Estudo da implementação paralela do quicksort no modelo mestre escravo

Diego Pinto da Jornada
pp12804

Daniel Antoniazzi Amarante
pp12802

Table 1: Algoritmo sem ordenação local

Processos	Núcleos	Tempo	Speed-Up	Eficiência
1.000.000	3	1.200	1,0	0,3
1.000.000	7	303,2	4,0	0,6
1.000.000	15	80,7	14,9	1,0
1.000.000	31	35,8	33,4	1,1
100.000	3	12.1	1,0	0,3
100.000	7	4.1	2,9	0,4
100.000	15	1.9	6,4	0,4
100.000	31	1.4	8,1	0,3

Table 2: Algoritmo com ordenação local

Processos	Núcleos	Tempo	Speed-Up	Eficiência
1.000.000	3	502.361595	1,0	0,3
1.000.000	7	92.864404	5,4	0,8
1.000.000	15	22.246438	22,6	1,5
1.000.000	31	10.790653	46,6	1,5
100.000	3	5.4	1,0	0,3
100.000	7	2.0	2,6	0,4
100.000	15	1.4	3,9	0,3
100.000	31	1.2	4,5	0,1

1. INTRODUÇÃO

O objetivo do trabalho é desenvolver uma solução que implemente uma versão paralela, seguindo o modelo divisão e conquista, utilizando a biblioteca *MPI*, de um programa que ordena um grande vetor usando o algortimo *Bubble Sort*. O programa foi testado em vetores de tamanho 100.000 e 1.000.000, utilizando 2 nodos do cluster Atlantica do LAD da PUCRS, com número de processos variando entre 1, 3, 7, 15 e 31, este último com *hyper-threading*. Foram implementadas duas versões do programa, com e sem ordenação local. Foi possível executar o algoritmo em quase todos os casos de teste previstos, exceto o programa sequencial e o programa paralelo com 3 processos do algoritmo não otimizado, mas esses dados nos foram fornecidos pelo professor.

2. MODELO DA SOLUÇÃO

Foram modeladas duas soluções diferentes, uma com ordenação local e a outra sem. Na solução sem ordenação local, o primeiro processo vai pegar o vetor inteiro desordenado, dividir em dois, e passar adiante para que os próximos processos o ordenem, quando ordenado ele recebe de volta e faz o merge dos dois vetores que recebeu ordenado. Os processos vão delegando a função de ordenar o vetor pra baixo até não haver mais processos para passar adiante, nas folhas da árvore. Chegando nas folhas, o processo executa o bubblesort sobre o vetor e envia ele de volta para cima ordenado.

A segunda solução utiliza ordenação local, então, em vez de dividir o vetor em dois, cada processo divide ele em três, guardando uma parte para ordenar localmente, com o bubblesort. Isso trás um ganho de eficiência, pois diminui o tempo que os processos ficam parados, que será analisado posteriormente. A quantidade escolhida para ordenação total é o tamanho total do vetor dividido pelo número de processos, para balancear a carga da melhor maneira.

3. ANÁLISE DOS RESULTADOS

O tempo do processo sequencial foi de 70 minutos, e os outros tempos estão descritos nas tabelas 1 e 2 em segundos. A versão paralela do algoritmo obteve um desempenho superior a versão sequencial, desempenho que foi aumentando conforme o número de processos cresce. isso se dá porque não há dependência entre os processos, e, pela complexidade do *bubblesort* de $O(n^2)$, ordenar as duas metades do vetor paralelamente faz o trabalho 4 vezes mais rápido idealmente. O desempenho só degrada quando os vetores começam a ficar muito pequenos, e o tempo de dividir o vetor, mandar as mensagens e unir de novo não compensa, pois ordenar seria mais rápido.

O balanceamento de carga se mostrou viável para o algoritmo, que teve um desempenho melhor na sua versão otimizada, pois o balanceamento é mais igualitário. Na versão simples a carga é dividida apenas pelas folhas da árvore, contendo $(n + 1)/2$ processos, por a árvore de processamento se tratar de uma árvore binária cheia, devido aos números de processos de teste, mas na versão com ordenação local, todos os processos tentam manter um equilíbrio de trabalho.

