Programação Paralela Trabalho 2

Bruno Brandelli, Rodrigo Pacheco, Usuário pp12804

I. Introdução

Este trabalho tem como principal objetivo analisar a eficiência, speed-up e alguns outros fatores da execução de um algoritmo de divisão e conquista realizando o ordenamento de um vetor utilizando processos paralelos. Tal analise será feita apartir da observação da execução do algoritmo em forma sequencial, com três variações de carga, 10000, 100000 e 1000000 elementos organizados de forma inversa de contagem dentro de um vetor, de forma a utilizar o pior caso no algoritmo de ordenamento bubblesort. Outro cenário que será observado é a diferença de eficiência quanto ao ordenamento apenas nas ultimas folhas da árvore, e a utilização de ordenamento local junto do envio de carga aos filhos.

II. Divisão e Conquista

Serão desenvolvidas duas versões do algoritmo executando de forma paralela, uma mais simples que utiliza o conceito básico de divisão e conquista, onde o trabalho é dividido, e em algum ponto é conquistado, e por fim intercalado os resultados da divisão. E uma versão que busca o menor tempo ocioso dos processos, realizando uma ordenação local enquanto espera o resultado da divisão.

Na versão mais simplória do algoritmo desenvolvido, os processos que ficam na camada mais abaixo da arvore realizam o bubblesort, enquanto os processos que estão mais acima apenas fazem a combinação destes resultados, já na versão que busca um aproveitamento maior dos processos, todos os processos farão o bubblesort em uma fração do vetor. A técnica de divisão e conquista aplicada neste trabalho consegue tirar muito proveito da programação paralela, visto que a divisão das tarefas que ocorre é repassada entre os processos.

III. Análise de Resultados

Os algoritmos foram executados usando 2 núcleos do cluster Grad, onde cada núcleo tem 8 processadores, e são capazes de rodar 16 processos utilizando o Hyper-Threading. Na figura 1 temos os resultados encontrados utilizando o vetor com 1000000 elementos, tal decisão foi tomada devido à maior necessidade de processamento, embora os testes com as outras configurações de vetores tenham sido realizadas.

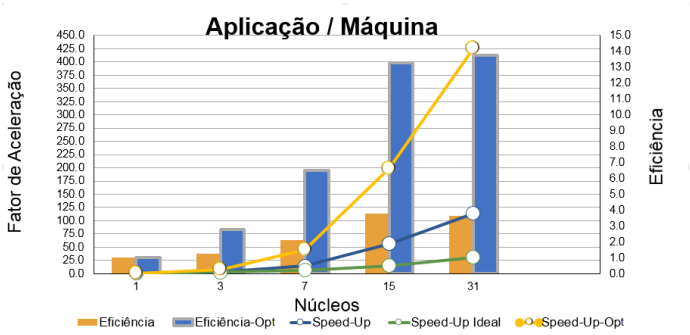

Figure 1. Gráfico de eficiência/speed-up

Este trabalho mostrou um comportamento interessante, porque devido ao fato da ordenação ter uma complexidade quadrática, ao dividir o problema entre os processos, o ganho de desempenho também é quadrático. Isto leva a execução dos algoritmos em paralelo à um speed-up maior do que o calculado como ideal.

Ao utilizar o algoritmo sem otimização, a carga de trabalho mais pesado encontrava-se nas folhas do ultimo nível da árvore, onde era realizado o ordenamento dos pedaços menores do vetor, e as camadas superiores apenas intercalavam estes pedaços de vetores, que é uma tarefa muito mais fácil computacionalmente. No caso do algoritmo otimizado, o pai sempre ficava com metade do vetor para ordenar localmente, e após receber o trabalho dos seus filhos, ele intercalava os 3 vetores, ou seja, quanto mais ao topo da árvore, mais carga de trabalho o processo recebia, mas apesar disto, como pode ser visto na figura 1, ele tem um ganho de eficiência devido ao fato de haver menos processos ociosos, porque com uma divisão de metade da carga de trabalho entre pai e filhos , o pai gasta quase a mesma quantidade de tempo ordenando sua porção do vetor que os filhos que talvez tenham que dividir seu trabalho novamente.

