Navegação

Tarefa 5 - Contagem de Números Primos com OpenMP

Descrição

Este programa implementa um algoritmo para contar números primos entre 2 e um valor máximo `n`, comparando o desempenho entre versões sequencial e paralela usando OpenMP. O projeto demonstra conceitos fundamentais de programação paralela, incluindo paralelização de loops, operações de redução, e análise de performance.

Objetivo

Demonstrar a aplicação prática de OpenMP em um problema computacionalmente intensivo, analisando quando a paralelização oferece vantagens reais e quais são os desafios envolvidos na programação paralela.

Funcionalidades

- **Versão Sequencial**: Conta primos usando um único thread
- **Versão Paralela**: Usa `#pragma omp parallel for` com redução para paralelizar o loop principal
- **Comparação de Performance**: Mede tempos de execução e calcula speedup e eficiência

- **Análise de Overhead**: Demonstra quando a paralelização compensa ou não
- **Verificação de Correção**: Confirma que ambas as versões produzem o mesmo resultado

Análise do Código

Elementos-chave da paralelização:

- `parallel for` : Distribui iterações do loop entre threads
- `reduction(+:contador)` : Evita condições de corrida na soma
- Cada thread mantém uma cópia local do contador
- Ao final, todas as cópias são somadas automaticamente

8x

Speedup Teórico Máximo

3-4x

Speedup Prático Observado

40-60%

Eficiência Esperada

Análise de Overhead e Performance

Por que Versão Sequencial é Mais Rápida para n Pequenos?

O **overhead de paralelização** consiste em vários componentes que podem superar os benefícios para problemas pequenos:

1. Custos de Criação de Threads

```
Tempo para criar 8 threads: ~0.0001-0.0002 segundos Tempo
para computar 1000 primos: ~0.00001 segundos Resultado:
Overhead é 10-20x maior que o trabalho útil!
```

2. Sincronização e Redução

- **Barrier implícita**: Threads esperam umas pelas outras
- **Operação de redução**: Combinar contadores parciais
- **Cache coherency**: Sincronização entre núcleos do processador

3. Distribuição de Carga

Para n = 1000 com 8 threads:

- Thread 0: números 2, 10, 18, 26... (125 números)
- Thread 1: números 3, 11, 19, 27... (125 números)
- ...
- Cada thread processa muito poucos números

Análise Quantitativa do Overhead

n	Tempo Seq	Tempo Par	Overhead	Overhead/Trabalho
1000	0.000011s	0.000113s	**0.000102s**	**9.3x**
10000	0.000190s	0.004460s	**0.004270s**	**22.5x**
100000	0.003703s	0.001484s	**-0.002219s**	**Speedup!**

> **Ponto de Break-even**: ~100.000 números, onde o overhead se torna desprezível comparado ao trabalho computacional.

Resultados Experimentais

Performance Observada

n	Primos	Tempo Seq	Tempo Par	Speedup	Eficiência
1000	168	0.000011s	0.000113s	**0.09x**	**1.1%**
10000	1229	0.000190s	0.004460s	**0.04x**	**0.5%**
100000	9592	0.003703s	0.001484s	**2.50x**	**31.3%**
500000	41538	0.032372s	0.007992s	**4.05x**	**50.6%**
1000000	78498	0.083585s	0.021391s	**3.91x**	**48.9%**

Interpretação dos Resultados

1. Região de Overhead (n < 100.000):

- Speedup < 1: Versão sequencial vence
- Overhead de criação de threads domina
- Eficiência muito baixa (<5%)

2. Região de Transição (n ≈ 100.000):

- Ponto de break-even
- Speedup começa a aparecer
- Eficiência ainda baixa (~30%)

3. Região Eficiente (n > 500.000):

- Speedup estável ~4x
- Eficiência ~50%
- Trabalho computacional justifica overhead

