Programação Concorrente (ICP-361)
Cap. 3: Comunicação entre threads por memória compartilhada*

Profa. Silvana Rossetto

[1]Instituto de Computação (IC)
Universidade Federal do Rio de Janeiro (UFRJ)
Agosto de 2023
silvana@ic.ufrj.br

1. Introdução

Os exemplos que vimos no Capítulo 2 são relativamente simples de paralelizar pois é possível dividí-los em subproblemas menores que podem ser completamente resolvidos de forma independente, isto é, sem a necessidade dos fluxos de execução concorrente interagirem. A maioria dos problemas, entretanto, requerem um esforço maior para serem paralelizados pois a computação dos subproblemas exige que os fluxos de execução troquem informações entre si e sincronizem suas ações.

Vamos tomar como exemplo o problema de **somar os elementos de um vetor**. Considere que temos um vetor A com N elementos e precisamos somar todos os seus valores. O programa a seguir mostra a implementação de um algoritmo sequencial para resolver esse problema. A função `main` cria uma variável local para armazenar o resultado da soma de todos os elementos do vetor (`soma_total`), chama a função que executa a soma e imprime o resultado.

```
#define N 1000000 //N igual ao número de elementos do vetor
//Soma os n elementos do vetor a
float somaElemVetorSeq(const float a[], int n) {
   int i;
   float soma = 0;
   for(i=0; i<n; i++) {
      soma = soma + a[i];
   }
   return soma;
}
void main() {
   float a[N], soma_total;
   int i;
   //inicializa o vetor a
   ...
   soma_total = somaElemVetorSeq(a, N);
   //imprime o valor da variável 'soma_total'
}
```

Para paralelizar esse algoritmo vamos usar a estratégia de dividir o número total de elementos do vetor (N) em grupos consecutivos de P elementos e criar um fluxo de execução concorrente para somar os elementos de cada grupo. O programa seguinte mostra essa solução.

*Texto baseado no livro "Introdução à programação paralela em GPU para a implementação de métodos numéricos", D. Alfaro e S. Rossetto, Notas em Matemática Aplicada, vol 84, 2016, disponível em: https://www.sbmac.org.br/wp-content/uploads/2022/08/livro_84.pdf

```
1.#define N 1000000 //N igual ao número de elementos do vetor
2.#define P 4 //P igual ao número de unidades de processamento
3.float soma_total = 0;
4.//Soma uma parte dos n elementos do vetor a
5.void somaGrupoConsElemVetor(const float a[], int inicio, int fim) {
6.   int i;
7.   float soma_local = 0;
8.   for(i=inicio; i<fim; i++)
9.      { soma_local = soma_local + a[i]; }
10.   // !!! solicita exclusão mútua !!!
11.   soma_total = soma_total + soma_local;
12.   // !!! libera a exclusão mútua !!!
13.}
14.void main() {
15.   float a[N];
16.   int i, inicio, fim;
17.   //inicializa o vetor a
18.   ...
19.   //soma os n elementos de a
20.   for(i=0; i<P; i++) {
21.      inicio = i * (N/P);
22.      //o último fluxo trata os elementos restantes
23.      if (i<P-1) {
24.         fim = inicio + (N/P);
25.      } else {
26.         fim = N;
27.      }
28.      //dispara um fluxo de execução f para executar:
29.           // somaGrupoConsElemVetor(a, inicio, fim);
30.   }
31.   // !!! espera todos os fluxos de execução terminarem !!!
32.   //imprime o valor da variável 'soma_total'
33.   printf("soma total = %f", soma_total);
34.}
```

Há vários pontos a destacar nesse programa. O primeiro deles é que, diferente dos exemplos de **soma de vetores** e de **multiplicação de matrizes**, que possuíam como saída uma estrutura de dados com a mesma dimensão da estrutura de dados de entrada, no problema de somar os elementos de um vetor temos como entrada um vetor e como saída um único valor que deverá ser preenchido com a contribuição do processamento realizado por diferentes fluxos de execução. No programa declaramos a variável `soma_total` como variável global (linha 3) para permitir que a função executada pelos fluxos de execução (`somaGrupoConsElemVetor`) possa alterá-la diretamente. (Outra alternativa seria passar o endereço dessa variável como argumento para a função executada pelos fluxos de execução.)

