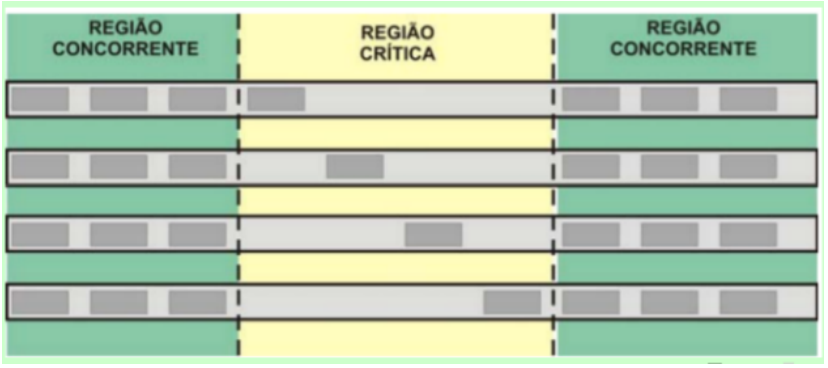
Acadêmica: Jênifer Vieira Goedert

1. Explique o que é *race conditions*.

São situações que acontecem quando duas ou mais threads tentam atualizar, ao mesmo tempo, uma variável ou quando uma thread atualiza uma variável e outra thread acessa o valor dessa variável ao mesmo tempo. Quando acontece uma dessas condições, o resultado vai depender da ordem de execução das threads e não há nenhuma garantia que a variável seja atualizada com o valor correto. Assim, as diretivas de sincronização garantem que o acesso/atualização de uma variável aconteça no momento certo.

2. Explique como funciona o *#pragma omp critical*.

O construtor *critical* restringe a execução de uma determinada tarefa a apenas uma thread por vez. É utilizado para evitar condições de corrida. Quando uma thread encontra uma sessão crítica, ela espera no início da mesma até que nenhuma thread esteja executando as instruções da região crítica de mesmo nome. Quando nenhuma thread estiver executando essa sessão crítica, a thread que estava esperando, entra na região e executa as instruções.



3. Qual a diferença do *#pragma omp critical* para o *#pragma omp single*.

No *critical* todas as threads executam o mesmo código, porém cada thread por vez. Não existem problemas relacionados a acesso a região crítica ou condição de corrida. Já no *single*, somente a primeira thread disponível executa o bloco dentro do single, as outras aguardam o final da execução na barreira implícita.

4. Explique como funciona o *#pragma omp atomic*.

O construtor *atomic* permite que certa região da memória seja atualizada atomicamente, impedindo que várias threads acessem essa região ao mesmo tempo. Ele também habilita múltiplas threads a atualizarem dados compartilhados sem interferência. Essa diretiva é usada para proteger uma atualização única para uma variável compartilhada, sendo utilizada para uma única instrução. É importante destacar que essa restrição é aplicada a todas as threads que executam o programa, não somente as threads pertencentes a um mesmo bloco (time). Esse construtor é aplicado somente à instrução localizada logo abaixo da diretiva.

5. Quais as diferenças entre *#pragma omp critical* e *#pragma omp atomic*.

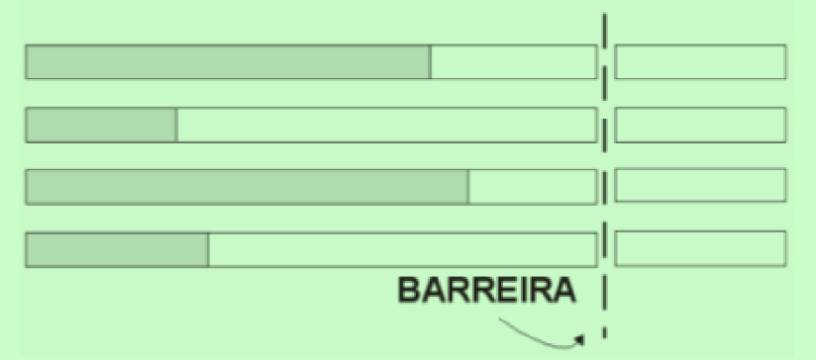
A utilização do construtor *atomic* é muito parecida com a utilização do construtor *critical*.

