Computação Concorrente - UFRJ - 2021.1 - Lista 2

Gabriele Jandres Cavalcanti

DRE: 119159948

Questão 01

a Para a resolução do problema apresentado, temos como solução um algoritmo iterativo, onde cada posição do vetor de saída representa a soma parcial dos valores até aquela posição do vetor de entrada. A solução concorrente utilizando threads possibilita que esse trabalho de calcular somas parciais seja feito de forma paralela.

No entanto, como cada iteração depende da anterior, então todas as threads devem completar a iteração corrente antes de qualquer uma delas iniciar a próxima iteração. Por causa disso, a estratégia de sincronização por barreira é utilizada, de forma a garantir que as threads trabalhem sempre em fase.

De forma mais detalhada, a função que será executada pelas threads na solução concorrente recebe como argumento o identificador da thread, como de costume. Além do identificador, a função trabalha com duas variáveis adicionais, *aux* e *salto*, que representam o valor de uma posição auxiliar e o valor do salto dado no vetor, respectivamente.

A solução parte do princípio que para uma dada posição x no vetor, não serão necessárias mais que x iterações para calcular o valor da soma parcial correspondente. Por isso o salto já começa com o valor 1 (posição 0 não precisa ser manipulada).

Em cada iteração (salto), quando o id é menor que o valor do salto não ocorre manipulação dos dados do vetor. Mas para os valores de ids maiores do que ele, já ocorre manipulação dos dados. Na primeira iteração por exemplo, como o salto tem valor 1 nesse momento, aux terá o valor da posição imediatamente anterior à posição em questão no vetor. Aqui é feita uma chamada à função barreira para $nthreads - salto$, já que em cada iteração somente $nthreads - salto$ entram no *if* e portanto somente elas realizam modificações no vetor. A necessidade dessa barreira será melhor explicada na questão (c).

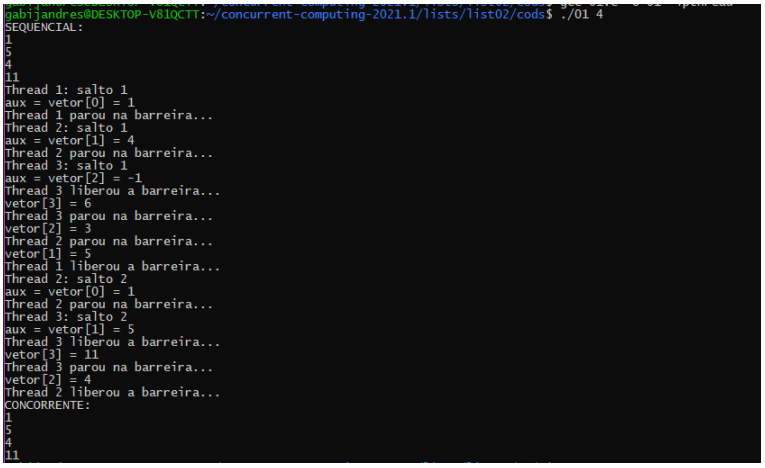
Após esse cálculo dos valores parciais, cada thread sobrescreve o valor correspondente em $vetor[id]$ utilizando seu valor atual e o auxiliar. Novamente, é feita uma chamada à função barreira para $nthreads - salto$ para garantir que na próxima iteração o valor que aux irá armazenar já esteja atualizado.

Seguindo assim, a cada iteração haverá manipulação dos dados do vetor e soma dos valores acumulados se o id da thread for maior do que o valor do salto, para que não haja iterações desnecessárias com $vetor[id]$.

Como essa soma é feita utilizando recorrência, de forma que escrevemos em $vetor[id]$ o valor de soma acumulado que estamos calculando a cada iteração, ao final teremos em $vetor[i]$ o que desejamos, que é o valor da soma parcial da posição i.

b Sim, a solução está correta devido ao que foi explicado anteriormente. O resultado apresentado ao executar o programa é equivalente ao que seria obtido com o algoritmo sequencial também apresentado.

Como exemplo, rodando o programa sequencial e concorrente com o vetor dado no enunciado temos como saída:



c São necessárias duas chamadas de sincronização coletiva para que não haja condição de corrida.

