UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 9 - REGIÕES CRÍTICAS NOMEADAS E LOCKS EXPLÍCITOS
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 27 DE ABRIL DE 2025

SUMÁRIO

1. INTRODUÇÃO

Este relatório descreve o desenvolvimento de um programa utilizando a *API OpenMP* para manipulação de listas encadeadas de forma paralela, visando a resolução de problemas típicos de concorrência, como as condições de corrida. O objetivo principal é realizar inserções em múltiplas listas encadeadas de maneira segura, utilizando técnicas de sincronização adequadas para garantir a integridade das listas durante a execução de tarefas paralelas.

A tarefa foi dividida em duas partes principais: a primeira parte implementa um modelo de inserção em duas listas utilizando *tasks* do *OpenMP* e regiões críticas nomeadas. Essa abordagem assegura que as inserções em listas diferentes não bloqueiem uma à outra, permitindo um desempenho mais eficiente. A segunda parte do trabalho generaliza a solução para um número variável de listas, implementando uma estratégia de *locks* explícitos para garantir que as inserções em listas compartilhadas sejam feitas de forma segura, sem interferência de outras tarefas paralelas.

A tarefa de sincronização em ambientes paralelos é crítica para garantir que a manipulação de dados compartilhados, como listas encadeadas, não resulte em inconsistências ou falhas durante a execução. O uso de *OpenMP* para gerenciar múltiplas threads e operações concorrentes é uma forma eficaz de otimizar a execução de tarefas computacionalmente intensivas, especialmente quando se trabalha com grandes volumes de dados ou operações paralelizadas.

Este trabalho foi desenvolvido com o intuito de compreender e aplicar conceitos de concorrência, como sincronização e controle de acesso a recursos compartilhados, além de explorar os métodos oferecidos pelo *OpenMP* para facilitar a implementação de programas paralelos.

2. METODOLOGIA

A metodologia adotada para o desenvolvimento deste trabalho foi estruturada em duas abordagens principais: uma utilizando regiões críticas nomeadas, para o caso de duas listas, e outra com a implementação de múltiplas listas, utilizando *locks* explícitos para gerenciar o acesso concorrente.

Na primeira parte da tarefa, o objetivo era implementar a inserção paralela em duas listas encadeadas de forma independente. Para isso, foi utilizado o *OpenMP* para criar tarefas paralelizadas com a diretiva *#pragma omp task*. O controle de acesso a cada lista foi realizado por meio de regiões críticas nomeadas. Essas regiões críticas são um mecanismo que garante que, durante a execução de uma tarefa, apenas uma *thread* possa acessar uma lista específica, evitando condições de corrida.

A estrutura geral do código consistiu nas seguintes etapas:

1. **Criação das listas:** Duas listas encadeadas foram inicializadas no início da execução. As listas são representadas por ponteiros para estruturas do tipo *Node*, que contêm um valor e um ponteiro para o próximo nó.
2. **Criação de tarefas:** Utilizando *#pragma omp task*, cada inserção foi organizada dentro de uma tarefa paralela. As tarefas foram distribuídas entre as *threads* de forma que cada uma delas escolhesse aleatoriamente em qual lista inserir um número gerado aleatoriamente.
3. **Sincronização com regiões críticas:** A sincronização das inserções nas listas foi garantida utilizando regiões críticas nomeadas, especificando que a inserção em uma lista não pudesse ocorrer simultaneamente em múltiplas *threads*. Dessa forma, a inserção em *list1* e *list2* foi protegida por regiões críticas específicas, respectivamente denominadas *list1_section* e *list2_section*.

Na segunda parte da tarefa, a solução foi generalizada para permitir a manipulação de um número arbitrário de listas. A abordagem para garantir a segurança nas inserções foi modificada para utilizar *locks* explícitos, por meio do tipo de dado *omp_lock_t* do *OpenMP*. O uso de *locks* explicitamente associados a cada lista assegura que uma única *thread* possa inserir elementos em uma lista ao mesmo tempo, sem interferência de outras *threads*.

