Estudo Dirigido

Aluno: Caio Gomes Monteiro - 120036373

- Questão 1

  - Alternativa a
    Pode-se dizer que um programa é concorrente quando faz uso da execução simultânea de várias tarefas computacionais, que podem ser implementadas como programas separados ou como um conjunto de threads criadas por um único programa. Essas tarefas simultâneas podem ser feitas por um único processador, vários processadores em um único equipamento ou processadores distribuídos por uma rede.
  - Alternativa b
    Se as três tarefas consomem um tempo x de execução, mas apenas 2 tarefas podem ser paralelizadas, então a lei de Amdahl diz que o tempo de execução mínimo não pode ser menor que x daquela única tarefa que não pode ser paralelizada, o que faz com que a aceleração máxima seja de 3 vezes.

    Como cada vez que aumentamos o número de threads e assim a divisão da tarefa concorrente, o tempo de execução vai decaindo e decaindo, quando o número de threads tende ao infinito, a parte concorrente leva um tempo negligenciável, ou seja, pode-se determinar esse tempo como sendo 0, e a parte que não pode ser paralelizada é responsável pelo tempo de execução de todo o programa. Ou seja, para que essa aceleração máxima seja atingida, o requisito da execução é que o programa tenha um numero de threads que tenda ao infinito.
  - Alternativa c
    A seção crítica de um programa concorrente é toda e qualquer variável, atributo ou função que, sem a implementação de sincronização de threads, seriam acessados simultaneamente pelas threads, o que poderia gerar um comportamento indesejado quando essas threads tentassem acessar essa parte do código ao mesmo tempo. Existem várias maneiras de sincronização para assegurar que a área de código de um algoritmo que acessa um recurso compartilhado não possa ser acessado concorrentemente por mais de uma linha de execução.

- Questão 2

  - Alternativa a
    Um algoritmo que poderia ser implementado de forma concorrente é o algoritmo de

Gram-Schmidt, visto em álgebra linear algorítmica. O algoritmo recebe como entrada uma base de um subespaço qualquer do $\mathbb{R}^n$ e retorna uma base ortonormal do $\mathbb{R}^n$ que ainda representa o mesmo subespaço, ou seja, recebe uma lista de vetores qualquer e retorna uma lista de vetores ortonormais que ainda representam o mesmo subespaço original.

- Alternativa b
  O algoritmo, para poder calcular a base ortonormal resultante, precisa passar por todos os vetores da lista dada como entrada e calcular o vetor ortonormalizado que será retornado junto com os outros vetores da lista de saída. Temos então um loop do tamanho do número de elementos da lista de entrada. A programação concorrente poderia dividir a tarefa feita nesse loop com cada thread olhando para um grupo de vetores da lista de entrada diferente. Duas formas diferentes de dividir essas tarefas internas do algoritmo seriam fazer com que cada thread olhe para um vetor da lista de entrada alternadamente e outra forma seria dividir a lista de entrada pelo número de threads, fazendo blocos desse tamanho e cada thread olhando apenas para seu respectivo bloco. Como vimos nas nossas aulas, essas duas maneiras geram um bom balanceamento de carga, já que estamos sempre tentando dividir a lista de entrada pelo número de threads e o único desbalanceamento seria o resto dessa divisão.
- Alternativa c
  No caso desse algoritmo, não é necessário nenhuma sincronização, já que a única variável global que todas as threads visitam seria a lista de vetores dada como entrada, mas como definido anteriormente, cada thread acessaria elementos ou blocos de elementos distintos dentro dessa lista, o que faz com que a sincronização não seja necessária.

- Questão 3

  - Alternativa a
    Sim. Mesmo sem sincronização das threads, ainda é possível que as threads acessem a seção crítica do código uma de cada vez, o que faria com que a variável *saldo* tivesse seis acréscimos de 100, ou seja, o valor imprimido seria de 600.
  - Alternativa b
    Não. O menor valor possível para a variável *saldo* em qualquer ordem das threads é de 200. Isso acontece quando as três threads entram na primeira iteração dos seus loops e fazem $saldo = saldo + 100$ simultaneamente, o que faz com que apenas um acréscimo seja efetivamente realizado, e o valor de *saldo* mude para 100. Na segunda iteração, análogo a primeira iteração, novamente as threads realizam os acréscimos simultaneamente e isso faz com que o valor de *saldo* fique 200. Por isso que qualquer valor abaixo de 200, incluindo 100, é impossível de ocorrer.
  - Alternativa c
    Sim. Para o valor de 200 ser impresso, como já explicado na questão anterior, as três threads precisam entrar na primeira iteração dos seus loops e fazer $saldo = saldo + 100$

simultaneamente, o que faz com que apenas um acréscimo seja efetivamente realizado, e o valor de *saldo* mude para 100. Na segunda iteração, análogo a primeira iteração, novamente as threads precisam realizar os acréscimos simultaneamente e isso faz com que agora o valor de *saldo* fique 200.

