UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TAREFA 7 - UTILIZAÇÃO DE TASKS
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 18 DE ABRIL DE 2025

SUMÁRIO

1. INTRODUÇÃO

Com o avanço da computação paralela, o uso eficiente de múltiplas *threads* tem se tornado essencial para aproveitar ao máximo os recursos disponíveis nos processadores modernos. O *OpenMP* surge como uma das soluções mais acessíveis para paralelização em *C*, oferecendo diretivas simples para controle de concorrência, criação de tarefas e divisão de trabalho. Neste contexto, este trabalho propõe a implementação de um programa em *C* que simula o processamento paralelo de arquivos fictícios utilizando uma lista encadeada de nós, onde cada nó representa um nome de arquivo. A ideia central é percorrer essa lista dentro de uma região paralela e, para cada nó, criar uma tarefa com a diretiva *#pragma omp task*, que será executada por uma *thread* do time paralelo. O foco da atividade é compreender o comportamento da criação e execução de tarefas com *OpenMP*, observando se todos os nós são processados corretamente, se ocorrem repetições ou perdas, e se o comportamento se mantém consistente entre diferentes execuções. Além disso, busca-se analisar como garantir que cada tarefa seja única e vinculada exclusivamente a um nó da lista, evitando efeitos colaterais causados por condições de corrida ou má sincronização entre as threads.

2. METODOLOGIA

Para realizar o experimento proposto, foi implementado um programa em linguagem *C* que constrói uma lista encadeada contendo dez nós, onde cada nó armazena o nome de um arquivo fictício no formato *"fileX.txt"*, com *X* variando de *0* a *9*. A estrutura utilizada para representar cada nó inclui um campo para o nome do arquivo e um ponteiro para o próximo nó da lista. Após a criação da lista, o programa entra em uma região paralela controlada pelo *OpenMP*, onde apenas uma *thread* é responsável por percorrer a lista sequencialmente e, para cada nó, criar uma tarefa com a diretiva *#pragma omp task*. Para garantir que cada tarefa receba uma cópia independente do ponteiro para o nó correspondente — e evitar que todas as tarefas acessem o mesmo endereço —, foi utilizado o modificador *firstprivate(node)*, que garante que cada tarefa tenha sua própria cópia do ponteiro atual no momento da criação.

Durante o desenvolvimento, observou-se a necessidade de encapsular o laço de criação de tarefas dentro de uma região *#pragma omp single*, assegurando que apenas uma *thread* do time paralelo crie as tarefas, evitando comportamentos indesejados, como múltiplas *threads* tentando avançar o ponteiro *current* simultaneamente. Embora tenha sido considerado o uso de *#pragma omp critical* para proteger o acesso a *current*, essa abordagem se mostrou desnecessária no contexto em que apenas uma *thread* está encarregada da criação das tarefas. A sincronização do término de todas as tarefas foi feita com o uso da diretiva *#pragma omp taskwait*, colocada ao final da região *single*, garantindo que o programa somente avance após a conclusão de todas as tarefas pendentes. Por fim, a memória alocada para a lista foi liberada após o processamento.

A execução do programa foi feita repetidamente em diferentes ambientes, especialmente em um sistema *macOS* com arquitetura *Apple Silicon* e em um sistema *Linux* com compilador *GCC*, a fim de observar variações no comportamento da execução paralela. Durante essas execuções, foi registrado o nome do arquivo processado, o endereço de memória do nó e o identificador da *thread* responsável, como forma de análise e rastreamento da correta distribuição das tarefas.

3. RESULTADOS

A execução do programa permitiu analisar o comportamento da criação e execução de tarefas em um ambiente paralelo utilizando *OpenMP*. Foram observados resultados distintos a depender da plataforma e da forma como o código foi estruturado. Em execuções realizadas no *macOS*, com arquitetura *Apple Silicon*, o programa apresentou falhas de segmentação (*segmentation fault*) sempre que a criação das tarefas não era controlada adequadamente. Em especial, quando a região de criação das tarefas não estava envolvida pela diretiva *#pragma omp single*, múltiplas *threads* tentavam acessar e modificar o ponteiro current simultaneamente, resultando em acessos inválidos à memória e, consequentemente, na interrupção do programa.

Por outro lado, ao manter a criação das tarefas dentro de uma região *single* e utilizando *firstprivate* para isolar o ponteiro de cada nó, o comportamento tornou-se mais estável. Nessas condições, o programa passou a distribuir corretamente as tarefas entre diferentes threads, com cada thread processando um subconjunto dos nós da lista. A saída do programa confirmava que todos os arquivos da lista foram processados, e não foram observadas repetições nem omissões, desde que as diretivas de sincronização estivessem corretamente aplicadas.

Em contrapartida, em ambientes *Linux* com compilador *GCC*, mesmo quando o controle sobre a criação das tarefas era menos rigoroso, o programa frequentemente não apresentava falhas de segmentação. No entanto, foram observados comportamentos inconsistentes, como tarefas repetidas ou nós não processados, especialmente em execuções onde a proteção da variável *current* não era garantida. Isso reforça a ideia de que a ausência de erro explícito não implica em execução correta, sendo fundamental garantir a sincronização e isolamento adequado das variáveis envolvidas na criação de tarefas.

Além disso, verificou-se que a distribuição das tarefas entre as *threads* variava de uma execução para outra, o que é esperado em ambientes de execução paralela, dado que o agendamento das tarefas é dinâmico. Apesar dessa variação, uma vez que o código foi estruturado corretamente com as diretivas apropriadas, todas as execuções passaram a apresentar resultados coerentes, com cada arquivo sendo processado uma única vez por uma única *thread*, independentemente da ordem ou distribuição das tarefas.

4. CONCLUSÃO

A implementação proposta demonstrou, na prática, como o uso de tarefas com *OpenMP* pode ser uma ferramenta poderosa para paralelizar o processamento de estruturas dinâmicas em *C*, como listas encadeadas. Ao criar uma tarefa para cada nó da lista, foi possível distribuir o trabalho entre várias *threads*, melhorando o aproveitamento dos recursos computacionais disponíveis. No entanto, o experimento também evidenciou a importância crítica de uma correta sincronização e gerenciamento do compartilhamento de dados entre as *threads*. Problemas como condições de corrida e segmentações de memória ocorreram em execuções onde não havia uma delimitação clara de responsabilidade na criação das tarefas, mostrando que a paralelização mal coordenada pode comprometer a estabilidade e a confiabilidade do programa.

Além disso, os testes em diferentes sistemas operacionais revelaram que o comportamento do *OpenMP* pode variar sutilmente entre plataformas, o que reforça a necessidade de escrever código paralelizado com atenção redobrada a portabilidade e robustez. A utilização das diretivas *#pragma omp single*, *#pragma omp task* e *firstprivate* foi essencial para garantir que cada nó fosse processado exatamente uma vez, por uma única tarefa, e sem interferência entre threads.

Portanto, conclui-se que, embora simples em sua estrutura, o exercício foi bastante eficaz para ilustrar os desafios práticos do paralelismo com *OpenMP*, e como decisões aparentemente pequenas — como onde colocar uma diretiva ou como proteger uma variável — podem impactar significativamente o resultado final e a confiabilidade do sistema como um todo.

5. ANEXOS

- Repositório no Github com o programa desenvolvido: https://github.com/ErnaneJ/parallel-programming-dca3703