Conclusões e Lições Aprendidas

Quando Usar Paralelização

✅ **SIM - Quando:**

- Problema é computacionalmente intensivo
- Dados são independentes entre iterações
- Overhead é pequeno comparado ao trabalho
- n > 100.000 (para este problema específico)

❌ **NÃO - Quando:**

- Problema muito pequeno (overhead domina)
- Dependências entre dados
- Sincronização frequente necessária
- Recursos limitados (single-core, memory-bound)

Principais Aprendizados

1. **Overhead é Real**: Paralelização não é sempre benéfica
2. **Granularidade Importa**: Muito trabalho fino gera overhead
3. **Medição é Essencial**: Sempre medir antes de otimizar
4. **Correção Primeiro**: Garantir resultado correto antes de otimizar
5. **Escalabilidade Limitada**: Lei de Amdahl sempre se aplica

Conceitos Fundamentais Demonstrados

- **Paralelização de Loops**: `#pragma omp parallel for`
- **Operações de Redução**: `reduction(+:contador)`
- **Medição Precisa**: `omp_get_wtime()`
- **Análise de Performance**: Speedup e eficiência
- **Overhead Analysis**: Identificação de custos escondidos
- **Load Balancing**: Distribuição de trabalho entre threads

Ambiente de Teste: 8 threads disponíveis, GCC com flags `-fopenmp -O2 -lm`, Sistema Linux

Este projeto demonstra que programação paralela eficiente requer compreensão profunda dos trade-offs entre overhead, granularidade, e distribuição de carga. Sempre meça antes de otimizar!

Código

```c
// Função para verificar se um número é primo
int eh_primo(int n) {
    if (n < 2) return 0;
    if (n == 2) return 1;
    if (n % 2 == 0) return 0;

    int limite = (int)sqrt(n);
    for (int i = 3; i <= limite; i += 2) {
        if (n % i == 0) return 0;
    }
    return 1;
}

// Versão sequencial
int contar_primos_sequencial(int n) {
    int contador = 0;

    for (int i = 2; i <= n; i++) {
        if (eh_primo(i)) {
            contador++;
        }
    }

    return contador;
}

// Versão paralela com OpenMP
int contar_primos_paralelo(int n) {
    int contador = 0;

    #pragma omp parallel for reduction(+:contador)
    for (int i = 2; i <= n; i++) {
        if (eh_primo(i)) {
            contador++;
        }
    }

    return contador;
}

// Função para medir tempo de execução
double medir_tempo(int (*funcao)(int), int n) {
    double inicio = omp_get_wtime();
    int resultado = funcao(n);
    double fim = omp_get_wtime();
    return fim - inicio;
}

int main() {
    int n;
    scanf("%d", &n);

    printf("\n=== CONTAGEM DE NÚMEROS PRIMOS ===\n");
    printf("Intervalo: 2 a %d\n", n);
    printf("Número de threads disponíveis: %d\n", omp_get_max_threads());

    // Medindo tempo da versão sequencial
    double tempo_seq = medir_tempo(contar_primos_sequencial, n);
    int primos_seq = contar_primos_sequencial(n);

    // Medindo tempo da versão paralela
    double tempo_par = medir_tempo(contar_primos_paralelo, n);
    int primos_par = contar_primos_paralelo(n);

    // Exibindo resultados
    printf("\n=== RESULTADOS ===\n");
```

```
    printf("\nTempo sequencial: %.6f segundos\n", tempo_seq);
    printf("Tempo paralelo:   %.6f segundos\n", tempo_par);

    if (tempo_seq > 0) {
        double speedup = tempo_seq / tempo_par;
        printf("Speedup: %.2fx\n", speedup);
        printf("Eficiência: %.2f%%\n", (speedup / omp_get_max_threads()) * 100);
    }

    // Verificação de correção
    if (primos_seq != primos_par) {
        printf("\n✗ ERRO! Resultados diferentes entre as versões.\n");
    }

    return 0;
}
```