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

Implementação Atual - Paralelização Básica:

- `parallel for`: Distribui iterações do loop entre threads
- **SEM reduction**: Demonstra problemas de condição de corrida (race condition)
- Cada thread acessa diretamente a variável compartilhada `contador`
- Resultados incorretos devido à falta de sincronização

4x

Threads Fixas

~3x

Speedup Observado

ERRO

Race Condition

Análise da Race Condition

O que Acontece na Race Condition?

Cenário de Race Condition:

TIMELINE

```
// Timeline da Race Condition (contador inicial = 100)
Tempo    Thread A                Thread B                Resultado
T1       LOAD contador → 100                               contador = 100
T2                               LOAD contador → 100      contador = 100
T3       ADD 1 → reg_A = 101                             contador = 100
T4                               ADD 1 → reg_B = 101      contador = 100
T5       STORE 101 → contador                            contador = 101
T6                               STORE 101 → contador     contador = 101
```

```
ESPERADO: contador = 102 (duas operações de incremento)
RESULTADO: contador = 101 (uma operação foi perdida!)
```

Por que a Perda Varia com o Tamanho do Problema?

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

1. Custos de Criação de Threads

```
ANALYSIS
// Análise de Overhead para n = 1000
Tempo para criar 4 threads:     ~0.0001-0.0002 segundos
Tempo para computar 1000 primos:  ~0.00001 segundos

Resultado: Overhead é 10-20x maior que o trabalho útil!

// Para problemas pequenos, o custo de paralelização
// supera os benefícios do processamento paralelo
```

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 4 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 (4 Threads Fixas - SEM Reduction)

n	Primos Corretos	Primos Paralelo	Diferença	Tempo Seq (s)	Tempo Par (s)	Speedup	Status
1000	168	**112**	**-56 (-33%)**	0.000047	0.000168	**0.28x**	**ERRO - Race Condition**
10000	1229	**994**	**-235 (-19%)**	0.000654	0.000256	**2.55x**	**ERRO - Race Condition**
100000	9592	**8843**	**-749 (-8%)**	0.009823	0.002585	**3.80x**	**ERRO - Race Condition**
1000000	78498	**76514**	**-1984 (-3%)**	0.112938	0.056190	**2.01x**	**ERRO - Race Condition**
10000000	664579	**651086**	**-13493 (-2%)**	2.683373	0.909031	**2.95x**	**ERRO - Race Condition**

Análise Detalhada da Race Condition

1. Padrão de Perdas por Race Condition:

- **n = 1000: Perde 56 primos (-33%)** - Overhead massivo + muitas colisões
- **n = 10000: Perde 235 primos (-19%)** - Alta frequência de colisões
- **n = 100000: Perde 749 primos (-8%)** - Colisões moderadas
- **n = 1000000: Perde 1984 primos (-3%)** - Menos colisões relativas
- **n = 10000000: Perde 13493 primos (-2%)** - Proporção menor de colisões

2. Por que a Perda Percentual Diminui?

- **Densidade de Operações**: Mais trabalho computacional por thread
- **Menos Contenção Relativa**: Operações de incremento se tornam menos frequentes
- **Distribuição Temporal**: Threads passam mais tempo calculando que competindo
- **MAS ainda há perdas absolutas significativas!**

3. Variabilidade dos Resultados:

- Cada execução produz **resultados diferentes**
- Depende do timing exato das threads
- Torna debugging extremamente difícil
- Comportamento não-determinístico é inaceitável

4. Speedup Ilusório:

- Speedup existe mas os resultados estão **incorretos**
- Para n = 1000: Overhead ainda domina (speedup 0.28x)
- Para n ≥ 10000: Speedup aparente de ~2.0-3.8x
- **Performance sem correção é completamente inútil**

Por que é mais desafiador programar em paralelo?

A programação paralela introduz complexidades que não existem na programação sequencial. Este projeto demonstra alguns dos principais desafios que tornam a programação paralela mais difícil de dominar.

