Tarefa 9: Regiões Críticas Nomeadas vs Locks Explícitos

📝 Objetivo da Tarefa

Desenvolver um programa OpenMP que demonstra a diferença entre **regiões críticas nomeadas** e **locks explícitos** para sincronização de estruturas de dados paralelas, especificamente listas encadeadas.

> **Requisitos principais:**
>
> - Usar OpenMP Tasks para realizar N inserções
> - Implementar duas listas encadeadas com regiões críticas nomeadas
> - Escolha aleatória da lista para inserção
> - Generalizar para N listas usando locks explícitos
> - Explicar por que regiões críticas nomeadas têm limitações

🔐 Tipos de Locks em OpenMP

OpenMP oferece diferentes mecanismos de sincronização, cada um com características específicas para diferentes cenários de uso.

1. Regiões Críticas Simples

A forma mais básica de sincronização, onde apenas uma thread pode executar o bloco por vez:

```
#pragma omp critical
{
    // Apenas uma thread executa este bloco por vez
    shared_variable++;
}
```

> ⚠ **Limitação:** Todas as regiões críticas simples compartilham o mesmo lock global, criando gargalos desnecessários.

2. Regiões Críticas Nomeadas

Permitem múltiplas seções críticas independentes com nomes únicos:

```
#pragma omp critical(nome_do_lock)
{
    // Protegido pelo lock "nome_do_lock"
    lista1_operations();
}

#pragma omp critical(outro_nome)
{
    // Protegido pelo lock "outro_nome" - independente do anterior
    lista2_operations();
}
```

> ✔ **Vantagem:** Diferentes nomes permitem paralelismo entre seções críticas distintas.

3. Locks Explícitos (omp_lock_t)

Oferecem controle total sobre a sincronização com gerenciamento manual:

```
// Declaração e inicialização
omp_lock_t my_lock;
omp_init_lock(&my_lock);

// Uso do lock
omp_set_lock(&my_lock);     // Bloqueia até adquirir
{
    // Seção crítica
    critical_operations();
}
omp_unset_lock(&my_lock);   // Libera o lock

// Limpeza
omp_destroy_lock(&my_lock);
```

Comparação Detalhada dos Tipos de Locks

Tipo de Lock	Sintaxe	Flexibilidade	Performance	Uso Recomendado
Critical Simples	`#pragma omp critical`	✕ Baixa	✕ Baixa (lock global)	Prototipagem rápida
Critical Nomeadas	`#pragma omp critical(nome)`	⚠ Média	✔ Boa	Pequeno número fixo de recursos
Locks Explícitos	`omp_set_lock(&lock)`	✔ Alta	✔ Alta	Estruturas dinâmicas, arrays

Ciclo de Vida dos Locks Explícitos

```
// 1. DECLARAÇÃO
omp_lock_t lock;

// 2. INICIALIZAÇÃO (obrigatória)
omp_init_lock(&lock);

// 3. USO REPETIDO
for (int i = 0; i < n; i++) {
    omp_set_lock(&lock);        // Adquire
    {
        // Operações thread-safe
    }
    omp_unset_lock(&lock);      // Libera
}

// 4. DESTRUIÇÃO (obrigatória para evitar vazamentos)
omp_destroy_lock(&lock);
```

⚠ **Importante:** Sempre emparelhar `omp_init_lock()` com `omp_destroy_lock()` para evitar vazamentos de recursos.

📖 Comandos de Lock Explícito

🔷 `omp_lock_t lock;` - Declaração do Lock

- **Tipo opaco do OpenMP** para representar um lock
- **Estado indefinido** após declaração - não pode ser usado
- **Deve ser inicializado** antes do primeiro uso

🔷 `omp_init_lock(&lock);` - Inicialização do Lock

- **Obrigatório** antes de qualquer uso do lock
- **Aloca recursos internos** do OpenMP (mutexes, semáforos)
- **Lock fica desbloqueado** e pronto para uso
- **Não é thread-safe** - chame em região sequencial

