Relatório de Entrega de Trabalho
Disciplina de Programação Paralela (PP) – Prof. César De Rose

Alunos: João Langhanz – Paulo Jitsukawa
Usuários: pp12810 – pp12815
Exercicio: trabalho 2 de MPI (D&C)
Entrega: 18/05/2017

1) Introdução

Com o objetivo de avaliar o desempenho de um algoritmo de ordenação em um ambiente de processamento distribuído de alto desempenho, foi implementado um algoritmo para realizar a distribuição de carga de trabalho em diversos processos, para que estes executem suas tarefas separadamente. O problema consiste na ordenação de um grande vetor utilizando o modelo de Divisão e Conquista (D&C), em que o processo analisa se deve dividir ou conquistar o trabalho recebido. Se optar por dividir, ele quebra o problema em dois e repassa cada parte a um processo, que vai executar o mesmo algoritmo; se optar por conquistar, ele ordena o vetor e devolve para quem o enviou. Quando dois processos retornam seus vetores para o processo pai, este realiza uma intercalação dos vetores antes de retornar o processo para o seu próprio pai.

3) Testes

Foram realizados testes com a alocação de 2 nós do cluster *grad*, que possui 8 processadores físicos por nó. O algoritmo foi executado com 1, 3, 7, 15 e 31 processos; esses números foram escolhidos a fim de manter a árvore balanceada, e medido o tempo que cada um levou para terminar. Também foi testada a versão sequencial em uma maquina local, com 4 núcleos físicos e 8 threads. Implementou-se também uma versão do algoritmo em que cada processo, mesmo que opte por dividir, pega uma porcentagem do trabalho para realizar localmente. A porcentagem é fixa durante a execução do programa; foram feitos testes com a porcentagem de 10%, 20% e 30%. Foram utilizados vetores de tamanho 10.000, 100.000 e 1.000.000 para os testes. Como os dois primeiros não apresentaram diferenças significativas, analisamos aqui apenas os resultados do de maior tamanho.

4) Análise de Desempenho

A versão sequencial do algoritmo que foi executado na máquina local se saiu melhor do que o sequencial que rodou no cluster *grad*. A máquina local com processador Intel Core i5 levou 41 minutos para ordenar o vetor, contra os 73 minutos (dado fornecido pelo professor) do cluster. Uma provável razão para isso é que a máquina local possui um processador mais atual e mais veloz que o cluster, logo, ela se sai melhor para processos sequenciais, o que não aconteceria com muitos processos paralelos, já que possui apenas 4 núcleos.

O desempenho medido na execução com 3 processos foi bem melhor do que a versão sequencial. O tempo de execução foi quase 4 vezes mais rápido com 2 processos realizando o trabalho de ordenação em paralelo, assim como de 3 para 7 e 7 para 15. O cluster utiliza Hyper-Threading (HT) com 31 processos, por isso era esperada uma queda na eficiência, mas não foi o que aconteceu. A eficiência aumentou, mesmo que pouco, ao contrário do que era esperado, já que há mais de um processo por núcleo e eles ficam sendo escalonados pelo SO. Um dos provavéis motivos pra isso é o próprio modelo de D&C. Se o analisarmos, veremos que se divide formando uma árvore binária e, quando um nodo (processo) divide o trabalho e o repassa para seus filhos, ele fica ocioso, apenas aguardando a resposta. Assim, só as folhas da árvore realizam o trabalho pesado de ordenação, gerando um grande disperdicio, pois aproximadamente metade da árvore está parada esperando o retorno para realizar a intercalação. Ao fazer uma conta rápida, apenas 16 dos 31 processos estão fazendo o trabalho de ordenação. Por isso, ao utilizar HT, a eficiência não é tão afetada; provavelmente o escalonador do SO vai rodar os processos que estão fazendo o trabalho pesado nos mesmos núcleos onde se encontram os processos ociosos, o que tem um custo no desempenho, mas não tanto quanto um caso em que, por exemplo, todos os 31 processos estivessem realizando o trabalho de ordenação. Na figura 1 podemos ver um gráfico dos desempenhos coletados.

