Investigando Paralelismo ao Nível de Instrução (ILP) em C

🎯 Objetivo

Esta atividade tem como objetivo investigar os efeitos do **paralelismo ao nível de instrução (ILP)** em programas C, analisando como dependências entre iterações afetam o desempenho dos laços sob diferentes níveis de otimização do compilador.

> **Meta principal:** Compreender como o compilador explora paralelismo interno e como dependências de dados podem limitar ou facilitar essas otimizações, especialmente em operações vetoriais e loops computacionalmente intensivos.

📚 Teoria

O **paralelismo ao nível de instrução (ILP)** refere-se à capacidade dos processadores modernos de executar múltiplas instruções simultaneamente, desde que não haja dependências de dados entre elas. O compilador pode explorar ILP reordenando instruções e utilizando recursos internos do processador, mas dependências no código podem limitar esse paralelismo.

🔗 Tipos de Dependência

- **Dependência de dados (RAW - Read After Write):** Uma instrução precisa do resultado de outra anterior
- **Dependência anti (WAR - Write After Read):** Uma instrução escreve em um local que será lido por outra
- **Dependência de saída (WAW - Write After Write):** Duas instruções escrevem no mesmo local

🛠️ Técnicas de Otimização do Compilador

- **Vetorização SIMD:** Uso de instruções que processam múltiplos dados simultaneamente
- **Loop Unrolling:** Desenrolar loops para reduzir overhead de controle
- **Instruction Scheduling:** Reordenação de instruções para maximizar paralelismo
- **Register Allocation:** Otimização do uso de registradores

🧪 Experimentos Implementados

Nesta atividade, três laços são implementados para ilustrar diferentes cenários de dependência:

1 Inicialização de Vetor (Sem Dependências)

```
for (long i = 0; i < N; i++) {
    vector[i] = i * 0.5 + 1.0;
}
```

Características:

- Cada elemento do vetor é inicializado de forma independente
- Não há dependência entre iterações
- O compilador pode paralelizar e otimizar facilmente este loop
- Permite vetorização SIMD eficiente

Gargalo: Limitado pela largura de banda da memória (memory bandwidth) para escrita, especialmente em vetores grandes.

2 Soma Acumulativa (Com Dependência)

```
double sum_sequential = 0.0;
for (long i = 0; i < N; i++) {
    sum_sequential += vector[i];
}
```

Características:

- Os elementos do vetor são somados sequencialmente em uma única variável acumuladora
- **Forte dependência entre iterações:** cada soma depende do resultado anterior
- Limita significativamente o paralelismo disponível
- O compilador pode aplicar vetorização limitada

Limitação: A dependência sequencial força serialização parcial, impedindo paralelismo total mesmo com otimizações agressivas.

3 Soma com Múltiplos Acumuladores (Quebra de Dependência)

```
double sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
for (long i = 0; i <= N - 4; i += 4) {
    sum0 += vector[i];
    sum1 += vector[i+1];
    sum2 += vector[i+2];
    sum3 += vector[i+3];
}
double sum_parallel = sum0 + sum1 + sum2 + sum3;
```

Características:

- A soma é dividida entre múltiplas variáveis acumuladoras
- Reduz drasticamente as dependências entre iterações
- Permite ao compilador explorar mais ILP e paralelismo interno

- Facilita vetorização SIMD eficiente
- Aproveita múltiplas unidades funcionais da CPU

Vantagem: Loop unrolling manual combinado com quebra de dependência permite ao compilador aplicar otimizações agressivas, resultando em speedups significativos.

🔬 Procedimento Experimental

1. **Implementar os três loops em C** conforme descrito acima
2. **Compilar o programa com diferentes níveis de otimização:**
    - `-O0`: Sem otimizações (baseline)
    - `-O2`: Otimizações moderadas
    - `-O3`: Otimizações agressivas, incluindo vetorização e paralelismo
3. **Medir e comparar os tempos de execução** de cada loop em cada nível de otimização
4. **Analisar os resultados** e identificar padrões de performance

📊 Análise de Resultados Esperados

Loop	-O0 (Baseline)	-O2 (Moderado)	-O3 (Agressivo)	Speedup Esperado
1. Inicialização	Lento	Rápido	Muito Rápido	Alto (5-10x)
2. Soma Simples	Lento	Médio	Rápido	Moderado (2-4x)
3. Soma Múltipla	Lento	Rápido	Muito Rápido	Muito Alto (8-15x)

📈 Padrões de Performance

🔷 **Loop 1 (Inicialização):**

Deve apresentar **grande ganho de desempenho** com otimizações, pois não há dependências. O compilador pode aplicar vetorização SIMD agressiva, mas será limitado pela largura de banda da memória para escritas.

🔷 **Loop 2 (Soma Acumulativa):**

O ganho com otimização será **limitado** devido à dependência sequencial entre as iterações. Mesmo com -O3, o compilador não pode quebrar completamente a cadeia de

dependências.

🔷 **Loop 3 (Soma com Múltiplas Variáveis):**

A quebra de dependência permite ao compilador aplicar ILP de forma efetiva, resultando em **desempenho superior**, especialmente com otimizações mais agressivas (-O3).

🧠 Conceitos Fundamentais Demonstrados

1. 🔄 Dependência de Dados vs. Paralelismo

A diferença entre os loops 2 e 3 ilustra perfeitamente como dependências de dados podem ser gargalo para performance, e como técnicas simples (múltiplos acumuladores) podem quebrar essas dependências.

2. 🚀 Poder das Otimizações do Compilador

A comparação entre -O0, -O2 e -O3 demonstra o impacto dramático das otimizações automáticas do compilador, especialmente vetorização SIMD e loop unrolling.

3. ⚖️ Trade-offs de Design

O loop 3 mostra como pequenas mudanças no código (usar 4 variáveis em vez de 1) podem ter impacto dramático na performance, ilustrando a importância de escrever código "amigável ao compilador".

4. 🏗️ Arquitetura de Processadores Modernos

Os resultados refletem características de CPUs modernas: múltiplas unidades funcionais, pipelines superescalares, e instruções SIMD (SSE, AVX).

💡 Lições Práticas

- **Evite dependências desnecessárias:** Use múltiplos acumuladores quando possível

- **Confie no compilador:** Otimizações modernas são muito eficazes quando o código permite
- **Profile antes de otimizar:** Meça o impacto real das diferentes abordagens
- **Entenda sua arquitetura:** Diferentes CPUs podem apresentar resultados distintos
- **Loop unrolling:** Pode ser benéfico mesmo quando feito manualmente

---

Programação Paralela - Tarefa 2: Análise de Paralelismo ao Nível de Instrução (ILP)

Documento gerado automaticamente - Para conversão em PDF use: Ctrl+P → Salvar como PDF