Relatório de Entrega de Trabalho
Disciplina de programação Paralela (PP) – Prof. César De Rose
Aluno: Fábio Delazeri Riffel; Lásaro Curtinaz Dumer **Usuário:** pp12806; pp12818
Exercício: Trabalho 3 de MPI e OpenMP **Entrega:** 01/12/2015

1) Implementação

Conforme o enunciado, utilizamos o modelo mestre escravo para a divisão do trabalho, onde o mestre (utilizando MPI) delega a cada escravo um vetor para ser ordenado, o escravo realiza o trabalho de ordenação do vetor com o uso do OpenMP e o devolve ao mestre.

Para gerar a carga de trabalho, basta especificar o tamanho do vetor. Este será gerado com valores inteiros ordenados de forma decrescente.

A ordenação no escravo pode utilizar um dos dois algoritmos solicitados: bubble sort ou quick sort. E também pode ser paralela (usando OpenMP) ou não. O padrão é ser executado um bubble sort paralelo, para fazer a alternância com quick sort e não paralelo, utiliza-se os seguintes argumentos, em conjuntos ou separadamente:

-q : para utilização do quick sort
-s : para execução sequencial

2) Dificuldades encontradas

Primeiramente fizemos uma implementação inicial que funcionava corretamente, porém não utilizava as versões do Bubble Sort e do Quick Sort descritas no enunciado, sendo elas versões que embutiam em seu funcionamento o OpenMP.

Uma vez descoberto nosso engano, trocamos nossa execução de modo a separar o vetor recebido pela parte MPI em pequenas partes, e estas sim são utilizadas em paralelo na execução dos dois métodos com o OpenMP.

Dificuldades no acesso fora da PUCRS, através do servidor sparta. O que nos atrasou para a medição de tempos de execução do programa, devido a contratempos e a alta demanda e alocação das maquinas do LAD, e posteriores erros que elas nos apresentavam, fomos obrigados a executar em nossas maquinas pessoais, que tem configurações inferiores e limitaram nossos testes.

3) Testes

Conforme já mencionado, para termos comparações consistentes utilizamos sempre como entrada um vetor de valores inteiros ordenados de forma decrescente, que é o pior caso do Bubble Sort e para o Quick Sort. Desta forma temos uma maior quantidade de esforço, o que facilita a análise.

Todos os testes foram executados utilizando um notebook com um processador Intel Celeron de 1.5 GHz com dois núcleos físicos e dois em Hyper-threading.

Para a comparação utilizamos 1000 vetores de 100.000 elementos utilizando o quick sort e para o bubble sort utilizamos 100 vetores de 100000 elementos. Sendo executados tanto de forma sequencial, quanto paralela.

Apesar de mantermos o uso de quatro processos (núcleos) MPI, executamos com o uso de 4 e 8 threads no OpenMP.

4) Analise do desempenho

Com base no tempo sequencial e nos tempos paralelos, podemos ver que para o algoritmo de quick sort o ganho não foi significativo, já para o algoritmo bubble sort temos um ganho mais significativo de performance ao usar um número maior de threads, e também podemos ver que os resultados foram melhores do que os esperados no Speed-Up ideal.

.

Algoritmo	Núcleos	Threads	Tempo de Execução (s)	Speed-Up	Speed-Up Ideal	Eficiência
Quicksort	4	1	32,050	1,0	1	0,250
	4	8	31,660	1,0	8	0,032
Bubblesort	4	1	1316,080	1,0	1	0,250
	4	4	270,430	4,9	4	0,304
	4	8	143,030	9,2	8	0,288

