Relatório Lab 3

Lucas Toscano Pimentel Appolinário Cerqueira - 2110695
Pedro Henrique Cariello de Freitas - 2011367

Objetivo

O objetivo principal é trabalhar com o paralelismo por meio das threads.

Estrutura do programa e Solução

Breve descrição de cada módulo e as funções implementadas;

- Ex1

Arquivo “paralelismo.c”

```c
#define TAM_VETOR 1000000
#define NUM_THREADS 100

void *paralelismo(void *threadid) {
    int i = (int)threadid;
    int segmento1, segmento2, segmento_clock, *soma, *vetor, size = TAM_VETOR;

    clock_t *time;

    segmento1 = shmget(7000, sizeof(int) * size, IPC_CREAT | S_IRUSR | S_IWUSR);
    segmento2 = shmget(7001, sizeof(int) * NUM_THREADS, IPC_CREAT | S_IRUSR | S_IWUSR);
    segmento_clock = shmget(7002, sizeof(clock_t) * 2, IPC_CREAT | S_IRUSR | S_IWUSR);

    // Attach shared memory segments
    vetor = (int *)shmat(segmento1, 0, 0);
    soma = (int *)shmat(segmento2, 0, 0);
    time = (clock_t *)shmat(segmento_clock, 0, 0);

    soma[i] = 0;
    int forsize = size/NUM_THREADS;
    clock_t time_ver = clock();
    if (time_ver < time[0]) {
        time[0] = time_ver;
    }
    for (int j = 0; j < forsize; j++) {
        vetor[i * forsize + j] = vetor[forsize * i + j] * 2;
        soma[i] = soma[i] + vetor[i * forsize + j];
        // printf("soma %d = %d\n",i,soma[i]);
    }
    time_ver = clock();
    if (time_ver > time[1]) {
        time[1] = clock();
    }

    return NULL;
}
```

1

2

3

4

Esta função é a função que cada thread executará em paralelo.

1- Acessa os segmentos memória compartilhada criados na main utilizando o shmget e usando como referência o tamanho do vetor e o número de threads. Em seguida atribui os valores armazenados nesses espaços de memória às variáveis “vetor”, “soma” e “time”. Assim, é possível realizar a comunicação dos resultados da thread para fora dela.

2- Define o tamanho do vetor que será utilizado pela thread e inicia a medição do tempo de execução.

3- Multiplica cada elemento por 2 e adiciona o resultado ao elemento do vetor soma referente à essa thread.

4- Encerra a medição de tempo.

```c
int main() {
    double time_spent = 0.0;
    int segmento1, segmento2, segmento_clock, *vetor, *soma, size = TAM_VETOR, soma_total = 0;
    clock_t *time;

    segmento1 = shmget(7000, sizeof(int) * size, IPC_CREAT | S_IRUSR | S_IWUSR);
    segmento2 = shmget(7001, sizeof(int) * NUM_THREADS, IPC_CREAT | S_IRUSR | S_IWUSR);
    segmento_clock = shmget(7002, sizeof(clock_t) * 2, IPC_CREAT | S_IRUSR | S_IWUSR);

    // Attach shared memory segments
    vetor = (int *)shmat(segmento1, 0, 0);
    soma = (int *)shmat(segmento2, 0, 0);
    time = (clock_t *)shmat(segmento_clock, 0, 0);

    pthread_t threads[NUM_THREADS];
    int t;

    for(int i = 0; i < size; i++)
      vetor[i] = 5;

    for (t = 0; t < NUM_THREADS; t++) {
        pthread_create(&threads[t], NULL, paralelismo, (void *)t);
    }

    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        soma_total = soma_total + soma[i];
    }

    time_spent += (double)(time[1] - time[0]) / CLOCKS_PER_SEC;

    printf("Time spent: %f seconds\n", time_spent);
    printf("soma_total = %d\n", soma_total);

    shmctl (segmento1, IPC_RMID, 0);
    shmctl (segmento2, IPC_RMID, 0);
    shmctl (segmento_clock, IPC_RMID, 0);
    return 0;
}
```

1 2 3 4 5 6 7

A main realiza os seguintes comandos:

1- Aloca e anexa segmentos de memória compartilhada para o vetor, a soma total e o tempo.

2- Inicializa a array de threads com o quantidade definida na macro NUM_THREADS.

3- Cria um vetor com o número de elementos definidos em TAM_VETOR e inicializa todos eles com o valor 5.

4- Inicia as threads para executar a função paralelismo e aguarda que todas terminem sua execução.

5- Calcula a soma total dos elementos do vetor, que foi acumulada pelas threads.

6- Calcula o tempo total gasto durante a execução das threads e faz as impressões.

7- Libera a memória compartilhada quando o programa termina.

