Estudo da implementação Híbrida do quicksort e do bubblesort no modelo mestre escravo

Diego Pinto da Jornada
pp12804

Daniel Antoniazzi Amarante
pp12802

1. INTRODUÇÃO

O objetivo do trabalho é desenvolver uma solução que implemente uma versão híbrida, seguindo o modelo divisão e conquista, utilizando as biblioteca *MPI* e *OpenMP*, de um programa que ordena uma lista de vetores usando os algortimos *Quicksort* e *Bubblesort*. O programa foi testado em 10.000 vetores de tamanho 100.000, utilizando 4 nodos do cluster Atlantica do LAD da PUCRS, com 5 processos e número de threads variando entre 1, 2, 4 e 8. Foi possível executar o algoritmo utilizando *Quicksort* em todos os casos de teste previstos, porém, o algoritmo utilizando o *Bubblesort* não foi possível ter seu tempo analisado, devido às limitações do laboratório LAD, por seu tempo de execução ser muito grande e os laboratórios estarem disponíveis apenas localmente. O resultado foi comparado com a versão paralela utiizando somente *MPI*, com o mesmo número de processos, nas mesmas máquinas.

2. MODELO DA SOLUÇÃO

Foi implementado um modelo mestre escravo paralelo, onde existem 1 mestre e 4 escravos, 1 escravo operando em cada nodo. Os escravos pedem para o mestre sacos de trabalho para ordenar e devolvem os vetores ordenados. A cada transferência, são enviados 8 vetores para o escravo ordenar, sendo esse número igual ao número maior de threads testadas. Ao receber a submatriz para ordenar, o escravo itera sobre os vetores paralelamente, utilizando o OpenMP, e quando terminada a ordenação, envia os vetores de volta para o mestre ordenados. Quando os escravos pedirem matrizes para ordenar, mas o trabalho estiver finalizado, o mestre vai enviar para os escravos uma mensagem para se suicidar, finalizando sua execução.

3. ANÁLISE DOS RESULTADOS

O programa foi executado utilizando 5 nodos, sendo 4 desses escravos, e com threads variando entre 2, 4, 8 e 16 e o algoritmo de ordenação Quicksort. Não foram observadas melhorias, o motivo disso seria a eficiência do algoritmo quicksort. A troca das mensagens é tão custosa, com as mensagens longas para enviar as matrizes, que a eficiência extra ganha na paralelização da ordenação não parece fazer diferença. Se tivessemos utilizado um ambiente melhor, ou um trabalho mais pesado, seria possível ver essa diferença.

O programa também foi comparado com o programa paralelo MPI puro, utilizando a mesma quantidade de nodos, que envia os vetores um por vez, onde novamente o custo de mandar as mensagens longas pesou. O programa híbrido executou em 39.713 segundos e o puro terminou em 23.897 segundos, e o grande culpado foi a troca de mensagem. Enviar uma mensagem maior ocupa o mestre por mais tempo, e a ordenação dos vetores é tão rápida que logo os escravos estão inativos novamente querendo conversar com o mestre enquanto o mestre está ocupado enviando. Isso só mostra que, no caso previso, o formato utilizado não é eficiente. Para que uma eficiência fosse demonstrada, seria nescessário um problema diferente, como a execução de um algoritmo mais lento de ordenação em vez do quicksort.

Uma situação onde provavelmente veríamos um ganho de desempenho utilizando o programa híbrido seria quando ordenássemos com o bubblesort. O bubblesort é mais pesado computacionalmente, e seria mais fácil notar a melhoria com o uso das threads OpenMP.

Na figura 1, podemos observar o gráfico de speedup, com a variação do número de threads totais, sendo a primeira, com quatro threads, o programa MPI puro, e os outros os híbridos. O gráfico mostra bem a perda tida ao aumentar o tamanho das mensagens, comparado ao algoritmo que envia uma por vez, e também o quanto não foi possível observar aumento de performance utilizando a paralelização do OpenMP.

