Grupo 4 - Tiago Koji Castro Shibata (8988730), Adilson Torres Gregório de Souza (8988254)

Sistemas de Computação de Alto Desempenho

Aula 4

Relatório: Exclusão mútua

- **Introdução**

Em aula, estudamos métodos de exclusão mútua. No laboratório, otimizamos um programa que bloqueia excessivamente e implementamos um programa com uso de estratégias de exclusão mútua.

- **Programas**

No **exercício 1**, recebemos um programa que realiza soma dos elementos de um vetor em paralelo. O programa, no entanto, somava elemento por elemento em um acumulador sequencialmente; dessa forma, apenas uma *thread* tinha acesso ao acumulador por vez e o programa era mais lento que sua versão serial.

Otimizamos o programa para realizar a soma em paralelo. Dividimos o vetor em partições, e cada thread realizou a soma em sua partição. No final, os resultados de cada partição foram somados. A estratégia funciona, pois a função de soma é comutativa.

Implementamos o programa:

```
#include <stdio.h>
#include <sys/sysinfo.h>

#include "mede_time.h"

#define min(a, b)    ((a) < (b) ? (a) : (b))

#define SIZE 2000
shared int a[SIZE];
shared int sum;
shared Semaph S;
shared int nprocs;
shared int partition_size;

task spec init_vetor();
task body init_vetor()
{
    int i;
    forall i = 0 to (SIZE - 1) {
        a[i] = 1.0;
    }
}
```

```
task spec soma_elementos();
task body soma_elementos()
{
    int i;

    forall i = 0 to (nprocs - 1) {
        int private_sum = 0;
         for (int j = i * partition_size; j < min((i + 1) * partition_size, SIZE); j++)
{
            private_sum += a[j];
        }
        lock(&S);
        sum += private_sum;
        unlock(&S);
    }
}

main()
{
    nprocs = get_nprocs();
    partition_size = (SIZE + nprocs - 1) / nprocs;
    printf("nprocs = %d\n", nprocs);
    printf("Partition size: %d\n", partition_size);

    create_sem(&S, 1);
    create nprocs, init_vetor();
    wait_all();
    TIMER_CLEAR;
    TIMER_START;
    create nprocs, soma_elementos();
    wait_all();
    TIMER_STOP;
    printf("Vetor inicializado, Time: %.5fms\n", 1000 * TIMER_ELAPSED);
}
```

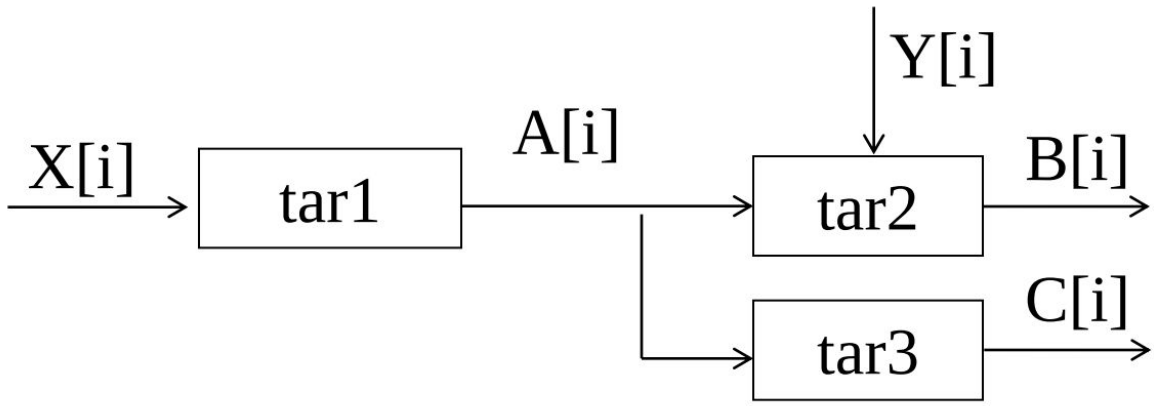
Implementamos também uma versão serial usando um loop *for*. Testamos e analisamos o desempenho de ambas com diferentes tamanhos de vetor:

Número de elementos	Paralelo	Serial
10	1.5ms	0.00012ms
100	1.5ms	0.00018ms
1000	1.5ms	0.00048ms
2000	1.5ms	0.00069ms

4000	1.6ms	0.00095ms
40000	1.6ms	0.012ms
400000	1.8ms	0.14ms
4000000	2.8ms	1.9ms
40000000	10.0ms	16.7ms

Observamos que o *overhead* de criação de thread influencia muito o tempo de processamento, e a versão paralela só mostra vantagens com número grande de elementos. Além disso, como a operação de soma é rápida, outros gargalos (como largura de banda de memória) podem influenciar mais do que o tempo de processamento, por isso os ganhos são muito baixos.

No **exercício 2**, implementamos um algoritmo do estilo *consumidor x produtor*. Uma tarefa (tar1) gera dados e, conforme eles são gerados, são consumidos por outras duas tarefas (tar2 e tar3) ao mesmo tempo, executando em paralelo:



Implementação do programa:

```
#include <stdio.h>

shared int A[2000];
shared int B[2000];
shared int C[2000];
shared int X[2000];
shared int Y[2000];
shared Semaph full_2;
```

```
shared Semaph full_3;
shared Semaph empty_2;
shared Semaph empty_3;

task spec tar_1();
task body tar_1()
{
  int i;

  for (i=0; i< 1999; i++) {
    lock(&empty_2);
    lock(&empty_3);
    A[i] = X[i] * 3;
    printf("A[%d] = %d gerado\n", i, A[i]);
    fflush(stdout);
    unlock(&full_2);
    unlock(&full_3);
  }
}

task spec tar_2();
task body tar_2()
{
  int i;
  B[0] = 1;
  for (i=1; i < 2000; i++) {
    lock(&full_2);
    B[i] = Y[i] + A[i-1];
    printf("B[%d] = %d gerado\n", i, B[i]);
    fflush(stdout);
    unlock(&empty_2);
  }
}

task spec tar_3();
task body tar_3()
{
  int i;
  C[0] = 1;
  for (i=1; i < 2000; i++) {
    lock(&full_3);
    C[i] = A[i-1] * 2;
    printf("C[%d] = %d gerado\n", i, C[i]);
    fflush(stdout);
    unlock(&empty_3);
  }
}
```

```
main()
{
  for (int i = 0; i < 2000; i++) {
    X[i] = i;
    Y[i] = i + 1;
  }
  create_sem(&full_2, 0);
  create_sem(&full_3, 0);
  create_sem(&empty_2, 1);
  create_sem(&empty_3, 1);
  printf("semaforos e vetores criado\n");
  create 1, tar_1();
  create 1, tar_2();
  create 1, tar_3();
  wait_all();
  rem_sem(&full_2);
  rem_sem(&full_3);
  rem_sem(&empty_2);
  rem_sem(&empty_3);
  printf("FIM DO PROGRAMA \n");
  fflush(stdout);
}
```

Para a solução, foi utilizado 2 pares de semáforos, para que a tar1 ao gerar os elementos do vetores sejam consumidos pelas tarefas tar2 e tar3, e só volte a gerar novamente quando ambas consumirem o elemento gerado.