Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 7 - Utilização de tasks
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

O presente relatório tem como objetivo entender o funcionamento do paralelismo com múltiplas threads e tasks, para isso desenvolvemos um programa em linguagem C utilizando a biblioteca `OpenMP` para implementar uma estrutura de lista encadeada cujos nós contêm nomes de arquivos fictícios e, dentro de uma região paralela, percorrer essa lista criando uma task individual para o processamento de cada nó. Cada task é responsável por imprimir o nome do arquivo e o identificador da thread que a executou.

O experimento também buscou responder perguntas-chave: todas as tarefas foram executadas? Houve repetição ou omissão na execução dos nós? O comportamento do programa muda a cada execução? Além disso, foram discutidas estratégias para garantir que cada tarefa seja executada uma única vez e por apenas uma thread.

2 Metodologia

O código foi estruturado de forma a atender aos requisitos da tarefa proposta. Para isso, foi definida uma `struct` representando os nós da lista encadeada, contendo dois atributos: `nome_arquivo`, utilizado para armazenar um nome fictício de arquivo, e `prox`, que é um ponteiro para o próximo nó da lista.

```
typedef struct Node {
 char nome_arquivo[100];
 struct Node* prox;
} Node;
```

Além disso, foram implementadas funções auxiliares para manipulação da lista: `criar_no()`, responsável por alocar e inicializar um novo nó; `adicionar_no()`, que insere o novo nó ao final da lista; e `liberar_lista()`, utilizada para desalocar toda a memória associada à lista encadeada.

```
Node* criar_no(const char* nome) {
    Node* novo = (Node*)malloc(sizeof(Node));
    strcpy(novo->nome_arquivo, nome);
    novo->prox = NULL;
    return novo;
}

void adicionar_no(Node** head, const char* nome) {
    Node* novo = criar_no(nome);
    if (*head == NULL) {
        *head = novo;
    } else {
        Node* atual = *head;
        while (atual->prox != NULL)
```

```
        atual = atual->prox;
        atual->prox = novo;
    }
}

void liberar_lista(Node* head) {
    Node* temp;
    while (head) {
        temp = head;
        head = head->prox;
        free(temp);
    }
}
```

A função `main()` foi desenvolvida com o objetivo de testar a criação da lista encadeada e implementar o processamento paralelo dos seus nós utilizando a biblioteca OpenMP. Inicialmente, os nós são criados e adicionados à lista com nomes fictícios de arquivos, simulando uma situação real de processamento de dados.

Em seguida, foi definida uma região paralela com a diretiva `#pragma omp parallel`, responsável por ativar múltiplas threads. Dentro dessa região, utilizou-se a diretiva `#pragma omp single` para garantir que apenas uma das threads fosse responsável por percorrer a lista e criar as tarefas (`tasks`) de processamento.

Para cada nó da lista, uma tarefa foi criada com a diretiva `#pragma omp task`. Cada tarefa recebe uma cópia do ponteiro para o nó atual e o identificador da thread que a criou (armazenado previamente). O código da tarefa imprime o nome do arquivo armazenado no nó, a thread que criou a tarefa e a thread que efetivamente a executou. Esse design permitiu observar de forma clara a distinção entre criação e execução de tarefas em um ambiente multithread.

Por fim, ao término da execução paralela, a função `liberar_lista()` é chamada para liberar a memória alocada para a lista encadeada, garantindo um uso adequado dos recursos do sistema

Dessa forma, nossa `main()` ficou da seguinte forma:

```
int main() {
    Node* lista = NULL;

    adicionar_no(&lista, "arquivo1.txt");
    adicionar_no(&lista, "arquivo2.txt");
    adicionar_no(&lista, "arquivo3.txt");
    adicionar_no(&lista, "arquivo4.txt");
    adicionar_no(&lista, "arquivo5.txt");

    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("Thread %d está criando as tasks\n", omp_get_thread_num());
            Node* atual = lista;
            while (atual != NULL) {
                Node* no = atual;
                int criadora = omp_get_thread_num();

                #pragma omp task firstprivate(no, criadora)
```

```
                {
                    printf("Arquivo: %s | Task criada pela thread: %d |
                    Executada pela thread: %d\n",
                    no->nome_arquivo, criadora, omp_get_thread_num());
                }
                atual = atual->prox;
            }
        }
    }

    liberar_lista(lista);
    return 0;
}
```

3 Resultados

Para a análise dos resultados, o código foi executado em diferentes configurações, permitindo observar os efeitos das decisões implementadas. Na execução do código completo, sem alterações, obtivemos os seguintes resultados:

Primeira execução:

```
Thread 1 está criando as tasks
Arquivo: arquivo1.txt | Task criada pela thread: 1 | Executada pela thread: 3
Arquivo: arquivo2.txt | Task criada pela thread: 1 | Executada pela thread: 2
Arquivo: arquivo5.txt | Task criada pela thread: 1 | Executada pela thread: 3
Arquivo: arquivo4.txt | Task criada pela thread: 1 | Executada pela thread: 0
Arquivo: arquivo3.txt | Task criada pela thread: 1 | Executada pela thread: 1
```

