Módulo 1 - Lista de Exercícios (2021/1 REMOTO)

Carlos Bravo
119136241

Agosto 2021

Questão 1

(a) Um algoritmo sequencial roda o código comando a comando, então só é possível rodar uma linha quando a anterior terminar. Um concorrente, no entanto, é possível definir um bloco de comandos, chamados de thread, que pode ser executado em paralelo, se a ordem de execução não for importante. Dessa forma é possível definir blocos para serem rodados em cada um dos núcleos do processador e, como a ordem não faz diferença, poderão ser executados quantas threads seus núcleos permitirem, ao invés de apenas 1 na versão sequencial.

(b) É indicado para qualquer problema em que tenha um bloco de comandos que pode ser executado independente de sua ordem, mesmo que em apenas um pedaço do algoritmo. Alguns exemplos são:

- Multiplicação de Matriz: O cálculo de uma célula independe do resultado de outras células
- Aprendizado de Máquina: Por consequência do último item
- Processamento de Imagens: Processamentos que ocorrem pixel a pixel, como alterar paleta de cores, não dependem do resultado dos outros pixels
- Séries: Como a ordem dos fatores não altera o resultado, é possível somar em qualquer ordem

(c) A aceleração máxima é calculada pela fórmula $\frac{T_{sequencial}}{T_{concorrente}}$. Se rodarmos as 5 tarefas sequencialmente, seu tempo será de $5t$ (Sendo t a unidade de tempo gasta por uma tarefa). Se rodarmos de forma concorrente, 3 delas poderão ser executadas ao mesmo tempo em núcleos diferentes, então o tempo concorrente será dado pelas tarefas sequenciais + o bloco concorrente, $2t + 1t = 3t$. Dessa forma, a aceleração máxima será de $\frac{5t}{3t} = \frac{5}{3}$, o que é aproximadamente 60% mais rápido.

(d) Seção crítica é o nome dado a uma seção do código que faz parte das threads mas que depende da ordem de execução. Um exemplo é soma de variável, pois para realizar essa soma internamente a máquina precisa realizar duas operações: Resgatar o valor da variável e atribuir seu valor antigo + a soma. No entanto, se for executado a primeira parte (De resgatar o valor) e o processador cortar a execução para outra thread, ao voltar realizará apenas a atribuição do resultado, sobrescrevendo o valor que estivesse ali.

(e) Sincronização por exclusão mútua é um método para impedir que seções críticas se tornem problemáticas. Analogamente falando, a seção crítica possui um cadeado e, quando uma thread

quer acesso, trava o cadeado. Quando outra thread quiser acesso a essa seção e a mesma estiver bloqueada, a thread é pausada e outra entra em execução. Quando a thread que inicialmente travou a seção terminar a parte crítica, basta abrir o cadeado. Agora com a seção destravada, outra thread poderá travá-la para que possa usar.

Questão 2

```
void *task(void *arg){
  tArgs *args = (tArgs *)arg;
  int nthreads = args->nthreads;
  int n = args->n;
  int id = args->id;

  double* soma = (double *)malloc(sizeof(double));
  *soma = 0;

  for(int i = id; i <= n; i += nthreads){
    *soma += 4 * pow(-1,i) / (2*i + 1);
  }

  pthread_exit((void *)soma);
}
```

Esse algoritmo começa recebendo um parâmetro do tipo *tArgs*, que contém o número de threads, o valor de *n* e um identificador da thread.

É criado um espaço de memória do tipo *double* para armazenar o resultado da soma e é inicializado com 0.

O *for* começa do identificador da thread (Indo de 0 até $nthreads - 1$) e soma *nthreads* ao contador, dessa forma as threads podem pegar termos alternados da série. Pra cada repetição encontra o valor da função, multiplica por 4 e soma na variável.

Por fim retorna o valor da soma parcial de π, ficando a cargo da *main* somar os resultados parciais para obter o resultado final.