🔷 `omp_set_lock(&lock);` - Aquisição do Lock

- **Comportamento bloqueante** - espera até conseguir o lock
- **Exclusão mútua** - apenas uma thread por vez
- **Thread torna-se dona** do lock após aquisição
- **Operação atômica** sem timeout

🔷 `omp_unset_lock(&lock);` - Liberação do Lock

- **Deve ser chamado pela mesma thread** que fez `omp_set_lock()`
- **Libera o lock** para outras threads esperando
- **Thread perde ownership** do lock
- **Emparelhamento obrigatório** com `omp_set_lock()`

🔷 `omp_destroy_lock(&lock);` - Destruição do Lock

- **Libera recursos do sistema** alocados na inicialização
- **Lock deve estar desbloqueado** antes da destruição
- **Previne vazamentos de memória** dos recursos OpenMP
- **Emparelhar sempre** com `omp_init_lock()`

> 🎯 **Sequência Obrigatória:**
>
> **DECLARAÇÃO → INICIALIZAÇÃO → [AQUISIÇÃO → LIBERAÇÃO]* → DESTRUIÇÃO**

✔ **Vantagens dos Locks Explícitos:**

- **Escalabilidade Total:** Funciona para qualquer número N de listas
- **Criação Dinâmica:** Locks criados em runtime conforme necessário
- **Flexibilidade Máxima:** Cada estrutura pode ter seu próprio lock
- **Performance Superior:** Paralelismo otimizado entre todas as listas
- **Encapsulamento:** Lock faz parte da estrutura de dados

🎯 Por que Cada Tipo foi Escolhido

🔍 **Decisão de Design do Programa:**

Primeira Parte - Regiões Críticas Nomeadas:

- Demonstrar paralelismo entre **exatamente 2 listas**
- Mostrar que `lista1` e `lista2` podem operar simultaneamente
- Evidenciar a **limitação** quando precisamos de mais listas

Segunda Parte - Locks Explícitos:

- Resolver o problema de **N listas dinâmicas**
- Demonstrar **escalabilidade real** (testamos com 20 listas)
- Mostrar **performance superior** (74.2% mais throughput)

📊 Análise Comparativa

Regiões Críticas Nomeadas

- **Vantagens:**
  - Sintaxe simples
  - Paralelismo entre diferentes nomes
  - Gerenciamento automático
- **Limitações:**

Locks Explícitos

- **Vantagens:**
  - Criação dinâmica em runtime
  - Totalmente escalável
  - Controle fino de sincronização
  - Flexibilidade máxima

- Nomes devem ser conhecidos em tempo de compilação
- Não escalável para N dinâmico
- Impossível criar nomes em runtime

- **Desvantagens:**
  - Gerenciamento manual necessário
  - Sintaxe mais verbosa
  - Possibilidade de deadlocks

🔍 Passo a Passo: Execução das Duas Listas com Regiões Críticas Nomeadas

🎯 Entrada do Usuário:

- **Inserções:** 5000
- **Threads:** 4

Fluxo de Execução - Primeira Parte:

1. 🏗️ **Inicialização:**
   - Criar duas listas globais: `global_list1` e `global_list2`
   - Inicializar cada lista com: head = NULL, count = 0 → `init_simple_list()`
   - Marcar tempo inicial: `start_time = omp_get_wtime()`
2. 📡 **Região Paralela:**
   - Criar 4 threads com `#pragma omp parallel num_threads(4)`
   - Thread principal (single) cria 5000 tasks
   - Outras 3 threads aguardam tasks na fila de trabalho
3. ⚡ **Execução das Tasks:**
   - Cada task gera seed único: `time() + omp_get_thread_num() + task_id`
   - Escolhe aleatoriamente: lista 1 (choice=0) ou lista 2 (choice=1) → `rand_r()`
   - Gera valor aleatório entre 0-999 para inserir → `rand_r()`
4. 🔒 **Inserção com Regiões Críticas:**
   - **Lista 1:** `#pragma omp critical(lista1)` → `insert_list1_critical()`
   - **Lista 2:** `#pragma omp critical(lista2)` → `insert_list2_critical()`
   - **Criação de nós:** Aloca memória para novos nós → `create_node()`
   - **Proteção:** Apenas uma thread por lista por vez
5. ⏱️ **Finalização:**