- Alternativa d
  Sim. Primeiro, na primeira iteração dos loops, seria necessário as 3 threads realizarem os acréscimos separadamente, como se hovesse sincronização na seção crítica do código, fazendo com que $saldo = 300$. Em seguida, na segunda iteração dos loops, seria necessário que uma instrução de acréscimo fosse feita de forma normal e sem interrupção das outras threads, fazendo com que $saldo = 400$ e as duas últimas instruções de acréscimo restantes fossem feitas simultaneamente, o que faria com que, como explicado antes, apenas houvesse um acréscimo sendo realmente feito, fazendo com que, finalmente, o valor de *saldo* seja de 500.

- Questão 4

  - Alternativa a
    A ordem de execução das threads necessária para que o erro ocorra é que a thread 1 execute $S1$ primeiro, fazendo com que a thread entre dentro do if satetement, já que $thd \rightarrow proc\_info$ tem valor diferente de NULL. Depois, na thread 2, é executado $S3$, fazendo agora com que $thd \rightarrow proc\_info$ tenha valor de NULL. Agora, voltando para a thread 1 dentro do if statement, a thread tentará agora executar $S2$ com essa veriável com valor NULL, quebrando a premissa de atomicidade, causando um erro na aplicação.
  - Alternativa b
    Como o valor de $thd \rightarrow proc\_info$ pode mudar entre a execução de $S1$ e $S2$ com o código do enunciado, uma possível solução para isso seria reconhecer essa parte do código como uma seção crítica, tendo assim que implementar uma sincronização por exclusão mútua, o que faria com que outras threads não executem ao mesmo tempo da thread 1, o que garantiria a condição da variável $thd \rightarrow proc\_info$ quando ela entrou inicialmente no if statement.
  - Alternativa c
    Uma correção no código que faria com que a thread 2 apenas executasse seus comandos depois que a variável $mThread$ fosse devidamente inicializada é a implementação de sincronização condicional usando variáveis de condição. Dessa forma, a thread 2 apenas seria executada depois que a thread 1 sinalizasse, através de uma variável de condição, que foi executada e inicializou a variável $mThread$ com sucesso, fazendo com que a thread 2 apenas seja executada depois da execução da thread 1.

- Questão 5

– Alternativa a
A implementação da thread 2 e as alterações necessárias para tal na thread 1 são implementadas a seguir:

```
/*Variaveis de escopo global*/
pthread_cond_t cond_1;
pthread_mutex_t x_mutex;
int condicao=0;
int contador = 0;

void *Thread1 (void *t) {
  for (int i = 0; i<N ; i++){
      pthread_mutex_lock(&x_mutex);
      contador++;
      if (contador%100 == 0){
        condicao = 1;
        while (condicao == 1) {
          pthread_cond_wait(&cond_1, &x_mutex);
        }
      }
      pthread_mutex_unlock(&x_mutex);
  }
  pthread_exit(NULL);
}

void *Thread2 (void *t) {
  while(contador<N){
      pthread_mutex_lock(&x_mutex);
      if (condicao == 1) {
        printf("hooray! Multiplo de 100 encontrado!!! %d \n", contador);
        condicao=0;
        pthread_cond_signal(&cond_1);
      }
      pthread_mutex_unlock(&x_mutex);
  }
  pthread_exit(NULL);
}

```

– Alternativa b
Essa implementação atende a todos os requisitos. Quando o contador não é múltiplo de 100, a thread 1 apenas incrementa o contador em 1 e a thread 2 não faz nada pois a variável $condicao = 0$. Assim que o contador é um múltiplo de 100, a thread 1 entra no if statement, alterando o valor de *condicao* para 1 e entra em um loop esperando o sinal da thread 2 enquanto essa *condicao* continuar sendo igual a 1. Na thread 2, ela reconhece que agora $condicao = 1$, então imprime na tela o múltiplo de 100, altera o valor de *condicao* para 0 e sinaliza a thread 1. Dessa forma, a thread 1 espera a thread 2 imprimir totalmente antes de realizar qualquer acréscimo ao *contador* e a thread 2 é finalizada quando não há mais necessidade da sua atuação pois a última iteração do for loop da thread 1 faz com que $contador = N$, o que faz com que a thread 2 saia do *while* loop e seja finalizada.