A primeira barreira serve para menter a integridade dos valores auxiliares calculados, porque ela faz com que o programa aguarde todas as threads obterem o valor acumulado em $vetor[id - salto]$ e o atribuírem a aux. Essa barreira é necessária porque há atribuição de novos valores a $vetor[id]$ em linhas posteriores, o que pode comprometer o valor obtido em aux para outras threads caso essa barreira não exista. Por exemplo, assumindo o vetor de entrada do enunciado, na primeira iteração ($salto = 1$), a thread 1 vai obter $aux = vetor[0] = 1$ e atribuirá $vetor[1] = 1 + 4 = 5$. Com isso, nessa mesma iteração, a thread 2 vai obter $aux = vetor[1] = 5$, quando deveria obter $aux = 4$.

Já a segunda barreira espera todas as threads escreverem um novo valor acumulado em $vetor[id]$ para garantir a integridade dos valores de aux calculados nas próximas iterações.

Por exemplo, ainda utilizando o vetor de entrada do enunciado, se removermos a segunda barreira, observamos que a thread 3 na segunda iteração utiliza o valor de $vetor[1]$ que ainda não foi atualizado pela thread 1 na primeira iteração, conforme ilustrado abaixo:

Com a barreira, há a garantia que o valor de $vetor[1]$ só é utilizado pela thread 3 após a atualização pela thread 1:

Então, para evitar condição de corrida, elas não poderiam ser substituídas por uma única chamada.

Questão 02

```
long long int contador = 0;
pthread_mutex_t mutex;
pthread_cond_t cond; // variavel para sincronizacao condicional
bool novo_multiplo = false; // variavel de controle para evitar repetidas
    impressoes do mesmo multiplo

void *T1 (void *arg) {
  while(1) {
    pthread_mutex_lock(&mutex); // inicia a sincronizacao por exclusao
        mutua para acesso a variavel global
    FazAlgo(contador);
    contador++;
    if (contador % 100 == 0) {
      novo_multiplo = true; // indico que encontrei um novo multiplo
      pthread_cond_signal(&cond); // sinalizo para que T2 consiga imprimir
      pthread_cond_wait(&cond, &mutex); // espero a sinalizacao de T2 para
          continuar
    }
    pthread_mutex_unlock(&mutex);
  }

  pthread_exit(NULL);
}

void *T2 (void *arg) {
  while(1) {
    pthread_mutex_lock(&mutex); // inicia a sincronizacao por exclusao
        mutua para acesso a variavel global
    if (contador % 100 != 0 || !novo_multiplo) { // t2 ficara em espera se
        o contador nao for multiplo de 100 ou se o numero nao for um novo
        multiplo e ja tiver sido impresso em tela
      pthread_cond_wait(&cond, &mutex);
    }
    printf("%lld \n", contador);
    novo_multiplo = false; // reinicia o indicador
    pthread_cond_signal(&cond); // sinaliza para que T1 prossiga
    pthread_mutex_unlock(&mutex);
  }

  pthread_exit(NULL);
}
```

Questão 03

a O pool de threads cria threads e as gerencia. Em vez de criar uma thread e descartá-la assim que a tarefa for concluída, o pool de threads reutiliza threads em várias tarefas. No caso da aplicação de exemplo, teremos 10 threads disponíveis para realizar as 100 tarefas. A ideia é que assim que uma thread termina a execução de uma tarefa ela já está liberada para executar outra tarefa. Isto é, a tarefa pode ser executada por qualquer thread de trabalho livre.

Basicamente, na implementação apresentada, o método *execute* é responsável por escalonar uma tarefa para execução pelo pool adicionando-a na fila *queue*, o método *shutdown* é responsável por mudar o estado da variável de controle *shutdown* que indica o encerramento do pool e a classe *MyPoolThreads* representa o pool de threads que fica observando a fila *queue*, aguardando tarefas para serem executadas. Quando a fila não está mais vazia e não há sinal de encerramento (há tarefas), o primeiro elemento é removido da fila *queue* e executado. No caso de a fila estar vazia e *shutdown* ser verdadeiro, quer dizer que as tarefas foram finalizadas e é possível retornar.

b O erro no código está na falta de notificação das threads, fazendo com que elas em alguns casos fiquem presas aguardando no *while* mesmo que o pool tenha encerrado, com isso acontece uma situação de deadlock. Esse caso pode ocorrer quando as threads conseguirem executar todas as tarefas antes do sinal de *shutdown*. Com isso, como não haverá tarefas na fila as threads entrarão em espera e quando a função *shutdown* executar o deadlock vai ocorrer no join, que vai ficar esperando as threads acabarem a execução mas elas estarão presas no loop. Para resolver o problema, basta adicionarmos uma notificação a todas as threads no bloco *syncronized* da função *shutdown* para "acordar"as threads:

```
// Encerra o pool depois que todas as tarefas escalonadas foram finalizadas
public void shutdown() {
    synchronized (queue) {
        this.shutdown = true; // indica o encerramento do pool
        queue.notifyAll();
    }

    // espera as threads finalizarem
    for (int i = 0; i < nThreads; i++)
      try {
        threads[i].join();
      } catch (InterruptedException e) {
        return;
      }
}
```