- Aguardar todas as 5000 tasks terminarem (taskwait implícito)
- Sincronizar todas as 4 threads
- Medir tempo final e calcular duração → `omp_get_wtime()`

6. 🧮 **Resultados:**
   - Imprimir conteúdo das listas → `print_simple_list()`
   - **Lista 1:** 2542 elementos (50.84%)
   - **Lista 2:** 2458 elementos (49.16%)
   - **Tempo:** 2.9895 segundos
   - **Throughput:** 1672 operações/segundo
7. 🧹 **Limpeza:**
   - Liberar memória de todos os nós → `destroy_simple_list()`

📋 Passo a Passo: Execução de N Listas com Locks Explícitos

> 🎯 **Entrada do Usuário:**
>
> - **Listas:** 20
> - **Inserções:** 5000 (mesmo valor anterior)
> - **Threads:** 4 (mesmo valor anterior)

Fluxo de Execução - Segunda Parte:

1. 🏗️ **Alocação Dinâmica:**
   - Alocar array de 20 estruturas `LockedList` → `malloc()`
   - Cada estrutura contém: head, count, id, **omp_lock_t lock**
   - Inicializar cada lista com locks → `init_locked_list()` → `omp_init_lock()`
2. 🚀 **Região Paralela:**
   - Mesmo padrão: 4 threads, 5000 tasks
   - Thread principal cria tasks, outras executam
   - Cada task recebe cópia do array de listas
3. ⚡ **Execução das Tasks:**
   - Seed único para cada task → `time() + omp_get_thread_num() + task_id`
   - Escolha aleatória entre as **20 listas** → `rand_r()`
   - Acesso dinâmico: `lists[choice]`
4. 🔒 **Inserção com Locks Explícitos:**
   - **Criação de nós:** Aloca memória → `create_node()`

- **Aquisição:** `omp_set_lock(&list->lock)`
- **Inserção:** Modificar head, incrementar count → `insert_locked_list()`
- **Liberação:** `omp_unset_lock(&list->lock)`
- **Vantagem:** Cada lista tem lock independente

5. ⏱️ **Finalização:**
   - Aguardar 5000 tasks (mesmo processo)
   - Sincronização automática das threads
   - Cálculo de tempo de execução → `omp_get_wtime()`
6. 📊 **Resultados:**
   - Imprimir conteúdo de todas as listas → `print_locked_list()`
   - **20 Listas:** 212-288 elementos cada (distribuição equilibrada)
   - **Tempo:** 1.7177 segundos (**42.5% mais rápido!**)
   - **Throughput:** 2911 operações/segundo
   - **Escalabilidade:** 10x mais listas com melhor performance
7. 🧹 **Limpeza:**
   - Destruir todos os 20 locks → `destroy_locked_list()` → `omp_destroy_lock()`
   - Liberar memória de todos os nós das listas → `free()`
   - Liberar array principal → `free(lists)`

📈 Análise de Resultados

Exemplo de Saída do Programa - Teste de Alta Carga

```
TAREFA 9: Regiões Críticas Nomeadas vs Locks Explícitos
============================================================
Digite o número de inserções: Digite o número de threads:

=== DUAS LISTAS COM REGIÕES CRÍTICAS NOMEADAS ===
Inserções: 5000 | Threads: 4

Resultados após 5000 inserções:
Lista 1 (2542 elementos): 197 393 197 197 375 294 294 147 300 300 48 48 801... Lista 2 (2458 elementos): 827 112 56 614 506 310 437 455 455 437 466 391 686... Tempo total: 2.9895 segundos Total de elementos: 5000 Digite o número de listas para a versão generalizada:

=== 20 LISTAS COM LOCKS EXPLÍCITOS ===
Inserções: 5000 | Threads: 4

Resultados após 5000 inserções em 20 listas:
```

```
Lista 1 (228 elementos): 780 813 813 446 200 200 804 804 860 981
819... Lista 2 (242 elementos): 112 260 758 313 850 850 276 276 424
601 328... Lista 3 (268 elementos): 801 801 585 585 738 738 216 691
697 697 911... Lista 4 (277 elementos): 130 130 614 827 614 614 981
635 518 518 847... Lista 5 (288 elementos): 147 147 694 362 621 626
626 735 735 735 668... ... (15 listas adicionais) ... Lista 20 (212
elementos): 488 791 304 304 325 388 959 959 959 844 436... Tempo
total: 1.7177 segundos Total de elementos: 5000
```

