Implementação do Algoritmo *Bubble Sort* Utilizando Padrão Mestre-Escravo

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 mestre-escravo utilizando a biblioteca MPI.**

Keywords*—*Programação paralela, MPI, Mestre-Escravo.

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^2) no seu pior caso. Com a implementação do algoritmo, buscou-se a solução de um problema que envolvia a ordenação de diversos vetores. A implementação desse algoritmo partiu do padrão mestre-escravo, aonde vetores foram distribuídos entre processos e *hosts* do cluster.

II. Metodologia e Implementação

O algoritmo foi desenvolvido em linguagem C utilizando a biblioteca MPI. A proposta foi utilizar o algoritmo com a finalidade de realizar a ordenação de mil vetores contendo dez mil posições cada. Com o objetivo de analisar o comportamento do problema paralelizado, cada vetor foi propositalmente criado considerando o pior caso para realizar a ordenação, resultando no grau máximo de complexidade.

O algoritmo desenvolvido foi segmentado em duas partes. Uma parte é responsável em realizar as operações do mestre que consiste na criação dos vetores, distribuição dos vetores entre os escravos e coletas destes após a ordenação. Outra parte do código é de responsabilidade dos escravos. Está parte será responsável em executar o algoritmo *bubble sort* e entregando o resultado ao mestre.

Os resultados foram executados seguindo o aumento de processos utilizando sempre dois nodos do *cluster*. O ambiente utilizado para os testes foi o Cluster Amazônia, 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. A forma de tratar a paralelização do problema neste trabalho permite que a curva de *speed up* acompanhe a linha do ideal. Isso é possível pois a maior parte do tempo os vetores estão sendo processados, realizando a troca de mensagem somente quando se recebe e quando se concluí a ordenação dos vetores. Outro fator que contribui em pequena escala no ganho de *speed up* é a maneira que o mestre está distribuindo as cargas. Inicialmente, ele entrega para cada processo um vetor para ser ordenado. A medida que esses vetores vão sendo entregues ao mestre, o mesmo vai distribuindo mais cargas entre os processos ociosos. Isso permite que todos os processos estejam sempre em execução e a proporção que não existem mais vetores para ordenação, os processos vão sendo eliminados através de uma *tag* de *kill*.

Observa-se que ao executar com 32 processos a curva de *speed up* tem um acentuado declínio. Por ser uma aplicação *cpu-bound* os processos que estão sendo executados no processador, compartilham recursos com processos que estão em *threads* físicas e virtuais (*Hyper Threading*).

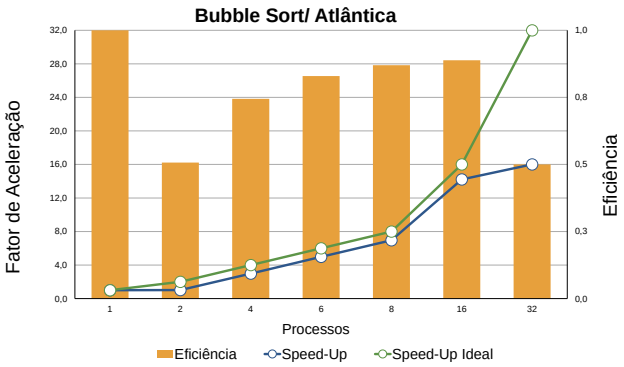

Figura 1: Resultados de *speedup* e eficiência

IV. Dificuldades Encontradas

A depuração dos algoritmos em MPI ainda é um desafio a medida que foi sendo desenvolvido o código, os principais problemas estavam relacionados à alocação de memória para todo o conjunto de vetores. Era frequente os problema de acesso indevido e *segmetantion fault*, comum quando não se tem um gerenciamento adequado da memória.

Outro fator que resultou em vários problemas foi a distribuição dos vetores para cada escravo. Inicialmente tudo era feito uma única vez, entregando todos os vetores para todos os escravos. Porém, quando se trabalhava com vetores grandes e em grande quantidade, fazia com que o *buffer* do MPI se esgotassem, e o programa não tinha sucesso na execução. Esse problema foi resolvido alterando a maneira da distribuição dos vetores para os escravos, entregando-os de acordo como os vetores fossem entregues ao mestre.

V. Considerações Finais

A modelagem de um programa MPI exige diversos pontos que devem ser analisados levando em conta o tipo de padrão utilizado. No mestre-escravo, essa modelagem é bem definida pois você tem um mestre gerenciado os trabalhos e escravos realizando-os.

Para solução do problema elencado neste trabalho, essa modelagem garantiu que o algoritmo atingisse um ganho de aceleração aceitável para um algoritmo paralelo.