Esse método foi escolhido, ao invés de usar exclusão mútua para acessar uma variável soma global, porque o cálculo da função não consome tanto tempo de processamento, então as threads estão acessando muitas vezes a variável *soma* para somar o resultado. Se utilizasse desse método, provavelmente ia demorar mais para terminar de rodar, pois iria ficar travando a variável para todas as threads.

Questão 3

- -1: T1.1, T1.2, T1.3, T1.4, T1.5
- 0: T3.1, T3.2, T1.1, T3.3
- 2: T3.1, T3.2, T2.1, T3.3

- -2: T1.1, T1.2 <-> T2.1, T1.3, T1.4, T2.2, T1.5

  T1.2 <-> T2.1 (Em linguagem de máquina, seguindo os comandos da leitura complementar):

  T1.2.1 (Lê o valor de x, -1)

  T1.2.2 (Soma 1, 0)

  T2.1.1 (Lê o valor de x, -1)

  T2.1.2 (Soma 1, 0)

  T2.1.3 (Atribui resultado a x, 0)

  T1.2.3 (Atribui resultado a x, 0)

- 3: T1.1 <-> T3.1, T3.2, T1.2, T2.1, T3.3

  T1.1 <-> T3.1 (Em linguagem de máquina, seguindo os comandos da leitura complementar):

  T3.1.1 (Lê o valor de x, 0)

  T3.1.2 (Soma 1, 1)

  T1.1.1 (Lê o valor de x, 0)

  T1.1.2 (Soma -1, -1)

  T1.1.3 (Atribui resultado a x, -1)

  T3.1.3 (Atribui resultado a x, 1)

- -3: Não é possível. Na linha T1.4 x==-1, no entanto não executado sobraram apenas 1 linha de subtração (T2.2), necessitando de 2. Na linha T3.2 x==1, no entanto não executado sobraram apenas 3 linhas de subtração (T1.1, T1.3 e T2.2), necessitando de 4

- 4: Não é possível. Pela mesma lógica anterior, não há somas o suficiente para alcançar este resultado

Questão 4

(a) Quando a T0 chegar na linha 3, a primeira condição do while é o valor da variável $queroEntrar\_1$, que é definida como false. Dessa forma, o while já é cancelado e passa para a linha 4, que contém a sessão crítica.

(b) Acontecerá a mesma coisa que explicado anteriormente, mas com a variável $queroEntrar\_0$.

(c) Se ambas executarem a linha 1 antes de entrar no while, ambas variáveis se tornam $true$. A thread que rodar a linha 2 primeiro irá se travar no while, pois após ambas terminarem essa linha, o valor da variável $TURN$ será 0 ou 1. Se for 0, T0 poderá entrar na seção crítica enquanto T1 ficará travado até que T0 chegue na linha 5. Se for 1, acontece a mesma coisa com a outra thread, com T1 podendo executar e T0 esperando.

(d) Sim. Todos os casos possíveis já foram conferidos nos itens anteriores:

- Se T0 chegar ao while antes de T1 rodar, os valores serão $queroEntrar\_0 = true$ & $TURN = 1$. Se esperar para que T1 chegue a seção crítica ao mesmo tempo, na linha 2 sobreescreverá $TURN = 0$, com isso seu while se torna verdadeiro e ficará travada
- Se T1 chegar ao while antes de T0 rodar, acontecerá o contrário do anterior, com T1 rodando a seção crítica e T0 travado
- Se chegarem ao mesmo tempo no while, como $queroEntrar\_0$ & $queroEntrar\_1$ com certeza são $true$, dependerá da segunda condição, do valor de $TURN$. No entanto, só há 2 valores possíveis para esssa variável, sendo 0 a T1 ficará travada e sendo 1 a T0 ficará travada, então não há como permitir que ambas threads passem do while

Como não há condições para que acessem a seção crítica ao mesmo tempo, foi garantida exclusão mútua