Observações dos Resultados

✔ **Distribuição Aleatória em Alta Escala:** Com 5000 inserções, a distribuição foi praticamente equilibrada: Lista 1 recebeu 2542 elementos (50.84%) e Lista 2 recebeu 2458 elementos (49.16%). Entre 20 listas, a distribuição variou de 212 a 288 elementos por lista, demonstrando aleatoriedade efetiva.

✔ **Integridade dos Dados em Alta Carga:** O total de elementos inseridos (5000) corresponde exatamente à soma dos elementos em todas as listas em ambas as execuções, comprovando que não houve condições de corrida mesmo com alta concorrência.

✔ **Performance Superior com Locks Explícitos:** Para 5000 inserções, os locks explícitos (1.7177s) foram **42.5% mais rápidos** que regiões críticas nomeadas (2.9895s), demonstrando a superioridade da abordagem para cenários de alta carga.

✔ **Paralelismo Efetivo:** O tempo de execução é baixo, indicando que as inserções estão acontecendo em paralelo sem bloqueios desnecessários.

Análise de Performance

Abordagem	Inserções	Listas	Tempo (s)	Throughput (ops/s)	Escalabilidade

Regiões Críticas Nomeadas	5000	2	2.9895	1672	✕ Limitado a nomes fixos
Locks Explícitos	5000	20	1.7177	2911	✓ Ilimitado

Conclusões

Por que Regiões Críticas Nomeadas não são suficientes para N listas?

1. **Limitação de Compilação:** Os nomes das regiões críticas devem ser literais conhecidos em tempo de compilação
2. **Impossibilidade de Generalização:** Não podemos criar nomes dinamicamente como "lista1", "lista2", ..., "listaN"
3. **Código Estático:** Para 100 listas, precisaríamos escrever 100 seções críticas diferentes no código

Vantagens dos Locks Explícitos:

1. **Criação Dinâmica:** Locks podem ser criados em arrays ou estruturas em runtime
2. **Escalabilidade Total:** Funciona para qualquer número N de listas
3. **Flexibilidade:** Permite implementações mais complexas de sincronização
4. **Eficiência:** Cada lista tem seu próprio lock independente

Conceitos Aprendidos

- **OpenMP Tasks:** Criação dinâmica de unidades de trabalho paralelo
- **Sincronização Granular:** Diferentes abordagens para proteger estruturas de dados
- **Thread Safety:** Uso de rand_r() para geração de números aleatórios thread-safe
- **Gestão de Memória:** Alocação e liberação adequada de estruturas dinâmicas
- **Análise de Performance:** Medição de tempo e avaliação de paralelismo

> **Lição Principal:** A escolha entre diferentes mecanismos de sincronização deve considerar não apenas a funcionalidade, mas também a escalabilidade e flexibilidade da solução. Regiões críticas nomeadas são excelentes para casos simples e conhecidos, enquanto locks explícitos oferecem máxima flexibilidade para soluções escaláveis.

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

// ============================================================================
// ESTRUTURAS DE DADOS
// ============================================================================

// Estrutura do nó da lista encadeada
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Estrutura para lista simples (usada com regiões críticas nomeadas)
typedef struct {
    Node* head;
    int count;
    int id;
} SimpleList;

// Estrutura para lista com lock explícito (usada com múltiplas listas)
typedef struct {
    Node* head;
    int count;
    int id;
    omp_lock_t lock; // Cada lista tem seu próprio lock independente
} LockedList;

// ============================================================================
// FUNÇÕES AUXILIARES
// ============================================================================

// Função para criar um novo nó
Node* create_node(int data) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    if (new_node == NULL) {
        fprintf(stderr, "Erro ao alocar memória para novo nó\n");
        exit(1);
    }
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Inicializar lista simples
void init_simple_list(SimpleList* list, int id) {
    list->head = NULL;
    list->count = 0;
    list->id = id;
}

