrão do sistema esse nodo deve receber antes de tudo uma mensagem *tag_nivel* com nível da árvore do pai, então realiza-se um cálculo para saber o nível da árvore desse nodo $nível + 1$, também utiliza-se o nível calcular o tamanho do array que os filhos deste trabalham, com a fórmula: $tam/2^{(nivel-1)}$. A segunda mensagem que deve chegar é com tag de recebimento de array *tag_vetor* (enviada pelo pai), assim o nodo tem duas opções, se estiver na medida do padrão fixo então ordena o array e retorna ao pai, caso contrário este faz a divisão em duas outras partes para enviar aos seus filhos, calcula o id dos filhos, envia a informação do nível e o array logo em seguida. Também existe um terceiro tipo de mensagem com a tag que representa o recebimento de um array retornando *tag_r_vetor*, neste caso aguarda receber duas vezes este tipo de mensagem (um para cada filho) e realiza a junção desses vetores que estão ordenados e em seguida envia este array único como a mensagem do mesmo tipo ao vetor pai. Existe ainda outro tipo de mensagem para encerrar esses processos que são disparadas no final do processamento pelo nodo raiz.

3. Resultados

Os resultados obtidos são em relação a um vetor com 2000 números, o motivo deste tamanho se deve a que tivemos um erro na execução de vetores maiores que 2000 na execução da divisão e conquista, a imagem do erro está na Figura 3 anexada no final deste relatório após o código fonte. Tendo em vista deste problema executamos as soluções sequenciais para vetores deste tamanho, e realizamos a análise de SpeedUp e Eficiência em relação a este vetor.

Tipo de Ordenação	Tempo em Segundos
Bubble Sort	0.00417
Quick Sort	0.000297
Divisão e Conquista	0.01633

Tabela 1. Tempo de execução

Como o vetor que estamos ordenando é pequeno não temos uma diferença significativa nos tempos de execução, porém podemos perceber que a execução da divisão em conquista demora mais que os algoritmos sequênciais, isso se dá por que durante a troca de mensagens entre os processos, os pais precisamos dividir e criar novos vetores para enviar aos filhos, estes mesmo serão unidos novamente no processo de *Interleaving*, esses dois processos percorrem todos esses vetores, fazendo com que o tempo de execução aumente em relação ao sequencial. Provavelmente se o vetor fosse de um tamanho maior, o processo de divisão e conquista teria um melhor desempenho, pois a ordenação de fato na ponta (Nas folhas da Árvore) seriam realizadas de forma mais rápida.

Tipo de Ordenação	SpeedUp
Bubble Sort	0.01818
Quick Sort	0.25523

Tabela 2. SpeedUp da Divisão e Conquista em Relação as Ordenações Sequenciais.

Tanto para eficiência quanto para o SpeedUp ambos tiveram valores abaixo do teórico, ou seja valores ruins provando que a implementação da divisão e conquista realizada, e para este tamanho de vetor não é eficiente, tornando a versão sequencial tanto do Bubble Sort quanto QuickSort melhores.

