# 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**.

**Duas_listas.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Estrutura do Nó e da Lista Encadeada
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct LinkedList {
    Node* head;
} LinkedList;

// Função para inserir um novo nó no início da lista
void insert(LinkedList* list, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        fprintf(stderr, "Falha na alocação de memória\n");
        return;
    }
    newNode->data = value;
    newNode->next = list->head;
    list->head = newNode;
}

// Função para liberar a memória de uma lista
void free_list(LinkedList* list) {
    Node* current = list->head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    list->head = NULL;
}

int main() {
    const int N_INSERTIONS = 100000;

    // Inicializa as duas listas com a cabeça apontando para NULL
    LinkedList listA = { NULL };
    LinkedList listB = { NULL };

    // A diretiva 'parallel for' distribui as iterações entre as threads
    #pragma omp parallel for
    for (int i = 0; i < N_INSERTIONS; ++i) {
        // Cada thread precisa de sua própria seed para rand_r ser thread-safe
        unsigned int seed = (unsigned int)time(NULL) ^ omp_get_thread_num();
```

```c

        int value_to_insert = rand_r(&seed) % 1001; // Valor aleatório de 0 a 1000
        int list_choice = rand_r(&seed) % 2;       // Escolha aleatória: 0 ou 1

        if (list_choice == 0) {
            // Região Crítica Nomeada para a lista A.
            #pragma omp critical (lock_A)
            {
                insert(&listA, value_to_insert);
            }
        } else {
            // Região Crítica Nomeada para a lista B.
            #pragma omp critical (lock_B)
            {
                insert(&listB, value_to_insert);
            }
        }
    }

    printf("Inserções concluídas.\n");

    // Contagem para verificação
    long countA = 0;
    for (Node* current = listA.head; current != NULL; current = current->next) countA++;
    long countB = 0;
    for (Node* current = listB.head; current != NULL; current = current->next) countB++;

    printf("Elementos na Lista A: %ld\n", countA);
    printf("Elementos na Lista B: %ld\n", countB);
    printf("Total de inserções: %ld (esperado: %d)\n", countA + countB, N_INSERTIONS);

    // Libera a memória alocada
    free_list(&listA);
    free_list(&listB);

    return 0;
}
```

**N_listas.c**

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Estrutura do Nó e da Lista Encadeada
typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct LinkedList {
    Node* head;
} LinkedList;

// Implementação da função para inserir um novo nó no início da lista
void insert(LinkedList* list, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        // Em um programa real, um tratamento de erro mais robusto seria necessário
        return;
    }
    newNode->data = value;
    newNode->next = list->head;
    list->head = newNode;
}

// Implementação da função para liberar a memória de uma lista
void free_list(LinkedList* list) {
    Node* current = list->head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    list->head = NULL;
}

int main() {
    const int N_INSERTIONS = 100000;
    int M_LISTS;

    printf("Digite o número de listas: ");
    scanf("%d", &M_LISTS);

    if (M_LISTS <= 0) {
        printf("Número de listas deve ser positivo.\n");
        return 1;
    }
```

```
50
51      // 1. Aloca dinamicamente um array de listas
52      LinkedList* lists = (LinkedList*)malloc(M_LISTS * sizeof(LinkedList));
53      // 2. Aloca dinamicamente um array de locks
54      omp_lock_t* locks = (omp_lock_t*)malloc(M_LISTS * sizeof(omp_lock_t));
55
56      // Verificação de robustez da alocação
57      if (lists == NULL || locks == NULL) {
58          fprintf(stderr, "Falha ao alocar memória para listas ou locks.\n");
59          free(lists);
60          free(locks);
61          return 1;
62      }
63
64      // Inicializa cada lista e seu respectivo lock
65      for (int i = 0; i < M_LISTS; ++i) {
66          lists[i].head = NULL;
67          omp_init_lock(&locks[i]); // Inicializa o lock
68      }
69
70      #pragma omp parallel for
71      for (int i = 0; i < N_INSERTIONS; ++i) {
72          unsigned int seed = (unsigned int)time(NULL) ^ omp_get_thread_num();
73
74          int value_to_insert = rand_r(&seed) % 1001;
75          int list_index = rand_r(&seed) % M_LISTS; // Escolhe uma das M listas
76
77          // 3. Adquire o lock explícito para a lista escolhida
78          omp_set_lock(&locks[list_index]);
79
80          // --- Início da Região Crítica ---
81          insert(&lists[list_index], value_to_insert);
82          // --- Fim da Região Crítica ---
83
84          // 4. Libera o lock
85          omp_unset_lock(&locks[list_index]);
86      }
87
88      printf("Inserções concluídas.\n");
89
90      long long total_count = 0;
91      for (int i = 0; i < M_LISTS; ++i) {
92          long count = 0;
93          for (Node* current = lists[i].head; current != NULL; current = current-
    >next) count++;
94          total_count += count;
95      }
96      printf("Total de inserções: %lld (esperado: %d)\n", total_count,
    N_INSERTIONS);
97
98      // 5. Destrói os locks e libera toda a memória
99      for (int i = 0; i < M_LISTS; ++i) {
100         omp_destroy_lock(&locks[i]);
```

```
        free_list(&lists[i]);
    }
    free(lists);
    free(locks);

    return 0;
}
```