// Inicializar lista com lock
void init_locked_list(LockedList* list, int id) {
    list->head = NULL;
    list->count = 0;
    list->id = id;
    omp_init_lock(&list->lock); // Inicializa o lock antes do uso
}

// Destruir lista simples
void destroy_simple_list(SimpleList* list) {
    Node* current = list->head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
}

// Destruir lista com lock
void destroy_locked_list(LockedList* list) {
```

```
    Node* current = list->head;
    while (current != NULL) {
        Node* temp = current;
        current = current->next;
        free(temp);
    }
    omp_destroy_lock(&list->lock); // Libera recursos do lock
}

// Imprimir lista simples
void print_simple_list(SimpleList* list) {
    printf("Lista %d (%d elementos): ", list->id, list->count);
    Node* current = list->head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// Imprimir lista com lock
void print_locked_list(LockedList* list) {
    printf("Lista %d (%d elementos): ", list->id, list->count);
    Node* current = list->head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

// ============================================================================
// IMPLEMENTAÇÃO COM DUAS LISTAS USANDO REGIÕES CRÍTICAS NOMEADAS
// ============================================================================

// Variáveis globais para as duas listas (necessário para regiões críticas nomeadas)
SimpleList global_list1, global_list2;

// Função para inserir na lista 1 usando região crítica nomeada
void insert_list1_critical(int data) {
    Node* new_node = create_node(data);

    #pragma omp critical(lista1) // Lock específico para lista1 - permite paralelismo
com lista2
    {
        new_node->next = global_list1.head;
        global_list1.head = new_node;
        global_list1.count++;

        // Simula processamento
        usleep(1000);
    }
}

// Função para inserir na lista 2 usando região crítica nomeada
void insert_list2_critical(int data) {
    Node* new_node = create_node(data);

    #pragma omp critical(lista2) // Lock específico para lista2 - independente de
lista1
    {
        new_node->next = global_list2.head;
        global_list2.head = new_node;
        global_list2.count++;

        // Simula processamento
        usleep(1000);
    }
}

// Programa principal com duas listas usando regiões críticas nomeadas
void program_two_lists_named_critical(int num_insertions, int num_threads) {
    printf("\n=== DUAS LISTAS COM REGIÕES CRÍTICAS NOMEADAS ===\n");
    printf("Inserções: %d | Threads: %d\n\n", num_insertions, num_threads);

    // Inicializar as duas listas globais
    init_simple_list(&global_list1, 1);
    init_simple_list(&global_list2, 2);
```

```
    double start_time = omp_get_wtime(); // Marca tempo inicial

    // Região paralela usando TASKS
    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp single // Apenas uma thread cria as tasks
        {
            // Criar tasks para cada inserção
            for (int i = 0; i < num_insertions; i++) {
                #pragma omp task firstprivate(i) // Cada task tem sua própria cópia de
i
                {
                    unsigned int local_seed = time(NULL) + omp_get_thread_num() + i; //
Seed único por task

                    // Escolha aleatória entre lista 1 ou 2
                    int choice = rand_r(&local_seed) % 2; // rand_r é thread-safe
                    int value = rand_r(&local_seed) % 1000;

                    if (choice == 0) {
                        insert_list1_critical(value);
                    } else {
                        insert_list2_critical(value);
                    }
                }
            }
        }
        // Todas as tasks terminam aqui automaticamente (taskwait implícito)
    }

    double end_time = omp_get_wtime();

    printf("\nResultados após %d inserções:\n", num_insertions);
    print_simple_list(&global_list1);
    print_simple_list(&global_list2);
    printf("Tempo total: %.4f segundos\n", end_time - start_time);
    printf("Total de elementos: %d\n", global_list1.count + global_list2.count);

    destroy_simple_list(&global_list1);
    destroy_simple_list(&global_list2);
}

// IMPLEMENTAÇÃO GENERALIZADA COM N LISTAS USANDO LOCKS EXPLÍCITOS

// Função para inserir em lista com lock explícito
void insert_locked_list(LockedList* list, int data) {
    Node* new_node = create_node(data);

    omp_set_lock(&list->lock); // Adquire lock específico desta lista
    {
        new_node->next = list->head;
        list->head = new_node;
        list->count++;

