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

Sistemas de Computação de Alto Desempenho
Aula 8

Relatório:
OpenMP - Deadlocks

- **Introdução**

Em aula, estudamos o uso de OpenMP para paralelismo, com foco no uso de deadlocks.

- **Configuração do sistema**

Testes foram realizados em um processador Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz, com 4 núcleos. O sistema utilizado é Linux e os processos foram executados com *niceness* mínimo para que outros processos do sistema interferissem menos nos resultados.

Todos os programas foram compilados com *`-O3 -DNDEBUG -Wall -Wextra -pedantic`*.

- **Programas**

No **exercício 1**, implementamos uma versão paralela sem deadlock para o programa *filosofo-deadlock.c*, que consiste na versão paralela do problema dos filósofos, porém a solução permite a ocorrência de deadlock.

Versão paralela com deadlock:

```c
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <omp.h>
#define think 20
#define eat 10
sem_t sem[5];

void thinking(i)
{ printf("filosofo %d thinking\n",i);
  fflush(stdout);
  usleep(think);
}
void eating(i)
{ printf("filosofo %d eating\n",i);
  fflush(stdout);
  usleep(eat);
}
void filosofo(int i)
{ int k;
```

```
  for (k=0;k<100;k++) {
     thinking(i);
     sem_wait(&sem[i]);
     usleep(1);
     sem_wait(&sem[(i+1)%5]);
     eating(i);
     sem_post(&sem[i]);
     sem_post(&sem[(i+1)%5]);
  }
}

void main()
{
  printf("INICIO \n");
  fflush(stdout);
  sem_init(&sem[0],0,1);
  sem_init(&sem[1],0,1);
  sem_init(&sem[2],0,1);
  sem_init(&sem[3],0,1);
  sem_init(&sem[4],0,1);
  #pragma omp parallel num_threads(5)
  {
      #pragma omp sections
      {
         #pragma omp section
         filosofo(0);
         #pragma omp section
         filosofo(1);
         #pragma omp section
         filosofo(2);
         #pragma omp section
         filosofo(3);
         #pragma omp section
         filosofo(4);
      }
  }
      printf("FIM\n");
}
```

Na versão acima, após alguns testes, notamos a ocorrência de deadlock, tendo uma média de 20% das vezes ocorrido deadlock.

Uma das soluções possíveis é a solução assimétrica, onde um dos filósofos inverte o sentido de pegar os garfos, evitando assim o deadlock onde cada um pega o garfo da direita. Se montarmos o grafo de recursos (filósofos, que alocam recursos e garfos, os recursos adquiridos), vemos que essa inversão remove o ciclo do grafo, removendo a espera circular.

Versão paralela sem deadlock:

```c
#include <stdio.h>
#include <semaphore.h>
#include <stdlib.h>
#include <omp.h>
#define think 20
#define eat 10
sem_t sem[5];

void thinking(i)
{ printf("filosofo %d thinking\n",i);
  fflush(stdout);
  usleep(think);
}
void eating(i)
{ printf("filosofo %d eating\n",i);
  fflush(stdout);
  usleep(eat);
}
void filosofo(int i)
{ int k;
  for (k=0;k<100;k++) {
     thinking(i);
     sem_wait(&sem[i]);
     usleep(1);
     sem_wait(&sem[(i+1)%5]);
     eating(i);
     sem_post(&sem[i]);
     sem_post(&sem[(i+1)%5]);
  }
}

void filosofo5(int i)
{ int k;
  for (k=0;k<100;k++) {
     thinking(i);
     sem_wait(&sem[0]);
     usleep(1);
     sem_wait(&sem[4]);
     eating(i);
     sem_post(&sem[4]);
     sem_post(&sem[0]);
  }
}

void main()
{
  printf("INICIO \n");
  fflush(stdout);
  sem_init(&sem[0],0,1);
  sem_init(&sem[1],0,1);
  sem_init(&sem[2],0,1);
  sem_init(&sem[3],0,1);
  sem_init(&sem[4],0,1);
  #pragma omp parallel num_threads(5)
  {
```

```
      #pragma omp sections
      {
         #pragma omp section
         filosofo(0);
         #pragma omp section
         filosofo(1);
         #pragma omp section
         filosofo(2);
         #pragma omp section
         filosofo(3);
         #pragma omp section
         filosofo5(4);
      }
  }
      printf("FIM\n");
}
```

Em todos os testes realizados não foi obtido a ocorrência de deadlocks.

No **exercício 2**, implementamos uma versão paralela para o programa *ex2-1-seq.c*, onde dado o arquivo *seq-teste.txt* realiza a busca de uma sequência de caracteres, digitada no teclado.

