Relatório de Entrega de Trabalho

Disciplina de programação Paralela (PP) – Prof. César De Rose

Aluno: Renan Alves Silveira, Thomaz Silveira.

Exercício: trabalho 3 de MPI: Fases Paralelas (FP)

1) Introdução

O Trabalho consiste na implementação paralela de um programa que realiza a ordenação de um vetor de tamanho 1.000.000 utilizando o modelo fases paralelas. Para a ordenação serão utilizados o algoritmo de ordenação *Bubble Sort* e com o número de processos variando entre 16 e 32. O Objetivo principal é mensurar os tempos de execução das versões para comparar com os resultados obtidos utilizando as mesma configurações porém com o modelo divisão e conquista e avaliando os tempos de ordenação.

2) Implementação

A implementação é baseada no pseudo-código fases paralelas disponibilizado no moodle. O Algoritmo inicializa o vetor de trabalho com uma quantidade 1/np. Utilizando SPMD, o código é enxuto e todos processos utilizam o mesmo código, pois todos realizam a mesma tarefa, apenas realizando um tratamento diferente para os nodos das extremidades e nodos intermediários. A implementação é dividida em 3 principais tarefas que cada nodo realiza:

1) Processamento local, onde o nodo ordena o seu saco de trabalho localmente utilizando o algoritmo Bubble Sort.
2) teste de condição de parada, onde o nodo verifica se está ordenado com o nodo adjacente, e informa para os demais, atraves da função MPI_Bcast().
3) troca para convergência, é feita em três etapas: primeiro envia para esquerda, realiza uma ordenação com a parcela recebida, devolve a parcela que sobrou.

estas etapas se repetem até que todos estejam ordenados com seus nodos adjacentes. No processamento local cada nodo utilizando o algoritmo de ordenação, ordena o seu vetor de trabalho. Após a ordenação, cada nodo envia para direita o seu maior elemento, o nodo da direita recebe o elemento e compara com o seu menor elemento, e se o seu menor elemento for maior que o maior elmento recebido, então seta ‘1’ no vetor de controle ‘vet_ctrl[]’, caso contratio o valor sera ‘0’. Apos todos processos realizarem as verificações, cada um envia via MPI_Bcast uma mensagem contendo o valor referente ao seu campo no vetor de controle, para que todos possam atualizar os seus vetores locais, assim mantendo uma sincronia do estado da ordenação. Um teste é realizado neste vetor para caso todos estejam ordenados, isto é com o valor ‘1’ em todas posições do vetor então a ordenação está completa e o proceso é finalizado. Caso contrário, os nodos trocam parte do seu vetor de trabalho local com os demais nodos e repetem-se todas etapas novamente. A definição da parcela se da por uma fração do tamanho do vetor local, ou seja uma porcentagem do vetor é calculada e alocada no vetor, para que possamos realizar as trocas sem perda da informação que o vetor contém. Uma das vantagens dessa implementação é que torna a programaçã mais simples e intuitiva, mostrando que o MPI é uma ferramenta robusta e versátil para implementação de algoritmos paralelos e distribuidos.

3) Dificuldades encontradas

Sair do pseudo-código para o programa em c, Realizar debug em partes como a convergencia , entender como seria realizadas no MPI as trocas, foram necessários alguns testes de mesa para o endentimento deste procedimento, onde a implementação é razoavelmente simples porem o entendimento exige um aprofundamento sobre oque está acontecendo em todo o processo.

4) Análise de Desempenho

As medida dos tempos das versões paralela e sequencial do modelo fases paralelas para os diferentes números de processos:

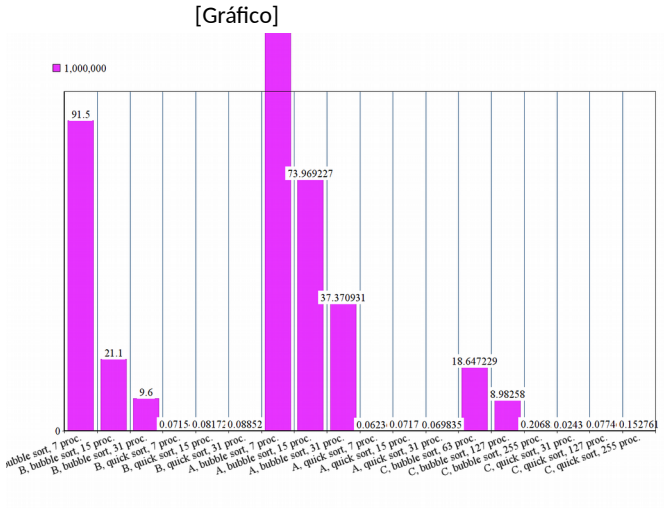
MODELO FASES PARALELAS

MEDIDAS DE EXECUÇÃO PARA 1’000’000 ELEMENTOS NO SACO DE TRABALHO.

N° Processos	Algoritmo	Código	Tempo
16	Bubble Sort	Sequencial	370.50
32	Bubble Sort	Sequencial	366.15

Obtivemos o seguinte gráfico de comparação entre o modelo fases paralelas e divisão e conquista, para 1’000’000 elementos e 32 processos, com o mesmo algoritmo de ordenação Quick Sort em 2 nós na Grad:

[Gráfico]



5) Observações Finais

Como o esperado a versão paralela obteve resultados de tempo melhores que a versão sequencial, concluindo que mesmo com as trocas de mensagens entre os processos na versão paralela, ainda foi superior a versão sequencial.

