Implementação do Algoritmo *Bubble Sort* Utilizando o Padrão Divisão e Conquista

Carlos Alberto Franco Maron
Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre – Brasil
carlos.maron@acad.pucrs.br

***Resumo*—Este trabalho propõem a implementação de um algoritmo clássico de ordenação, paralelizados no padrão divisão e conquista utilizando a biblioteca MPI.**

Keywords*—*Programação paralela, MPI, Divisão e Conquista.

I. Introdução

Bubble Sort é um dos mais simples algoritmos para ordenação de vetores. O seu funcionamento consiste em verificar todos os elementos dos vetores, e elevando o maior valor sempre ao topo do vetor [2]. É um algoritmo de complexidade quadrática ($O(n^2)$) no seu pior caso. Por se um algoritmo de baixa eficiência na ordenação, esta implementação propôs o padrão divisão e conquista, para buscar diminuir o tempo de ordenação do vetor.

II. Metodologia e Implementação

A algoritmo foi implementado utilizando a linguagem C em conjunto com a biblioteca MPI. O algoritmo é iniciado com o valor já definido para o tamanho do vetor que deverá ser ordenado, realizando uma alocação dinâmica desse em memória. Utilizando a quantidade de processos para a execução, existe um tratamento no código que verifica se o processo tem a identificação 0. Este é considerado a raiz de toda a árvore, sendo responsável em realizar a carga de valores no vetor anteriormente alocado. O vetor é carregado considerando o pior caso de ordenação. Ao saber o tamanho do vetor e quantos processos folhas (usado no processamento), é necessário encontrar o valor do DELTA. Essa variável determinará se o processo filho que receberá uma fatia do vetor poderá realizar a ordenação com o *bubble sort* (conquista) ou irá dividi-lo em mais partes (divisão) e repassá-las para outros níveis. Se um processo recebe uma fatia do vetor para ser ordenado, ele guarda a identificação do precesso de origem, que será usada posteriormente para devolver o vetor ordenado após a conclusão. Para se ter uma média mais precisa dos tempos de execução, o algoritmo paralelo foi executado 10 vezes. As execuções seguiram 3,7,15,31 e 63 processos utilizando um, dois e quatro nodos.

O ambiente utilizado para os testes foi o Cluster Atlântica, localizado no Laboratório de Alto Desempenho (LAD). Cada servidor possuí as seguintes características: processadores Xeon Quad-Core E5520 2.27 GHz Hyper Threading (HT) com 16 *cores* incluindo HT. E 16 GB de memória RAM.

III. Resultados dos testes

A Figura 1 apresenta os resultados de *speed up* e eficiência utilizando dois nodos. A divisão dos processos folhas deve ser feita de maneira equilibrada. Portanto, no gráfico podemos observar que utilizando 3 processos (2 folhas) o ganho de desempenho não é significante devido a carga (vetor) nos *cores* do processador ainda ser alta. A medida que os processos vão aumentando, a granularidade do vetor vai diminuindo e permitindo que cada processo ordene com mais rapidez uma fatia do vetor.

Tabela I: Tabela de Tempos com Variação dos *Nodes*

Vetor:800.000	1 Node	2 Nodes	4 Nodes
Proc.	Time(s)	Time(s)	Time(s)
3	682	714	714
7	179	210	181
15	85	48	46
31	40	35	13
63	22	12	7

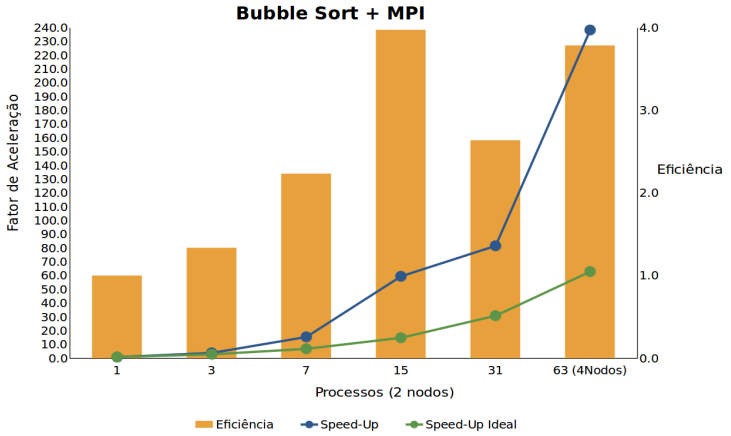

Figura 1: Resultados de *speedup* e eficiência

A tabela I exibe valores em segundos das execuções comparadas com o aumento de nodos. Percebe-se que mesmo havendo concorrência dos processos para uso do processador, a granularidade baixa de cada processo torna a ordenação mais rápida. A Tabela nos ajuda a entender também que o tempo para a troca de mensagens ainda afeta o desempenho do algoritmo. Contudo, utilizando 4 nodos com 63 processos, cada folha receberá menos de um 1MB de dados para fazer o cálculo. Isso significa que o tamanho do pacote gerado pelo MPI será pequeno, efetuando assim uma troca rápida, e permitindo também que esses valores sejam armazenados em cache, já que a arquitetura oferece 8MB no processador.