O hyper-threading demonstra um ganho de eficiência até um certo ponto, já que para ativa-lo é necessário chamar mais processos. E uma grande quantidade de processos pode gerar muitas camadas para a árvore, que reduz a eficiência devido ao processo de intercalação que ocorre em cada camada.

```c
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 10000      // trabalho final com o valores 10.000, 100.000, 1.000.000

void bubbleSort(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 vetor[ARRAY_SIZE];

    for(int i = 0 ; i<ARRAY_SIZE; i++){
        vetor[i] = ARRAY_SIZE-i;
    }

    bubbleSort(ARRAY_SIZE, vetor);
    return 0;
}
```

```
#include <stdio.h>
#include "mpi.h"
#define ARRAY_SIZE 100

void bubbleSort(int size, int *array); //metodo que faz o ordenamento
void initializeArray(int size, int *array); // metodo que inicializa o array na ordem inversa
void showArray(int *array);
int * interleaving(int size,int *array); //metodo que faz a intercalação dos arrays vindos dos
filhos
int father(int proc); // metodo que retorna o pai de um processo
int leftChild(int proc); // metodo que retorna o numero do processo filho a esquerda
int rightChild(int proc); // metodo que retorna o numero do processo filho a direita
int calculateDelta(int size, int proc_n);

void main(int argc, char** argv){
    int* aux; // usado para armazenar o array que volta da intercalação dos arrays
    int array[ARRAY_SIZE]; //inicia o array que será ordenado
    int count;    // faz o receive do tamanho do array pelo MPI_GET_count
    int my_rank;  // Identificador do processo
    int proc_n;   // Número de processos
    int a_size;   //guarda o tamanho do array dentro do processo
    int delta;    // delta que define a divisao ou conquista
    double start, end; // usados para medir o tempo
    MPI_Init(&argc, &argv);
    MPI_Status status; // Status de retorno
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

    //tratamento para processos filhos
    if (my_rank != 0){
       delta = calculateDelta(ARRAY_SIZE,proc_n);
       MPI_Recv(array,ARRAY_SIZE,MPI_INT,father(my_rank),MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       MPI_Get_count(&status, MPI_INT, &count);
       a_size = count;
    }else{
        //tratamento para a raiz dos processos
       start = MPI_Wtime();
       a_size = ARRAY_SIZE;               // defino tamanho inicial do array
       initializeArray(a_size,array);    // inicializa o array na ordem reversa
       delta = calculateDelta(ARRAY_SIZE,proc_n);
    }
    // dividir ou conquistar
    if(a_size <= delta){
       bubbleSort(a_size,array);  // conquisto
       MPI_Send(array,a_size,MPI_INT,father(my_rank), 1, MPI_COMM_WORLD);
    }else{
        // dividir
        // quebrar em duas partes e mandar para os filhos
        MPI_Send (&array[0],(a_size/2),MPI_INT,leftChild(my_rank), 1, MPI_COMM_WORLD);  // mando
metade inicial do array
        MPI_Send (&array[a_size/2],(a_size/2),MPI_INT,rightChild(my_rank),1, MPI_COMM_WORLD);  //
mando metade final
        // receber dos filhos
        MPI_Recv (&array[0],a_size,MPI_INT,leftChild(my_rank),MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        MPI_Recv (&array[a_size/2],a_size,MPI_INT,rightChild(my_rank),MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        // intercalo array inteiro
        aux = interleaving(a_size,array);
        if (my_rank != 0)
           MPI_Send(aux,a_size,MPI_INT,father(my_rank), 1, MPI_COMM_WORLD);
    }
    if(my_rank == 0){
        end = MPI_Wtime();
        printf("%f segundos \n", end-start);
    }

    MPI_Finalize();
}
```

```
void bubbleSort(int n, int *array){
    int c=0, d, troca, trocou =1;

    while (c < (n-1) & trocou){
        trocou = 0;
        for (d = 0 ; d < n - c - 1; d++){
            if (array[d] > array[d+1]){
                troca      = array[d];
                array[d]   = array[d+1];
                array[d+1] = troca;
                trocou = 1;
            }
        }
        c++;
    }
}

int * interleaving(int size,int *array){
    int *array_auxiliar;
    int i1, i2, i_aux;

    array_auxiliar = (int *)malloc(sizeof(int) * size);

    i1 = 0;
    i2 = size / 2;

    for (i_aux = 0; i_aux < size; i_aux++) {
        if (((array[i1] <= array[i2]) && (i1 < (size / 2)))
            || (i2 == size))
            array_auxiliar[i_aux] = array[i1++];
        else
            array_auxiliar[i_aux] = array[i2++];
    }

    return array_auxiliar;
}

void initializeArray(int size, int *array){
    for(int i = 0 ; i < size; i++){
        array[i] = size-i;
    }
}

void showArray(int *array){
    for(int i = 0 ; i<ARRAY_SIZE; i++){
        printf("%d\n",array[i]);
    }
}

int father(int proc){
    return (proc-1)/2;
}

int leftChild(int proc){
    return proc * 2 + 1;
}

int rightChild(int proc){
    return proc * 2 + 2;
}

int calculateDelta(int size, int proc_n){
    return size/((proc_n + 1)/2);
}
```

```
#include <stdio.h>
#include "mpi.h"