Tipo de Ordenação	Eficiência
Bubble Sort	0.00114
Quick Sort	0.00026

Tabela 3. Eficiência da Divisão e Conquista em Relação as Ordenações Sequenciais.

```c
#include <stdio.h>
#include <math.h>
#include "mpi.h"


//Fun  o de compara  o QS
int compare(const void *a, const void *b)
{
        return (*(int *)a - *(int *)b);
}

main(int argc, char **argv)
{
    // Parametros
    int tam_vet = 50, nums_vet = 1000; // tamanho do vetor inicial e range dos numeros aleatorios de
        ex 100 = 0~99
    int tam_fixo = 25;

    // Variaveis
    int id_proc, qtd_proc;
    int id_pai, id_filho_esq, id_filho_dir;
    int nivel[1];
    int i = 0;
    int tam_esq, tam_dir, tam_novo_vetor;
    int vetor[tam_vet];
    int recebido[tam_vet];
    int cont = 0;
    int i_esq = 0;
    int i_dir = 0;
    int espera = 0;
    int vet_aux_esq[tam_vet];
    int vet_aux_dir[tam_vet];

    double ti, tf; // Tempos

    // TAGS
    int t_vetor = 50;
    int t_morte = 99;
    int t_nivel = 1;
    int t_r_vetor = 25;


    MPI_Status status; /* Status de retorno */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id_proc);
    MPI_Comm_size(MPI_COMM_WORLD, &qtd_proc);

    if (id_proc == 0) /* Raiz */ {

        nivel[0] = 1; //Raiz inicializa o nivel como um, e passa para os filhos
        for (i = 0; i < tam_vet; i++)
            vetor[i] = rand() % nums_vet + 1;

        //Por ser o raiz pular algumas etapas tipo calcular lados do array

        ti = MPI_Wtime();

        int vet_esq[tam_vet/2]; // 1000 / 2 = 500 0..499 500...999
        int vet_dir[tam_vet/2];

        for (i = 0; i < tam_vet; ++i) {
            if(i < tam_vet/2)
                vet_esq[i] = vetor[i];
            else {
                vet_dir[cont] = vetor[i];
                cont++;
            }
        }
        id_filho_esq = id_proc * 2 + 1;
        id_filho_dir = id_proc * 2 + 2;
        nivel[0] = 2;
        MPI_Send(nivel, tam_vet, MPI_INT, id_filho_esq, t_nivel, MPI_COMM_WORLD);
        MPI_Send(nivel, tam_vet, MPI_INT, id_filho_dir, t_nivel, MPI_COMM_WORLD);
```

```
        MPI_Send(vet_esq, (tam_vet/2), MPI_INT, id_filho_esq, t_vetor, MPI_COMM_WORLD);
        MPI_Send(vet_dir, (tam_vet/2), MPI_INT, id_filho_dir, t_vetor, MPI_COMM_WORLD);
        int verificacao = 0;

        while(verificacao == 0){
            MPI_Recv(recebido, (tam_vet/2), MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
                status);

            if(status.MPI_TAG == t_r_vetor){
                    espera++;
                    if(status.MPI_SOURCE == id_filho_esq){
                        for(i = 0; i < tam_vet/2; i++)
                            vet_aux_esq[i] = recebido[i];


                    }else{
                        for(i = 0; i < tam_vet/2; i++)
                            vet_aux_dir[i] = recebido[i];
                    }
                    if(espera == 2){
                        int vetor_final[tam_vet];
                        // ~~~ Processo de Interleaving ~~~
                        i_esq = 0;
                        i_dir = 0;
                        for (i = 0; i < tam_vet; i++) {
                            if (((vet_aux_esq[i_esq] <= vet_aux_dir[i_dir]) && (i_esq < (tam_vet/2))
                                ) || (i_dir == (tam_vet/2)))
                                vetor_final[i] = vet_aux_esq[i_esq++];
                            else
                                vetor_final[i] = vet_aux_dir[i_dir++];
                        }
                        verificacao = 1;



                        printf("\n\n ---------------- FIM ----------------------");
                    }
            }
        }
        // MSG da MORTE
        for(i = 0; i < qtd_proc; i ++){
            MPI_Send(0, 0, MPI_INT, i, t_morte, MPI_COMM_WORLD);
        }

    } else /* Nodos abaixo da raiz, varios niveis */ {
        int tam_vetor_n;
