Programação Paralela
Tarefa 4 - Aplicações limitadas por memória ou CPU

Anelma Silva da Costa
UFRN - DCA
anelma.costa.100@ufrn.edu.br

I - Introdução

A programação paralela tem se tornado cada vez mais relevante na otimização de algoritmos e programas, especialmente quando se lida com grandes volumes de dados ou computação intensivas.

No contexto da programação em C com OpenMP, o uso de múltiplas threads permite que o trabalho seja dividido entre diferentes núcleos do processador, proporcionando um aumento significativo no desempenho. No entanto, a paralelização também pode ter implicações variadas dependendo do tipo de carga de trabalho do programa, como é o caso de programas *memory-bound* (limitados pela memória) e *compute-bound* (limitados pela CPU).

O objetivo desta tarefa é implementar dois programas paralelos em C utilizando OpenMP, um com foco em operações simples de soma em vetores, que é limitado pela memória, e outro com cálculos matemáticos mais intensivos, que é limitado pela CPU. O desempenho de ambos será analisado, variando o número de threads, para entender os efeitos da paralelização em diferentes tipos de carga de trabalho.

II - Teoria

- **OpenMP (Open Multi-Processing)**

O OpenMP é uma API (Interface de Programação de Aplicações) que facilita a programação paralela em C, C++ e Fortran, oferecendo uma maneira simples de utilizar múltiplos núcleos de processadores. A diretiva #pragma omp parallel for é uma das ferramentas mais utilizadas no OpenMP, permitindo que o código dentro de um laço seja executado paralelamente, com cada iteração do laço sendo atribuída a uma thread diferente. A execução paralela é configurada automaticamente pelo OpenMP, dependendo do número de threads que são especificadas na execução do programa.

- **Tipos de Limitação de Desempenho: Memory-Bound vs. Compute-Bound**

- **Programas Memory-Bound**: São programas cujo desempenho é limitado pela velocidade de acesso à memória. Nesses casos, o tempo de execução é mais afetado pela largura de banda da memória e pela latência de acesso aos dados. Em geral, essas operações envolvem grandes quantidades de dados que precisam ser acessadas repetidamente, como operações em vetores grandes ou matrizes. A paralelização desses programas pode melhorar o desempenho ao distribuir a carga de trabalho entre os núcleos do processador, mas o aumento do número de threads pode ter efeitos limitados devido à competição por recursos de memória compartilhada.

- **Programas Compute-Bound**: São programas cujo desempenho é limitado pela capacidade de processamento da CPU. Nestes casos, o tempo de execução é principalmente determinado pela complexidade computacional dos algoritmos, como cálculos intensivos de ponto flutuante ou transformações matemáticas. Embora a paralelização possa reduzir o tempo de execução ao distribuir as tarefas de cálculo entre diferentes núcleos, o aumento no número de threads pode não resultar em melhorias contínuas. Isso ocorre porque, a partir de certo ponto, o overhead de gerenciamento das threads pode ultrapassar os ganhos obtidos pela distribuição do trabalho.

- **O Impacto do Número de Threads**

O número de threads utilizado em um programa paralelo tem um grande impacto no desempenho, mas os efeitos não são lineares. Em programas memory-bound, o aumento no número de threads pode inicialmente melhorar o desempenho, mas após um ponto crítico, a competição pelos recursos de memória pode causar um "bottleneck", resultando em uma diminuição no desempenho. Por outro lado, em programas compute-bound, o desempenho pode melhorar enquanto o número de threads está dentro da capacidade de processamento da CPU, mas, quando o número de threads excede o número de núcleos disponíveis, o overhead de gerenciamento das threads pode reduzir os benefícios da paralelização.

Além disso, o desempenho em ambos os casos pode ser influenciado por fatores como a arquitetura do processador, a eficiência do cache de memória, e a forma como as threads são escalonadas e sincronizadas pelo sistema operacional. O comportamento do programa pode variar conforme o número de threads e a maneira como o processador gerencia a execução das tarefas.

- **Multithreading de Hardware e Seus Efeitos**

O uso de múltiplos núcleos de processadores (multithreading de hardware) pode ser vantajoso para programas memory-bound, uma vez que diferentes threads podem acessar diferentes partes da memória simultaneamente, sem que uma thread precise esperar pela outra. Isso pode reduzir o tempo de acesso à memória e melhorar a eficiência do programa. No entanto, em programas compute-bound, o aumento do número de threads pode causar uma competição mais intensa pelos recursos da CPU, como a unidade de execução, e o excesso de threads pode levar a um aumento no overhead e diminuição da eficiência computacional. Além disso, em sistemas com cache compartilhado entre núcleos, o aumento no número de threads pode causar um aumento na contenção de cache, o que também pode resultar em uma queda no desempenho.