1. Sincronização - O Desafio Central

Problema Demonstrado:

- **Race Conditions**: Múltiplas threads acessando `contador` simultaneamente
- **Operações Não-Atômicas**: `contador++` é na verdade 3 instruções assembly
- **Resultados Imprevisíveis**: Cada execução produz resultado diferente

Soluções de Sincronização:

```c
// ✗ ERRADO - Race Condition (implementação atual)
int contar_primos_paralelo(int n) {
    int contador = 0;  // Variável compartilhada - PERIGOSO!

    #pragma omp parallel for  // SEM reduction
    for (int i = 2; i <= n; i++) {
        if (eh_primo(i)) {
            contador++;  // Race condition aqui!
        }
    }

    return contador;  // Resultado incorreto
}

// ✓ CORRETO - Com Reduction
int contar_primos_correto(int n) {
    int contador = 0;

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

    return contador;  // Resultado correto
}

// ✓ ALTERNATIVA - Com Critical Section
int contar_primos_critical(int n) {
    int contador = 0;

    #pragma omp parallel for
    for (int i = 2; i <= n; i++) {
        if (eh_primo(i)) {
            #pragma omp critical
            contador++;  // Acesso mutuamente exclusivo
        }
```

```
    }

    return contador;
}
```

2. Comunicação entre Threads

Desafios de Comunicação:

- **Dados Compartilhados**: Quais variáveis são seguras para compartilhar?
- **Consistência de Cache**: Diferentes cores podem ter caches desatualizados
- **False Sharing**: Threads modificando dados próximos na mesma linha de cache
- **Overhead de Sincronização**: Custo de coordenação entre threads

3. Equilíbrio de Carga (Load Balancing)

O Problema:

- **Distribuição Desigual**: Algumas threads terminam antes que outras
- **Trabalho Variável**: Testar primalidade tem custo diferente para cada número
- **Idle Time**: Threads rápidas ficam esperando as lentas

Estratégias de Balanceamento:

> **STATIC**
>
> Divisão fixa - pode gerar desbalanceamento

> **DYNAMIC**
>
> Distribuição dinâmica - melhor balanceamento

> **GUIDED**

Chunks adaptativos - equilibra overhead e balanceamento

Conclusões e Lições Aprendidas

Lições Críticas sobre Race Conditions

✗ PROBLEMA DEMONSTRADO:

- **Paralelização SEM sincronização = Resultados incorretos**
- Race conditions ocorrem quando múltiplas threads acessam dados compartilhados
- Operações como `contador++` NÃO são atômicas
- Speedup sem correção é completamente inútil

✓ SOLUÇÕES NECESSÁRIAS:

- Usar `reduction(+:contador)` para operações de soma
- Implementar sincronização com mutexes quando necessário
- Sempre validar correção antes de medir performance
- Entender que paralelização correta pode ter overhead

Principais Aprendizados - Race Conditions

1. **Race Conditions São Perigosas**: Resultados imprevisíveis e incorretos
2. **Sincronização é Obrigatória**: Dados compartilhados precisam de proteção
3. **Testes de Correção são Críticos**: Sempre validar resultados paralelos
4. **Operações Atômicas**: `contador++` não é thread-safe
5. **Performance vs Correção**: Correção sempre vem primeiro

Conceitos Fundamentais Demonstrados

- **Paralelização Básica**: `#pragma omp parallel for`
- **Race Conditions**: Problemas de acesso concorrente a dados compartilhados
- **Medição de Tempo**: `omp_get_wtime()`
- **Análise de Speedup**: Mesmo com resultados incorretos
- **Validação de Correção**: Comparação entre versões sequencial e paralela
- **Overhead Analysis**: Identificação de custos mesmo com race conditions

