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 2 de MPI (Divisão e Conquista) **Entrega:** 20/10/2015

1) Implementação

Conforme o enunciado, utilizamos o modelo divisão e conquista para aplicar o algoritmo Bubble Sort. Assim como visto em aula, utilizamos um delta (valor utilizado como base para decidir quando dividir e quando conquistar) variável, e este é calculado com base no tamanho especificado do vetor (carga de trabalho) e no número de processos paralelos.

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

Uma vez iniciado o programa, a execução se dá pelo cálculo do delta, seguido de avaliações que optam por dividir quando o valor é maior que o delta, sendo que a divisão é sempre em duas partes, ou conquistar, aplicando o algoritmo Bubble Sort. Uma vez executada a conquista o pedaço ordenado é devolvido ao processo pai, assim sucessivamente até a ordenação completa do vetor.

2) Dificuldades encontradas

Primeiramente fizemos uma implementação inicial que parecia funcionar corretamente, porém ao testar com um número grande de processos paralelos, percebemos que haviam erros decorrentes de arredondamentos ao dividir partes de tamanho ímpar, tendo então que fazer alterações para o tratamento desses casos.

Como havia sido citado no enunciado, fizemos uma segunda versão, onde ao invés de dividir o trabalho apenas para os filhos, deixamos também parte do trabalho para ser processado pelo pai. Porém nossa implementação não foi satisfatória e por isso não foi utilizada nos testes e análise.

3) Testes

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. Desta forma temos uma maior quantidade de esforço, o que facilita a análise.

Todos os testes foram executados utilizando de forma exclusiva dois nodos do cluster Atlantica.

Para a melhor comparação utilizamos dois tamanhos de vetor, sendo estes 100.000 e 1.000.000. Além da variação de número de processos, inclusive utilizando Hyper-threading no caso de 31 processos.

4) Analise do desempenho

Com base no tempo sequencial e nos tempos paralelos, podemos ver que nos dois casos, temos um ganho significativo de performance ao usar um número maior de processos, e também podemos ver que os resultados foram bem melhores do que os esperados no Speed-Up ideal.

No caso da execução com o vetor de tamanho 100.000, vemos que mesmo com uma leve melhora de tempo na execução com 15 e 31 processos, chegamos em um plateau, ou seja, podemos ver que com 15 processos nosso resultado é aproximadamente 45 vezes mais rápido e o de 31 é somente em torno de 50 vezes.

Núcleos	Tempo de Execução (s)	Speed-Up	Speed-Up Ideal	Eficiência
1	44,973	1,0	1	1,
3	11,21	4,0	2	1,
7	2,906	15,5	4	2,
15	0,973	46,2	8	3,
31	0,862	52,2	16	1,

Tabela 1-Execuções para 100000

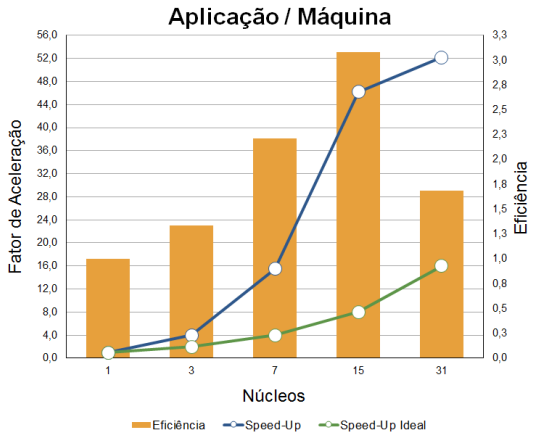

Figura 1-Execuções para 100000

Onde não temos um ganho representativo. Isto se dá, pois para esta quantidade de trabalho e este grande número de processos, acabamos gastando muito esforço nas divisões sem possuir carga de trabalho suficiente para justificar todas estas divisões, ainda mais somando o custo do uso de Hyper-threading, subutilizando os processos.

Já quando temos uma maior carga de trabalho (1.000.000), percebemos que o ganho foi realmente significativo, inclusive ao compararmos os resultados de 15 e 31 processos, com ganhos de aproximadamente 60 e 110 vezes respectivamente.