Dentro da função `somaGrupoConsElemVetor`, cada fluxo de execução calcula a soma dos elementos do vetor no intervalo definido e armazena essa soma em uma variável local chamada `soma_local` (linhas 7 a 9). Essa abordagem permite que os fluxos de execução trabalhem de forma completamente independente enquanto somam os valores dos elementos dentro do intervalo dado.

Quando terminam de realizar essa soma, os fluxos de execução precisam acrescen-

tar na soma total o valor da sua soma local. Isso é feito na linha 11. Porém, se deixarmos dois ou mais fluxos de execução atualizarem a variável global `soma_total` ao mesmo tempo, o resultado das várias atualizações poderá ficar incorreto.

Vejamos porque isso poderia acontecer. Quando um fluxo de execução processa a linha 11, ele primeiro carrega os valores atuais das variáveis `soma_total` e `soma_local` e guarda esses valores em áreas de memória especiais — e exclusivas de cada fluxo de execução — chamadas **registradores**. Em seguida, a unidade de processamento aritmético do computador soma o conteúdo desses dois registradores e armazena o resultado em um terceiro registrador. Após isso, o conteúdo desse terceiro registrador é carregado de volta para atualizar a variável `soma_total`. Então, se dois ou mais fluxos de execução processarem essa sequência de instruções ao mesmo tempo, eles armazenarão nos seus registradores o mesmo valor inicial de `soma_total` (vamos considerar, por exemplo, que esse valor seja 0). Em seguida, acrescentarão a esse valor a sua soma local (fazendo 0 + `soma_local`), e finalmente atualizarão a variável `soma_total` com o resultado dessa última soma. O que acontecerá ao final é que o novo valor da variável `soma_total` será o valor atribuído pelo último fluxo de execução, que conterá apenas o incremento da sua soma local, as demais atualizações serão perdidas.

Sincronização por exclusão mútua Para evitar esse problema, é preciso garantir que os fluxos de execução concorrente executarão a linha 11 com **exclusão mútua**, isto é, apenas um fluxo de execução a cada vez. Podemos dizer de outra forma que a sentença da linha 11 deve ser **atômica**, isto é, ela deve ser completamente executada antes que outro fluxo de execução comece a processá-la. Assim, quando um fluxo de execução terminar de executar a linha 11, teremos a garantia de que o valor da variável global `soma_total` está atualizado. O próximo fluxo de execução irá ler o valor atual de `soma_total` e acrescentar a sua soma local.

A **exclusão mútua** é um tipo de **sincronização** que pode ser requerida dentro de um algoritmo concorrente. Os fluxos de execução precisam sincronizar suas ações para evitar a ocorrência de erros de execução próprios da programação concorrente. No programa acima, enfatizamos isso nas linhas 10 e 12, assumindo por ora que algum mecanismo de exclusão mútua (provido pela linguagem C) será invocado nessas linhas para sinalizar, respectivamente, a entrada e saída de um trecho de código atômico. Na próxima seção vamos detalhar essas questões.

2. Comunicação e sincronização entre threads via memória compartilhada

A facilidade provida por um espaço de endereçamento compartilhado é normalmente usada para implementar a **comunicação** entre as threads de uma aplicação. Quando uma thread precisa trocar/compartilhar uma informação com outra thread, ela o faz alterando o valor de uma variável comum para as duas threads, ou seja, uma variável que está armazenada em um endereço de memória que as duas threads compartilham (conhecem e podem acessar).

O modelo de organização tipicamente adotato nos computadores multiprocessadores é baseado no uso de memória compartilhada, i.e., um espaço de endereçamento físico único é compartilhado entre todos os processadores. Esse modelo de organização facilita o desenvolvimento de aplicações para essas arquiteturas, pois é uma extensão

direta do modelo de acesso à memória usado nos computadores uniprocessadores: diferentes linhas de execução (threads) podem acessar variáveis de um programa que estão em endereços de memória igualmente acessíveis por outras threads.