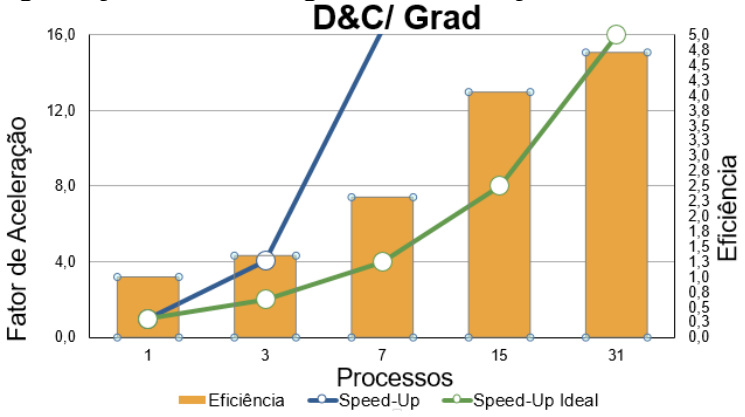

Figura 1 – resultados observados

Com a finalidade de aproveitar melhor esses processos ociosos, utilzamos a versão do algoritmo que realiza a ordenação local de uma parte de vetor. Para este teste desconsideramos a versão sequencial, pois não faz sentido aplicá-la aqui. O maior ganho ocorre quando executamos para 3 processos: uma diferença de mais de 50 minutos, o que não acontece quando se coloca mais processos, situação em que não houve uma diferença significativa. Um dos motivos para isso ter ocorrido é a maneira que foi escolhida para pegar uma parte do vetor, que é fixa.

O que ocorre é que a porcentagem representa uma grande parte do vetor no primeiro nodo, de foma que com 3 processos tem um desempenho melhor. À medida em que se desce na árvore, os vetores são menores e essa porcentagem representa uma quantidade menor de elementos, tornando-se menos significativa a ajuda que o processo está fornecendo.

Outro fator observado foi que os testes com diferentes porcentagens de ordenação local não demonstraram grande diferença no resultado. Uma das causas provavéis foi a já citada, de que a porcentagem fixa significa cada vez menos, além do fato de que é necessário uma rotina de intercalação a mais com a ordenação local.

5) Observações Finais

O modelo de D&C se mostrou bastante eficiente para tratar do problema proposto. O que foi percebido é que se faz necessário realizar ajustes no modelo para que ele tenha um melhor desempenho quando paralelizado em processos, como a ordenação local, para evitar que os processos fiquem ociosos.

/* Versão simples */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

int *interleaving(int vetor[], int tam)
{
int *vetor_auxiliar;
int i1, i2, i_aux;

vetor_auxiliar = (int *)malloc(sizeof(int) * tam);

i1 = 0;
i2 = tam / 2;

for (i_aux = 0; i_aux < tam; i_aux++) {
        if (((vetor[i1] <= vetor[i2]) && (i1 < (tam / 2))) || (i2 == tam))
                vetor_auxiliar[i_aux] = vetor[i1++];
        else
                vetor_auxiliar[i_aux] = vetor[i2++];
        }
        return vetor_auxiliar;
}
/* Bubble Sort */
void bs(int n, int * vetor)
{
        int c=0, d, troca, trocou =1;
        while (c < (n-1) & trocou ){
                trocou = 0;
                for (d = 0 ; d < n - c - 1; d++)
                        if (vetor[d] > vetor[d+1]){
                                troca = vetor[d];
                                vetor[d] = vetor[d+1];
                                vetor[d+1] = troca;
                                trocou = 1;
                        }
                        c++;
        }
}

int main(int argc, char** argv){

int my_rank;
int proc_n;
int ARRAY_SIZE = atoi(argv[1]);
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
int delta = ARRAY_SIZE / (((proc_n-1)/2) + 1); // Numero de folhas da minha arvore

int filhoDireita = (my_rank*2)+1;
int filhoEsquerda = (my_rank*2)+2;

int *vetor;
vetor = (int *)malloc(sizeof(int) * ARRAY_SIZE);

int size;
int newsize;

double t1,t2;
t1 = MPI_Wtime(); // inicia a contagem do tempo

if ( my_rank == 0 ) // NODO PAI
{

        /*preenche vetor */
        int i;
        for (i=0 ; i<ARRAY_SIZE; i++){
        vetor[i] = ARRAY_SIZE-i;
}
size = ARRAY_SIZE;
newsize = ARRAY_SIZE/2;

if(proc_n == 1){
        bs(size, vetor);
}else{

        /* Divide o vetor para seus dois filhos, se existirem */
        if(proc_n > filhoDireita){
```

```
        MPI_Send(&vetor[0], newsize, MPI_INT, filhoDireita, 1, MPI_COMM_WORLD);
        }
        if(proc_n > filhoEsquerda){
                MPI_Send(&vetor[size/2], newsize, MPI_INT, filhoEsquerda, 1, MPI_COMM_WORLD);
        }