Núcleos	Tempo de Execução (s)	Speed-Up	Speed-Up Ideal	Eficiência
1	4267,809	1,0	1	1,0
3	1121,496	3,8	2	1,3
7	281,600	15,2	4	2,2
15	74	57,6	8	3,8
31	37,882	112,7	16	3,6

Tabela 2-Execuções para 100000

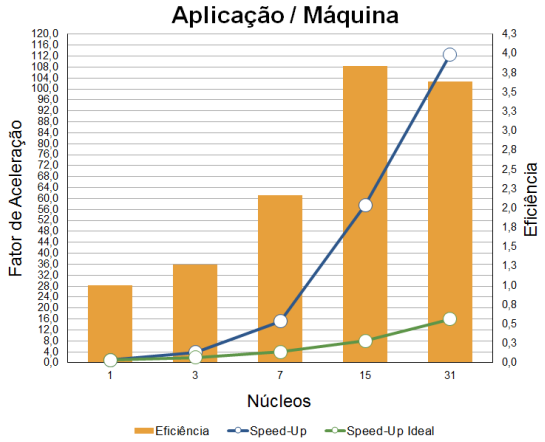

Figura 2-Execuções para 1000000

5) Observações Finais

Com a análise do desempenho concluímos que um número maior de processos nem sempre oferece um maior ganho em performance. O aumento no número de processos deve ocorrer em conjunto com o tamanho de carga de trabalho, senão acabamos subutilizando o sistema.

```
#include "mpi.h"
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <sys/time.h>

#define ARRAY_SIZE 100000
//#define DEBUG 1
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 *interleaving(int array[], int len)
{
        int *array_aux;
        int i1, i2, i_aux;

        array_aux = (int *)malloc(sizeof(int) * len);

        i1 = 0;
        i2 = len / 2;

        for (i_aux = 0; i_aux < len; i_aux++)
        {
                if (((array[i1] <= array[i2]) && (i1 < (len / 2)))
                                || (i2 == len))
                        array_aux[i_aux] = array[i1++];
                else
                        array_aux[i_aux] = array[i2++];
        }

        return array_aux;
}

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;  /* Identificador do processo */
        int proc_n;   /* Número de processos */
        int tam_vetor = 0;
   int *vetor_aux;         /* ponteiro para o vetor resultantes que sera alocado dentro da rotina */
   int pai;

   //time measure
   double t_before, t_after;

        int message[ARRAY_SIZE]; /* Buffer para as mensagens */
        MPI_Status status; /* Status de retorno */
```

```
    MPI_Init (&argc , & argv);

  t_before = MPI_Wtime();

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
  int delta = ARRAY_SIZE/((proc_n+1)/2);
    int vetor[ARRAY_SIZE];

  if ( my_rank != 0 )
  {
    MPI_Recv(&vetor[0],ARRAY_SIZE,MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_INT, &tam_vetor);
    pai = status.MPI_SOURCE;
  }
  else
  {
    tam_vetor = ARRAY_SIZE;
    // sou a raiz e portanto gero o vetor - ordem reversa
    int i;
    for (i=0 ; i<ARRAY_SIZE; i++)          /* init array with worst case for sorting */
      vetor[i] = ARRAY_SIZE-i;
  }

  //root
  if(tam_vetor <= delta+1)
  {
    //conquista (bubble sort)
    bs(tam_vetor, vetor);    // conquisto
    if ( my_rank !=0 ){
      MPI_Send(&vetor,tam_vetor,MPI_INT, pai, 1, MPI_COMM_WORLD);
    }
  }
  else
  {
    //dividir e mandar para os filhos
    //2 * my_rank + 1 e 2 * my_rank + 2
    int tam1 = tam_vetor/2;
    int tam2 = tam_vetor/2;
    int filhoE = 2 * my_rank + 1;
    int filhoD = 2 * my_rank + 2;
    if (tam_vetor%2 != 0)
      tam2 = (tam_vetor+1)/2;

    MPI_Send(&vetor[0], tam1, MPI_INT, filhoE, 1, MPI_COMM_WORLD);
    MPI_Send(&vetor[tam1], tam2, MPI_INT, filhoD, 1, MPI_COMM_WORLD);
    //recebe dos filhos
    MPI_Recv(&vetor[0],tam1,MPI_INT, filhoE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Recv(&vetor[tam1],tam2,MPI_INT, filhoD, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    // intercalo vetor inteiro
    vetor_aux = interleaving(vetor, tam_vetor);
    if ( my_rank !=0 )
    {
      MPI_Send(&vetor_aux[0],tam_vetor,MPI_INT, pai, 1, MPI_COMM_WORLD);
    }
  }
  // mando para o pai
  if ( my_rank ==0 )
  {
    t_after = MPI_Wtime();
    printf("[%f]@master[%d]:Time measured=%1.3fs ;Array len=%d\n",curMilis(),my_rank, t_after - t_before,tam_vetor);
    }

  MPI_Finalize();
}
```