Quando duas ou mais threads de um programa precisam trocar informações entre si, o programador define variáveis de acesso compartilhado entre elas. Desse modo, quando uma thread tem um valor novo que deve ser comunicado para as demais threads, ela simplesmente escreve esse valor na variável compartilhada, independente das demais threads estarem esperando por esse valor naquele momento. Quando outra thread precisar saber qual é o valor atual dessa informação, ela simplesmente lê o conteúdo atual da variável compartilhada.

Assim, toda a comunicação entre as threads se dá de forma **assíncrona**, i.e., as threads escrevem/lêem valores nas variáveis compartilhadas a qualquer tempo. Para garantir que a comunicação ocorra de forma correta (por exemplo, que o valor da variável não seja sobreescrito antes que as outras threads leiam o valor anterior), a interação entre threads via memória compartilhada gera a necessidade de **sincronização**. **A necessidade de sincronização é um dos principais desafios para a programação concorrente com memória compartilhada**.

Neste texto serão discutidos aspectos relacionados com a **comunicação** e **sincronização** entre linhas de execução distintas dentro de uma mesma aplicação, usando o modelo de **memória compartilhada**.

2.1. Seções críticas e ações atômicas

Quando escrevemos programas concorrentes com compartilhamento de variáveis entre threads, normalmente esperamos que a execução das expressões aritméticas ou sentenças de atribuição da linguagem de programação sejam feitas de forma *atômica*, i.e., que uma vez iniciada a ação por uma thread ela seja executada por completo antes que outra thread opere sobre a mesma variável. Entretanto, as expressões aritméticas e sentenças de atribuição das linguagens de alto nível (C, Java, etc.) são normalmente traduzidas em mais de uma instrução de máquina. Por conta disso, durante a execução das sentenças ou expressões da linguagem de alto nível por threads distintas, pode ocorrer entrelaçamentos das instruções de máquina correspondentes, dando possibilidade de ocorrência de resultados inesperados.

Como exemplo, considere duas threads, T_1 e T_2, associadas aos seguintes trechos de código (as variáveis y e z são inicialmente iguais a 0):

```
T1:                    T2:
x = y + z;             y = 1;
                       z = 2;
```

Se assumirmos (incorretamente) que a execução de cada sentença de atribuição é uma ação atômica, o valor final de x, computado pela thread T_1, poderá ser 0, 1 ou 3, representando as somas $0 + 0$, $1 + 0$ e $1 + 2$, respectivamente. Entretanto, as instruções de máquina para as threads T_1 e T_2 poderão ser da seguinte forma:

```
T1:                          T2:
(1) mov y, %eax              (4) mov 1, y
(2) add z, %eax              (5) mov 2, z
(3) mov %eax, x
```

Desse modo, os seguintes entrelaçamentos de instruções de máquina poderão ocorrer:

```
(1), (2), (3), (4), (5) -> x=0
(4), (1), (2), (3), (5) -> x=1
***(1), (4), (5), (2), (3) -> x=2 ***
(4), (5), (1), (2), (3) -> x=3
```

Como o exemplo ilustra, uma vez que as instruções de máquina é que são ações atômicas (e não as expressões e sentenças da linguagem de alto nível), o entrelaçamento da execução das instruções de máquina é que determina o resultado da computação. No terceiro entrelaçamento, temos como resultado $x = 2$ (resultado não esperado). Esse tipo de erro, gerado pela execução concorrente dos fluxos de execução que acessam uma mesma área de memória e ao menos uma das operações é de escrita, é chamado **corrida de dados** (do termo em inglês *data race*).

Toda referência a uma variável que pode ser acessada/modificada por outra thread é uma **referência crítica**. Para impedir a ocorrência de resultados indesejáveis para uma determinada computação, os trechos de código que contêm referências críticas **devem ser executados de forma atômica**. Esses trechos de código são denominados de **seção crítica** do código. Identificar os trechos de seção crítica e coordenar a sua execução é um dos problemas fundamentais da programação concorrente.