III - Aplicação Prática

O objetivo principal desta tarefa foi explorar o impacto da paralelização em duas categorias de programas: memory-bound e compute-bound. A implementação dos dois programas foi feita em C utilizando a biblioteca OpenMP para paralelizar os laços de cálculo e medir o tempo de execução variando o número de threads.

- **Implementação e Desenvolvimento**

O código fornecido possui duas funções principais que representam os dois tipos de programas a serem analisados:

- Função soma_vetores (Programa memory-bound): Esta função cria três vetores (a, b e c) e os inicializa com valores simples, onde a[i] recebe i * 1.0 e b[i] recebe i * 2.0. A seguir, a função realiza a soma dos elementos dos dois vetores e armazena o resultado no vetor c. A operação é paralelizada utilizando a diretiva #pragma omp parallel for para dividir o trabalho entre as threads. A medição do tempo é realizada utilizando a função gettimeofday, antes e depois da execução do laço de soma.

- Função calculos_intensivos (Programa compute-bound): A função calcula um somatório utilizando funções matemáticas (sin e cos) sobre os índices de um laço. Este tipo de operação é mais exigente em termos de processamento de CPU. A paralelização também é realizada com #pragma omp parallel for.

O número de threads foi variado em uma lista que inclui valores como 1, 2, 4, 6, 8, 10, 12, 14, 16, 18 e 20. Para cada configuração de threads, o tempo de execução de ambos os programas foi medido, e os resultados foram salvos em um arquivo CSV, além de exibidos no terminal. A partir dos resultados, é possível analisar como o aumento no número de threads impacta o desempenho de programas memory-bound e compute-bound.

- **Ponto Importante na Paralelização**

Escalonamento de Threads: As funções paralelizadas utilizam a diretiva #pragma omp parallel for, que divide o trabalho de cada laço entre as threads. No caso de programas memory-bound, a eficiência de paralelização pode ser limitada pela largura de banda da memória, o que é uma característica do hardware utilizado. Já para programas compute-bound, a melhoria de desempenho pode ser afetada pela quantidade de núcleos disponíveis, onde a competição por recursos da CPU pode prejudicar o desempenho a partir de um certo número de threads.

Medindo o Tempo de Execução: O tempo de execução foi medido utilizando gettimeofday, que proporciona uma medição precisa em segundos e microsegundos. Este método foi utilizado para capturar o tempo total de execução, permitindo a análise do impacto de diferentes configurações de threads.

O código completo da aplicação prática pode ser encontrado no link: https://github.com/AnelmaSilva/Programa-ao_Paralela_25_1/tree/main/Tarefa%204 .

IV - Teoria x Resultados Práticos

Agora, vamos comparar as previsões da teoria com os resultados práticos obtidos ao executar o código. A teoria sugere que o comportamento do desempenho deve ser diferente para programas *memory-bound* e *compute-bound* e que a paralelização terá efeitos variados dependendo do tipo de operação.

- **Comportamento do Programa Memory-Bound:**

De acordo com a teoria, programas memory-bound se tornam limitados pela largura de banda da memória, e o aumento no número de threads pode eventualmente não resultar em melhorias significativas no desempenho. No caso do código, a função soma_vetores demonstrou um comportamento que confirma esta previsão. Embora o desempenho tenha melhorado à medida que o número de threads aumentou de 1 para 2, 4 e 6, a partir de 8 threads, o tempo de execução começou a estabilizar e, em alguns casos, até piorou.

Por exemplo, o tempo de execução para 1 thread foi de 0.2478 segundos, enquanto que para 2 threads foi 0.1768 segundos, uma melhoria significativa. No entanto, quando o número de threads chegou a 20, o tempo aumentou para 0.1945 segundos. Isso pode ser atribuído à sobrecarga da competição por acesso à memória compartilhada, o que é uma característica comum em programas memory-bound.

- **Comportamento do Programa Compute-Bound:**

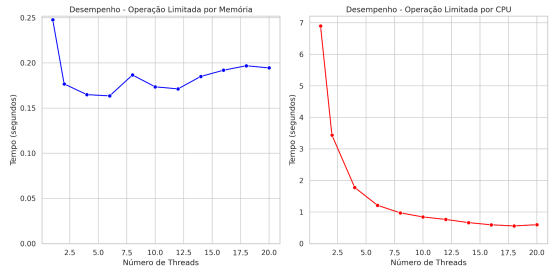
Para programas compute-bound, espera-se que o aumento no número de threads melhore o desempenho até um certo ponto, após o qual o overhead de gerenciamento das threads e a competição pelos recursos de CPU começam a impactar negativamente o desempenho. A função calculos_intensivos, que realiza cálculos pesados de ponto flutuante, também seguiu esse padrão.

