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

Tarefa 09: Regiões Críticas Nomeadas e Locks Explícitos

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

I. INTRODUÇÃO

O presente relatório descreve a implementação e análise de dois programas paralelos utilizando OpenMP para inserir números aleatórios em listas encadeadas, explorando os mecanismos de sincronização *#pragma omp critical* e *omp_lock_t*. O objetivo é garantir a integridade das estruturas de dados em ambientes concorrentes, evitando condições de corrida durante as inserções paralelas. A condição de corrida ocorre quando duas ou mais *threads* tentam acessar ou modificar simultaneamente uma mesma estrutura de dados compartilhada, o resultado final é a perda de dados à medida que um nó sobrescreve outro. Para evitar essa condição de corrida existente utiliza-se mecanismos de sincronização, como:

I.I Regiões Críticas:

A diretiva ***#pragma omp critical*** define uma seção crítica, garantindo que apenas uma thread execute o bloco de código por vez. Ao atribuir um nome à região crítica (*#pragma omp critical(nome)*), múltiplas seções podem ser protegidas de forma independente, permitindo execução paralela desde que acessem recursos distintos.

I.II Locks Explícitos

Locks explícitos com ***omp_lock_t*** oferecem controle manual sobre a exclusão mútua. São úteis em cenários onde a sincronização dinâmica é necessária, como no caso de estruturas com tamanho variável definido em tempo de execução. As funções utilizadas incluem:

- **omp_init_lock(&lock)**: inicializa o lock.
- **omp_set_lock(&lock)**: adquire o lock.
- **omp_unset_lock(&lock)**: libera o lock.
- **omp_destroy_lock(&lock)**: finaliza o lock.

II. METODOLOGIA

A metodologia consistiu na implementação[1] de dois programas paralelos utilizando OpenMP para realizar inserções seguras em listas encadeadas. Na primeira versão, foram utilizadas duas listas fixas, e cada tarefa criada escolheu aleatoriamente em qual delas inserir um número aleatório. Para evitar condições de corrida, foram aplicadas regiões críticas nomeadas específicas para cada lista, permitindo que duas *threads* inserissem simultaneamente em listas diferentes sem bloqueios desnecessários.

[1] Link do código desenvolvido para a tarefa:

Na segunda versão, o número de listas foi definido dinamicamente como quatro. Cada lista foi associada a um lock explícito do tipo *omp_lock_t,* permitindo controle individual sobre a exclusão mútua durante as inserções. A cada tarefa, um número aleatório era gerado e inserido em uma das listas escolhida aleatoriamente, com a operação protegida pelo *lock* correspondente. Essa abordagem possibilitou a sincronização adequada mesmo com número de listas indefinido em tempo de compilação.

III. RESULTADOS E DISCUSSÕES

```
=== VERSÃO 1: 2 listas com regiões críticas nomeadas ===
Lista 1: 575 -> 208 -> 583 -> 993 -> 409 -> 399 -> 52 -> 681 -> 633 -> 999 -> 897 -> 717 -> 391 ->
 676 -> 371 -> 666 -> 74 -> 269 -> 659 -> 739 -> 699 -> 808 -> 291 -> 415 -> 190 -> 656 -> 618 ->
506 -> 780 -> NULL
Lista 2: 595 -> 974 -> 42 -> 393 -> 376 -> 515 -> 49 -> 30 -> 559 -> 492 -> 126 -> 999 -> 456 -> 1
81 -> 544 -> 693 -> 164 -> 803 -> 783 -> 17 -> 895 -> NULL
```

Imagem 1: Resultados de Compilação para a Versão 1

Na primeira versão do programa, as inserções foram realizadas com sucesso em duas listas encadeadas, com distribuição variável dos elementos entre elas. A utilização de regiões críticas nomeadas (*#pragma omp critical(lista1)* e *#pragma omp critical(lista2)*) permitiu que diferentes threads acessassem listas distintas simultaneamente, sem interferência mútua. Isso aumentou a eficiência do paralelismo e garantiu a integridade das listas, pois cada uma possuía sua própria seção crítica protegida. A saída confirmou a eficácia da abordagem, com inserções bem-sucedidas e listas consistentes.

```
=== VERSÃO 2: 4 listas com locks explícitos ===
Lista 1: 931 -> 857 -> 126 -> 857 -> 883 -> 173 -> 317 -> 235 -> 691 -> 882 -> 7 -> NULL
Lista 2: 819 -> 117 -> 837 -> 765 -> 941 -> 209 -> 573 -> 951 -> 349 -> 794 -> 554 -> 149 -> 43 ->
 513 -> 689 -> NULL
Lista 3: 308 -> 176 -> 936 -> 734 -> 890 -> 334 -> 810 -> 671 -> NULL
Lista 4: 319 -> 852 -> 609 -> 842 -> 686 -> 33 -> 989 -> 750 -> 655 -> 306 -> 709 -> 222 -> 354 ->
 939 -> 776 -> 318 -> NULL
```

Imagem 2: Resultados de Compilação para a Versão 2

Na segunda versão, o número de listas foi generalizado para quatro, definido dinamicamente. Como *#pragma omp critical(nome)* exige que os nomes das regiões críticas sejam fixos e conhecidos em tempo de compilação, essa abordagem se tornou inviável para múltiplas listas criadas em tempo de execução. Regiões críticas nomeadas não podem ser instanciadas dinamicamente, o que limitaria a sincronização apenas a listas predefinidas.

Diante disso, a utilização de *locks* explícitos com *omp_lock_t* foi essencial. Cada lista recebeu um *lock* individual, inicializado e manipulado manualmente. Isso permitiu o controle preciso sobre a exclusão mútua, protegendo a inserção de forma independente em cada lista, mesmo com o número de listas sendo variável. A saída demonstrou uma distribuição aleatória dos elementos entre as quatro listas, com as inserções ocorrendo de forma segura e paralela, comprovando que os *locks* explícitos garantiram a integridade das estruturas mesmo em um contexto mais dinâmico e escalável.

IV. CONCLUSÃO

Os experimentos demonstraram a importância de mecanismos de sincronização apropriados para garantir a integridade de estruturas de dados compartilhadas em ambientes paralelos. Na primeira versão, o uso de regiões críticas nomeadas foi suficiente e eficiente para proteger duas listas fixas, permitindo concorrência segura entre threads ao evitar bloqueios desnecessários entre listas diferentes.

Entretanto, essa abordagem se mostrou limitada quando o número de listas passou a ser definido dinamicamente. Como regiões críticas nomeadas exigem nomes fixos em tempo de compilação, não são adequadas para cenários com estruturas criadas em tempo de execução. Nesse contexto, a segunda versão utilizou *locks* explícitos com *omp_lock_t*, possibilitando uma sincronização flexível e segura para qualquer quantidade de listas. Essa técnica garantiu exclusão mútua adequada e escalabilidade da solução, evidenciando a necessidade de se escolher o mecanismo de sincronização com base na natureza e complexidade da aplicação paralela.