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

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

double tempo(){
struct timeval tv;
gettimeofday(&tv,0);
return tv.tv_sec + tv.tv_usec/1e6;
}

void bs(int vectorSize, int *vectorSort)
{
    int i, j, aux;
    for(i = 0; i < vectorSize - 1; i++)
    {
        for(j = i+1; j < vectorSize ; j++)
        {
            if(vectorSort[i] > vectorSort[j])
            {
                aux = vectorSort[i];
                vectorSort[i] = vectorSort[j];
                vectorSort[j] = aux;
            }
        }
    }
}

int main(int argc, char **argv)
{
    //-----------------MPI
    int nWorkers,
        procN,
        myRank,
        dst=1,
        MASTER = 0,
        TAG,
        rankSource,
        workIndex;

        MPI_Status status;

    double endTime, startTime;
    //-----------------MPI


    //-----------------Bubble Sort
    int **vectorSort, *vector;
    int i=0,
        j=0,
        vectorSize = 10000,
        qntVectors = 1000,
        tagKill = qntVectors;

    FILE *vectorIn;
    FILE *vectorOut;

    srand (time (NULL));

    //-----------------Bubble Sort


    MPI_Init (&argc , &argv);


    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &procN);

    nWorkers = procN-1;

    if ( myRank == MASTER)
    {
```

```
        vectorSort = (int **) malloc( qntVectors * sizeof (int *) );
        vector = (int *) malloc( vectorSize * sizeof (int));

        for (i=0; i<qntVectors; i++)
        {
            vectorSort[i] = malloc( vectorSize * sizeof (int) );
        }

        for (j=0; j<qntVectors; j++)
        {
            for (i=0; i < vectorSize; i++)
            {
                vectorSort[j][i] = vectorSize - i;
            }
        }


        //vectorIn = fopen("vetor_gerado.txt","w");
/*      for (j = 0; j < qntVectors; j++)
        {
            for (i = 0; i < vectorSize; i++)
            {
                //fprintf(vectorIn, "V[%d]P[%d][Gerado]-> %d\n",j,i, vectorSort[j][i]);
                printf("V[%d]P[%d][Gerado]-> %d\n",j,i, vectorSort[j][i]);
            }
        }*/
        //fclose(vectorIn);

        //Enviando vetores para os escravos

        startTime = tempo();
        dst=1;
        workIndex = 0;

        for (workIndex = 0; workIndex < nWorkers; workIndex++)
        {
            TAG = workIndex;

            MPI_Send(vectorSort[workIndex], vectorSize, MPI_INT, dst, TAG, MPI_COMM_WORLD);

            dst++;
        }

        for (j = workIndex; j < qntVectors; j++)
        {
            MPI_Recv(vector, vectorSize, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

            TAG = status.MPI_TAG;
            rankSource = status.MPI_SOURCE; //Último RANK Recebido

            for (i=0; i < vectorSize; i++)
                vectorSort[TAG][i] = vector[i]; //Armazena o vetor recebido

            TAG = j;

            MPI_Send(vectorSort[j], vectorSize, MPI_INT, rankSource, TAG, MPI_COMM_WORLD);

        }

        for (i = 0; i < workIndex; i++)
        {
            MPI_Recv(vector, vectorSize, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
&status);

            TAG = status.MPI_TAG;

            for (i=0; i < vectorSize; i++)
                vectorSort[TAG][i] = vector[i];
        }

        for (dst=1; dst < procN; dst++)
```

```
        {
            printf("Destino: %d\n", dst);
            MPI_Send (&tagKill, 1, MPI_INT, dst, tagKill, MPI_COMM_WORLD);
        }
        endTime = tempo();

        printf("Tempo de Execução: %f\n", endTime-startTime);

        //vectorOut = fopen("vetor_ordenado.txt","w");
        /*for (j=990; j <qntVectors; j++)
        {
            for (i=0; i < vectorSize; i++)
            {
                //fprintf(vectorOut, "V[%d]P[%d][Ordenado]-> %d\n", j,i, vectorSort[j][i]);
                printf("V[%d]P[%d][Ordenado]-> %d\n",j,i, vectorSort[j][i] );
            }
        }

        //fclose(vectorOut);*/

        printf("free vectors\n");

        for (i=0; i<qntVectors; i++)
            free(vectorSort[i]);

        free(vectorSort);
        free(vector);

        printf("master sleep 5\n");
        sleep(5);

        printf("master done\n");

    }
/*
        Processo Escravo
*/
    else
    {
        int *vectorSort;

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

        while(1)
        {

            MPI_Recv(vectorSort, vectorSize, MPI_INT, MASTER, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            TAG = status.MPI_TAG;

            int source = status.MPI_SOURCE;

            if (TAG != tagKill)
            {
                bs(vectorSize, vectorSort);
                MPI_Send(vectorSort, vectorSize, MPI_INT, MASTER, TAG, MPI_COMM_WORLD);
            }
            else
            {
                free(vectorSort);
                break;
            }
        }
        printf("slave %d done\n", myRank);
    }

    MPI_Finalize();
    return 0;
}
```