Questão 04

a Um escritor pode esperar por um longo intervalo de tempo pela autorização para escrever se por exemplo a demanda por leituras for alta o suficiente para quase nunca termos um número de leitores igual 0. Isso ocorre quando há um número muito grande de threads leitoras ou quando a tarefa das threads leitoras é mais demorada do que a escrita.

Em questões de tempo de processamento das operações de leitura e de escrita, podemos afirmar que em geral a leitura é uma operação mais rápida. Porém, pode acontecer de além de realizar leitura a thread leitora tenha também que fazer algum tipo de verificação do valor lido, que dependendo da verificação pode demorar bem mais do que uma escrita simples. Nesse tipo de situação podemos nos deparar com o problema da inanição, onde os leitores realizam tarefas mais complexas e demoradas do que os escritores, que realizam escritas simples. Com isso, como os leitores podem ler juntamente a outros leitores, pode-se entrar em uma sequência longa de leitura, dificultando a chance de escritores executarem.

Um exemplo prático é a atividade do laboratório 7, onde as threads leitoras liam a variável central e tinham que realizar uma iteração para verificar se o número era primo, enquanto a thread escritora apenas tinha que atribuir o valor do seu identificador à variável central.

b Solução concorrente em Java para o padrão leitores e escritores que minimiza o tempo de espera dos escritores (todas as vezes que um escritor tentar escrever e existir leitores lendo, a entrada de novos leitores fique impedida ate que todos os escritores em espera sejam atendidos):

```
// Monitor que implementa a logica do padrao leitores/escritores
class Monitor {
  private int leitores, escritores, escritoresAguardando;

  // Construtor
  Monitor() {
    this.leitores = 0; // leitores lendo (0 ou mais)
    this.escritores = 0; // escritor escrevendo (0 ou 1)
    this.escritoresAguardando = 0; // indica a quantidade de escritores
        aguardando pra realizar escrita
```

```
  }

  // Entrada para leitores
  public synchronized void EntraLeitor(int id) {
    try {
      // enquanto tiver algum escritor escrevendo ou houverem novos
          escritores querendo escrever, as threads leitoras devem se
          bloquear
      while (this.escritores > 0 || this.escritoresAguardando > 0) {
        System.out.println("Leitor " + id + " bloqueado...");
        wait(); // bloqueia pela condicao logica da aplicacao
      }
      this.leitores++; // registra que ha mais um leitor lendo
      System.out.println("Leitor " + id + " lendo...");
    } catch (InterruptedException e) {
        System.err.println(e);
    }
  }

  // Saida para leitores
  public synchronized void SaiLeitor(int id) {
    this.leitores--; // registra que um leitor saiu
    if (this.leitores == 0) this.notify(); // libera escritor no caso de
        n o haverem mais leitores (caso exista escritor bloqueado)
    System.out.println("Leitor " + id + " saindo...");
  }

  // Entrada para escritores
  public synchronized void EntraEscritor(int id) {
    try {
      this.escritoresAguardando++; // indica que ha um novo escritor
          querendo escrever
      System.out.println("Escritor " + id + " quer escrever...");
      System.out.println(this.escritoresAguardando + " escritores
          aguardando para escrever...");
      while ((this.leitores > 0) || (this.escritores > 0)) {
        System.out.println("Escritor " + id + " bloqueado...");
        wait(); // bloqueia pela condicao logica da aplicacao
      }
      this.escritoresAguardando--; // indica que nao esta mais aguardando
          pois ira escrever
      this.escritores++; // registra que ha um escritor escrevendo
      System.out.println("Escritor " + id + " escrevendo...");
    } catch (InterruptedException e) {
        System.err.println(e);
    }
  }

  // Saida para escritores
  public synchronized void SaiEscritor(int id) {
    this.escritores--; // registra que o escritor saiu
    notifyAll(); // libera leitores e escritores (caso existam leitores ou
        escritores bloqueados)
    System.out.println("Escritor " + id + " saindo...");
  }
}
```