Versão serial, com medição de tempo de execução:

```
// Exercicio 1: Programa busca uma sequencia, definida pelo usuario, em um arquivo
com varias sequencias
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SEQ_SIZE 12
#define TRUE 0
#define FALSE 1

int main()
{
    int i = 0;
    int j;
    int igual;
    char * buff;
    char sequencia[SEQ_SIZE];
    int file_size;
    char ** seq_vet;
    int n_seq;
    int k;
    FILE *f = fopen("seq-teste.txt", "r");
       if (f == NULL)
       {
           printf("Erro na abertura do arquivo\n");
           exit(1);
       }
    fseek(f,0L,SEEK_END);
    file_size=ftell(f);
```

```
    fseek(f,0L,SEEK_SET);
    n_seq=(file_size+1)/(SEQ_SIZE-1);
    seq_vet=(char **)malloc(n_seq*sizeof(char *));
    for (k=0;k<n_seq;k++)
        seq_vet[k]=malloc(SEQ_SIZE*sizeof(char));
    while(!feof(f)){
       fgets(seq_vet[i], SEQ_SIZE, f);
        printf("seq_vet[%d] =  %s\n ",i,seq_vet[i]);
        i=i+1;
   }

   printf("file_size=%d\n",file_size);

   printf("Digite a sequencia que deseja buscar: ");
   scanf("%s", sequencia);
   struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
   igual = FALSE;
   for (i=0;i<n_seq;i++){
        buff=seq_vet[i];
        // printf("i=%d buff %s\n ",i,buff);
        for (j = 0; j < SEQ_SIZE-2; j++){
             if (sequencia[j] != buff[j]){
                   break;
             }
       }
        if (j== SEQ_SIZE-2) {
              igual=TRUE;
              break;
        }
   }
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Time: %.5fms\n",
        (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));

   if (igual == TRUE)
       printf("Sequencia encontrada na linha %d\n",i);
   else
       printf("Sequencia nao encontrada i=%d\n",i);

   fclose(f);
   return 0;
}
```

Após 20 execuções, adquirimos o menor tempo de 0.24ms na busca. Esse tempo foi medido em torno do laço de busca, sem *printf*s.

Implementamos uma versão paralela. Nela, se qualquer thread encontrar uma sentença equivalente todas devem parar a execução.

Realizamos manualmente a divisão do trabalho entre as threads usando uma seção crítica para atribuir *i* (início da busca) e *stop* (fim da busca). Criamos uma variável

found, compartilhada entre as threads, que é atômica - ou seja, é atualizada atomicamente ao invés de passar por um ciclo *read-modify-write*, e leituras não são otimizadas (o compilador não vai trocar a leitura da variável pelo uso de um registrador local). Dessa maneira, temos uma variável compartilhada sem utilizar uma seção crítica ou semáforo, melhorando o desempenho do programa.

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

#define SEQ_SIZE 12

static inline int is_same_sequence(char *a, char *b) {
    for (int j = 0; j < SEQ_SIZE - 2; j++) {
        if (a[j] != b[j]) {
            return 0;
        }
    }
    return 1;
}

int search(char **v, int size, char *s) {
    int search_base = 0;
    _Atomic int found = 0;
    #pragma omp parallel
    {
        int i, stop;
        #pragma omp critical
        {
            i = search_base;
            search_base += size / omp_get_num_threads();
        }
        stop = search_base < size ? search_base : size;
        printf("Thread %d: %d -> %d\n", omp_get_thread_num(), i, stop);

        while (i < stop && !found) {
            if (is_same_sequence(s, v[i])) {
                found = i;
                break;
            }
            i++;
        }
    }
    return found;
}
```

```c
int main() {
    int k;
    FILE *f = fopen("seq-teste.txt", "r");
       if (f == NULL) {
         perror("fopen");
            exit(1);
       }
    fseek(f, 0L, SEEK_END);
    int file_size = ftell(f);
    fseek(f, 0L, SEEK_SET);

    int n_seq = (file_size + 1) / (SEQ_SIZE - 1);
    char **seq_vet = (char **)malloc(n_seq * sizeof(char *));
    for (k = 0; k < n_seq; k++)
        seq_vet[k] = malloc(SEQ_SIZE * sizeof(char));
    int i = 0;
    while (!feof(f)) {
           fgets(seq_vet[i], SEQ_SIZE, f);
        printf("seq_vet[%d] =  %s\n", i, seq_vet[i]);
        i++;
    }

    printf("Search sequence: ");
    char sequencia[SEQ_SIZE];
    scanf("%s", sequencia);


    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);
    int match = search(seq_vet, n_seq, sequencia);
    clock_gettime(CLOCK_MONOTONIC, &end_time);
    printf("Time: %.5fms\n",
          (end_time.tv_sec * 1000 + 1e-6 * end_time.tv_nsec) - (start_time.tv_sec *
1000 + 1e-6 * start_time.tv_nsec));

    if (match)
        printf("Match at line %d\n", match);
    else
        printf("Sequence not found\n");
   fclose(f);
   return 0;
}
```

Ao executar o código, vemos:

```
...
Thread 3: 7500 -> 10000
```

```
Thread 0: 0 -> 2500
Thread 1: 2500 -> 5000
Thread 2: 5000 -> 7500
…
```

Indicando que a divisão entre threads funcionou como esperado. Testamos uma série de buscas e o código encontrou os elementos corretamente. Após 20 execuções, o menor tempo que atingimos foi 0.86ms.

Novamente observamos que o uso de multithreading piorou a performance do programa. Atribuímos a deterioração no tempo de execução à natureza do problema: por ser um laço curto de busca em memória, muito pouco processamento é utilizado por iteração, enquanto que os acessos à memória e tamanho do cache se tornam gargalos.