Tabela 1-Execuções

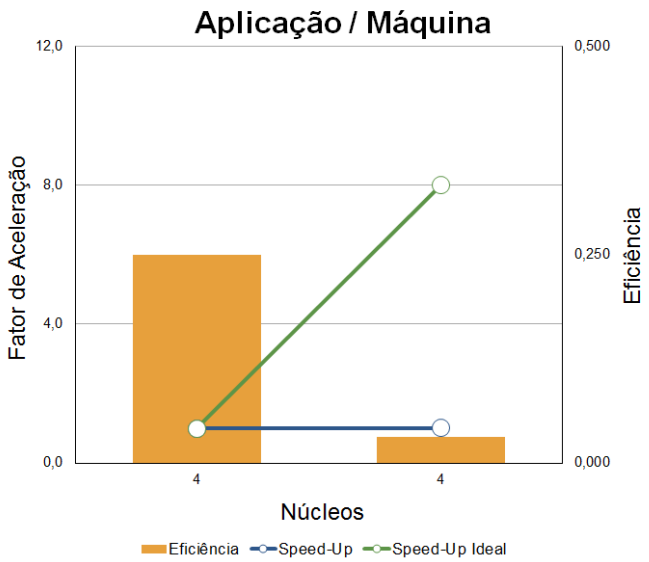

Figura 1-Execuções para Quick Sort

Como podemos verificar na tabela e no gráfico acima, o algoritmo de quick sort não obteve o ganho desejado ao se usar uma quantidade maior de threads, tornando a eficiência das mesmas muito inferior a versão sequencial.

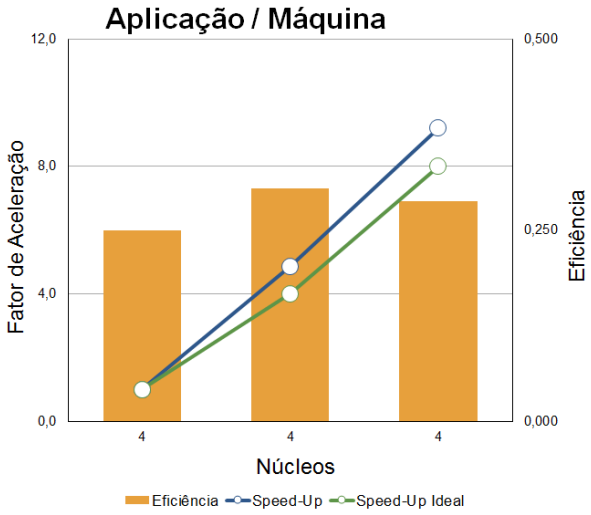

Figura 2-Execuções para Bubble Sort

Para o algoritmo de bubble sort podemos concluir que o ganho de performance foi satisfatório, já que superou o Speed-Up ideal. A eficiência por thread não diminuiu muito, atingindo um plateau, que acreditamos que poderá diminuir para um número meio de threads, o qual não testamos devido a nossas restrições de hardware.

5) Observações Finais

A necessidade de avaliar dois algoritmos para a paralelização utilizando threads se mostrou importante para concluirmos que este processo as vezes é melhor aproveitado dependendo do problema em que é utilizado, como vimos no caso do bubble sort.

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include "mpi.h"
#define GET_WORK 0
#define WORK_DONE 1
#define WORK 2
#define SUICIDE 3
#define NUM_ARRAYS 2
#define ARRAYS_SIZE 20
#define FALSE 0
#define TRUE 1
//#define DEBUG 1
//#define PRINTV 0
#define OPENMPTHREADNUMBER 4

// BUBBLESORT
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 compare (const void * a, const void * b){return ( *(int*)a - *(int*)b );}
const char * printTag(int tag){
    if(tag== GET_WORK){return "GET_WORK";}
    else if(tag==WORK_DONE){return "WORK_DONE";}
    else if(tag==WORK){return "WORK";}
    else if(tag==SUICIDE){return "SUICIDE";}
    else{return "Invalid Tag";}
}

const double curMilis(){
    struct timeval  tv;
    gettimeofday(&tv, NULL);
    return ((tv.tv_sec) * 1000 + (tv.tv_usec) / 1000.0) +0.5; // convert tv_sec & tv_usec to millisecond
}


main(int argc, char** argv){
    int my_rank;
    int proc_n;
    int **saco;
    int toOrder[ARRAYS_SIZE];
    int ordered[ARRAYS_SIZE];
    int i,j,s;
    int *vetor_aux;
    double t1, t2;

    srand(time(NULL));
    int r = rand();

    int qkSort = FALSE;
    int parallel = TRUE;
    size_t optind;
```

```
        for (optind = 1; optind < argc && argv[optind][0] == '-'; optind++) {
            switch (argv[optind][1]) {
                case 'q': qkSort = TRUE; break;
                case 's': parallel = FALSE; break;
                default:
                fprintf(stderr, "Usage: %s [-qs]\n", argv[0]);
                exit(EXIT_FAILURE);
            }
        }

