Programação Paralela
Trabalho I

Giovanni Cupertino, Matthias Nunes, *Usuário pp12820*

I. Introdução

O objetivo do trabalho é desenvolver uma solução que ordene diversos vetores utilizando o algoritmo Quick Sort. Os vetores contém cem mil elementos e são, no total, mil vetores que estão na ordem inversa de valores indo de noventa e nove mil novecentos e noventa e nove na primeira posição até zero na ultima posição. Para a abordagem paralela do trabalho, utilizou-se a estrutura mestre escravo utilizando biblioteca MPI. Para esta abordagem temos um dos processos como mestre, que é responsável por mandar mensagens com o vetor a ser ordenado, avisar aos escravos para que terminem suas funções (suicídio) e para reorganizar o vetor na sua posição original na estrutura. Os outros processos (escravos) recebem o vetor, o ordenam e devolvem ao mestre e, enquanto estão ativos, ficam pedindo um vetor para ordenar.

II. Análise dos Resultados Obtidos

Em primeira análise é possível observar que o tempo de resposta para dois processos é maior que quando executado sequencialmente, isso ocorre pelo fato de que só temos um processo realizando a ordenação e o outro como mestre, havendo um tempo extra principalmente pela troca de mensagens entre os dois, como pode ser visto na tabela I.

Núcleos	Tempo de Execução(s)	Speed-Up	Speed-Up Ideal	Eficiência
1	6,59000	1,00	1	1,00
2	10,31665	0,63877	2	0,31939
4	3,77560	1,74542	4	0,43635
5	2,51584	2,61941	5	0,52388
8	2,63324	2,50262	8	0,31283
11	2,26792	2,90575	11	0,26416
12	2,31676	2,84449	12	0,23704
16	2,38038	2,76846	16	0,17303
20	2,70099	2,43985	20	0,12199
24	2,58161	2,55267	24	0,10636
28	2,53986	2,59463	28	0,09267
32	2,69780	2,44273	32	0,07634

Tabela I: Resultados obtidos

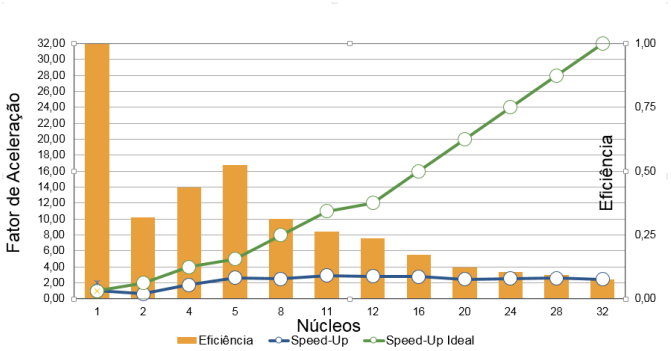

Figura 1: Gráfico gerado a partir da tabela

Essas pequenas diferenças, como a troca de mensagens, a reorganização dos vetores nas suas posições inicias e a comunicação entre os 2 nós utilizados também reduzem a eficiência e criam diferenças entre o speed-up ideal e o real. Devido a velocidade do algoritmo de ordenação o tempo para realizar a tarefa depois de cinco processos permaneceu bastante semelhante, já que os escravos terminam a sua tarefa antes mesmo do mestre conseguir enviar uma nova para todos os processos que estão solicitando. Algumas alternativas para maior utilização dos núcleos, para os casos com mais de cinco, seriam possuir mais de um mestre permitindo maior distribuição de vetores aos escravos(cada um controlando escravos diferentes e passando os vetores quando lhes forem requeridos) e a utilização de um algoritmo de ordenação mais lento que proporcionaria mais tempo ao mestre para distribuir vetores aos escravos, ou seja, antes que tivesse novas requisições.