Código

```c
#include <stdio.h>  // Para printf, scanf
#include <stdlib.h> // Para funções padrão
#include <math.h>   // Para sqrt()
#include <omp.h>    // Para OpenMP
#include <time.h>   // Para medição de tempo

// Função para verificar se um número é primo
int eh_primo(int n) {
    if (n < 2) return 0;        // Números menores que 2 não são primos
    if (n == 2) return 1;       // 2 é o único primo par
    if (n % 2 == 0) return 0;   // Outros números pares não são primos

    int limite = (int)sqrt(n);  // Só precisa testar até raiz quadrada
    for (int i = 3; i <= limite; i += 2) {  // Testa apenas números ímpares
        if (n % i == 0) return 0;           // Se divisível, não é primo
    }
    return 1;  // Se chegou até aqui, é primo
}

// Versão sequencial - executa em uma única thread
int contar_primos_sequencial(int n) {
    int contador = 0;  // Contador seguro (sem concorrência)

    for (int i = 2; i <= n; i++) {      // Loop sequencial
        if (eh_primo(i)) {              // Testa se é primo
            contador++;                 // Incrementa sem risco de race condition
        }
    }

    return contador;  // Retorna contagem correta
}

// Versão paralela com OpenMP - DEMONSTRA RACE CONDITION
int contar_primos_paralelo(int n) {
    int contador = 0;  // Variável compartilhada entre threads (PERIGOSO!)

    #pragma omp parallel for  // Paraleliza o loop SEM reduction
    for (int i = 2; i <= n; i++) {      // Cada thread processa algumas iterações
        if (eh_primo(i)) {              // Teste de primalidade independente
            contador++;                 // RACE CONDITION: acesso não sincronizado!
        }
    }

    return contador;  // Retorna contagem INCORRETA devido à race condition
}

// Função para medir tempo de execução usando ponteiro para função
double medir_tempo(int (*funcao)(int), int n) {
    double inicio = omp_get_wtime();  // Marca tempo de início (alta precisão)
    int resultado = funcao(n);        // Executa a função (seq ou par)
    double fim = omp_get_wtime();     // Marca tempo de fim
    return fim - inicio;              // Retorna tempo decorrido em segundos
}

int main() {
    // Fixar número de threads em 4 para testes consistentes
    omp_set_num_threads(4);

    printf("\n=== CONTAGEM DE NÚMEROS PRIMOS ===\n");
    printf("Número de threads fixo: %d\n", omp_get_max_threads());  // Confirma configuração

    // Array com valores de teste crescentes para demonstrar comportamento
    int valores_n[] = {1000, 10000, 100000, 1000000, 10000000};
    int num_testes = sizeof(valores_n) / sizeof(valores_n[0]);  // Calcula quantidade de testes

    printf("\n=== RESULTADOS DOS TESTES ===\n");
    // Cabeçalho da tabela de resultados com formatação alinhada
    printf("%-12s %-12s %-12s %-15s %-15s %-10s %-20s\n", "N", "Primos Seq", "Primos Par", "Tempo Seq
(s)", "Tempo Par (s)", "Speedup", "Status");
    printf("%-12s %-12s %-12s %-15s %-15s %-10s %-20s\n", "============", "============",
"============", "===============", "===============", "==========", "====================");

    // Loop principal de testes com diferentes tamanhos de problema
    for (int i = 0; i < num_testes; i++) {
        int n = valores_n[i];                    // Valor atual de n
        printf("\nTestando com n = %d...\n", n); // Feedback visual do progresso

        // Medindo tempo e resultado da versão sequencial (referência correta)
        double tempo_seq = medir_tempo(contar_primos_sequencial, n);
        int primos_seq = contar_primos_sequencial(n);  // Resultado CORRETO

        // Medindo tempo e resultado da versão paralela (com race condition)
        double tempo_par = medir_tempo(contar_primos_paralelo, n);
        int primos_par = contar_primos_paralelo(n);    // Resultado INCORRETO
```

```
        // Calculando speedup aparente (mesmo com resultados incorretos)
        double speedup = (tempo_seq > 0) ? tempo_seq / tempo_par : 0;

        // Verificação de correção comparando resultados
        const char* status = (primos_seq == primos_par) ? "CORRETO" : "ERRO - Race Condition";

        // Exibindo linha formatada com todos os resultados
        printf("%-12d %-12d %-12d %-15.6f %-15.6f %-10.2f %-20s\n",
               n, primos_seq, primos_par, tempo_seq, tempo_par, speedup, status);
    }

    return 0;  // Programa executado com sucesso
}

// Comando de compilação: gcc -fopenmp tarefa5.c -o tarefa5 -lm
```