        int nivel_aux[1];
        int meu_nivel;
        int esperando_filho = 0;
        id_pai = (id_proc - 1) / 2;
        while(1){
            MPI_Recv(recebido, tam_vet, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &
                status);
            if(status.MPI_TAG == t_vetor){
                // Verifica se deve ou nao ordenar
                if(tam_vetor_n <= tam_fixo){ // entao ordena
                    // Manda para o pai o vetor ordenado
                    int *vet_aux;
                    vet_aux = recebido;
                    qsort(vet_aux, tam_vetor_n, sizeof(int), compare);

                    MPI_Send(vet_aux, tam_vetor_n, MPI_INT, id_pai, t_r_vetor, MPI_COMM_WORLD);
                }else{// Manda para os filhos
                    int vet_esq_n[tam_vetor_n/2]; // 1000 / 2 = 500 0..499 500...999
                    int vet_dir_n[tam_vetor_n/2];
                    id_filho_esq = id_proc * 2 + 1;
                    id_filho_dir = id_proc * 2 + 2;
                    cont = 0;
                    for (i = 0; i < tam_vetor_n; ++i) {
                        if(i < tam_vetor_n/2)
                            vet_esq_n[i] = recebido[i];
                        else {
```

```
                            vet_dir_n[cont] = recebido[i];
                            cont++;
                        }
                    }
                    nivel_aux[0] = meu_nivel + 1;
                    // Manda o Nivel para os Filhos
                    MPI_Send(nivel_aux, tam_vet, MPI_INT, id_filho_esq, t_nivel, MPI_COMM_WORLD);
                    MPI_Send(nivel_aux, tam_vet, MPI_INT, id_filho_dir, t_nivel, MPI_COMM_WORLD);

                    // Manda o Vetor para os Filhos
                    MPI_Send(vet_esq_n, (tam_vetor_n/2), MPI_INT, id_filho_esq, t_vetor,
                        MPI_COMM_WORLD);
                    MPI_Send(vet_dir_n, (tam_vetor_n/2), MPI_INT, id_filho_dir, t_vetor,
                        MPI_COMM_WORLD);

                }
            }else if(status.MPI_TAG == t_morte){
                MPI_Finalize();
                return 0;
            }else if(status.MPI_TAG == t_nivel){
                meu_nivel = recebido[0];
                tam_vetor_n = (tam_vet / (pow(2, (meu_nivel - 1))));
            }else if(status.MPI_TAG == t_r_vetor){ // Aguardando o retorno do vetor
                esperando_filho++;
                if(status.MPI_SOURCE == id_filho_esq){
                    for(i = 0; i < tam_vetor_n; i++)
                        vet_aux_esq[i] = recebido[i];
                }else{
                    for(i = 0; i < tam_vetor_n; i++)
                        vet_aux_dir[i] = recebido[i];
                }
                // So se receber os dois faz o processo do interleaving e manda para o pai
                if(esperando_filho == 2){
                    int vetor_ord[tam_vetor_n];
                    // ~~~ Processo de Interleaving ~~~
                    i_esq = 0;
                    i_dir = 0;
                    for (i = 0; i < tam_vetor_n; i++)
                    {
                        if (((vet_aux_esq[i_esq] <= vet_aux_dir[i_dir]) && (i_esq < (tam_vetor_n/2))
                            ) || (i_dir == (tam_vetor_n/2)))
                            vetor_ord[i] = vet_aux_esq[i_esq++];
                        else
                            vetor_ord[i] = vet_aux_dir[i_dir++];
                    }
                    // ~~~ Fim do Interleaving ~~~
                    MPI_Send(vetor_ord, tam_vetor_n, MPI_INT, id_pai, t_r_vetor, MPI_COMM_WORLD);
                }
            }
        }
    }
        if (id_proc == 0)
        {
                tf = MPI_Wtime();
                double total_time;
                total_time = tf - ti;
                printf("\n TEMPO TOTAL = %f \n", total_time);
        }

    MPI_Finalize();
    return 0;

}
```