O fato de se estar utilizando dois nós e a biblioteca MPI permite um balanceamento da carga igualitário dos processos nos núcleos físicos (8 em cada uma das máquinas com capacidade de simular 16 threads). Mesmo utilizando capacidades acima de dezesseis processos (*hyper-threading*), não foi possível observar melhoras significativas nos tempos de execução e por consequência no speed-up e na eficiência.

III. Dificuldades Encontradas

Não foram encontradas dificuldades na implementação desse trabalho, nem na utilização da biblioteca MPI.

src/sequential.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROWS 1000
#define COLUMNS 100000

int vet[ROWS][COLUMNS];

int
compare (const void* a, const void* b)
{
  return *((const int*) a) - *((const int*) b);
}

int
main (int argc, const char* argv[])
{
  time_t start, stop;

  start = clock();

  int i, j, k = COLUMNS;
  for (i = 0; i < ROWS; i++)
    {
      for (j = 0; j < COLUMNS; j++)
        {
          vet[i][j] = k;
          k--;
        }
      k = COLUMNS;
    }

  for (i = 0; i < ROWS; i++)
    {
      qsort(vet[i], COLUMNS, sizeof(int), compare);
    }

  stop = clock();

  float diff = ((float)(stop - start) / 1000000.F) * 1000;
  printf("Time: %.0fms\n\n", diff);

  return 0;
}
```

src/parallel.c

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

#define ROWS 1000
#define COLUMNS 100000
#define WORKTAG 1
#define DIETAG 2

int vet[ROWS][COLUMNS];

int
compare (const void* a, const void* b)
{
  return *((const int*) a) - *((const int*) b);
}

int
master (void)
{
  double t1,t2;
  t1 = MPI_Wtime();

  int proc_n;
  int rank;
  int work = 0;

  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

  //Populate the matrix
  int i, j, k;
  for (i = 0; i < ROWS; i++)
    {
      k = COLUMNS;
      for (j = 0; j < COLUMNS; j++)
        {
          vet[i][j] = k;
          k--;
        }
    }

  //Seed the slaves
  for (rank = 1; rank < proc_n; rank++)
    {
      MPI_Send(vet[work], COLUMNS, MPI_INT, rank, WORKTAG, MPI_COMM_WORLD)
          ;
      work++;
    }

  //Receive a result from any slave and dispatch a new work request
  int save_path = 0;
```

```
52    while (work < ROWS)
53      {
54        MPI_Recv(vet[save_path], COLUMNS, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
55        MPI_Send(vet[work], COLUMNS, MPI_INT, status.MPI_SOURCE, WORKTAG,
              MPI_COMM_WORLD);
56        work++;
57        save_path++;
58      }
59
60    //Receive last results
61    for (rank = 1; rank < proc_n; rank++)
62      {
63        MPI_Recv(vet[save_path], COLUMNS, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
64        save_path++;
65      }
66
67    //Kill all the slaves
68    for (rank = 1; rank < proc_n; rank++)
69      {
70        MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
71      }
72
73    t2 = MPI_Wtime();
74    fprintf(stderr, "Time: %fs\n\n", t2-t1);
75
76    return 0;
77  }
78
79  int
80  slave (void)
81  {
82    int* work = malloc(COLUMNS * sizeof(int));
83    MPI_Status status;
84
85    //Receive and work until it dies
86    while (1)
87      {
88        MPI_Recv(work, COLUMNS, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &
              status);
89
90        if (status.MPI_TAG == DIETAG)
91          {
92            free(work);
93            return 0;
94          }
95
96        qsort(work, COLUMNS, sizeof(int), compare);
97
98        MPI_Send(work, COLUMNS, MPI_INT, 0, 0, MPI_COMM_WORLD);
99      }
100
101   return 1;
```

```
102 }
103
104 int
105 main (int argc, char** argv)
106 {
107   int my_rank;
108   int proc_n;
109
110   MPI_Init(&argc , &argv);
111
112   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
113   MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
114
115   if ( my_rank == 0 )
116     master();
117   else
118     slave();
119
120   MPI_Finalize();
121
122   return 0;
123 }
```