#define ARRAY_SIZE 1000000

void bubbleSort(int size, int *array); //metodo que faz o ordenamento
void initializeArray(int size, int *array); // metodo que inicializa o array na ordem inversa
void showArray(int *array);
int * interleaving(int size,int size_child,int *array); //metodo que faz a intercalação dos arrays
vindos dos filhos
int father(int proc); // metodo que retorna o pai de um processo
int leftChild(int proc); // metodo que retorna o numero do processo filho a esquerda
int rightChild(int proc); // metodo que retorna o numero do processo filho a direita
int calculateDelta(int size, int proc_n);

void main(int argc, char** argv){
    int* aux; // usado para armazenar o array que volta da intercalação dos arrays
    int size_child; // tamanho de cada processo filho
    int array[ARRAY_SIZE]; //inicia o array que será ordenado
    int count;    // faz o receive do tamanho do array pelo MPI_GET_count
    int my_rank;  // Identificador do processo
    int proc_n;   // Número de processos
    int a_size;   //guarda o tamanho do array dentro do processo
    int delta;    // delta que define a divisao ou conquista
    double start, end; // usados para medir o tempo
    MPI_Init(&argc, &argv);
    MPI_Status status; // Status de retorno
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

    //tratamento para processos filhos
    if (my_rank != 0){
       delta = calculateDelta(ARRAY_SIZE,proc_n);
       MPI_Recv(array,ARRAY_SIZE,MPI_INT,father(my_rank),MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       MPI_Get_count(&status, MPI_INT, &count);
       a_size = count;
    }else{
        //tratamento para a raiz dos processos
       start = MPI_Wtime();
       a_size = ARRAY_SIZE;               // defino tamanho inicial do array
       initializeArray(a_size,array);    // inicializa o array na ordem reversa
       delta = calculateDelta(ARRAY_SIZE,proc_n);
    }
    // verifica se divide ou conquista
    if(a_size < 2 * delta){
       // conquista
       bubbleSort(a_size,array);  // conquisto
       MPI_Send(array,a_size,MPI_INT,father(my_rank), 1, MPI_COMM_WORLD);
    }else{
        // divide
        size_child = (a_size - delta)/2; // subtrai o delta do tamanho atual e divide por dois,
tendo assim o tamanho de cada filho
        MPI_Send (&array[0],size_child,MPI_INT,leftChild(my_rank), 1, MPI_COMM_WORLD);  //  mando
uma parte para o filho da esquerda
        MPI_Send (&array[size_child],size_child,MPI_INT,rightChild(my_rank),1, MPI_COMM_WORLD);
// mando uma parte para o filho da direita
        bubbleSort(a_size - 2 * size_child,&array[2 * size_child]); //ordeno metade do array
localmente enquanto espero resposta
        // receber dos filhos
        MPI_Recv (&array[0],a_size,MPI_INT,leftChild(my_rank),MPI_ANY_TAG, MPI_COMM_WORLD,
&status);
        MPI_Recv (&array[size_child],a_size,MPI_INT,rightChild(my_rank),MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
        // intercalo array inteiro
        aux = interleaving(a_size,size_child,array);
        if (my_rank != 0)
           MPI_Send(aux,a_size,MPI_INT,father(my_rank), 1, MPI_COMM_WORLD);
    }
    if(my_rank == 0){
        end = MPI_Wtime();
```

```
        printf("%f segundos \n", end-start);
    }

    MPI_Finalize();
}

void bubbleSort(int n, int *array){
    int c=0, d, troca, trocou =1;

    while (c < (n-1) & trocou){
        trocou = 0;
        for (d = 0 ; d < n - c - 1; d++){
            if (array[d] > array[d+1]){
                troca      = array[d];
                array[d]   = array[d+1];
                array[d+1] = troca;
                trocou = 1;
            }
        }
        c++;
    }
}

int * interleaving(int size,int size_child,int *array){
    int *array_auxiliar;
    int i1,size_i1,i2,size_i2,i3,i_aux;
    array_auxiliar = (int *)malloc(sizeof(int) * size);

    //defino o inicio e o limite de cada array que será intercalado
    i1 = 0;
    size_i1 = size_child;
    i2 = size_i1;
    size_i2 = 2 * size_child;
    i3 = size_i2;

    for (i_aux = 0; i_aux < size; i_aux++) {
        if((array[i1] <= array[i2]) && (array[i1] <= array[i3]) && (i1 < size_child) || ((i2 ==
(size_i2))
                && (i3 == size))){
            array_auxiliar[i_aux] = array[i1++];
        }else{
            if ((array[i2] <= array[i3]) && (i2 < (size_i2))
                || (i3 == size))
                array_auxiliar[i_aux] = array[i2++];
            else
                array_auxiliar[i_aux] = array[i3++];
        }
    }
    return array_auxiliar;
}

void initializeArray(int size, int *array){
    for(int i = 0 ; i < size; i++){
        array[i] = size-i;
    }
}

void showArray(int *array){
    for(int i = 0 ; i<ARRAY_SIZE; i++){
        printf("%d\n",array[i]);
    }
}

int father(int proc){
    return (proc-1) / 2;
}

int leftChild(int proc){
    return proc * 2 + 1;
```

```
}

int rightChild(int proc){
    return proc * 2 + 2;
}

int calculateDelta(int size, int proc_n){
    return size / proc_n;
}
```