Análise de Estratégias de Sincronização em OpenMP

Regiões Críticas Nomeadas vs. Locks Explícitos na Manipulação de Listas Encadeadas

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)

Disciplina de Programação Paralela - DCA3703

12 de setembro de 2025

Introdução

- A manipulação de estruturas de dados dinâmicas, como listas encadeadas, em paralelo apresenta desafios significativos de sincronização para evitar **condições de corrida**.
- O objetivo do projeto é analisar o impacto de duas diferentes estratégias de sincronização do OpenMP para o problema de inserções concorrentes.
- Foram explorados dois cenários distintos de gerenciamento de recursos:
  - **Cenário Estático:** Com um número fixo de listas, usando regiões críticas nomeadas.
  - **Cenário Dinâmico:** Com um número `M` de listas, exigindo o uso de locks explícitos.
- O estudo busca ilustrar as diferenças de **aplicabilidade, flexibilidade e complexidade** entre as abordagens.

Metodologia: O Problema Proposto

Inserções Paralelas em Listas Encadeadas

Lógica do Algoritmo

- O problema consiste na execução de `N` inserções em paralelo, distribuídas por um conjunto de `M` listas.
- A cada iteração, uma thread escolhe aleatoriamente uma das `M` listas para inserir um novo nó.
- A função `rand_r()` é usada para garantir a geração de números aleatórios de forma *thread-safe*.

Desafio

- Como gerenciar o acesso simultâneo ao ponteiro `head` de cada lista de forma **correta e eficiente**?
- A modificação do ponteiro `head` é a **seção crítica** que precisa ser protegida.

Metodologia: As Duas Versões Analisadas

Foram implementadas duas versões em C com OpenMP para isolar diferentes estratégias de sincronização:

Versão 1: Cenário Estático (Regiões Críticas Nomeadas)

- Aborda o problema para um número fixo de duas listas.
- Utiliza `#pragma omp critical (name)` para criar locks independentes (`lock_A` e `lock_B`).
- Permite que inserções em listas diferentes ocorram simultaneamente.

Versão 2: Cenário Dinâmico (Locks Explícitos)

- Generaliza o problema para um número `M` de listas, definido em tempo de execução.
- Utiliza um array de `omp_lock_t` alocado dinamicamente, onde o `lock[i]` protege a `lista[i]`.
- Permite travamento granular e dinâmico.

Análise dos Mecanismos Utilizados

- `#pragma omp parallel for`: Usado em ambas as versões para distribuir as `N` iterações do laço entre as threads disponíveis.
- `#pragma omp critical (name)`:
  - Garante a exclusão mútua de forma declarativa e de alto nível.
  - É uma solução rígida, aplicável apenas quando os recursos são **fixos e conhecidos em tempo de compilação**.
- **Array de** `omp_lock_t`:
  - Estratégia de "locks explícitos"que oferece controle total ao programador.
  - Solução flexível e escalável, **necessária para cenários dinâmicos** com número variável de recursos.
  - Exige gerenciamento manual do ciclo de vida do lock (init, set, unset, destroy).

Análise de Desempenho: A Importância da Granularidade

Fine-Grained vs. Coarse-Grained Locking

Estratégia Adotada: Alta Granularidade (Fine-Grained)

Ambas as implementações adotam uma estratégia de alta granularidade, onde cada lista possui seu próprio lock.

- **Benefício:** Esta é a abordagem de melhor desempenho, pois **minimiza a contenção**. Uma thread só bloqueia outra se ambas tentarem acessar a *mesma lista* simultaneamente.
- **Alternativa Ineficiente:** Se usássemos um único lock para todas as listas (*coarse-grained*), como um `#pragma omp critical` sem nome, o desempenho seria drasticamente pior.
  - Todas as inserções seriam serializadas, criando um **gargalo sequencial** e eliminando os benefícios do paralelismo.

Conclusão

- O experimento demonstrou que a escolha de mecanismos de sincronização é uma **decisão de design** ditada pela natureza do problema (estático vs. dinâmico).
- Abstrações de alto nível (`critical (name)`) são convenientes, mas insuficientes para problemas com um número dinâmico de recursos, onde **locks explícitos são obrigatórios**.
- A implementação de uma estratégia de **alta granularidade** foi crucial em ambas as versões para permitir a máxima concorrência e evitar a serialização desnecessária das tarefas.
- A tarefa reforça que a compreensão de mecanismos de mais baixo nível, como os locks explícitos, é fundamental para desenvolver **soluções paralelas robustas e escaláveis**.