

LABORATÓRIO DE PROGRAMAÇÃO PARALELA

TRABALHO FINAL

Alunos:

Vitor Costa Hardoim
Rafael Dantas Amancio

OpenMP

Nessa sessão foram estudados os seguintes métodos de ordenação: BubbleSort, MergeSort, QuickSort e BitonicSort. Após a análise do funcionamento desses algoritmos concluímos que dentre eles, o mais interessante para implementação por threads foi o BitonicSort, pois o comportamento do seu loop de criação da rede de ordenação favorece a implementação paralela, já que nós sempre comparamos os elementos de maneira pré-definida e a sequência de comparações não depende dos valores do array.

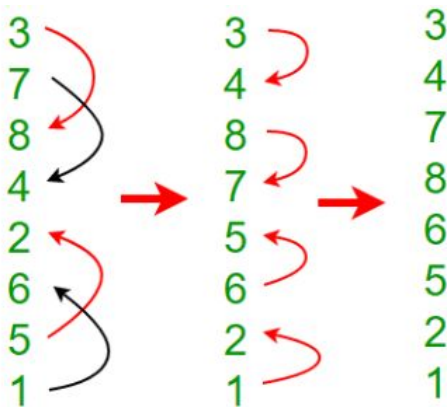
- BitonicSort

Vetor [3, 7, 4, 8, 6, 2, 1, 5]

- 1º passo: Consideramos cada 2 elementos consecutivos como uma sequência bitônica, ou seja, os 2 primeiros nós ordenamos de maneira crescente e os 2 seguintes em ordem decrescente, assim por diante, aumentando os grupos em potências de 2(4, 8, 16, 32...)



- 2º passo: Teremos 2 sequências bitônicas: (3, 7, 8, 4) e (2, 6, 5, 1).



Porém, agora o tamanho da comparação é de 2, assim comparamos x_0 com x_2 , x_1 com x_3 e assim por diante, a paralelização é feita nesse momento da geração da sequência bitônica de tamanho n , como a

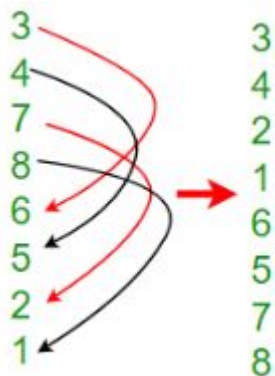
ordem em que as trocas são feitas é pré-definida, esse acaba sendo momento mais favorecido pela paralização.

Depois desse passo, teremos uma sequência bitônica de 8 elementos.

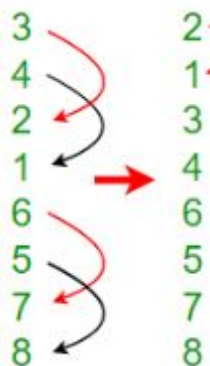
Vetor [3, 4, 7, 8, 6, 5, 2, 1]

Com a sequência bitônica completa, faremos o seguinte passo, depois dele teremos a primeira metade ordenada em ordem crescente e a segunda decrescente

Nós vamos comparar o primeiro elemento da primeira metade, com o primeiro da segunda e assim por diante, trocando caso o da primeira metade for menor:



Nesse ponto, nós teremos 2 sequência bitônicas de tamanho $n/2$: **(3, 4, 2, 1)** e **(6, 5, 7, 8)**, nós repetimos o mesmo processo, agora com as 2 sequências e vamos obter 4 de tamanho $n/4$.



Por último, repetiremos esse processo para as 4 sequências, terminando com 8 de tamanho $n/8$, que é 1, e o vetor está ordenado.



- Implementação:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define SIZE 1024
```

void trocaCrescente(int index1, int index2, int *array) //Troca 2 valores de posição para ficarem em ordem crescente no array.

```
{
    if (array[index2] < array[index1])
    {
        int temp = array[index2];
        array[index2] = array[index1];
        array[index1] = temp;
    }
}
```

void trocaDecrescente(int index1, int index2, int *array) //Troca 2 valores de posição para ficarem em ordem crescente no array.

```
{
    if (array[index1] < array[index2])
    {
        int temp = array[index2];
        array[index2] = array[index1];
        array[index1] = temp;
    }
}
```

void bitonicSortFromBitonicSequence(int startIndex, int lastIndex, int dir, int *ar) // Cria um array crescente ou decrescente dependendo da direção de entrada.

