Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia da Computação e Automação - DCA
DCA3703 - Programação Paralela

Tarefa 10: Mecanismos de Sincronização: Atomic e Reduction

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

I. INTRODUÇÃO

Na programação paralela, diferentes partes de um programa são executadas ao mesmo tempo por várias threads. Quando essas threads precisam acessar ou modificar os mesmos dados, é preciso ter cuidado para que não aconteçam erros, chamados de condições de corrida. Para evitar esses problemas, existem mecanismos de sincronização que organizam esse acesso compartilhado. Um deles é o ***critical***, que faz com que apenas uma *thread* de cada vez possa executar um certo trecho do código, garantindo segurança, mas podendo causar lentidão quando muitas threads tentam usar esse trecho ao mesmo tempo. O ***atomic*** é um pouco mais leve, pois funciona apenas com operações simples e rápidas, reduzindo o tempo de espera. Já o ***reduction*** é uma forma mais inteligente de lidar com somas e contagens feitas por várias threads, permitindo que cada uma trabalhe separadamente e junte os resultados no final, de forma rápida e eficiente. Esses mecanismos são essenciais para manter a correção dos programas paralelos, mas cada um tem suas vantagens e deve ser usado conforme a necessidade.

Para garantir a correta agregação de resultados parciais em ambientes paralelos, é necessária a aplicação desses mecanismos de sincronização. Este relatório apresenta a análise comparativa de três abordagens em OpenMP: *critical*, *atomic* e *reduction*, avaliando seu impacto no desempenho e na produtividade durante a execução paralela de uma simulação de lançamento de pontos na estimativa estocástica de π.

II. METODOLOGIA

A implementação[1] utilizou a linguagem C com a biblioteca OpenMP para paralelizar a estimativa de π com 1 bilhão de pontos lançados aleatoriamente em um quadrado de lado unitário. Cada versão aplicou uma abordagem de sincronização distinta para somar o número de pontos dentro do círculo de raio 1. Os testes foram realizados com 4 *threads*, utilizando *rand_r()* com sementes individuais por *thread* para garantir a independência dos valores gerados. O tempo de execução de cada abordagem foi medido com precisão, e o valor estimado de π foi comparado para validar a consistência dos resultados.

III. RESULTADOS E DISCUSSÕES

[1] Link do código desenvolvido para a tarefa:
https://github.com/quelita2/programacao-paralela/blob/main/topico01/tarefa10/src/main.c

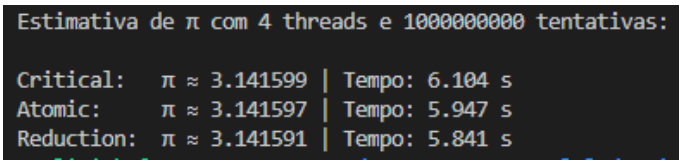

Imagem 1: Resultado de Compilação da tarefa

A principal vantagem do uso de *reduction* está no seu bom desempenho, que vem do fato de ele evitar a necessidade de sincronizar constantemente as *threads* durante a execução. Enquanto as diretivas *critical* e *atomic* fazem as *threads* acessarem diretamente uma variável compartilhada, que exige bloqueios frequentes para evitar erros, enquanto o *reduction* permite que cada *thread* faça seu próprio cálculo de forma independente. Só no final esses resultados são combinados. Isso reduz bastante a disputa por acesso à mesma variável e evita os atrasos causados por bloqueios.

Além disso, o *reduction* facilita a vida do programador e funciona muito bem mesmo quando o número de *threads* aumenta, ou seja, ele é mais escalável. Já o *atomic*, embora mais leve que o *critical*, ainda exige um controle por trás a cada vez que a variável é usada, o que pode atrapalhar quando há muitas atualizações. O *critical*, por fim, é o mais custoso de todos: ele impede que mais de uma thread entre em uma parte do código ao mesmo tempo, o que diminui bastante o paralelismo.

Com base nisso, é possível seguir um caminho prático para escolher a melhor opção de sincronização em OpenMP:

- Use ***reduction*** quando a operação for acumulativa, como somar, contar ou calcular médias. Ele traz o melhor desempenho e exige pouco do programador.
- Use ***atomic*** quando a operação sobre a variável for simples, como um incremento ou subtração, e quando *reduction* não for uma opção. Ele é um meio-termo entre desempenho e simplicidade.
- Use ***critical*** apenas quando for realmente necessário proteger várias instruções ao mesmo tempo ou estruturas mais complexas. Por ser mais pesado, deve ser usado com cuidado.

Em resumo, a escolha do melhor mecanismo depende do tipo de tarefa, de como a variável compartilhada é usada e de quanto se quer aproveitar o paralelismo. A ideia é sempre equilibrar desempenho, segurança e simplicidade no código.

IV. CONCLUSÃO

A tarefa demonstrou, de forma prática, como a escolha do mecanismo de sincronização influencia diretamente no desempenho de aplicações paralelas. Entre os mecanismos analisados, o uso da cláusula *reduction* se destacou como a alternativa mais eficiente e produtiva para somatórios paralelos, pois evita contenção entre as threads ao realizar o acúmulo de forma local e apenas combinar os resultados no final. Em contraste, o uso de atomic e, especialmente, *critical*, introduziu maior sobrecarga devido ao controle mais restritivo do acesso à variável compartilhada. Assim, conclui-se que mecanismos como *reduction* devem ser priorizados sempre que possível em operações associativas e comutativas simples, reservando *atomic* para atualizações pontuais e *critical* para

trechos mais complexos que exigem proteção abrangente. Essa escolha cuidadosa é essencial para alcançar maior escalabilidade, eficiência e produtividade em sistemas paralelos.