Enunciado T1: Trabalho MPI - Divisão e Conquista (D&C)

O objetivo do trabalho é implementar, usando a biblioteca MPI, uma versão paralela seguindo o modelo divisão e conquista, de um programa que ordena um grande vetor usando o algortimo *Bubble Sort* (o programa sequencial está no final desta especificação). Após implementado, o programa deve ser executado no cluster atlantica com até 2 nós (variando de 1, 3, 7, 15 e 31 processos, este último com HT) para realização das medições de desempenho para três tamanhos de vetor, 100.000 e 1.000.000 elementos (sem os prints de tela). Deve ser feita também uma versão onde cada nó separa uma parte do vetor para ordenar localmente. Cada grupo (de dois integrantes) deve entregar um relatório em .pdf de uma página com a análise dos resultados e uma página com o código (seguir modelo proposto).

O nó raiz da árvore verifica se o vetor inicial será dividido ou conquistado. A verificação da condição de conquista é uma comparação com um valor fixo, para o vetor com 40 elementos pode ser por exemplo <= 10 (assim o algoritmo só desce 2 níveis da árvore binária e se utiliza dos 7 processadores alocados). Se for dividido, cada filho da árvore binária recebe uma metade do vetor. Por sua vez, cada filho verifica se ira conquistar ou novamente dividir o vetor. Quando as folhas da árvore decidirem que o vetor já está pequeno o bastante para ser conquistado, o vetor é ordenado com o algoritmo de *Bubble Sort* (código abaixo) e devolvido para o respectivo pai. O pai por sua vez, intercala os dois vetores recebidos (código sequencial abaixo) e devolve ao seu pai, até que o nó folha restabeleça o vetor original ordenado (Figura 1).

Para ordenar um vetor de 1.000.000 de elementos com um processo rodando em núcleo na atlântica o algoritmo de ordenação troca simples (bs) abaixo precisou de 74 minutos. Para 3 processos (um raiz e dois filhos) rodando em 3 núcleos precisou de 18 minutos.