        /* Aguarda a resposta dos filhos */
        if(proc_n > filhoDireita) MPI_Recv(&vetor[0], newsize, MPI_INT, filhoDireita, MPI_ANY_TAG, MPI_COMM_WORLD,
                &status);
        if(proc_n > filhoEsquerda) MPI_Recv(&vetor[size/2], newsize, MPI_INT, filhoEsquerda, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);
        /* Merge do vetor */
        vetor = interleaving(vetor,size);
        }
}
else // NODO FILHO
{
/* Descubro quem vai me enviar (pai) calculando pelo meu rank */
int mysource = (my_rank-1)/2;

/* Espero uma parte do vetor para ordenar */
MPI_Recv(vetor, ARRAY_SIZE , MPI_INT, mysource, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_INT, &size); // descubro tamanho da mensagem recebida
/* Pego o rank e a tag de quem me enviou */
int source = status.MPI_SOURCE;
int tag = status.MPI_TAG;

if(size <= delta){ //Conquista
        /* Ordena o vetor */
        bs(size, vetor);
}else{ //Divide

        newsize = size/2;
        /* Divide o vetor para seus dois filhos, se existirem */
        if(proc_n > filhoDireita){
        MPI_Send(&vetor[0], newsize, MPI_INT, filhoDireita, 1, MPI_COMM_WORLD);
        }
        if(proc_n > filhoEsquerda){
        MPI_Send(&vetor[size/2], newsize, MPI_INT, filhoEsquerda, 1, MPI_COMM_WORLD);
        }
        /* Aguarda a resposta dos filhos */
        if(proc_n > filhoDireita) MPI_Recv(&vetor[0], newsize, MPI_INT, filhoDireita, MPI_ANY_TAG, MPI_COMM_WORLD,
        &status);
        if(proc_n > filhoEsquerda) MPI_Recv(&vetor[size/2], newsize, MPI_INT, filhoEsquerda, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
        /* Merge do vetor */
        vetor = interleaving(vetor,size);

        }
        /* Envia de volta para o pai */
        MPI_Send(vetor, size, MPI_INT, source, 1, MPI_COMM_WORLD);
        }

        if(my_rank==0){
                t2 = MPI_Wtime();
                printf("\nTempo de execucao: %f\n\n", t2-t1);
        }