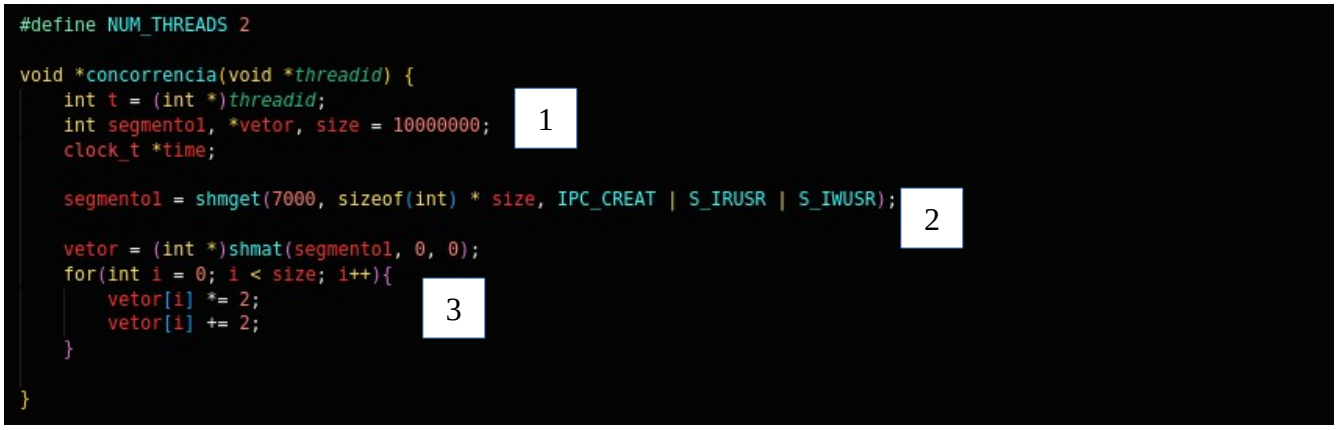
Arquivo “forma_linear_paralelismo.c”

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

#define TAM_VETOR 1000000

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

double time_spent =0.0;
clock_t begin = clock();

// Criando o vetor
int vetor[TAM_VETOR];
int soma = 0;
for(int i = 0; i < TAM_VETOR; i++)
  vetor[i] = 5;

// Somando todos os valores
for(int i = 0; i < TAM_VETOR; i++){
  vetor[i] = vetor[i]*2;
  soma = soma + vetor[i];
}

clock_t end = clock();
time_spent += (double)(end - begin)/CLOCKS_PER_SEC;
printf("Time spent: %f seconds\n", time_spent);

printf("soma = %d\n", soma);
return 0;

}
```

Aqui o cálculo desejado é feito linearmente (um elemento por vez).

- Ex2

Arquivo “concorrencia.c”



A função concorrencia é a função que será executada por cada thread.

1- Recebe um identificador de thread como argumento e o armazena na variável ‘t’, e define o tamanho do vetor.

2- Acessa o segmento memória compartilhada criado na main e atribui o valor armazenado nesse espaço de memória à variável “vetor”.

3- Realiza operações de multiplicação e adição em cada elemento do vetor. Cada elemento é multiplicado por 2 e depois tem 2 adicionado a ele.

A função main é o ponto de entrada do programa e executa as seguintes etapas:

1- Aloca um segmento de memória compartilhada para armazenar um vetor de inteiros com size elementos.

2- Anexa o segmento de memória compartilhada ao espaço de endereçamento do processo.

3- Inicializa o vetor com o valor 5 em cada elemento.

4- Cria duas threads pthreads para executar a função concorrencia. As threads compartilham o mesmo segmento de memória compartilhada, permitindo a concorrência.

5- Aguarda as threads terminarem com pthread_join.

6- Verifica se os valores do vetor são diferentes de 26 e os imprime.

7- Libera o segmento de memória compartilhada quando o programa termina usando shmctl

Observações e conclusões

Ex1

```
[lucas@fedora lab3]$ ./paralelismo
Time spent: 0.030590 seconds
soma_total = 10000000
[lucas@fedora lab3]$ ./forma_linear_paralelismo
Time spent: 0.007681 seconds
soma = 10000000
[lucas@fedora lab3]$
```

Foram testadas diversos valores para as quantidades de elementos no vetor e de threads. Após todas as execuções, percebeu-se que a forma linear de calcular foi sempre mais rápida do que a solução com threads. Uma possível hipótese para esse fato inesperado é que a ocorrência dessa difereça é consequêncua do custo de performance que o processo de criação de threads e de mudança de contexto quando se altera-se entre elas

Ex2

Após a execução, percebe-se que foi impressa uma grande quantidade de valores sendo eles principalmente “12” mas também tendo alguns valores “14”, “22” e “24”. A explicação para isso é uma falta de sincronização entre as duas threads que faz com que o acesso simultâneo à um elemento altere a execução dos comandos. O “12”, por exemplo, ocorre quando só as instruções de uma thread foram executadas sobre o elemento em questão.