UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
IDCA3703 - PROGRAMAÇÃO PARALELA

TARFA 2 - PIPELINE E VETORIZAÇÃO
RELATÓRIO DE EXECUÇÃO

ERNANE FERREIRA ROCHA JUNIOR

NATAL/RN, 31 DE MARÇO DE 2025

SUMÁRIO

1. INTRODUÇÃO

A eficiência do código executado em processadores modernos vai além da lógica do algoritmo implementado, sendo fortemente influenciada pela forma como o código-fonte é otimizado pelo compilador. Este trabalho tem como objetivo investigar os efeitos do paralelismo ao nível de instrução (*ILP - Instruction-Level Parallelism*) em programas sequenciais, avaliando o impacto de diferentes níveis de otimização do compilador. O programa desenvolvido para este estudo realiza operações simples e acumulativas sobre grandes vetores, permitindo uma análise detalhada do desempenho do código sob distintas configurações de otimização.

Foram testados diversos níveis de otimização do compilador, nomeadamente **-O0**, **-O1**, **-O2** e **-O3**, para avaliar como essas otimizações afetam tanto o tempo de execução quanto a consistência dos resultados. Além disso, o estudo busca entender de que forma o paralelismo e a eliminação de dependências entre operações podem influenciar o desempenho geral do programa. A análise foca, portanto, em como as otimizações aplicadas pelo compilador podem maximizar o aproveitamento do paralelismo de instruções, reduzindo o impacto de dependências de dados e melhorando a performance do código.

2. METODOLOGIA

Este estudo tem como objetivo investigar os efeitos das otimizações do compilador no desempenho de programas que operam sobre vetores grandes, realizando tanto uma inicialização simples quanto uma soma acumulativa com dependência entre as iterações. O código foi desenvolvido para executar uma série de operações sobre um vetor, com foco em avaliar como diferentes otimizações do compilador podem influenciar o desempenho do programa.

O processo inicia com a criação de um vetor de tamanho *N*, onde cada elemento é inicializado com um valor inteiro sequencial, começando de 2. Em seguida, a soma dos elementos do vetor é realizada de duas maneiras distintas. A primeira abordagem consiste em uma soma acumulativa com dependência entre as iterações, ou seja, o resultado de cada iteração depende do cálculo realizado na iteração anterior. Na segunda abordagem, a soma acumulativa é realizada utilizando múltiplas variáveis acumuladoras, de modo a eliminar as dependências entre as iterações e permitir um maior paralelismo durante a execução.

Para avaliar o impacto das otimizações do compilador no desempenho do código, foram realizados testes utilizando três níveis de otimização distintos. O primeiro nível, **-O0**, não aplica nenhuma otimização, compilando o código de forma direta. O segundo nível, **-O2**, é uma otimização agressiva que visa melhorar o desempenho sem alterar a semântica do código. Esse nível aplica melhorias, como a eliminação de código redundante e otimizações em *loops*, sem modificar a estrutura do programa original. Por fim, o nível **-O3** representa uma otimização ainda mais agressiva, com foco em maximizar o desempenho. Nesse nível, o compilador utiliza técnicas como *Loop Unrolling*[1] e *Inline Expansion*[2], que visam reduzir a sobrecarga de controle de *loops* e otimizar a execução paralela do código.

Para medir o impacto das otimizações, o tempo de execução de cada versão do código foi registrado utilizando a função *gettimeofday*, que captura o tempo de início e término de cada execução. Além disso, foi verificado se os resultados das somas acumulativas eram consistentes entre as diferentes versões do código, garantindo que a precisão das operações fosse mantida. A comparação dos tempos de execução entre as versões compiladas com diferentes otimizações permitiu avaliar como cada nível de otimização influenciou o desempenho geral do programa.

O código foi desenvolvido em linguagem C e compilado com o compilador *gcc-14 (Homebrew GCC 14.2.0_1) 14.2.0* em um *Macbook Air chip Apple M2 com macOS Sequoia 15.3.2*. O tamanho do vetor foi fixado em *N = 100.000.000*, o que garantiu um ambiente controlado e consistente para a realização dos testes, permitindo uma análise mais precisa dos efeitos das otimizações no tempo de execução e na consistência dos resultados.