        MPI_Finalize();
}
```

/* Versão com ordenação local */

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"

//#define ARRAY_SIZE 100000
//#define PERCENT 10

int *interleavingParcial(int vetor[], int tam,int tam2)
{
int *vetor_auxiliar;
int i1, i2, i_aux;

vetor_auxiliar = (int *)malloc(sizeof(int) * tam);

i1 = 0;
i2 = tam2;

for (i_aux = 0; i_aux < tam; i_aux++) {
```

```
if (((vetor[i1] <= vetor[i2]) && (i1 < tam2)) || (i2 == tam))
vetor_auxiliar[i_aux] = vetor[i1++];
else
vetor_auxiliar[i_aux] = vetor[i2++];
}
return vetor_auxiliar;
}

int *interleaving(int vetor[], int tam)
{
int *vetor_auxiliar;
int i1, i2, i_aux;

vetor_auxiliar = (int *)malloc(sizeof(int) * tam);

i1 = 0;
i2 = tam / 2;

for (i_aux = 0; i_aux < tam; i_aux++) {
if (((vetor[i1] <= vetor[i2]) && (i1 < (tam / 2))) || (i2 == tam))
vetor_auxiliar[i_aux] = vetor[i1++];
else
vetor_auxiliar[i_aux] = vetor[i2++];
}
return vetor_auxiliar;
}

/* Bubble Sort */
void bs(int n, int * vetor)
{

int c=0, d, troca, trocou =1;
// printf("Ordenando...\n");
while (c < (n-1) & trocou ){
trocou = 0;
for (d = 0 ; d < n - c - 1; d++)
if (vetor[d] > vetor[d+1]){
troca = vetor[d];
vetor[d] = vetor[d+1];
vetor[d+1] = troca;
trocou = 1;
}
c++;
}
}

int main(int argc, char** argv){

int my_rank;
int proc_n;
int ARRAY_SIZE = atoi(argv[1]);
int PERCENT = atoi(argv[2]);
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
int delta = ARRAY_SIZE / (((proc_n-1)/2) + 1); // Numero de folhas da minha arvore

int filhoDireita = (my_rank*2)+1;
int filhoEsquerda = (my_rank*2)+2;

int *vetor;
vetor = (int *)malloc(sizeof(int) * ARRAY_SIZE);

int size;
int newsize;
int localsize;
int pos1, pos2;
int par;

double t1,t2;
t1 = MPI_Wtime(); // inicia a contagem do tempo

if ( my_rank == 0 ) // NODO PAI
{

/*preenche vetor */
int i;
for (i=0 ; i<ARRAY_SIZE; i++){
vetor[i] = ARRAY_SIZE-i;
}
printf("Primeiro [%d]\n",vetor[0]);
```

```
/* Variaveis de Controle */
size = ARRAY_SIZE;
localsize = ARRAY_SIZE/PERCENT;
newsize = (size-localsize)/2;
pos1 = 0+localsize;
pos2 = (size/2)+(localsize/2);
par = (int)(newsize%2);

/* Divide o vetor para seus dois filhos, se existirem */
if(proc_n > filhoDireita){
MPI_Send(&vetor[pos1], newsize, MPI_INT, filhoDireita, 1, MPI_COMM_WORLD);
}
if(proc_n > filhoEsquerda){
MPI_Send(&vetor[pos2], newsize+par, MPI_INT, filhoEsquerda, 1, MPI_COMM_WORLD);
}

/* Ordena porcentagem local */
bs(localsize,vetor);

/* Aguarda a resposta dos filhos */
if(proc_n > filhoDireita) MPI_Recv(&vetor[pos1], newsize, MPI_INT, filhoDireita, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
/* intercala a ordenação local com a recebida do filho */
vetor = interleavingParcial(vetor,size,localsize);
if(proc_n > filhoEsquerda) MPI_Recv(&vetor[pos2-localsize], newsize+par, MPI_INT, filhoEsquerda, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
/* Merge do vetor */
vetor = interleavingParcial(vetor,size,newsize);

}
else // NODO FILHO
{
        /* Descubro quem vai me enviar (pai) calculando pelo meu rank */
        int mysource = (my_rank-1)/2;

        /* Espero uma parte do vetor para ordenar */
        MPI_Recv(vetor, ARRAY_SIZE , MPI_INT, mysource, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_INT, &size); // descubro tamanho da mensagem recebida
        /* Pego o rank e a tag de quem me enviou */
        int source = status.MPI_SOURCE;
        int tag = status.MPI_TAG;

        /* Se eu recebi uma msg de suicidio */
        if(tag == 0){
                printf("Suicidando-se, rank %d\n",my_rank);
                MPI_Finalize();
        }

        if(!(proc_n > filhoDireita)){//size <= delta){ //Conquista
        /* Ordena o vetor */
                bs(size, vetor);
        }else{ //Divide

                /* Variaveis de controle */
                localsize = size/PERCENT;
                newsize = (size-localsize)/2;
                pos1 = 0+localsize;
                pos2 = (size/2)+(localsize/2);
                par = (int)(newsize%2);
                /* Divide o vetor para seus dois filhos, se existirem */
                if(proc_n > filhoDireita){
                MPI_Send(&vetor[pos1], newsize, MPI_INT, filhoDireita, 1, MPI_COMM_WORLD);
        }
        if(proc_n > filhoEsquerda){
                MPI_Send(&vetor[pos2], newsize+par, MPI_INT, filhoEsquerda, 1, MPI_COMM_WORLD);
        }

        /* Ordena porcentagem local */
        bs(localsize,vetor);
        /* Aguarda a resposta dos filhos */
        if(proc_n > filhoDireita) MPI_Recv(&vetor[pos1], newsize, MPI_INT, filhoDireita, MPI_ANY_TAG, MPI_COMM_WORLD,
        &status);
        /* Intervacala local com o recebido do filho */
        vetor = interleavingParcial(vetor,size,localsize);
        if(proc_n > filhoEsquerda) MPI_Recv(&vetor[pos2-localsize], newsize+par, MPI_INT, filhoEsquerda, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
        /* Merge do vetor */
        vetor = interleavingParcial(vetor,size,newsize);

        }
        /* Envia de volta para o pai */
        MPI_Send(vetor, size, MPI_INT, source, 1, MPI_COMM_WORLD);
```

```
}

if(my_rank==0){
        t2 = MPI_Wtime();
        printf("\nTempo de execucao: %f\n\n", t2-t1);
}

MPI_Finalize();
}
```