UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 09 - Análise de Estratégias de Sincronização para Listas Encadeadas

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 12 de setembro de 2025

Lista de Figuras

Sumário

1 Introdução

A manipulação de estruturas de dados dinâmicas, como listas encadeadas, em ambientes de execução paralela apresenta desafios significativos de sincronização. A necessidade de garantir a integridade dos dados durante inserções e remoções concorrentes exige o uso de mecanismos que evitem condições de corrida, um problema clássico da ciência da computação. O acesso simultâneo ao ponteiro que indica o início da lista (‘head‘) é uma seção crítica que, se não for devidamente protegida, pode levar à corrupção da estrutura e perda de dados.

O objetivo deste projeto é analisar o impacto de diferentes estratégias de implementação paralela em OpenMP para o problema de múltiplas inserções concorrentes em listas encadeadas. Foram desenvolvidas duas versões de um algoritmo para explorar cenários distintos de gerenciamento de recursos compartilhados:

1. **Cenário Estático:** Uma implementação com um número fixo de duas listas, utilizando a diretiva de *regiões críticas nomeadas* do OpenMP.
2. **Cenário Dinâmico:** Uma implementação generalizada para um número ‘M‘ de listas (definido em tempo de execução), que requer o uso de *locks explícitos* para garantir a sincronização.

Este relatório busca, através da análise das duas implementações, ilustrar a importância de selecionar o mecanismo de sincronização adequado à natureza do problema, destacando as diferenças de aplicabilidade, flexibilidade e complexidade entre abordagens estáticas e dinâmicas.

2 Metodologia do Experimento

O experimento consiste na implementação e análise teórica de duas versões de um programa que realiza inserções paralelas em listas encadeadas.

2.1 Estrutura de Dados e Problema Proposto

A base do experimento é a estrutura de dados de lista encadeada simples, implementada em Linguagem C. O problema central é a execução de `N` inserções em paralelo, distribuídas por um conjunto de `M` listas. Para cada inserção, a thread responsável escolhe aleatoriamente uma das `M` listas e insere um novo nó. A função `rand_r()`, por ser reentrante, foi escolhida para a geração de números aleatórios de forma segura em ambiente multithread.

2.2 Implementação das Duas Versões Paralelas

As duas versões foram desenvolvidas em C com a biblioteca OpenMP para explorar as diferentes estratégias de sincronização.

2.2.1 Versão 1: Regiões Críticas Nomeadas (Cenário Estático)

- **Descrição:** Esta versão aborda o problema para um número fixo de duas listas.
- **Estratégia de Sincronização:** Utiliza a diretiva `#pragma omp critical (name)`. Ao fornecer nomes distintos para as regiões críticas que protegem cada lista (`lock_A` e `lock_B`), permite-se que inserções em listas diferentes ocorram de forma concorrente, já que os locks são independentes.

```
int main() {
    const int N_INSERTIONS = 100000;
    // Inicializa as duas listas com HEAD apontando para NULL
    LinkedList listA = { NULL };
    LinkedList listB = { NULL };
  #pragma omp parallel for
    for (int i = 0; i < N_INSERTIONS; ++i) {
        // Cada thread precisa de sua propria seed para rand_r ser thread-
            safe
        unsigned int seed = (unsigned int)time(NULL) ^ omp_get_thread_num()
            ;
        int value_to_insert = rand_r(&seed) % 1001; // Valor aleatorio de 0
             a 1000
        int list_choice = rand_r(&seed) % 2;        // Escolha aleatoria: 0
            ou 1

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

Listing 2.1 – Regioes Criticas Nomeadas

2.2.2 Versão 2: Locks Explícitos (Cenário Dinâmico)

- **Descrição:** Esta versão generaliza o problema para um número `M` de listas, definido pelo usuário em tempo de execução.
- **Estratégia de Sincronização:** Utiliza um array de locks do tipo `omp_lock_t`, alocado dinamicamente com tamanho `M`. Cada lista no índice `i` é protegida pelo lock no mesmo índice do array. Esta abordagem permite o travamento granular e dinâmico, selecionando o lock específico em tempo de execução.

```
int main() {
    .
        .
            .

    #pragma omp parallel for
        for (int i = 0; i < N_INSERTIONS; ++i) {
            unsigned int seed = (unsigned int)time(NULL) ^
                omp_get_thread_num();

            int value_to_insert = rand_r(&seed) % 1001;
            int list_index = rand_r(&seed) % M_LISTS; // Escolhe uma das M
                listas

            // 3. Adquire o lock explicito para a lista escolhida
            omp_set_lock(&locks[list_index]);

            // --- In cio da Regiao Critica ---
            insert(&lists[list_index], value_to_insert);
            // --- Fim da Regiao Critica ---

