**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

1. 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.

1. 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.

1. 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.

.

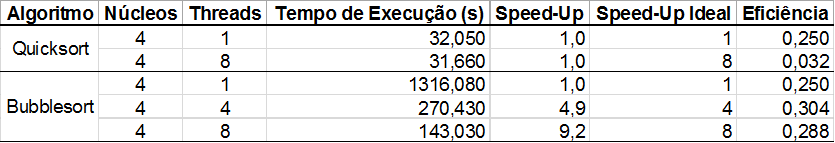


Tabela -Execuções

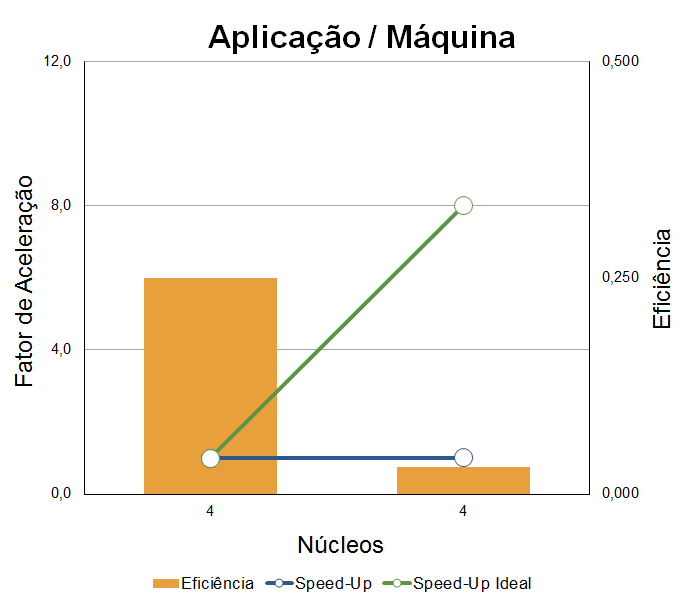


Figura -Execuções para Quick Sort

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.

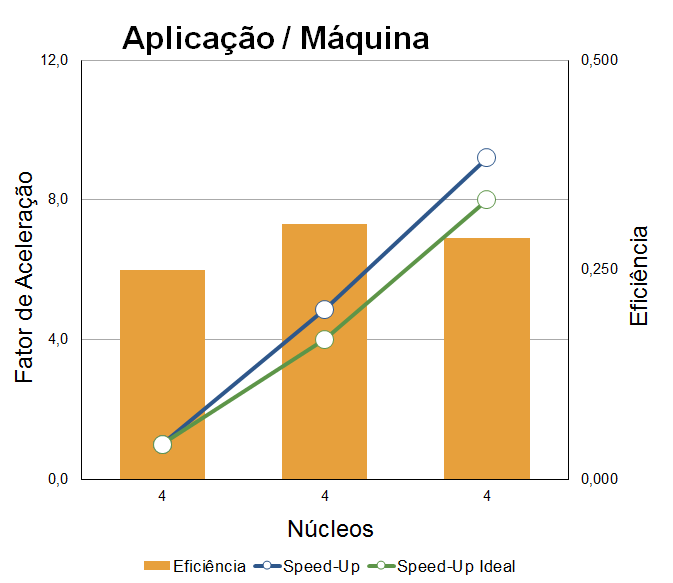


Figura -Execuções para Bubble Sort

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.

1. 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();

}