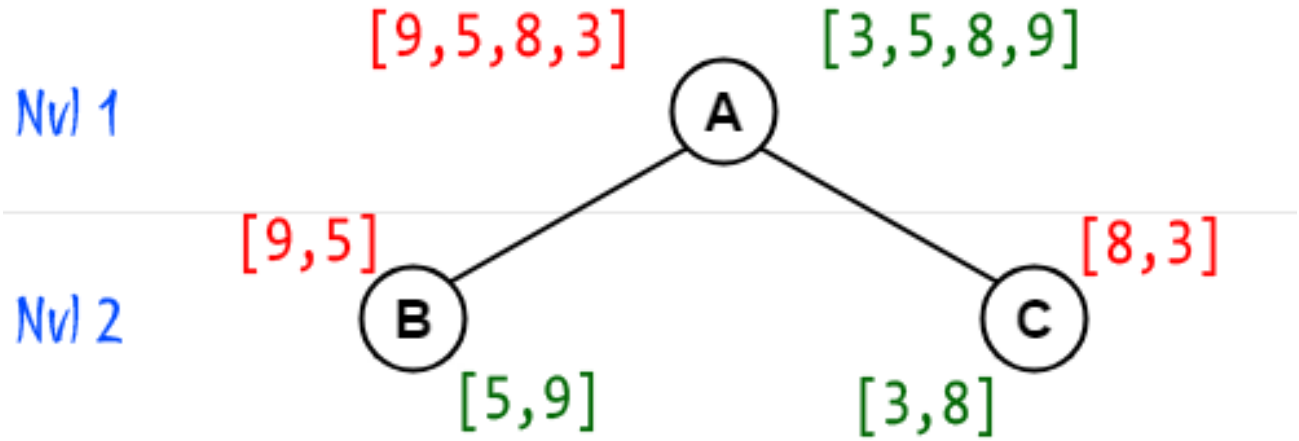

Figura 1. Árvore e seus níveis.

```
[grad05:04890] *** Process received signal ***
[grad05:04890] Signal: Segmentation fault (11)
[grad05:04890] Signal code: Address not mapped (1)
[grad05:04890] Failing at address: 0x7fffa6a68000
[grad05:04890] [ 0] /lib/x86_64-linux-gnu/libpthread.so.0(+0xfcb0) [0x7fb0b72bccb0]
[grad05:04890] [ 1] /lib/x86_64-linux-gnu/libc.so.6(+0x147738) [0x7fb0b7036738]
[grad05:04890] [ 2] /usr/local/openmpi/openmpi-1.4.5/lib/libmpi.so.0(ompi_convertor_pack+0x199) [0x7fb0b780a379]
[grad05:04890] [ 3] /usr/local/openmpi/openmpi-1.4.5/lib/openmpi/mca_btl_sm.so(+0x1c68) [0x7fb0b3bc1c68]
[grad05:04890] [ 4] /usr/local/openmpi/openmpi-1.4.5/lib/openmpi/mca_pml_ob1.so(+0xc5df) [0x7fb0b43df5df]
[grad05:04890] [ 5] /usr/local/openmpi/openmpi-1.4.5/lib/openmpi/mca_pml_ob1.so(+0x7998) [0x7fb0b43da998]
[grad05:04890] [ 6] /usr/local/openmpi/openmpi-1.4.5/lib/openmpi/mca_btl_sm.so(+0x4018) [0x7fb0b3bc4018]
[grad05:04890] [ 7] /usr/local/openmpi/openmpi-1.4.5/lib/libopen-pal.so.0(opal_progress+0x5b) [0x7fb0b6a5a03b]
[grad05:04890] [ 8] /usr/local/openmpi/openmpi-1.4.5/lib/openmpi/mca_pml_ob1.so(+0x6255) [0x7fb0b43d9255]
[grad05:04890] [ 9] /usr/local/openmpi/openmpi-1.4.5/lib/libmpi.so.0(PMPI_Send+0x155) [0x7fb0b7821f35]
[grad05:04890] [10] tv(main+0xd6b) [0x4018ef]
[grad05:04890] [11] /lib/x86_64-linux-gnu/libc.so.6(__libc_start_main+0xed) [0x7fb0b6f1076d]
[grad05:04890] [12] tv() [0x400aa9]
[grad05:04890] *** End of error message ***
```

Figura 2. Bug para vetores maiores do que 2000.

```
ppd59003@grad03:~/T4/T2$ ladcomp -env mpicc tv.c -o tv
/usr/local/openmpi/openmpi-1.4.5/bin/mpicc tv.c -o tv
ppd59003@grad03:~/T4/T2$ ladrun -np 31 tv
--------------------------------------------------------------------------
mpirun noticed that process rank 2 with PID 17815 on node grad03 exited on signal 11 (Segmentation fault).
--------------------------------------------------------------------------
```

Figura 3. Bug para vetores maiores do que 2000.