            // 4. Libera o lock
            omp_unset_lock(&locks[list_index]);
        }
    }
}
```

Listing 2.2 – Locks Explicitos

3 Análise e Discussão

A análise comparativa das duas implementações revela insights fundamentais sobre a aplicabilidade e os trade-offs dos mecanismos de sincronização em OpenMP.

3.1 Flexibilidade e Aplicabilidade

A principal diferença entre as abordagens reside na flexibilidade.

- **Regiões Críticas Nomeadas:** Demonstram ser uma abstração de alto nível e de fácil uso, porém rígida. Sua aplicabilidade se restringe a cenários onde o número de recursos a serem protegidos é fixo e conhecido em tempo de compilação, pois o nome do lock deve ser um identificador estático.
- **Locks Explícitos:** São a solução necessária para cenários dinâmicos. Por serem objetos (variáveis) que podem ser armazenados em arrays ou outras estruturas, eles permitem que o programador crie e gerencie um número arbitrário de locks em tempo de execução. Isso torna a solução escalável e adaptável a um número variável de recursos.

3.2 Impacto na Complexidade do Código

A flexibilidade dos locks explícitos introduz uma maior complexidade no gerenciamento. Enquanto a região crítica nomeada é uma única diretiva, os locks explícitos exigem um ciclo de vida manual: inicialização (`omp_init_lock`), aquisição (`omp_set_lock`), liberação (`omp_unset_lock`) e destruição (`omp_destroy_lock`).

3.3 Análise Teórica de Desempenho

Ambas as implementações adotam uma estratégia de **locking de alta granularidade (fine-grained)**, onde cada lista possui seu próprio mecanismo de exclusão mútua. Esta é a abordagem de melhor desempenho para este problema, pois minimiza a contenção. Uma thread tentando inserir na `lista[i]` não interfere em outra que esteja inserindo na `lista[j]` (para `i != j`).

Uma abordagem alternativa e ineficiente, de **baixa granularidade (coarse-grained)**, seria usar um único lock para todas as listas (ex: `#pragma omp critical` sem nome). Isso serializaria todas as inserções, transformando o laço paralelo em um gargalo

sequencial e eliminando os benefícios da paralelização, um cenário similar ao observado com o uso de `rand()` no relatório modelo da Tarefa 08.

3.4 Conclusão

A análise das duas implementações conclui que a escolha do mecanismo de sincronização não é apenas uma questão de preferência, mas uma decisão de design ditada pela natureza do problema.

1. **A escolha de mecanismos adequados é fundamental:** O experimento demonstrou que, para problemas com um número dinâmico de recursos compartilhados, abstrações estáticas como regiões críticas nomeadas são insuficientes, tornando o uso de locks explícitos obrigatório.
2. **A granularidade do locking dita o desempenho:** A implementação de uma estratégia de alta granularidade foi crucial em ambas as versões para permitir a máxima concorrência possível, evitando a serialização desnecessária das tarefas.

Em suma, a tarefa demonstrou de forma prática que, embora as abstrações de alto nível do OpenMP sejam poderosas, a compreensão de mecanismos de mais baixo nível, como os locks explícitos, é crucial 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;
    }
```

```c

    // 1. Aloca dinamicamente um array de listas
    LinkedList* lists = (LinkedList*)malloc(M_LISTS * sizeof(LinkedList));
    // 2. Aloca dinamicamente um array de locks
    omp_lock_t* locks = (omp_lock_t*)malloc(M_LISTS * sizeof(omp_lock_t));

    // Verificação de robustez da alocação
    if (lists == NULL || locks == NULL) {
        fprintf(stderr, "Falha ao alocar memória para listas ou locks.\n");
        free(lists);
        free(locks);
        return 1;
    }

    // Inicializa cada lista e seu respectivo lock
    for (int i = 0; i < M_LISTS; ++i) {
        lists[i].head = NULL;
        omp_init_lock(&locks[i]); // Inicializa o lock
    }

    #pragma omp parallel for
    for (int i = 0; i < N_INSERTIONS; ++i) {
        unsigned int seed = (unsigned int)time(NULL) ^ omp_get_thread_num();

        int value_to_insert = rand_r(&seed) % 1001;
        int list_index = rand_r(&seed) % M_LISTS; // Escolhe uma das M listas

        // 3. Adquire o lock explícito para a lista escolhida
        omp_set_lock(&locks[list_index]);

        // --- Início da Região Crítica ---
        insert(&lists[list_index], value_to_insert);
        // --- Fim da Região Crítica ---

        // 4. Libera o lock
        omp_unset_lock(&locks[list_index]);
    }

    printf("Inserções concluídas.\n");

    long long total_count = 0;
    for (int i = 0; i < M_LISTS; ++i) {
        long count = 0;
        for (Node* current = lists[i].head; current != NULL; current = current->next) count++;
        total_count += count;
    }
    printf("Total de inserções: %lld (esperado: %d)\n", total_count, N_INSERTIONS);

    // 5. Destrói os locks e libera toda a memória
    for (int i = 0; i < M_LISTS; ++i) {
        omp_destroy_lock(&locks[i]);
```

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

    return 0;
}
```