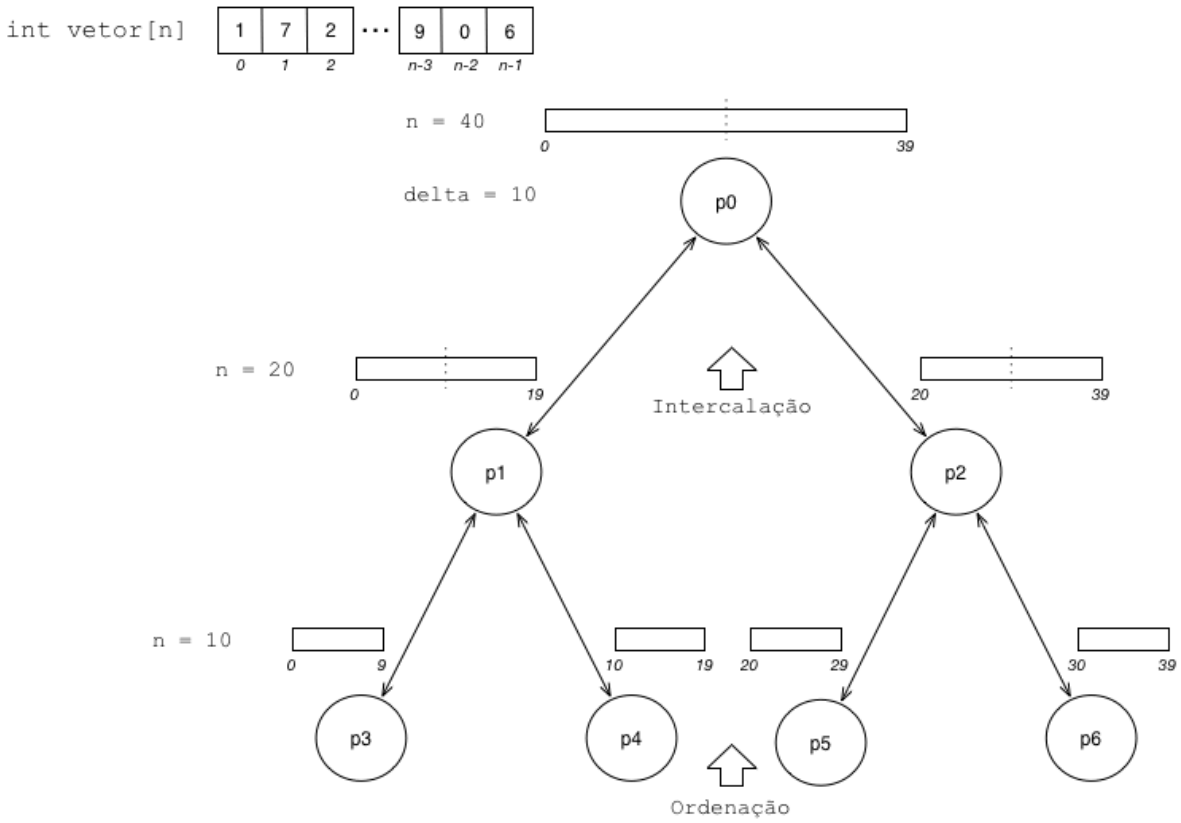

Figura 1: Funcionamento do modelo de divisão e conquista na ordenação de um único vetor

Os itens para avaliação são:

- execução da versão sequencial do algoritmo *Bubble Sort* para o vetor inteiro (inicializar com dados em ordem decrescente);
- implementação da versão paralela SPMD do algoritmo MPI descrito acima seguindo o modelo divisão e conquista;
- medição dos tempos de execução para a versão sequencial em uma máquina qualquer do aluno ou laboratório e da versão paralela para 100.000 e 1.000.000 de elementos (usando 2 nós exclusivos da máquina atlantica totalizando15 processadores - cada nó possui 8 processadores físicos);
- cálculo do *speed up* e da eficiência para os casos de teste (variando de 1, 3, 7, 15 e 31 processos, este último com HT);
- análise do balanceamento da carga na execução do programa paralelo nos diferentes níveis da árvore;
- análise do ganho obtido com HT;
- análise do ganho obtido com a inclusão de um percentual para ordenação local;
- clareza do código (utilização de comentários e nomes de variáveis adequadas);
- relatório no formato .pdf com duas páginas (coluna dupla), um apara a análise dos resultados e outra para o código, seguindo as recomendações fornecidas no moodle (submissão em sala de entrega moodle até as 17:30 do dia da entrega - duas semanas);

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

#define DEBUG 1           // comentar esta linha quando for medir tempo
#define ARRAY_SIZE 40     // trabalho final com o valores 10.000, 100.000, 1.000.000

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 vetor[ARRAY_SIZE];
   int i;

   for (i=0 ; i<ARRAY_SIZE; i++)          /* init array with worst case for sorting */
      vetor[i] = ARRAY_SIZE-i;


   #ifdef DEBUG
   printf("\nVetor: ");
   for (i=0 ; i<ARRAY_SIZE; i++)          /* print unsorted array */
      printf("[%03d] ", vetor[i]);
   #endif

   bs(ARRAY_SIZE, vetor);                 /* sort array */

   #ifdef DEBUG
   printf("\nVetor: ");
   for (i=0 ; i<ARRAY_SIZE; i++)                  /* print sorted array */
```

```
      printf("[%03d] ", vetor[i]);
   #endif


   return 0;
}
```

Rotina de Intercalação

```
/* recebe um ponteiro para um vetor que contem as mensagens recebidas dos filhos
e            */
/* intercala estes valores em um terceiro vetor auxiliar. Devolve um ponteiro
para este vetor */          */

int *interleaving(int vetor[], int tam)
{
        int *vetor_auxiliar;
        int i1, i2, i_aux;

        vetor_auxiliar = (int *)malloc(sizeof(int) * tam);

        i1 = 0;
        i2 = tam / 2;

        for (i_aux = 0; i_aux < tam; i_aux++) {
                if (((vetor[i1] <= vetor[i2]) && (i1 < (tam / 2)))
                    || (i2 == tam))
                        vetor_auxiliar[i_aux] = vetor[i1++];
                else
                        vetor_auxiliar[i_aux] = vetor[i2++];
        }

        return vetor_auxiliar;
}
Chamada para a rotina de Intercalação

int *vetor_auxiliar;         /* ponteiro para o vetor resultantes que sera
alocado dentro da rotina */

vetor_aux = interleaving(vetor, tam);
```

Última atualização: terça, 19 Set 2017, 16:21