        // Simula processamento
        usleep(1000);
    }
    omp_unset_lock(&list->lock); // Libera lock específico desta lista
}

// Programa generalizado para N listas usando locks explícitos
void program_n_lists_explicit_locks(int num_lists, int num_insertions, int num_threads)
{
    printf("\n=== %d LISTAS COM LOCKS EXPLÍCITOS ===\n", num_lists);
    printf("Inserções: %d | Threads: %d\n\n", num_insertions, num_threads);

    // Aloca memória para array de listas
    LockedList* lists = (LockedList*)malloc(num_lists * sizeof(LockedList)); //
Alocação dinâmica para N listas
    if (lists == NULL) {
        fprintf(stderr, "Erro ao alocar memória para as listas\n");
        exit(1);
    }

    // Inicializa todas as listas
    for (int i = 0; i < num_lists; i++) {
        init_locked_list(&lists[i], i + 1); // Cada lista recebe seu próprio lock
    }

    double start_time = omp_get_wtime(); // Marca tempo inicial
```

```
    // Região paralela usando TASKS
    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp single // Apenas uma thread cria as tasks
        {
            // Criar tasks para cada inserção
            for (int i = 0; i < num_insertions; i++) {
                #pragma omp task firstprivate(i, lists, num_lists) // Cada task tem
cópias privadas
                {
                    unsigned int local_seed = time(NULL) + omp_get_thread_num() + i; //
Seed único por task

                    // Escolha aleatória entre as N listas
                    int list_choice = rand_r(&local_seed) % num_lists; // Escolha
dinâmica entre N listas
                    int value = rand_r(&local_seed) % 1000;

                    insert_locked_list(&lists[list_choice], value); // Acesso direto
por índice
                }
            }
        }
        // Todas as tasks terminam aqui automaticamente (taskwait implícito)
    }

    double end_time = omp_get_wtime();

    printf("\nResultados após %d inserções em %d listas:\n", num_insertions,
num_lists);
    int total_elements = 0;
    for (int i = 0; i < num_lists; i++) {
        print_locked_list(&lists[i]);
        total_elements += lists[i].count;
    }

    printf("Tempo total: %.4f segundos\n", end_time - start_time);
    printf("Total de elementos: %d\n", total_elements);

    // Libera memória das listas
    for (int i = 0; i < num_lists; i++) {
        destroy_locked_list(&lists[i]); // Destrói lock e libera nós de cada lista
    }
    free(lists); // Libera array de listas
}

int main() {
    printf("TAREFA 9: Regiões Críticas Nomeadas vs Locks Explícitos\n");
    printf("=======================================================\n");

    srand(time(NULL)); // Inicializa gerador de números aleatórios

    int num_insertions, num_threads, num_lists;

    // Entrada do usuário
    printf("\nDigite o número de inserções: ");
    if (scanf("%d", &num_insertions) != 1 || num_insertions < 1) {
        fprintf(stderr, "Número de inserções inválido\n");
        return 1;
    }

    printf("Digite o número de threads: ");
    if (scanf("%d", &num_threads) != 1 || num_threads < 1) {
        fprintf(stderr, "Número de threads inválido\n");
        return 1;
    }

    // Demonstração com 2 listas usando regiões críticas nomeadas
    program_two_lists_named_critical(num_insertions, num_threads); // Primeira
abordagem: limitada a 2 listas

    // Entrada para número de listas
    printf("\nDigite o número de listas para a versão generalizada: ");
    if (scanf("%d", &num_lists) != 1 || num_lists < 1) {
        fprintf(stderr, "Número de listas inválido\n");
        return 1;
    }
```

```
    // Demonstração com N listas usando locks explícitos
    program_n_lists_explicit_locks(num_lists, num_insertions, num_threads); // Segunda
abordagem: escalável para N listas

    printf("\nPrograma concluído com sucesso!\n");
    return 0;
}
```