Segunda execução:

```
Thread 2 está criando as tasks
Arquivo: arquivo1.txt | Task criada pela thread: 2 | Executada pela thread: 3
Arquivo: arquivo5.txt | Task criada pela thread: 2 | Executada pela thread: 3
Arquivo: arquivo2.txt | Task criada pela thread: 2 | Executada pela thread: 1
Arquivo: arquivo3.txt | Task criada pela thread: 2 | Executada pela thread: 0
Arquivo: arquivo4.txt | Task criada pela thread: 2 | Executada pela thread: 2
```

A diretiva `#pragma omp single` permite que apenas uma thread entre em seu escopo. Em nosso programa, ela foi utilizada para proteger o processo de criação das tasks, garantindo que apenas uma thread seja responsável por criá-las e adicioná-las à fila de execução. As demais threads, enquanto isso, ficam disponíveis para executar as tasks criadas.

Nos resultados acima, observamos que, na primeira execução, a thread 1 foi responsável pela criação das tasks, enquanto na segunda execução, essa responsabilidade coube à thread 2. Isso mostra que a escolha da thread criadora é feita de forma não determinística, dependendo da estratégia de agendamento da OpenMP.

Caso a diretiva `#pragma omp single` não fosse utilizada, todas as threads presentes na região paralela executariam o mesmo trecho de código, criando múltiplas tasks para os mesmos arquivos. Isso resultaria em duplicações desnecessárias de tarefas, como ilustrado a seguir:

```
Thread 3 está criando as tasks
Arquivo: arquivo1.txt | Task criada pela thread: 3 | Executada pela thread: 3
Arquivo: arquivo2.txt | Task criada pela thread: 3 | Executada pela thread: 3
Arquivo: arquivo3.txt | Task criada pela thread: 3 | Executada pela thread: 3
Arquivo: arquivo4.txt | Task criada pela thread: 3 | Executada pela thread: 3
Arquivo: arquivo5.txt | Task criada pela thread: 3 | Executada pela thread: 3
Thread 1 está criando as tasks
Arquivo: arquivo1.txt | Task criada pela thread: 1 | Executada pela thread: 3
Arquivo: arquivo3.txt | Task criada pela thread: 1 | Executada pela thread: 3
Thread 0 está criando as tasks
Arquivo: arquivo5.txt | Task criada pela thread: 1 | Executada pela thread: 0
Arquivo: arquivo1.txt | Task criada pela thread: 0 | Executada pela thread: 0
Arquivo: arquivo2.txt | Task criada pela thread: 0 | Executada pela thread: 0
Arquivo: arquivo3.txt | Task criada pela thread: 0 | Executada pela thread: 0
Arquivo: arquivo4.txt | Task criada pela thread: 0 | Executada pela thread: 0
Arquivo: arquivo5.txt | Task criada pela thread: 0 | Executada pela thread: 0
Thread 2 está criando as tasks
Arquivo: arquivo1.txt | Task criada pela thread: 2 | Executada pela thread: 0
Arquivo: arquivo4.txt | Task criada pela thread: 1 | Executada pela thread: 3
Arquivo: arquivo3.txt | Task criada pela thread: 2 | Executada pela thread: 3
Arquivo: arquivo5.txt | Task criada pela thread: 2 | Executada pela thread: 3
Arquivo: arquivo2.txt | Task criada pela thread: 1 | Executada pela thread: 1
Arquivo: arquivo2.txt | Task criada pela thread: 2 | Executada pela thread: 2
Arquivo: arquivo4.txt | Task criada pela thread: 2 | Executada pela thread: 0
```

Esse comportamento demonstra a importância do uso da diretiva `single` quando se deseja garantir que apenas uma instância de determinado trecho de código seja executada dentro de uma região paralela.

Também foi realizada uma comparação entre o uso e a omissão da cláusula `firstprivate` na criação das tasks. Observou-se que, sem essa cláusula, todas as tarefas acabam acessando o mesmo ponteiro, resultando na repetição da mesma saída e na omissão dos demais arquivos. Com `firstprivate`, cada task recebe uma cópia independente do ponteiro para o nó atual, permitindo que o processamento seja feito corretamente e de forma paralela.

4 Conclusão

A implementação de um programa utilizando lista encadeada e tarefas com OpenMP permitiu observar na prática o funcionamento da diretiva `#pragma omp task` e seus impactos no paralelismo. O uso da diretiva `#pragma omp single` se mostrou essencial para garantir que todas as tasks fossem executadas e não houvesse duplicidade nem omissão de nós.

Por fim, o comportamento não determinístico da execução, com diferentes threads criando e executando tasks a cada execução, reforça a importância da sincronização e do controle de acesso aos dados compartilhados. O domínio dessas técnicas é essencial para o desenvolvimento de aplicações paralelas eficientes, corretas e seguras.