**Teste 2 – Computação Concorrente**

**Aluno: Carlos Cozzolino**

**DRE: 115086800**

Questão 1:

* Após determinar um vetor de comprimento 2n na função main, utilizaremos as threads para realizar a soma descrita no enunciado. Com o ‘pthread\_create’, criamos a quantidade de threads e passamos seus Identificadores para a função ‘tarefa’, aqui determinamos como iremos percorrer os valores do vetor e, como antes da posição ‘vetor[0]’ não temos nada para somar, começamos com o valor inteiro ‘salto’ igual a 1. Depois de determinado o laço de repetição, criamos uma condição onde apenas as threads com números identificadores maiores ou iguais ao atual valor do ‘salto’ podem modificar o inteiro ‘aux’ e a variável global ‘vetor’, a condição é utilizada para escolhermos quais threads irão atuar dependendo do valor de ‘salto’. Dentro do condicional precisamos determinar o valor do ‘aux’, que é a posição onde vetor[id-salto] cai e então precisamos criar a nossa primeira barreira, visto que sem ela poderíamos fazer as próximas somas sem antes determinar o valor correto do ‘aux’, resultando em valores errados como resposta, a barreira permite entrar a quantidade de threads menos o valor atual do ‘salto’, por exemplo, se estamos trabalhando com 4 threads e o atual valor de ‘salto’ for 1, as threads de id 1, 2 e 3 poderão atuar na primeira iteração do loop. Depois de determinado os valores do ‘aux’ da primeira iteração, podemos atualizar o o vetor na posição ‘id’, que ficará como ‘vetor[id]’ somado do ‘aux’, depois disso, outra barreira é criada para evitarmos uma possível condição de corrida e recebermos o valor incorreto no final. As threads de ‘id’ menor que o valor do ‘salto’ são dispensadas daquela iteração e a função tarefa termina de atualizar os valores da variável global e se encerra retornando um valor nulo.

O código da função ‘barreira’ é o mesmo apresentado nas aulas e na ‘main’ alocamos valores para o vetor e escolhemos o seu tamanho e quantidade de threads a serem utilizadas, dentro dela também criamos os valores de ‘mutex’ e ‘cond’, chamamos a função ‘tarefa’ e, por fim, liberamos a memória que esses dois valores ocupavam mais a da variável ‘vetor’.

Um exemplo de funcionamento da ‘tarefa’:

Dado o vetor com valores: [0, 1, 3, 6], teremos ‘nthreads’ = 4 e os ‘ids’: 0, 1, 2, 3, onde o valor vetor[0] é dispensado no condicional ‘id’>=’salto’, visto que salto começa valendo 1.

Na primeira iteração temos que ‘nthreads’ - ‘salto’ = 3, logo, os valores de aux serão: 0 para o id 1 (como aux = vetor[id-salto], se salto vale 1 e o id também 1, temos que aux = vetor[0]), 1 para o id 2, 3 para o id 3. Com esses valores, iremos somar aux + vetor[id] e atualizar o valor de vetor [id], ou seja, para o id = 1 fica: 0 + 1, para o id = 2: 1 + 3 e para o id = 3: 3 + 6. No final da primeira iteração teremos o vetor:[0, 1, 4, 9].

Agora o valor de ‘salto’ é atualizado e passa a valer 2\*1, logo as threads de ids menores do que 2 serão dispensadas e todo o processo anterior é realizado novamente, chegando nestes valores para o aux: 0 para a thread de id 2 e 1 para a thread de id 3(o valor da posição do vetor[3-2] é 1). E com isso podemos determinar os valores do vetor depois da segunda iteração, onde: para o id = 2 temos 0 + 4 e para o id = 3 temos 1 + 9, resultando em [0, 1, 4, 10]!

* A resposta concorrente apresentada está correta e, se usarmos as especificações apresentadas no enunciado, obtemos os mesmos valores que da forma concorrente. Um código sequencial e outro concorrente serão anexados para mostrar tal equivalência.
* Não poderiam, pois se removermos a primeira barreira algumas threads podem começar a somar os valores antes de o ‘aux’ estar devidamente determinado e, se removermos a segunda barreira, podemos ter uma condição de corrida ao rodar o código e uma thread ‘atropelar’ a outra, resultando em valores menores do que o esperado.

Questão 2:

* Código enviado junto com o PDF.

Questão 4:

* Algumas causas de inanição das threads poderiam ser:

1. Alguma thread lidando com algo de alta prioridade, ocupando a CPU com algo que demanda muito poder e tempo de processamento, fazendo com que as threads de prioridades mais baixas fiquem esperando esse processo finalizar.
2. Uma função ‘synchronized’ está bloqueando as threads para serem executadas em alguma forma específica, determinada pelo programa.
3. Como o método ‘Notify’ não possui nenhuma forma de controlar quais threads começam a atuar e quais estão paradas a muito tempo, pode acontecer que em um processo envolvendo múltiplas threads alguma delas nunca consiga participar da execução.

* Código enviado junto com o PDF.