Analisando o comparativo entre

Analisando o gráfico de comparação entre os modelos conclui-se que ....

Titulo: Trabalho 4 da disciplina de Programação

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

#define VET_SIZE 1000 // Trabalho Final com o valores 100.000 e 1.000.000
//#define DEBUG 1

// BSORT
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++;
}
}

int main(int argc, char **argv){
MPI_Status status; // Message status
double t1, t2; // Count exectuion time

int my_rank; // Process ID
int proc_n; // Number of process
  int aux;              // Store auxiliary values

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

t1 = (my_rank == 0)? MPI_Wtime(): 0; // Time counter on process 0

  #define LAST (proc_n - 1)       // Last process
int i;                          // for counters
  unsigned char end = 0;   // Control the main loop
  unsigned char k;                // Process counter
int psize = VET_SIZE/proc_n;    // tamanho do vetor do processo
  int pTochange = psize/25;       // Size to change date with others
  int left ,right;                // Auxiliar vector for change operation
  unsigned char vet_ctrl[proc_n]; // Control vector
  int *vector = (int *)malloc((pTochange + psize) * sizeof(int)); // Data vector with extra size

  left = (my_rank !=0) ? my_rank - 1:0 ;          // My left neighboor
  right = (my_rank < LAST) ? my_rank + 1: LAST;   // My right neighboor
  memset(vet_ctrl, 0, sizeof(vet_ctrl));

for (i = 0; i < psize; i++)
  {
    vector[i] = (proc_n - my_rank) * psize - i;
  }

  #ifdef DEBUG
  printf("[%d]vector: ", my_rank);
  for (i = 0; i < psize; i++)
    printf("%d ", vector[i]);
  printf("\n\n");
  #endif
  while(!end) {

    //*****************************
    //#1. Local ordenation
    //*****************************

    bs(psize, vector);

    // Send to the right, if im not the last process.
    if (my_rank != LAST){
      MPI_Send(&vector[psize - 1], 1, MPI_INT, right, 0, MPI_COMM_WORLD);

    }
    // Recieve from left if im not 0 process.
```

```
    if (my_rank != 0){
      MPI_Recv(&aux, 1, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
      /* If the left element size is less then my least element
      set ordenetion vector, if not clear the vector.
       */
      if (aux < vector[0]){
        vet_ctrl[my_rank] = 1;
      }
      else{
        vet_ctrl[my_rank] = 0;
      }
    }
    // Exeption case
    if (my_rank == 1){
      MPI_Send(vector, 1, MPI_INT, left, 0, MPI_COMM_WORLD);
    }
    // Exception case, recive from 1
    if (my_rank == 0){
      MPI_Recv(&aux, 1, MPI_INT, right, 0, MPI_COMM_WORLD, &status);
      /* If the element size from second process is greatter then my last element
      set ordenetion vector, if not clear the vector.
       */
      if(aux > vector[psize-1] ){
        vet_ctrl[my_rank] = 1;
      }else{
        vet_ctrl[my_rank] = 0;
      }
    }

    //*****************************
    //#2. Bcast status of ordenation
    //*****************************

    for (i = 0; i < proc_n; i++)
    {
      MPI_Bcast(&vet_ctrl[i], 1, MPI_UNSIGNED_CHAR, i, MPI_COMM_WORLD);
    }
    // Stop condition, verify if all neighbors are ordenate
    k = 0;
    for (i = 0; i < (proc_n*2)-2; i++){
      if (vet_ctrl[i] == 1) {
        k++;
      }
    }
    if (k == proc_n){
      end = 1;
      break;
    }

    //*****************************
    // #3. Converge
    //*****************************

    // Send to left if im not the first process.
    if (my_rank != 0) {
      /* first, send my portion, and wait to recive from neightbor if im no the least process*/
      MPI_Send(vector, pTochange, MPI_INT, left, 0, MPI_COMM_WORLD); // send to left
    }

    if (my_rank != LAST){
      // Wait for right to send
      MPI_Recv(&vector[psize], pTochange, MPI_INT, right, 0, MPI_COMM_WORLD, &status);
      // Ordenate vector without my left portion
      bs(psize,vector + pTochange);
      // Send back the right portion
      MPI_Send(&vector[psize], pTochange, MPI_INT, right, 0, MPI_COMM_WORLD);
    }
    if (my_rank !=0){
      MPI_Recv(vector, pTochange, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
    }

  } // End While
  // At this point, our vector is ordenate like a charm!!!

  // End time.
  t2 = (my_rank == 0)?MPI_Wtime():0;

  #ifdef DEBUG
  printf("[%d]Vector: ", my_rank);
```

```
  for (i = 0; i < psize; i++)
    printf("%d ", vector[i]);
  printf("\n\n");
  #endif

  if (my_rank == 0)
  {
    printf("Elapsed: %.4f s\n\n", t2 - t1);
  }

  free(vector);
  MPI_Finalize();
  return 0;
}
```