Implementação do Algoritmo *Bubble Sort* Utilizando o Padrão Fases Paralelas

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 fases paralelas utilizando a biblioteca MPI.**

Keywords*—*Programação paralela, MPI, Fases Paralelas.

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 sempre elevando o menor valor ao topo do vetor [2]. É um algoritmo de complexidade quadrática ($O(n^2)$) no seu pior caso. Por ser um algoritmo de baixa eficiência na ordenação, esta implementação propôs o padrão fases paralelas, para buscar diminuir o tempo de ordenação do vetor.

II. Metodologia e Implementação

O algoritmo foi implementado utilizando a linguagem C. Com o padrão fases paralelas, cada processo é iniciado com um vetor considerando o pior caso de ordenação. Esse vetor é alocado de maneira dinâmica, e juntamente nesse espaço de memória, existe um espaço extra para alocar a parcela de troca dos vetores. Os resultados se basearam no tempo de ordenação de um vetor com 800.000 posições, distribuídos em dois nodos, através de 4,6,8,16 e 32 processos. A parcela de troca, foi definida através de um cálculo de porcentagem, chegando ao valor de 25%.

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 da ordenação de um vetor de 800.000 posições. O *bubble sort* é um algoritmo ineficiente para ordenação de grandes vetores, pois seu grau de complexidade é quadrático. Porém, o fato de você conseguir dividir um problema desse grau, faz com que o seu ganho de aceleração seja praticamente quadrático. Utilizando 4 processos o ganho é considerável, levando em conta que cada processo deve realizar no mínimo nove interações entre seus vizinhos para que o vetor fique totalmente ordenado. A media que o vetor vai sendo divido em mais processos, o ganho de aceleração se mantém, porém sendo prejudicado pelo *hypertreading* (HT) com 32 processos.

O ganho neste modelo de programação fica limitado devido a ocorrências das chamadas do algoritmo de ordenação. Isso acontece pois, logo após o vetor ser iniciado, ele deve realizar uma primeira ordenação. Para verificar se os vizinhos estão ordenados entre si, cada processo já envia 25% do vetor para o processo vizinho. Ao confirmar a inconsistência, os processos realizam a ordenação da parcela recebida, com a mesma proporção do vetor local. Ou seja, uma segunda ordenação é feita com os 25% recebidos, com mais 25% do vetor local. Quando essa operação termina, o processo retorna a mesma quantia recebida, e novamente realiza uma ordenação.

A tabela I nos permite expandir o entendimento sobre o modelo de fases paralelas. Nela é apresentado uma comparação das parcelas de trocas utilizadas nos testes. Para esse exemplo, foi utilizado um vetor de 300.000 posições, distribuídos em dois nodos através de 2,4,6,8,16 e 32 processos. É evidente que nesse modelo, o desempenho está amarrado ao tanto que os processos trocarão para

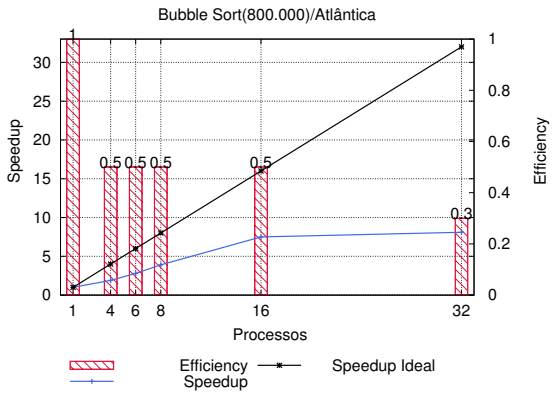

Figura 1: Resultados de *speedup* e eficiência

realizar a ordenação. Como a parcela é contrária ao aumento dos processos, utilizando mais processos equivale a realizar mais trocas. Contudo, a utilização da rede para realizar as trocas, não afeta com significância o tempo de execução, pois o pacote gerado pelo MPI se torna pequeno em relação à banda disponível. Porém, é necessário muito mais trocas quando usado uma porcentagem pequena, pois cada vetor recebido também é pequeno. E quando essa parcela aumenta pode acontecer dos processos estarem trocando valores que não precisariam ser trocados.

Nestes testes, como foi utilizado um vetor menor, a parcela de 50% se mostra com um bom desempenho. Porém, só é possível reduzir os tempos de execução, mas o ganho de aceleração fica equivalente aos testes com o vetor de 800.000.

Tabela I: Tabela contendo os tempos e a quantidade de interações.

Vetor:300.000	**2 Nodes Time/[Interactions]**		
Proc.	**50%**	**20%**	**5%**
2	401.22[3]	478.21[6]	1148.83[21]
4	177.93[5]	233.96[11]	584.97[41]
6	112.20[7]	154.13[16]	392.51[61]
8	83.23[9]	115.21[21]	295.74[81]
16	42.02[17]	60.35[41]	156.95[162]
32	39.44[34]	57.61[81]	146.62[322]

IV. Dificuldades Encontradas

A implantação desse modelo de programação foi o que exigiu mais trabalho. Inicialmente, a maneira de controlar o fluxo das trocas não fica muito clara, e isso resulta em diversos problemas, que vão desde a comunicação dos processos no MPI até a manipulação de memória.

Problemas foram encontrados durante a ordenação do vetor em cada processo. Durante a troca da parcela do vetor, a alocação do vetor recebido era feita no endereço de memória incorreto, e resultava ao final da execução um vetor com as posições totalmente zeradas.