2.2. Seções de entrada e saída da seção crítica

Mecanismos de sincronização de códigos permitem **transformar uma seção crítica em uma ação atômica** usando dois outros trechos de código especiais chamados **seção de entrada** e **seção de saída**. Essas duas seções adicionais de código tem por finalidade "cercar" a entrada e saída da seção crítica, garantindo que os requisitos de execução atômica sejam atendidos. O pseudocódigo abaixo ilustra essa estratégia:

```
while(true) {
  executa fora da seção crítica (...)
  requisita a entrada na seção crítica //seção de entrada
  executa a seção crítica (...)        //seção crítica
  sai da seção crítica                 //seção de saída
}
```

A Figura 1 mostra um exemplo de execução de dois processos que entram e saem das suas seções críticas (pedaço do código onde acessam um objeto compartilhado pelos dois processos).

Garantir a execução atômica de uma seção crítica de código consiste em escrever as seções de entrada e saída para essa seção crítica de modo que os seguintes requisitos sejam atendidos [1, 2]:

1. *Exclusão mútua*: quando uma thread está executando a sua seção crítica nenhuma outra thread pode executar a seção crítica relacionada com o mesmo objeto, nesse

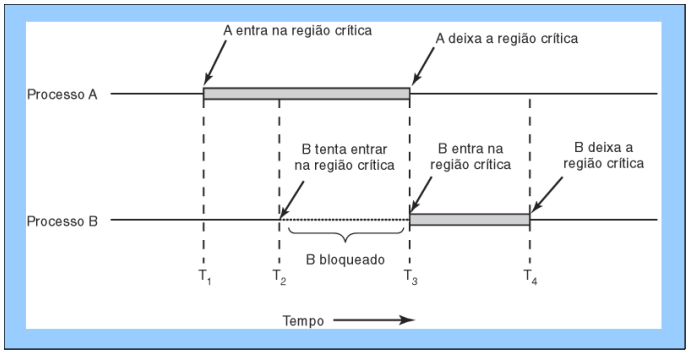

Figure 1. Exemplo de entrada e saída na seção crítica por dois processos (fonte: Pearson).

caso devem esperar na seção de entrada (se a thread na seção crítica perde a CPU, ela permanece com a posse da seção crítica).

2. *Ausência de starvation*: se uma thread está tentando entrar na seção crítica, então em algum momento ela deverá conseguir entrar.
3. *Independência do restante do código*: uma thread que termina sua execução fora da sua seção crítica (ou entra em loop infinito fora da seção crítica) não deve interferir nas seções de entrada e saída das threads restantes, e quando nenhuma thread estiver na seção crítica e há threads que desejam executá-la, apenas as threads que estão executando as seções de entrada e saída podem participar da decisão de qual delas ganhará o acesso à seção crítica.
4. *Garantia de entrada*: quando nenhuma thread está na seção crítica e uma thread deseja executar a seção crítica, ela deve passar pela seção de entrada imediatamente, e quando uma thread executa a seção de saída e há outras threads esperando na seção de entrada uma delas deverá ser escolhida para entrar na seção crítica.

Para que as threads possam executar corretamente e chegar ao final das suas execuções, é necessário que toda thread que entre na seção crítica saia dela em algum momento, i.e., **uma thread não deve executar um loop infinito ou terminar a sua execução dentro de uma seção crítica**.

Uma questão básica para o funcionamente desse tipo de mecanismo é que as **seções de entrada e de saída devem ser ações atômicas** (caso contrário a sincronização por exclusão mútua não será garantida).

2.3. Sincronização por exclusão mútua

A **sincronização por exclusão mútua** visa garantir que **os trechos de código em cada thread que acessam objetos compartilhados não sejam executados ao mesmo tempo**, ou que uma vez iniciados sejam executados até o fim sem que outra thread inicie a execução do trecho equivalente. Essa restrição é necessária para lidar com a possibilidade de inconsistência dos valores das variáveis compartilhadas.

Como exemplo, considere o caso de uma operação que incrementa o valor de uma variável global (*ex., s++*). Para realizar o incremento, três instruções de nível mais baixo (linguagem de máquina) são normalmente necessárias: (i) ler o valor atual da variável; (ii) incrementar esse valor; e (iii) escrever o novo valor na variável global. Se duas threads executam essa operação concorrentemente, as duas poderão ler o mesmo valor inicial de *s*

e realizar o incremento sobre esse valor. Ao final, embora duas operações de incremento tenham sido realizadas, o valor em *s* irá refletir apenas um incremento.