```
{
    if (dir == 1) // Ordenação crescente
    {
        int cont = 0; //contador para sabermos quais elementos já foram trocados de posição
        int n_elements = lastIndex - startIndex + 1;
        for (int j = n_elements / 2; j > 0; j /= 2)
        {
            cont = 0;
            for (int i = startIndex; i + j <= lastIndex; i++)
            {
                if (cont < j)
                {
                    trocaCrescente(i, i + j, ar);
                    cont++;
                }else
            }
        }
    }
}
```



```

int size = SIZE;
int array[size];
srand(time(NULL));
//gera o vetor aleatório
for (int i = 0; i < size; i++)
{
    array[i] = rand() % size;
}

//algoritmo de ordenação
bitonicSequenceGenerator(0, size - 1, array);

//printa o vetor de saída
for(int i = 0; i < size; i++)
{
    //printf("[%d]:%d ", i, array[i]);
}
}

```

- Tempos:

Nesse momento fizemos comparações com 4 e 1(não paralelizado) threads e 1024 elementos, obtendo o seguinte resultado:

4 threads - 1024 elementos:

0,008s
0,004s
0,005s
0,003s
0,005s

1 thread - 1024 elementos:

0,004s
0,003s
0,004s
0,004s
0,003s

Como para um vetor pequeno o tempo é muito curto, testamos para um vetor de tamanho 2^{20} , obtendo uma melhora perceptível com o paralelismo.

4 threads - 1048576 elementos:

0,417s
0,424s
0,419s
0,416s
0,422s

1 thread - 1048576 elementos:

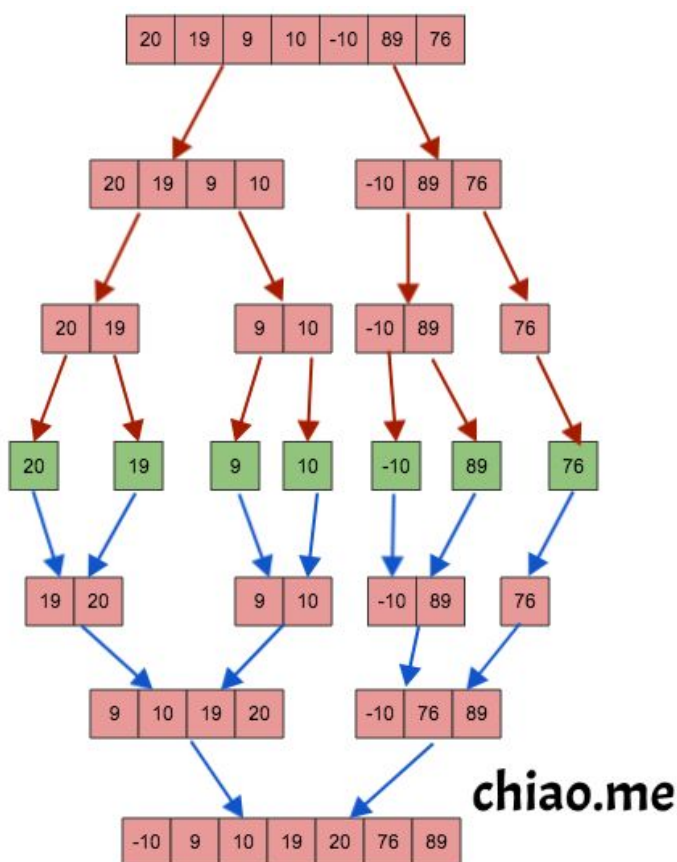
1.094s

1,153s
1,089s
1,096s
1,092s

MPI:

Nessa sessão o algoritmo de ordenação escolhido com o MergeSort, pois como o merge trabalha com subvetores que serão mesclados posteriormente à sua ordenação individual, podemos dividir essa tarefa em 4(processos) subtarefas diferentes, dessa forma temos um ganho de tempo razoável, enquanto a estrutura do algoritmo se mantém praticamente a mesma.

- MergeSort



No diagrama acima temos uma explicação bem simplificada do funcionamento do MergeSort, a ideia do algoritmo é dividir o vetor inicial desordenado em partes iguais, até que o subvetor tenha um tamanho igual à 1, a partir desse ponto é feito o merge. Os subvetores de tamanho 1 são comparados e mesclados em ordem crescente, um ponteiro percorre cada subvetor e adiciona o menor dos valores no vetor resultante, repetindo o processo até que não hajam mais subvetores.

- Implementação:

Abordagem 1:

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<time.h>

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```



```
void mergeSort(int arr[], int l, int r)
```

```
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

```
void printArray (int arr[], int n)
```

```
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

```
int main(int argc, char* argv[]){
```

```
    int my_rank;
    int np; // número de processos
    int n=1024; // número de elementos
    MPI_Status status;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
```