O processo de implementação para o modo generalizado foi o seguinte:

1. **Inicialização das listas e locks**: O número de listas (M) foi definido pelo usuário, e cada lista foi representada por um ponteiro para uma estrutura do tipo *Node*. Além disso, foi criado um *array* de *locks*, *locks[MAX_LISTS]*, onde cada índice corresponde a uma lista e contém um *lock* utilizado para controlar o acesso a ela.
2. **Criação de tarefas paralelas**: De maneira semelhante à parte anterior, as inserções nas listas foram feitas dentro de tarefas paralelas, utilizando novamente a diretiva *#pragma omp task*. A principal diferença aqui foi que a lista em que cada valor seria inserido era escolhida aleatoriamente entre as M listas.

3. **Controle de acesso com locks**: Para cada tarefa de inserção, a *thread* que a executa faz o bloqueio do *lock* correspondente à lista selecionada, insere o valor na lista e, por fim, libera o *lock*. Esse mecanismo de bloqueio explícito garante que não haja sobrecarga de inserções simultâneas na mesma lista, o que poderia causar inconsistências.
4. **Destruição dos locks**: Após a execução das tarefas, os *locks* foram destruídos, garantindo a liberação de recursos alocados para o controle de concorrência.

O uso de regiões críticas nomeadas na primeira parte foi adequado, pois tratava-se de apenas duas listas, o que permitia o uso de regiões críticas específicas para cada lista sem grandes custos de desempenho. No entanto, ao generalizar o número de listas, a implementação de *locks* explícitos foi fundamental para garantir a integridade dos dados, já que múltiplas listas poderiam ser acessadas de maneira simultânea, o que torna as regiões críticas nomeadas ineficazes para um número grande de listas.

3. RESULTADOS

A execução do programa foi dividida em duas partes, correspondendo aos dois modos de inserção em listas encadeadas: o Modo Nomeado e o Modo Generalizado. A seguir, serão apresentados os resultados obtidos para ambos os modos.

No Modo Nomeado, o programa foi configurado para realizar 10 inserções em duas listas encadeadas de forma paralela, com o uso de tarefas OpenMP e regiões críticas nomeadas. Cada tarefa paralela escolheu aleatoriamente em qual lista inserir um valor gerado aleatoriamente. A sincronização foi realizada por meio de regiões críticas específicas para cada lista, garantindo que as inserções fossem feitas de forma segura. A saída do programa apresentou duas listas, onde os valores foram inseridos de forma não ordenada, devido à escolha aleatória durante a execução das tarefas. O programa foi capaz de executar todas as inserções de forma segura, sem quaisquer erros de concorrência ou violação de integridade dos dados.

```
[Method Named] List 1: 74 -> 39 -> 88 -> 74 -> 88 -> 2 -> NULL
[Method Named] List 2: 46 -> 16 -> 95 -> 16 -> NULL
```

Esse comportamento demonstrou a eficácia da utilização de regiões críticas nomeadas para a sincronização entre as threads, garantindo que a inserção nas duas listas fosse feita de forma independente e sem sobrecarga de bloqueios.

No Modo Generalizado, o programa foi configurado para permitir a inserção em um número variável de listas, com o número de listas (M) e o número de inserções (N) definidos pelo usuário. No exemplo realiado, o número de listas foi definido como 15, e o número de inserções foi de 100. O controle de acesso às listas foi realizado por meio de *locks* explícitos, garantindo que cada lista fosse acessada por apenas uma thread por vez. Para cada tarefa paralela, uma lista foi escolhida aleatoriamente e o valor foi inserido nela. O uso de *locks* garantiu que não houvesse conflitos ao tentar acessar e modificar a mesma lista simultaneamente. Como resultado, os dados foram corretamente inseridos nas listas sem a ocorrência de condições de corrida. A saída do programa mostrou que as inserções foram distribuídas entre as listas de forma aleatória. O número de elementos em cada lista variou dependendo das escolhas feitas pelas *threads* durante a execução.

```
[Method Generalized] List 0: 84 -> 452 -> 943 -> 820 -> 188 -> 188 ->
NULL
[Method Generalized] List 1: 680 -> 189 -> 557 -> 48 -> 925 -> 293 ->
NULL
[Method Generalized] List 2: 172 -> 417 -> 294 -> 662 -> 153 -> 30 ->
NULL
[Method Generalized] List 3: 154 -> 277 -> 522 -> 890 -> 767 -> 258 ->
NULL
```

```
...
```

A distribuição dos valores nas listas mostrou que as tarefas foram corretamente atribuídas a diferentes listas, com um número aleatório de inserções em cada uma delas. A integridade dos dados foi preservada, e o uso dos *locks* assegurou que nenhuma lista fosse corrompida devido a acessos simultâneos.