No problema proposto, é possível utilizar *hyper-threading* com qualidade, pois o trabalho dos processos é bem independente uns dos outros, as únicas situações em que o uso de *hyper-threading* não trouxe muitos ganhos foi no caso de vetores menores, onde o problema era o número de processos, e não o *hyper-threading* em si.

Incluir um percentual para ordenação local permitiu um balanceamento melhor da carga do programa, trazendo um melhor aproveitamento dos processos e uma melhor eficiência do programa. Na versão simples do programa, apenas em torno da metade dos processos estavam realmente trabalhando fazendo a ordenação, enquanto o resto ficava ocioso esperando pelo resultado, utilizando um percentual local todos os processos trabalham na ordenação, minimizando o tempo ocioso ao máximo. A quantia escolhida para ordenação local é o tamanho do vetor dividido pelo número de processos, com um objetivo de todos os processos ordenarem um tamanho igual de vetor. Isso trás dois benefícios, o primeiro é que diminui o tempo ocioso dos processos, pois todos tem uma quantidade igual de trabalho para ordenar, o único trabalho extra será na hora de unir os vetores ordenados, onde, infelizmente, haverá uma disparidade da carga do trabalho e os vetores mais próximos da raiz trabalharão mais. O outro benefício é que igualando o tamanho dos vetores a serem ordenados diminui o tempo de duração do *bottleneck* do programa, que é o *bubblesort*, pois deixa os vetores ordenáveis no menor tamanho possível.

4. DIFICULDADES ENCONTRADAS

A indisponibilidade de acessar o LAD de fora da PUCRS foi uma dificuldade, pois o tempo para executar os testes era raro e limitado.

sequential.c

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

//#define DEBUG 1
#define ARRAY_SIZE 100000


int main(void)
{
    int vetor[ARRAY_SIZE];
    int i;

    for (i=0 ; i<ARRAY_SIZE; i++)
        vetor[i] = ARRAY_SIZE-i;


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

    bubble_sort(ARRAY_SIZE, vetor);


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

    return 0;
}
```

parallel.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include "bubblesort.h"

#define ARRAY_SIZE 100000
//#define DEBUG 1
#define SIZE_TAG 0
#define SEND_UP_TAG 1
#define SEND_DOWN_TAG 2

int
main(int argc, char *argv[])
```

```
{
    double t1, t2;
    t1 = MPI_Wtime();

    int my_rank;
    int proc_n;
    MPI_Status status;
    int i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_n);

    if (my_rank == 0)
    {
        int array[ARRAY_SIZE];
        for (i = 0; i < ARRAY_SIZE; i++)
            array[i] = ARRAY_SIZE - i;

        #ifdef DEBUG
        printf("Root process unordered:\n");
        for (i = 0; i < ARRAY_SIZE; i++)
            printf("[%03d] ", array[i]);
        printf("\n");
        #endif

        int *result = order(my_rank, array, ARRAY_SIZE, status);

        #ifdef DEBUG
        printf("Root process ordered:\n");
        for (i = 0; i < ARRAY_SIZE; i++)
            printf("[%03d] ", result[i]);
        printf("\n");
        #endif

    }
    else
    {
        int size;

        MPI_Recv(&size, 1,
                 MPI_INT, parent(my_rank), SIZE_TAG,
                 MPI_COMM_WORLD, &status);

        int *array = calloc(size, sizeof(int));

        MPI_Recv(array, size,
                 MPI_INT, parent(my_rank), SEND_DOWN_TAG,
                 MPI_COMM_WORLD, &status);

        if(is_leaf(size, ARRAY_SIZE, proc_n))
        {
            bubble_sort(size, array);

            MPI_Send(array, size,
                     MPI_INT, parent(my_rank), SEND_UP_TAG,
                     MPI_COMM_WORLD);
```

```
        }
        else
        {
            array = order(my_rank, array, size, status);
            MPI_Send(array, size,
                     MPI_INT, parent(my_rank), SEND_UP_TAG,
                     MPI_COMM_WORLD);
        }
    }

    if (my_rank == 0) {
        t2 = MPI_Wtime();
        printf( "Elapsed time is %f\n", t2 - t1 );
    }
    MPI_Finalize();
    return 0;
}

int*
order(int my_rank, int array[], int size, MPI_Status status)
{
    int half_size = size / 2;
    // Send size
    MPI_Send(&half_size, 1,
            MPI_INT, left_child(my_rank), SIZE_TAG,
            MPI_COMM_WORLD);

    MPI_Send(&half_size, 1,
            MPI_INT, right_child(my_rank), SIZE_TAG,
            MPI_COMM_WORLD);

    // Send array
    MPI_Send(array, half_size,
            MPI_INT, left_child(my_rank), SEND_DOWN_TAG,
            MPI_COMM_WORLD);

    MPI_Send(array + half_size, half_size,
            MPI_INT, right_child(my_rank), SEND_DOWN_TAG,
            MPI_COMM_WORLD);

    // Receive response
    MPI_Recv(array, half_size,
            MPI_INT, left_child(my_rank), SEND_UP_TAG,
            MPI_COMM_WORLD, &status);

    MPI_Recv(array + half_size, half_size,
            MPI_INT, right_child(my_rank), SEND_UP_TAG,
            MPI_COMM_WORLD, &status);

    return interleaving(array, size);
}
```