```
    int r=n/np;
    if(my_rank == 0){
```

```
        int vet[n];
```

```
        srand(time(NULL));
        //gera o vetor aleatório
        for (int i = 0; i < n; i++)    vet[i] = rand() % n;
```

```
        for(int i = 1; i<4;i++){
            MPI_Send(vet+(r*i),r,MPI_INT,i,0,MPI_COMM_WORLD);
        }
```

```
        int resp[n];
        int aux[r];
        for(int i = 0; i<r; i++) resp[i] = vet[i];
        mergeSort(resp,0,r-1);
```

```
        printf("Rank: %d\n", my_rank);
        printArray(resp, r);
        printf("\n");
```

```

        printf("Merge 0:\n");
        printArray(resp, r);
        printf("\n");

    for(int i = 1; i<4; i++){
        MPI_Recv(aux,r,MPI_INT,i,0,MPI_COMM_WORLD,&status);
        for(int j = 0; j < r; j++) resp[r*i+j] = aux[j];
        merge(resp, 0, r*i, r*i+r-1);

        printf("Merge %d: \n", i);
        printArray(resp, r*i + r);
        printf("\n");
    }

    printf("Resultado:\n");
    printArray(resp,n);
}
else{
    int vet[r];
    MPI_Recv(vet,r,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    mergeSort(vet,0,r-1);

    printf("Rank: %d\n", my_rank);
    printArray(vet, r);
    printf("\n");

    MPI_Send(vet,r,MPI_INT,0,0,MPI_COMM_WORLD);
}
MPI_Finalize();
}

```

- Tempos:

4 processos - 1024 elementos:

0,137s
 0,142s
 0,119s
 0,122s
 0,120s

Abordagem 2:

```

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define SIZE 1024

```

```

void merge(int vetor[], int comeco, int meio, int fim) {

```

```
int com1 = comeco, com2 = meio+1, comAux = 0, tam = fim-comeco+1;
```

```
int *vetAux;
```

```
vetAux = (int*)malloc(tam * sizeof(int));
```

```
while(com1 <= meio && com2 <= fim){
```

```
if(vetor[com1] < vetor[com2]) {
```

```
vetAux[comAux] = vetor[com1];
```

```
com1++;
```

```
} else {
```

```
vetAux[comAux] = vetor[com2];
```

```
com2++;
```

```
}
```

```
comAux++;
```

```
}
```

```
while(com1 <= meio){ //Caso ainda haja elementos na primeira metade
```

```
vetAux[comAux] = vetor[com1];
```

```
comAux++;
```

```
com1++;
```

```
}
```

```
while(com2 <= fim) { //Caso ainda haja elementos na segunda metade
```

```
vetAux[comAux] = vetor[com2];
```

```
comAux++;
```

```
com2++;
```

```
}
```

```
    for(comAux = comeco; comAux <= fim; comAux++){           //Move os elementos de volta para o vetor  
original
```

```
    vetor[comAux] = vetAux[comAux-comeco];
```

```
}
```

```
    free(vetAux);
```

```
}
```

```
void mergeSort(int vetor[], int comeco, int fim){
```

```
    if (comeco < fim) {
```

```
        int meio = (fim+comeco)/2;
```

```
        mergeSort(vetor, comeco, meio);
```

```
        mergeSort(vetor, meio+1, fim);
```

```
        merge(vetor, comeco, meio, fim);
```

```
    }
```

```
}
```

```
int array[SIZE];
```

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

```
    int meu_rank, np, tam;
```

```
    tam = SIZE/np;
```

```
    int parte[tam];
```

```
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &meu_rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &np);
```

```

if(meu_rank == 0){
    for(int i = 0; i < 1024; i++){
        array[i] = random()%1024;
    }
}

```

```

MPI_Scatter(array, tam, MPI_INT, parte, tam, MPI_INT, 0, MPI_COMM_WORLD);

```

```

mergeSort(parte, 0, tam-1);

```

```

MPI_Gather(parte, tam, MPI_INT, array, tam, MPI_INT, 0, MPI_COMM_WORLD);

```

```

if(meu_rank == 0) {

    mergeSort(array, 0, 1023);

    printf("Vetor ordenado: \n");
    for(int j = 0; j < 1024; j++) {

        printf("%d ", array[j]);

    }

    printf("\n");
    printf("\n");

}

```

```

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();

```

```

}

```

- **Tempos:**

4 processos - 1024 elementos:

0,092s

0,081s

0,070s

0,088s

0,104s