---

[1] *Loop unrolling* é uma técnica de otimização que consiste em expandir um loop para reduzir o número de iterações e, assim, diminuir o overhead de controle de fluxo, melhorando a performance.

[2] *Inline expansion* é uma técnica de otimização que substitui chamadas de funções por seu código diretamente no ponto de invocação, eliminando a sobrecarga de chamada de função e potencialmente melhorando a eficiência.

3. RESULTADOS

Os resultados obtidos durante os testes de desempenho evidenciam o impacto das otimizações no tempo de execução e na consistência dos cálculos. Ao comparar os tempos de execução para os diferentes níveis de otimização, notamos variações notáveis, especialmente entre os níveis **-O0**, **-O2** e **-O3**.

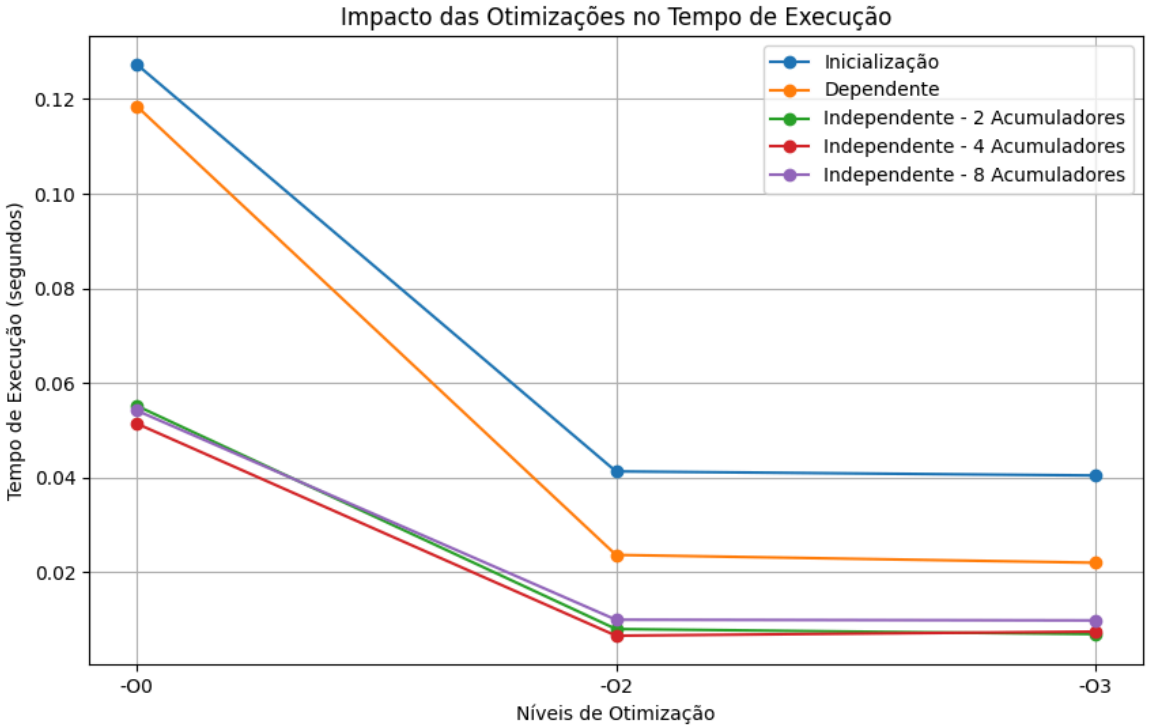
Otimização	Tempo de Inicialização (s)	Tempo Dependente (s)	Tempo Independente (2 acum.) (s)	Tempo Independente (4 acum.) (s)	Tempo Independente (8 acum.) (s)
-O0	0.127321	0.118464	0.055164	0.051404	0.054173
-O2	0.041316	0.023662	0.007997	0.006590	0.009964
-O3	0.040452	0.022007	0.006915	0.007453	0.009818

Primeiramente, o tempo de inicialização do vetor, que foi de 0,127 segundos com o nível **-O0**, sofreu uma redução significativa para 0,041 segundos com **-O2**. O nível **-O3** apresentou um tempo de inicialização de 0,040 segundos, ligeiramente menor que o de **-O2**, indicando que o impacto da otimização no processo de inicialização é considerável, mas se estabiliza em níveis muito próximos entre **-O2** e **-O3**.