*#pragma omp critica*l: Usado para proteger blocos de código arbitrários. É mais genérico e pode proteger qualquer bloco de código, mas com maior sobrecarga.

#pragma omp atomic: Usado para proteger operações simples e específicas, como incrementos e decrementos, com menor sobrecarga devido ao uso de operações atômicas de hardware.

6. Explique como funciona o *#pragma omp barrier*.

É utilizada para sincronizar todas as threads em um determinado ponto do código. Quando uma thread encontra uma barreira, ela espera até que todas as threads cheguem naquele ponto e, a partir daí, elas continuam a execução do código ao mesmo tempo.



7. Explique como funciona o *#pragma omp flush* e cite os pontos implícitos que ele é chamado.

É usada para garantir que todas as threads tenham acesso ao valor correto de uma variável compartilhada que foi recentemente atualizada, principalmente nos trechos entre os pontos de sincronização. Isso porque quando uma variável compartilhada é atualizada por uma thread, o novo valor dela fica armazenado somente na memória cache do processador (de acesso rápido) que não é visível pelos outros processadores. Assim, threads de outros processadores poderão ter acesso a um valor desatualizado dessa variável. As atualizações que ocorrem em pontos de sincronização ocorrem na memória principal, porém entre esses trechos não há nenhuma garantia de que as variáveis sejam atualizadas instantaneamente na memória principal.

Flush implícitos no OpenMP:

a) Em todas as barreiras explícitas e implícitas;
b) Na entrada e na saída de uma região crítica;
c) Na entrada e na saída de uma diretiva *ordered*;
d) Na entrada e na saída de rotinas lock;
e) Na saída de um construtor de divisão de trabalho.

Link do GitHub para os códigos das próximas questões: https://github.com/Jenifer19IFC/OpenMP-atividades/tree/main/aula5ativ.

8. **Atualização Segura de Variáveis com #pragma omp critical** - Implemente um programa que calcula o histograma de um vetor de inteiros. Utilize a diretiva #pragma omp critical para garantir que a atualização das contagens no histograma seja feita de forma segura quando múltiplas threads estão incrementando as mesmas posições simultaneamente. Crie um vetor de inteiros para o histograma e inicialize-o com zeros. Utilize #pragma omp parallel for para paralelizar a leitura do vetor original e #pragma omp critical para proteger a atualização do histograma.

9. **Contagem Paralela com Atualização Atômica** - Implemente um programa que conte o número de elementos pares e ímpares em um vetor de inteiros. Utilize a diretiva #pragma omp atomic para garantir que a contagem das variáveis de pares e ímpares seja feita de forma segura em um ambiente de execução paralela. Utilize #pragma omp parallel for para paralelizar a iteração sobre o vetor e #pragma omp atomic para proteger a atualização das contagens.

10. **Sincronização de Threads com #pragma omp barrier** - Implemente um programa onde múltiplas threads realizam um conjunto de operações em duas etapas distintas. Utilize a diretiva #pragma omp barrier para garantir que todas as threads completem a primeira etapa antes de prosseguirem para a segunda etapa. Divida as operações em duas seções dentro de um bloco #pragma omp parallel, e insira #pragma omp barrier entre as seções para sincronizar as threads.

11. **Comunicação de Dados entre Threads com #pragma omp flush** - Implemente um programa onde uma thread produtora gera dados e várias threads consumidoras processam esses dados. Utilize a diretiva #pragma omp flush para garantir que as threads consumidoras leiam os dados mais recentes produzidos pela thread produtora. Utilize uma variável compartilhada para armazenar os dados produzidos e utilize #pragma omp flush para sincronizar a memória entre a thread produtora e as threads consumidoras.

12. **Cálculo Paralelo de Pi com Redução e Sincronização** - Implemente um programa que utilize o método de Monte Carlo para calcular o valor de PI. Divida a execução em múltiplas threads que geram pontos aleatórios no intervalo [0, 1]. Utilize #pragma omp critical ou #pragma omp atomic para proteger a contagem de pontos dentro do círculo e utilize #pragma omp barrier para sincronizar todas as threads antes de calcular o valor final de PI. Utilize #pragma omp parallel para paralelizar a geração de pontos e #pragma omp barrier para garantir que todas as threads completem a geração de pontos antes do cálculo final.