        MPI_Status status; /* Status de retorno */
        MPI_Init(&argc , & argv); // funcao que inicializa o MPI, todo o cÃ³digo paralelo esta abaixo
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
        int slavesAlive = proc_n-1;
        int workDist[proc_n][1];

        if ( my_rank == 0 ){
            t1 = MPI_Wtime();       // contagem de tempo inicia neste ponto
            saco = (int **)malloc(NUM_ARRAYS * sizeof(int *));//alocação do vetor para os vetores
            if(saco == NULL)
            {
                printf("out of memory\n");
                return -1;
            }else{
                int valor = ARRAYS_SIZE * NUM_ARRAYS;//inicialização do valor
                for(i = 0; i < NUM_ARRAYS; i++)
                {
                    saco[i] =(int *) malloc(ARRAYS_SIZE * sizeof(int));//alocação de cada vetor e seu lugar
                    if(saco[i] == NULL)
                    {
                        printf("out of memory row %d\n",i);
                        return -1;
                    }else{
                        for(j = 0; j< ARRAYS_SIZE; j++)
                        {
                            saco[i][j] = valor;//atribuição do valor ao vetor
                            valor--;
                        }
                    }
                }
            }

            int next = 0;
            //mestre faz primeiro envio de tarefas para os escravos
            for(s=1;s<=slavesAlive;s++){
                if(next>=NUM_ARRAYS){//se o numero de tarefas ja se esgotou, termina o escravo
                    MPI_Send(&next, 1, MPI_INT,s, SUICIDE, MPI_COMM_WORLD);
                    slavesAlive--;
                }else {
                    MPI_Send(saco[next], ARRAYS_SIZE, MPI_INT,s, WORK, MPI_COMM_WORLD);//envia o vetor para o escravo 's'
                    workDist[s][0]=next;
                    next++;
                }
            }

            #ifdef DEBUG
            for(i = 0; i < proc_n; i++)
            {
                printf("workDist[%d]=%d\n",i,workDist[i][0]);
            }
            #endif
            while(slavesAlive > 0){//enquanto existirem escravos vivos, fica esperando mensagens
                MPI_Recv(ordered, ARRAYS_SIZE, MPI_INT,MPI_ANY_SOURCE, WORK_DONE, MPI_COMM_WORLD, &status);
//espera o vetor do mesmo escravo q enviou o indice
                memcpy(saco[workDist[status.MPI_SOURCE][0]],ordered,ARRAYS_SIZE*sizeof(int));//coloca o vetor ordenado na
matriz
                if(next>=NUM_ARRAYS){//se o numero de tarefas ja se esgotou, termina o escravo
                    MPI_Send(&next, 1, MPI_INT,status.MPI_SOURCE, SUICIDE, MPI_COMM_WORLD);
                    slavesAlive--;
                }else {//se nao envia a proxima tarefa
                    MPI_Send(saco[next], ARRAYS_SIZE, MPI_INT,status.MPI_SOURCE, WORK, MPI_COMM_WORLD);//envia o vetor
para o escravo
```

```
                workDist[status.MPI_SOURCE][0]=next;
                #ifdef DEBUG
                printf("workDist[%d]=%d\n",status.MPI_SOURCE,workDist[status.MPI_SOURCE][0]);
                #endif
                next++;
            }
        }
        printf("[%f]@master leaving...\n",curMilis());
        t2 = MPI_Wtime();       // contagem de tempo termina neste ponto
        printf("[%f]@[%d]MPI_Wtime measured work time to be: %1.2f\n",curMilis(),my_rank, t2-t1);
    }
    else
    {
        int tag = WORK;
        do{
            MPI_Recv(toOrder, ARRAYS_SIZE, MPI_INT,0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);//...fica esperando o
vetor em seguida
            tag = status.MPI_TAG;
            if(tag == WORK){//recebeu um vetor para ordenar
                if(parallel){
                    omp_set_num_threads(OPENMPTHREADNUMBER); // disparar 4 threads pois se trata de uma m◆quina Quad-
Core
                    if(qkSort)
                    {
                        int th_id;
                        int nthreads;