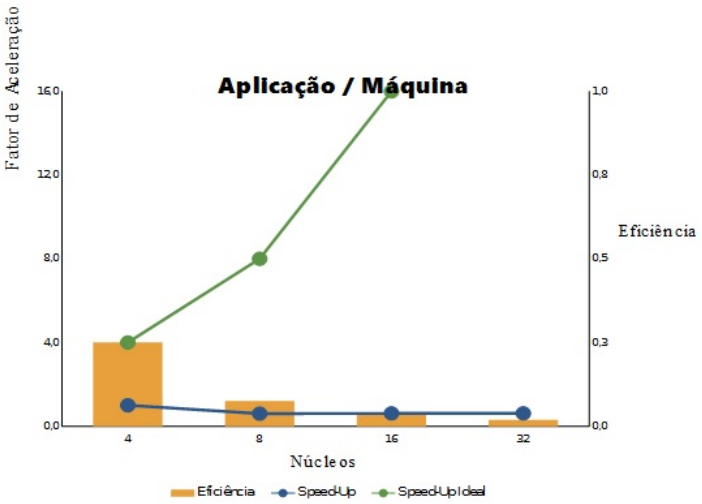

Figure 1: Gráfico de Speed-Up

4. DIFICULDADES ENCONTRADAS

A indisponibilidade do LAD tem sido uma dificuldade recorrente, pois limita o tempo e os horários que podem ser utilizados para testar as soluções. Horários esses que conflitam com aulas e trabalho dos integrantes do grupo.

Houve também dificuldade em entender a biblioteca OpenMP, e devido ao fato de não ter havido uma grande diferença nas amostras coletadas, dificuldade de saber se a paralelização estava, de fato, funcionando.

parallel.c

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

#define COLUMNS 100000
#define ROWS 10000
#define SUICIDE_TAG 2
#define WORK_TAG 1
#define THREADS 8

int matrix[ROWS][COLUMNS];
int run_quick = 1;

void populate_matrix() {
    int i,j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLUMNS; j++) {
            matrix[i][j] = COLUMNS - j;
        }
    }
}

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

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;
            }
        }
    }
}

void print_array(int array[]) {
    int i;
    for (i = 0; i < 10; ++i) {
        printf("%d ", array[i]);
    }
```

```
    printf("\n");
}

void print_matrix()
{
    int i,j;
    for (i = 0; i < ROWS; ++i) {
        for (j = 0; j < 30; ++j) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, char *argv[]){
    double t1, t2;
    int my_rank;
    int proc_n;
    int omp_rank;
    int i;
    int work_sent = 0;
    int work_received = 0;
    run_quick = atoi(argv[1]);
    t1 = MPI_Wtime();
    MPI_Status status;

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

    if (my_rank == 0){
        populate_matrix();
        while(work_sent < ROWS) {
                for (i = 0; i < proc_n - 1 && work_sent < ROWS; ++i){
                        MPI_Send(matrix[work_sent], THREADS*COLUMNS,
                                        MPI_INT, i+1, WORK_TAG,
                                        MPI_COMM_WORLD);
                        work_sent+=THREADS;
                }
                for (i = 0; i < proc_n - 1 && work_received < ROWS; ++i) {
                        MPI_Recv(matrix[work_received], THREADS*COLUMNS,
                                        MPI_INT, i+1, WORK_TAG,
                                        MPI_COMM_WORLD, &status);
                        work_received+=THREADS;
                }
        }
        int terminator = proc_n;
        while (--terminator)
            MPI_Send(0, 0, MPI_INT, terminator, SUICIDE_TAG, MPI_COMM_WORLD);

        print_matrix();
    }
    else{
        while(1) {
                int work_pool[THREADS][COLUMNS];
                MPI_Recv(work_pool, THREADS*COLUMNS,
                                MPI_INT, 0, MPI_ANY_TAG,
```

```
                                MPI_COMM_WORLD, &status);

                if (status.MPI_TAG == SUICIDE_TAG) {
                        MPI_Finalize();
                        return 0;
                }
                #pragma omp parallel for
                for (i = 0; i < THREADS; ++i) {
                    if(run_quick){
                        printf("quick\n");
                        qsort(work_pool[i], COLUMNS, sizeof(int), compare);
                    }
                    else{
                        printf("bubble\n");
                        bubble_sort(COLUMNS, work_pool[i]);
                    }
                }
                #pragma omp barrier
                MPI_Send(work_pool, THREADS*COLUMNS,
                                MPI_INT, 0, WORK_TAG,
                                MPI_COMM_WORLD);
        }
    }
    t2 = MPI_Wtime();
    MPI_Finalize();
    printf( "Elapsed time is %f\n", t2 - t1 );
}
```