Em relação à soma dependente, onde há uma dependência entre as iterações, o tempo de execução também apresentou uma redução com o aumento do nível de otimização. Com **-O0**, o tempo de soma dependente foi de 0,118 segundos, enquanto com **-O2** esse valor caiu para 0,023 segundos. Já com **-O3**, o tempo de execução foi de 0,022 segundos, demonstrando uma leve melhoria em relação ao nível **-O2**, mas sem uma grande diferença em termos de tempo.

Nos testes que envolvem somas independentes, onde múltiplos acumuladores são usados para eliminar as dependências de iteração, os tempos de execução caem de forma mais expressiva. Com o uso de 2 acumuladores, o tempo com **-O0** foi de 0,055 segundos, com **-O2** caiu para 0,008 segundos e com **-O3** o tempo foi de 0,007 segundos. Isso mostra claramente como a eliminação das dependências entre as iterações aumenta a possibilidade de paralelismo e melhora o desempenho. Quando o número de acumuladores é aumentado para 4, os tempos continuam a reduzir: de 0,051 segundos em -O0 para 0,007 segundos em -O2, e 0,007 segundos em -O3.

No entanto, a soma com 8 acumuladores apresentou um comportamento curioso, onde o tempo foi de 0,054 segundos em **-O0**, 0,010 segundos em **-O2** e 0,010 segundos em -O3, indicando que, ao utilizar um número muito alto de acumuladores, o ganho de desempenho não é tão significativo, provavelmente devido a um efeito de sobrecarga no gerenciamento dos acumuladores.



Todos os testes mostraram que os resultados das somas foram consistentes entre as versões, com o valor final sendo o mesmo para todos os níveis de otimização. Isso garante que as otimizações aplicadas não alteraram a precisão dos cálculos e preservaram a semântica do código.

4. CONCLUSÃO

Este estudo investigou o impacto das otimizações de compilador no desempenho de operações envolvendo grandes vetores e cálculos acumulativos, com o foco na exploração do paralelismo ao nível de instrução (*ILP*). Os resultados demonstraram que a aplicação de otimizações significativamente reduz o tempo de execução, especialmente quando dependências entre as iterações são eliminadas através do uso de múltiplos acumuladores.

Os níveis de otimização **-O2** e **-O3** apresentaram melhorias substanciais no tempo de execução em comparação com o nível **-O0**, particularmente nas operações que eliminam as dependências de iteração. A inicialização do vetor, por exemplo, foi consideravelmente acelerada com as otimizações **-O2** e **-O3**, refletindo o impacto das técnicas de otimização no processamento inicial. Embora o tempo de soma dependente tenha mostrado uma melhora com a otimização, a maior redução de tempo ocorreu nas versões independentes, especialmente com o uso de 2 e 4 acumuladores, onde o paralelismo foi mais eficaz.

Contudo, ao aumentar o número de acumuladores para 8, o ganho de desempenho foi menos significativo, sugerindo que, após certo ponto, o aumento de variáveis acumuladoras não traz benefícios adicionais e pode até gerar sobrecarga. Além disso, a consistência dos resultados em todas as versões, independentemente do nível de otimização, confirmou que as alterações feitas pelo compilador não comprometeram a precisão dos cálculos.

Em suma, este trabalho evidenciou como as otimizações de compilador, quando aplicadas adequadamente, podem melhorar substancialmente o desempenho de programas, especialmente ao explorar as possibilidades de paralelismo ao nível de instrução. A redução de dependências entre as iterações e o uso de múltiplos acumuladores foram cruciais para maximizar a eficiência do código, oferecendo uma visão valiosa sobre como o paralelismo pode ser explorado em programas sequenciais. A pesquisa reforça a importância de escolher o nível de otimização adequado para alcançar o melhor desempenho em diferentes tipos de operações computacionais.

5. ANEXOS

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