Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
**Tarefa 9 - Regiões críticas nomeadas e Locks explícitos**
**Aluno:** Daniel Bruno Trindade da Silva

# 1 Introdução

O presente trabalho tem como objetivo explorar o uso de paralelismo com OpenMP para manipulação segura de estruturas de dados compartilhadas, mais especificamente, listas encadeadas em ambientes concorrentes. Foram desenvolvidos dois programas que criam tarefas paralelas para realizar inserções em listas encadeadas, garantindo a integridade dos dados e evitando condições de corrida.

A seguir, serão apresentados os dois programas, acompanhados de uma análise das estratégias de sincronização utilizadas e das razões pelas quais o uso de locks se torna essencial na versão generalizada.

# 2 Metodologia

O primeiro programa lida com duas listas encadeadas, cada uma associada a uma região crítica nomeada. Isso permite que múltiplas inserções ocorram simultaneamente, desde que sejam em listas diferentes, promovendo paralelismo eficiente. Já o segundo programa generaliza o cenário anterior para um número arbitrário de listas definido pelo usuário.

No primeiro programa, foi criada uma região paralela com a diretiva `#pragma omp parallel`. Dentro dela, utilizou-se `#pragma omp single` para garantir que apenas uma thread gere as tarefas, evitando a criação duplicada de tarefas. Cada tarefa, por sua vez, executa uma inserção em uma das listas, utilizando regiões críticas nomeadas com `#pragma omp critical(nome)` para evitar condições de corrida sem bloquear o acesso à outra lista. A região paralela do código ficou assim:

```
#pragma omp parallel
{
    #pragma omp single
    {
        for (int i = 0; i < N; i++) {
            #pragma omp task
            {
                int valor = rand_r(&seed) % 100;
                int escolha = rand_r(&seed) % 2;

                if (escolha == 0) {
                    #pragma omp critical(lista1)
                    adicionar_no(&lista1, valor);
                } else {
                    #pragma omp critical(lista2)
                    adicionar_no(&lista2, valor);
                }
            }
        }
    }
}
```

Já no segundo programa, o número de listas é definido dinamicamente pelo usuário, o que inviabiliza o uso de regiões críticas nomeadas, pois o nome da região precisa ser conhecido em tempo de compilação. Para resolver esse problema, foram utilizados *locks explícitos* por meio da estrutura `omp_lock_t`. Esses locks permitem controlar manualmente o momento de criação, aquisição (com `omp_set_lock`) e liberação (com `omp_unset_lock`) da exclusão mútua, permitindo o acesso concorrente seguro às listas.

A região paralela do segundo código ficou assim:

```
Node** listas = (Node**)malloc(K * sizeof(Node*));
omp_lock_t* locks = (omp_lock_t*)malloc(K * sizeof(omp_lock_t));
for (int i = 0; i < K; i++) {
    listas[i] = NULL;
    omp_init_lock(&locks[i]);
}

int seed = time(NULL);

#pragma omp parallel
{
    #pragma omp single
    {
        for (int i = 0; i < N; i++) {
            #pragma omp task
            {
                int valor = rand_r(&seed) % 100;
                int indice = rand_r(&seed) % K;

                omp_set_lock(&locks[indice]);
                adicionar_no(&listas[indice], valor);
                omp_unset_lock(&locks[indice]);
            }
        }
    }
}

for (int i = 0; i < K; i++) {
      printf("Lista %d: ", i);
      imprimir_lista(listas[i]);
      liberar_lista(listas[i]);
      omp_destroy_lock(&locks[i]);
}
```

# 3 Resultados

O enunciado da tarefa nos questiona por que as regiões críticas nomeadas não são suficientes no segundo programa e por que o uso de *locks* se torna necessário. Durante o desenvolvimento da aplicação,

foi possível perceber que, no primeiro programa — com apenas duas listas — é viável utilizar regiões críticas nomeadas (`#pragma omp critical(lista1)` e `#pragma omp critical(lista2)`), já que o número de listas é fixo e conhecido em tempo de compilação. Cada região crítica é identificada por um nome estático, permitindo que múltiplas inserções ocorram em paralelo, desde que sejam em listas diferentes.

Contudo, no segundo programa, o número de listas ($K$) é definido dinamicamente em tempo de execução. Dessa forma, não é possível criar $K$ regiões críticas nomeadas, pois os nomes das regiões precisam ser conhecidos em tempo de compilação. A alternativa de usar uma única região crítica compartilhada impediria o paralelismo entre inserções em listas distintas, prejudicando o desempenho da aplicação.

Nesse contexto, o uso de *locks* explícitos (`omp_lock_t`) torna-se necessário. Com eles, é possível criar dinamicamente um vetor de $K$ locks — um para cada lista — e proteger individualmente cada inserção com o lock correspondente. Isso garante a integridade das estruturas sem comprometer o paralelismo, permitindo que inserções em listas diferentes ocorram simultaneamente de forma segura.

# 4 Conclusão

A implementação dos programas propostos permitiu compreender, na prática, os desafios relacionados à concorrência e à manipulação segura de estruturas de dados compartilhadas em ambientes paralelos. No primeiro programa, o uso de regiões críticas nomeadas mostrou-se eficiente para sincronizar o acesso a duas listas encadeadas, permitindo um bom nível de paralelismo sem comprometer a integridade dos dados.

Já no segundo programa, a necessidade de lidar com um número dinâmico de listas revelou a limitação das regiões críticas nomeadas, que não podem ser definidas em tempo de execução. Nesse contexto, a utilização de *locks* explícitos foi essencial para garantir exclusão mútua de forma flexível e escalável, mantendo o paralelismo entre as inserções em listas distintas.

Conclui-se, portanto, que o uso adequado de mecanismos de sincronização é fundamental para garantir a integridade dos dados em aplicações paralelas. Enquanto regiões críticas nomeadas funcionam bem em cenários com estruturas fixas e conhecidas em tempo de compilação, como no caso das duas listas, o uso de *locks* explícitos se mostra indispensável em contextos mais flexíveis e dinâmicos. Essa abordagem permite maior controle sobre o acesso concorrente e favorece a escalabilidade da aplicação, possibilitando a execução segura e eficiente de tarefas em paralelo.