                        int vetor_auxiliar[(ARRAYS_SIZE/OPENMPTHREADNUMBER)];
                        int toOrderCopy[ARRAYS_SIZE];

                        for (i=0;i<=ARRAYS_SIZE;i++)
                        {
                            toOrderCopy[i] = toOrder[i];
                        }

                        #pragma omp parallel private(th_id, nthreads, vetor_auxiliar) shared(toOrder)
                        {
                            th_id = omp_get_thread_num();
                            nthreads = omp_get_num_threads();
                            int ini = 0;
                            int end = 0;

                            ini = (th_id)*(ARRAYS_SIZE/nthreads);
                            end = ini+(ARRAYS_SIZE/nthreads);

                            for (i=0;i<(ARRAYS_SIZE/nthreads);i++)
                            {
                                vetor_auxiliar[i] = toOrderCopy[ini+i];
                            }

                            qsort (&vetor_auxiliar[0], (ARRAYS_SIZE/nthreads), sizeof(int), compare);//ordena o vetor

                            #pragma omp critical
                            {
                                for (i=0;i<(ARRAYS_SIZE/nthreads);i++)
                                {
                                    ini = ARRAYS_SIZE-((th_id+1)*(ARRAYS_SIZE/nthreads));
                                    toOrder[ini+i] = vetor_auxiliar[i];
                                }
                            }
                        }
                        //interleaving(toOrder,ARRAYS_SIZE,4);
                        //vetor_aux = interleavingNew(toOrder,ARRAYS_SIZE,4);
                    }
                    else // BUBBLESORT
                    {
                        int th_id;
                        int nthreads;

                        int vetor_auxiliar[(ARRAYS_SIZE/OPENMPTHREADNUMBER)];
                        int toOrderCopy[ARRAYS_SIZE];

                        for (i=0;i<=ARRAYS_SIZE;i++)
```

```
                {
                    toOrderCopy[i] = toOrder[i];
                }

                #pragma omp parallel private(th_id, nthreads, vetor_auxiliar) shared(toOrder)
                {
                    th_id = omp_get_thread_num();
                    nthreads = omp_get_num_threads();
                    int ini = 0;
                    int end = 0;

                    ini = (th_id)*(ARRAYS_SIZE/nthreads);
                    end = ini+(ARRAYS_SIZE/nthreads);

                    for (i=0;i<(ARRAYS_SIZE/nthreads);i++)
                    {
                        vetor_auxiliar[i] = toOrderCopy[ini+i];
                    }
                    bs ((ARRAYS_SIZE/nthreads),&vetor_auxiliar[0]);//ordena o vetor
                    #pragma omp critical
                    {
                        for (i=0;i<(ARRAYS_SIZE/nthreads);i++)
                        {
                            ini = ARRAYS_SIZE-((th_id+1)*(ARRAYS_SIZE/nthreads));
                            toOrder[ini+i] = vetor_auxiliar[i];
                        }
                    }
                }
                //interleaving(toOrder,ARRAYS_SIZE,4);
                //vetor_aux = interleavingNew(toOrder,ARRAYS_SIZE,4);
            }
        }
        else{
            if(qkSort){
                qsort (toOrder, ARRAYS_SIZE, sizeof(int), compare);//ordena o vetor
            }
            else{
                bs(ARRAYS_SIZE,toOrder);
            }
        }
        MPI_Send(toOrder,ARRAYS_SIZE, MPI_INT,0, WORK_DONE, MPI_COMM_WORLD);//envia o vetor para o mestre
      }
    }while(tag != SUICIDE);//se a ultima tag foi a de suicidio, termina execução
  }
  MPI_Finalize();
}
```