V. Considerações Finais

Neste trabalho foram apresentados resultados com o padrão de programação fases paralelas, ordenando um vetor de 800.000 posições utilizando o *bubble sort*. Em relação aos outros padrões estudados, que envolveram a ordenação com o algoritmo, fases paralelas permite um ganho no desempenho limitado. Utilizando tamanhos de vetores diferentes, as curvas de *speedup* ficam semelhantes.

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

void imprime(int *nVector, int n);
void bubble_sort(int *list, int n);

//----------Controle dos nVectores
int *nVector,
    sizeVector,
    partVector=0,
    startVector,
    endVector,
    i=0,
    stop,
    valueNeighbor,
    change;

    FILE *vectorIN;
    FILE *vectorOUT;

double startTime, endTime;
//----------Controle dos nVectores

//----------MPI
int procN,
    myRank,
    TAG=0;

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

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

    MPI_Status status;

    sizeVector = 800000;

    partVector = sizeVector / procN;
    change = (partVector * 0.20);


    int sizeAditional = partVector + change;
    int contador = 0;

    nVector = (int*) malloc(sizeAditional * sizeof(int));
    int statusVector[procN];

    startVector = sizeVector - (myRank * partVector);

    srand (time (NULL));

    //inicializa o vetor
    for (i=0; i<partVector; i++)
    {
        nVector[i] = startVector - i;
    }
    //fim


/*  vectorIN = fopen("vetor_gerado.txt","w");
    for (i=0; i<partVector; i++)
    {
        printf("[Rank %d gerado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
        fprintf(vectorIN, "[Rank %d gerado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
    }
    fclose(vectorIN); */

/*  char aux[256];
```

```
    sprintf (aux,"%d", myRank);
    strcat (aux, "_.txt");
    vectorIN = fopen(aux,"w");

    for (i=0; i<partVector; i++)
    {
        //printf("[Rank %d gerado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
        fprintf(vectorIN, "[Rank %d gerado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
    }
    fclose(vectorIN);*/



    if (myRank == 0)
    {
        startTime = MPI_Wtime();
    }

    stop = 0;
    while(stop != 1)
    {
        for (i=0; i<procN; i++)
            statusVector[i]=0;

        bubble_sort(&nVector[0], partVector);

        if (myRank>0)
        {
//          printf("Meu ID: %d vou eviar %d para %d\n", myRank,nVector[0], myRank-1 );
            MPI_Send(&nVector[0], change, MPI_INT, myRank-1, TAG, MPI_COMM_WORLD );
        }


        if (myRank != procN-1)
        {
            MPI_Recv(&nVector[partVector], change, MPI_INT, myRank+1, TAG, MPI_COMM_WORLD, &status);

//              printf("[%d] > [%d]\n", nVector[partVector-1], nVector[partVector]);
                if (nVector[partVector-1]>nVector[partVector])
                {
                    statusVector[myRank] = 1;
                }
        }

        for (i=0; i < procN; i++)
            MPI_Bcast(&statusVector[i], 1, MPI_INT, i,MPI_COMM_WORLD);

        stop = 1;
        for (i = 0; i < procN; i++)
        {
            if (statusVector[i] == 1)
            {
                stop = 0;
            }

        }

        if (stop == 0)
        {
            bubble_sort(&nVector[partVector-change], change*2);

            if (myRank != procN-1)
                MPI_Send(&nVector[partVector], change, MPI_INT, myRank+1, TAG, MPI_COMM_WORLD);
            if (myRank>0)
                MPI_Recv(&nVector[0], change, MPI_INT, myRank-1, TAG, MPI_COMM_WORLD, &status);
        }
        contador++;

    }

/*  vectorOUT = fopen("vetor_ordenado.txt","w");
    for (i = 0; i < partVector; i++)
```

```
    {
            fprintf(vectorOUT, "[Rank %d ordenado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
            printf("[ordenado]Posicao[%d] Valor[%d]\n", i, nVector[i] );
    }
    fclose(vectorOUT);*/

    if (myRank == 0)
    {
        endTime = MPI_Wtime();

        printf("Tempo de execução: %f\n", endTime-startTime);
        printf("Interações: %d\n", contador);
        printf("Numero de processos: %d\n", procN);

    }

/*  char aux2[256];
    sprintf (aux2,"%d", myRank);
    strcat (aux2, ".txt");
    vectorOUT = fopen(aux2,"w");
    for (i=0; i<partVector; i++)
    {
        //printf("[Rank %d gerado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
        fprintf(vectorOUT, "[Rank %d ordenado]Posicao[%d] Valor[%d]\n",myRank, i, nVector[i] );
    }
    fclose(vectorOUT);*/


    MPI_Finalize();
    free(nVector);
    return 0;
}


void imprime(int *nVector, int partVector){
    int i;
    for(i=0;i<partVector;i++){
        printf("[%d]\t",nVector[i]);

    }
    printf("\n");
}

void bubble_sort(int *list, int n)
{
  long c, d, t;

  for (c = 0 ; c < ( n - 1 ); c++)
  {
    for (d = 0 ; d < n - c - 1; d++)
    {
      if (list[d] > list[d+1])
      {
        t         = list[d];
        list[d]   = list[d+1];
        list[d+1] = t;
      }
    }
  }
}
```