parallel_optimized.c

```
#include <mpi.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "utils.h"
#include "bubblesort.h"

#define ARRAY_SIZE 100000
//#define DEBUG 1
#define SIZE_TAG 0
#define SEND_UP_TAG 1
#define SEND_DOWN_TAG 2

int
main(int argc, char *argv[])
{
    double t1, t2;
    t1 = MPI_Wtime();
    int my_rank;
    int proc_n;
    MPI_Status status;
    int i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_n);
    int piece_size = ARRAY_SIZE / proc_n;

    if (my_rank == 0)
    {
        int array[ARRAY_SIZE];
        for (i = 0; i < ARRAY_SIZE; i++)
            array[i] = ARRAY_SIZE - i;

#ifdef DEBUG
        printf("Root process unordered:\n");
        for (i = 0; i < ARRAY_SIZE; i++)
            printf("[%03d] ", array[i]);
        printf("\n");
#endif

        int *result = order(my_rank, array, ARRAY_SIZE, status, piece_size);

#ifdef DEBUG
        printf("Root process ordered:\n");
        for (i = 0; i < ARRAY_SIZE; i++)
            printf("[%03d] ", result[i]);
        printf("\n");
#endif

    }
    else
    {
        int size;

        MPI_Recv(&size, 1,
                 MPI_INT, parent(my_rank), SIZE_TAG,
                 MPI_COMM_WORLD, &status);
```

```
        int *array = calloc(size, sizeof(int));

        MPI_Recv(array, size,
                 MPI_INT, parent(my_rank), SEND_DOWN_TAG,
                 MPI_COMM_WORLD, &status);

        if(is_leaf3(size, ARRAY_SIZE, proc_n))
        {
            bubble_sort(size, array);

            MPI_Send(array, size,
                     MPI_INT, parent(my_rank), SEND_UP_TAG,
                     MPI_COMM_WORLD);
        }
        else
        {
            array = order(my_rank, array, size, status, piece_size);
            MPI_Send(array, size,
                     MPI_INT, parent(my_rank), SEND_UP_TAG,
                     MPI_COMM_WORLD);
        }
    }
    if (my_rank == 0) {
        t2 = MPI_Wtime();
        printf( "Elapsed time is %f\n", t2 - t1 );
    }
    MPI_Finalize();

    return 0;
}

int*
order(int my_rank, int array[], int size, MPI_Status status, int piece_size)
{

    int third_size = (size - piece_size) / 2;
    int left_size = size - 2 * third_size;
    // Send size
    MPI_Send(&third_size, 1,
             MPI_INT, left_child(my_rank), SIZE_TAG,
             MPI_COMM_WORLD);

    MPI_Send(&third_size, 1,
             MPI_INT, right_child(my_rank), SIZE_TAG,
             MPI_COMM_WORLD);

    // Send array
    MPI_Send(array, third_size,
             MPI_INT, left_child(my_rank), SEND_DOWN_TAG,
             MPI_COMM_WORLD);

    MPI_Send(array + third_size, third_size,
             MPI_INT, right_child(my_rank), SEND_DOWN_TAG,
             MPI_COMM_WORLD);

    //order rest locally
```

```
    bubble_sort(left_size, array + 2 * third_size);

    // Receive response
    MPI_Recv(array, third_size,
             MPI_INT, left_child(my_rank), SEND_UP_TAG,
             MPI_COMM_WORLD, &status);

    MPI_Recv(array + third_size, third_size,
             MPI_INT, right_child(my_rank), SEND_UP_TAG,
             MPI_COMM_WORLD, &status);

    return interleaving3(array, size, piece_size);
}
```