A solução para a **exclusão mútua** é definida agrupando *sequências contínuas de ações atômicas de hardware em seções críticas de software*. As **seções críticas** (trechos de código que acessam objetos compartilhados) devem ser transformadas em **ações atômicas**, de forma que a sua execução não possa ocorrer concorrentemente com outra seção crítica que referencia a mesma variável.

2.4. Mecanismos para sincronização por exclusão mútua entre threads

Existem diferentes mecanismos para implementar sincronização por exclusão mútua entre threads. Qualquer solução proposta deve prover a sincronização necessária para eliminar as **condições de corrida indesejáveis**, sem restringir as oportunidades de paralelismo na execução da aplicação. Há duas abordagens básicas [3]:

1. sincronização por espera ocupada;
2. sincronização por escalonamento.

A **sincronização por espera ocupada** faz com que a thread fique continuamente testando o valor de uma determinada variável até que esse valor lhe permita executar a sua seção crítica com exclusividade. Para implementar esse mecanismo de sincronização é necessário dispor de instruções de máquina que permitam ler e escrever em localizações da memória de forma atômica.

O principal problema da solução por espera ocupada é que ela gasta ciclos de CPU enquanto espera autorização para seguir com o seu fluxo de execução normal. A "espera ocupada" só faz sentido nos seguintes casos:

- não há nada melhor para a CPU fazer enquanto espera;
- o tempo de espera é menor que o tempo requerido para a troca de contexto entre threads.

Os mecanismos de **sincronização por escalonamento** são a alternativa mais usual. Nesse caso, quando uma thread não consegue entrar na seção crítica ela é bloqueada e a CPU pode ser cedida a outra thread, evitando assim o desperdício de ciclos de CPU.

2.5. Sincronização por escalonamento usando *locks*

Um ***lock* possui uma thread proprietária** e esta relação de posse determina características particulares das operações sobre *locks*:

- Uma thread requisita a posse de um *lock* L executando a operação `L.lock();`
- Uma thread que executa a operação `L.lock()` torna-se a proprietária do *lock* se nenhuma outra thread já possui o *lock*, caso contrário a thread é bloqueada;
- Um thread libera sua posse sobre o *lock* executando a operação `L.unlock` (se a thread não possui o *lock* a operação retorna com erro);
- Uma thread que já possua o *lock* L e executa `L.lock()` novamente não é bloqueada (apenas se o *lock* for recursivo), mas deve executar `L.unlock()` o mesmo número de vezes que executou `L.lock()` antes que outra thread possa ganhar a posse de L;

O uso de *locks* para implementar exclusão mútua se dá da seguinte forma: a operação `L.lock()` implementa a entrada na seção crítica e a operação `L.unlock()` implementa a saída da seção crítica, como ilustrado abaixo:

```
T1:                                T2:
L.lock();                          L.lock();
//seção crítica                    //seção crítica
L.unlock();                        L.unlock();
```

Locks na biblioteca Pthreads A biblioteca Pthreads oferece o mecanismo de sincronização por *locks* através de variáveis especiais do tipo ***pthread_mutex_t***. Por definição, Pthreads implementa **locks não-recursivos** (uma thread não deve tentar alocar novamente um *lock* que já possui). Para tornar o *lock* recursivo é preciso mudar suas propriedades básicas. O código abaixo ilustra o uso das operações de *lock* com a biblioteca Pthreads:

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutexAttr;
pthread_mutexattr_init(&mutexAttr);
pthread_mutexattr_settype(&mutexAttr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex, &mutexAttr);
pthread_mutex_lock(&mutex);
//seção crítica
pthread_mutex_unlock(&mutex);
```

Exercício

1. O que é **seção crítica** do código?
2. O que é **corrida de dados** no contexto de computação concorrente?
3. O que significa uma operação ser **atômica**?
4. O que é **sincronização por exclusão mútua?**
5. O que é **espera ocupada**, qual sua principal desvantagem e em quais situações se aplica?
6. Como funciona o mecanismo de exclusão mútua com **locks**?

References

[1] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.

[2] Gadi Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Prentice-Hall, 2006.

[3] M. L. Scott. *Programming Language Pragmatics*. Morgan-Kaufmann, 2 edition, 2006.