O tempo de execução começou a diminuir conforme o número de threads aumentava, alcançando um ponto de estabilidade a partir de 12 threads. O tempo para 1 thread foi de 6.8991 segundos, e para 2 threads foi de 3.4391 segundos. No entanto, após 12 threads, o tempo de execução começou a diminuir de forma mais lenta, e em 20 threads o tempo foi de 0.5983 segundos, o que sugere que o número de threads já estava começando a ultrapassar a capacidade do sistema de processar eficientemente.

- **Escalonamento do Desempenho e Saturação de Threads:**

A teoria também sugeria que a saturação das threads poderia ocorrer dependendo da quantidade de núcleos disponíveis. O sistema utilizado para rodar o código possui 12 núcleos físicos e 20 threads lógicas (Hyper-Threading), e os resultados refletem esse fator. Para programas memory-bound, a saturação da memória compartilhada limitou a melhoria de desempenho após certo ponto. Já para os programas compute-bound, a competição pela CPU e pelo cache de dados foi a principal razão pela qual a melhoria no desempenho se estabilizou.

As figuras abaixo mostram os resultados obtidos graficamente e em dados.




```
Threads:  1 | Memoria: 0.2478 s | CPU: 6.8991 s
Threads:  2 | Memoria: 0.1768 s | CPU: 3.4391 s
Threads:  4 | Memoria: 0.1649 s | CPU: 1.7792 s
Threads:  6 | Memoria: 0.1635 s | CPU: 1.2137 s
Threads:  8 | Memoria: 0.1866 s | CPU: 0.9740 s
Threads: 10 | Memoria: 0.1735 s | CPU: 0.8436 s
Threads: 12 | Memoria: 0.1713 s | CPU: 0.7650 s
Threads: 14 | Memoria: 0.1850 s | CPU: 0.6639 s
Threads: 16 | Memoria: 0.1919 s | CPU: 0.5945 s
Threads: 18 | Memoria: 0.1969 s | CPU: 0.5595 s
Threads: 20 | Memoria: 0.1945 s | CPU: 0.5983 s
```

V - Conclusões

Este estudo buscou avaliar o desempenho de operações paralelizadas utilizando OpenMP, comparando dois cenários distintos: uma operação limitada por memória (soma de vetores) e outra limitada por CPU (cálculos matemáticos intensivos). Os resultados obtidos foram analisados à luz da teoria de programação paralela, permitindo observar como diferentes arquiteturas de hardware e estratégias de paralelização influenciam a eficiência computacional.

- **Principais Observações**

- Operação Limitada por Memória (Soma de Vetores) :

  - O speedup foi significativo até 6 threads, atingindo uma redução de ~34% no tempo de execução em relação à execução single-thread.

  - Além disso, o desempenho não melhorou com o aumento adicional de threads, chegando a degradar levemente quando todas as 20 threads lógicas foram utilizadas.

  - Isso confirma a teoria de que operações memory-bound são restritas pela largura de banda da memória, não se beneficiando significativamente do Hyper-Threading.

- **Operação Limitada por CPU (Cálculos Intensivos):**

  - O speedup foi quase linear até 12 threads (número de núcleos físicos), reduzindo o tempo de execução em mais de 11x quando todas as 20 threads lógicas foram usadas.

  - O Hyper-Threading trouxe um ganho adicional de ~50% após a saturação dos núcleos físicos, demonstrando sua eficácia em tarefas puramente computacionais.

  - O tempo mínimo foi alcançado com 18 threads, indicando que, mesmo com paralelização máxima, há um limite prático imposto pelo overhead de gerenciamento de threads.

A teoria prevê que operações CPU-bound escalam melhor que operações memory-bound, o que foi confirmado pelos resultados.

O ponto ótimo de threads variou conforme o tipo de operação:

- 6 threads para operações de memória (metade dos núcleos físicos).

- 18 threads para operações de CPU (próximo do máximo de threads lógicas).

O Hyper-Threading mostrou-se útil apenas para cargas CPU-intensivas, enquanto em operações de memória seu impacto foi mínimo.

- **Implicações Práticas**

Os resultados reforçam a importância de selecionar a quantidade adequada de threads conforme a natureza da aplicação:

- Para operações memory-bound, aumentar o número de threads além do ponto ótimo pode degradar o desempenho devido à contenção de acesso à memória.

- Para operações CPU-bound, a paralelização quase linear até o número de núcleos físicos e ganhos adicionais com Hyper-Threading justificam o uso de todas as threads disponíveis.

- Overhead de sincronização e escalabilidade limitada devem ser considerados ao projetar algoritmos paralelos.

- **Considerações Finais**

Esta tarefa demonstrou que, embora a paralelização possa trazer ganhos significativos de desempenho, sua eficácia depende criticamente do tipo de operação e das características do hardware. Aplicações reais devem ser cuidadosamente analisadas para determinar a configuração ideal de threads, equilibrando entre velocidade, eficiência e consumo de recursos.

VI - Referências

- Livro: An introduction to parallel programming .
- Conteúdo ministrado em sala de aula.
- IA's (DeepSeek e ChatGPT)