Embora a tarefa tenha sido realizada com sucesso em ambos os modos, a parte do Modo Generalizado pode apresentar desafios de desempenho dependendo do número de listas e do número de inserções. O uso de *locks* explícitos, embora eficaz na sincronização, pode introduzir algum *overhead* devido ao tempo necessário para adquirir e liberar os *locks*.

A principal vantagem do Modo Nomeado está na sua simplicidade e na menor sobrecarga, já que o número de listas é fixo e o uso de regiões críticas nomeadas é eficiente para situações com um número limitado de recursos compartilhados. No entanto, à medida que o número de listas cresce no Modo Generalizado, o uso de *locks* torna-se necessário para garantir a integridade dos dados, embora possa resultar em uma leve queda no desempenho, especialmente em sistemas com muitas *threads*.

Dessa forma podemos dizer que o programa demonstrou boa escalabilidade na medida em que foi possível aumentar o número de listas e inserções sem comprometer a integridade dos dados. Contudo, à medida que o número de listas aumenta, o custo de gerenciamento dos *locks* também cresce, o que pode afetar o desempenho em situações de grande carga de trabalho.

4. CONCLUSÃO

O exercício desenvolvido permitiu explorar a implementação de operações paralelas em listas encadeadas, utilizando as técnicas de *OpenMP*, como tarefas paralelizadas e sincronização por meio de regiões críticas nomeadas e *locks* explícitos. Ao longo das duas abordagens implementadas, Modo Nomeado e Modo Generalizado, foi possível observar as vantagens e desafios de cada técnica em relação à integridade dos dados e ao desempenho da aplicação.

No Modo Nomeado, foi possível utilizar regiões críticas nomeadas para garantir a integridade de duas listas encadeadas, com um custo computacional relativamente baixo. O uso de regiões críticas nomeadas simplifica o controle de acesso a recursos compartilhados quando o número de listas é pequeno e fixo. As inserções ocorreram de forma segura, com boa performance, sem sobrecarga excessiva, e o programa foi capaz de lidar com inserções simultâneas em duas listas de maneira eficiente.

Já o Modo Generalizado, que envolveu um número variável de listas e inserções, exigiu o uso de *locks* explícitos para garantir a sincronização entre as *threads*, permitindo a inserção em um número indefinido de listas. Embora a abordagem tenha oferecido flexibilidade e escalabilidade, o uso de *locks* introduziu uma certa sobrecarga, que pode afetar o desempenho em cenários de alta concorrência. Mesmo assim, a solução foi capaz de garantir a integridade das listas e a correta distribuição dos valores, independentemente do número de listas ou inserções.

A análise dos resultados demonstrou que, enquanto o Modo Nomeado é adequado para situações com um número reduzido de listas, o Modo Generalizado é mais flexível e escalável, mas com um custo adicional em termos de desempenho. Além disso, ficou claro que, conforme o número de listas aumenta, o uso de técnicas de sincronização, como *locks*, torna-se necessário para evitar condições de corrida, mas também pode impactar o desempenho devido ao tempo de espera envolvido no bloqueio e desbloqueio das listas.

Em termos de escalabilidade, a solução se mostrou eficiente até um certo ponto. Com o aumento do número de listas e threads, o desempenho pode ser afetado pela sobrecarga das operações de sincronização, mas a integridade dos dados foi mantida em todos os cenários testados.

Em resumo, evidenciamos a importância de escolher a técnica de sincronização adequada para cada situação. O uso de regiões críticas nomeadas é ideal para cenários simples e com poucos recursos compartilhados, enquanto os *locks* são essenciais para garantir a integridade em sistemas mais complexos e escaláveis, embora com um custo computacional adicional.

5. ANEXOS

- Repositório no Github com o programa desenvolvido: https://github.com/ErnaneJ/parallel-programming-dca3703