utils.c

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

int *interleaving(int array[], int size) {
    int *aux_array;
    int i1, i2, i_aux;

    aux_array = calloc(size, sizeof(int));

    i1 = 0;
    i2 = size / 2;

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

    return aux_array;
}

int valid(int index, int limit);
int higher_or_invalid(int array[], int value_index, int limit, int
other_value_index);
int *interleaving3(int array[], int size, int slice_size) {
    int *aux_array;
    int i1, i2, i3, i1_limit, i2_limit, i3_limit, i_aux;

    int third_size = (size - slice_size) / 2;

    aux_array = calloc(size, sizeof(int));

    i1 = 0;
    i1_limit = third_size;
    i2 = third_size;
    i2_limit = 2 * third_size;
    i3 = 2 * third_size;
    i3_limit = size;
```

```
    for (i_aux = 0; i_aux < size; i_aux++) {

        if (valid(i1, i1_limit) &&
                higher_or_invalid(array, i2, i2_limit, i1) &&
                higher_or_invalid(array, i3, i3_limit, i1)) {
            aux_array[i_aux] = array[i1];
            i1++;
        } else if (valid(i2, i2_limit) &&
                higher_or_invalid(array, i3, i3_limit, i2)) {
            aux_array[i_aux] = array[i2];
            i2++;
        } else {
            aux_array[i_aux] = array[i3];
            i3++;
        }
    }

    return aux_array;
}

int valid(int index, int limit) {
    return index < limit;
}

int higher_or_invalid(int array[], int value_index, int limit, int
other_value_index) {
    return (array[value_index] > array[other_value_index]) || value_index >= limit;
}

int left_child(int index) {
    return 2 * index + 1;
}

int right_child(int index) {
    return 2 * index + 2;
}

int parent(int index) {
    return (index - 1) / 2;
}

int is_leaf(int current_size, int total_size, int number_of_process) {
    return current_size <= total_size / ((number_of_process + 1) / 2);
}

int is_leaf3(int current_size, int total_size, int number_of_process) {
//  printf("current size: %d, total_size: %d, delta: %d\n", current_size,
total_size, (total_size / number_of_process));
    return current_size < (total_size / number_of_process) * 2;
}
```

bubblesort.c

```
#include "bubblesort.h"
```

```
void
bubble_sort(int size, int* array)
{
    int holder, swap = 1;
    for (int i = 0; i < size-1 && swap; ++i)
    {
        swap = 0;
        int limit = size - i -1;
        for (int j = 0 ; j < limit; ++j)
        {
            int current = array[j];
            int next = array[j + 1];
            if (current > next)
            {
                holder     = current;
                array[j]   = next;
                array[j+1] = holder;
                swap       = 1;
            }
        }
    }
}
```