IV. Dificuldades Encontradas

A depuração dos algoritmos em MPI ainda se torna um desafio, a media que se trabalha com a linguagem, os erros e problemas vão sendo mais claros e não dificultam tanto a solução. Neste trabalho, novamente a estratégia para se obter o paralelismo se tornou um desafio. Contudo, até conseguir chegar em um nível de código aceitável, era comuns erros relacionados ao gerenciamento de memória, e também para atribuição de cargas aos processos filhos.

V. Considerações Finais

A modelagem de programas em paralelo, permite que a aplicações sejam executadas em menos tempo. O padrão de divisão e conquista aplicado nesse problema de ordenação, mostra claramente que ao dividir um algoritmo de complexidade quadrática, o usuário tem um ganho de desempenho quadrático.

Na paralelização deste problema de ordenação, outras extratégias poderiam ser tomadas para melhorar ainda mais o desempenho do algoritmo. Por exemplo, durante a divisão dos vetores para cada processo filho, o processo pai destes poderia ficar com uma porcentagem deste vetor. Dessa forma, enquanto o processo pai aguarda o resuldado do filho, o mesmo vai ordenando uma fatia pequena do vetor.

Referências

[1] Open MPI, *Open MPI: Open Source High Performance Computing*, Capturado em http://www.open-mpi.org/. Acessado em 25 de outubro de 2015.

[2] Bubble Sort, Capturado em https://en.wikipedia.org/wiki/Bubble_sort. Acessado em 25 de outubro de 2015.

Apêndice

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

#define vectorSize 800000

void bs(int n, int * vetor);
void merge(int a[], int m, int b[], int n, int sorted[]);
int *interleaving(int vetor[], int tam);


int main(int argc, char** argv){
    int myRank;
    int procN;
    int i, j, dest;
    int TAG;
    int *vectorN;
    int sizeVectorFolhas;
    int *vectorAUX;
    int halfVector;
    int DELTA;

    double startTime, endTime;

    MPI_Status status;
    MPI_Status masterStatus;

    TAG=1;
    halfVector = vectorSize / 2;
        srand(time(NULL));

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &procN);

    vectorN = (int *) malloc(sizeof(int) * vectorSize);

    DELTA = vectorSize / ((procN + 1) / 2);
    printf("DELTA %d\n", DELTA);



    if (myRank == 0)
    {
        sizeVectorFolhas = vectorSize;

        for(i=0; i<vectorSize; i++)
        {
            vectorN[i] = vectorSize -i;
        }

    }

    else
    {
        MPI_Recv(vectorN, vectorSize, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &masterStatus);
        MPI_Get_count(&masterStatus,MPI_INT,&sizeVectorFolhas);
    }

    if(sizeVectorFolhas <= DELTA)
    {
        bs(sizeVectorFolhas, vectorN);
    }
    else
    {

        halfVector = sizeVectorFolhas / 2;

        MPI_Send(&vectorN[0], halfVector, MPI_INT, myRank * 2 +1, TAG, MPI_COMM_WORLD);
        MPI_Send(&vectorN[halfVector], halfVector, MPI_INT, myRank * 2 +2, TAG, MPI_COMM_WORLD);
```

```
        MPI_Recv(&vectorN[0], halfVector, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD, &status);
        MPI_Recv(&vectorN[halfVector], halfVector, MPI_INT, MPI_ANY_SOURCE, TAG, MPI_COMM_WORLD,
&status);

        vectorN = interleaving(vectorN, vectorSize);

    }
    if(myRank != 0)
    {

        MPI_Send(vectorN, sizeVectorFolhas, MPI_INT, masterStatus.MPI_SOURCE, TAG, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    free(vectorN);

    printf("=>FIM: %d\n", myRank);


    return 0;
}


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

    vectorAUXiliar = (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))
            vectorAUXiliar[i_aux] = vetor[i1++];
        else
            vectorAUXiliar[i_aux] = vetor[i2++];
    }

    free(vetor);

    return vectorAUXiliar;
}

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++;
    }
}
```