Aproximação de π com Análise de Performance

📋 Descrição da Tarefa

Este projeto implementa um programa em C que calcula aproximações de π usando séries matemáticas, variando o número de iterações e medindo o tempo de execução. O objetivo é comparar os valores obtidos com o valor real de π e analisar como a acurácia melhora com mais processamento computacional.

> **Objetivo Principal:** Demonstrar a relação fundamental entre esforço computacional e precisão numérica, um princípio que governa desde simulações científicas até inteligência artificial moderna.

🧮 Teoria Matemática

1. Série de Leibniz

A **Série de Leibniz** é uma das fórmulas mais conhecidas para calcular π, descoberta pelo matemático alemão Gottfried Wilhelm Leibniz:

```
π/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + ...
```

Características da Série de Leibniz:

- **Convergência lenta:** O(1/n)
- **Simples de implementar:** Apenas divisões e alternância de sinais
- **Requer muitas iterações:** Para alta precisão
- **Erro aproximado:** |erro| ≈ 1/(2n+1)

2. Série de Nilakantha

A **Série de Nilakantha** oferece convergência mais rápida, desenvolvida pelo matemático indiano Nilakantha Somayaji no século XV:

$$\pi = 3 + 4/(2\times3\times4) - 4/(4\times5\times6) + 4/(6\times7\times8) - \ldots$$

Características da Série de Nilakantha:

- **Convergência rápida:** $O(1/n^3)$
- **Baseada em produtos consecutivos:** Três números consecutivos
- **Melhor precisão:** Com menos iterações
- **Erro aproximado:** $|erro| \approx 1/n^3$

⚙️ Funcionalidades Implementadas

Algoritmos de Cálculo

```
// Série de Leibniz double calculate_pi_leibniz(long long iterations) { double pi_approx = 0.0; int sign = 1; for (long long i = 0; i < iterations; i++) { pi_approx += sign * (1.0 / (2 * i + 1)); sign *= -1; } return 4.0 * pi_approx; } // Série de Nilakantha double calculate_pi_nilakantha(long long iterations) { double pi_approx = 3.0; int sign = 1; for (long long i = 1; i <= iterations; i++) { long long n = 2 * i; pi_approx += sign * (4.0 / (n * (n + 1) * (n + 2))); sign *= -1; } return pi_approx; }
```

Análise de Performance

- **measure_time():** Mede tempo de execução com precisão de clock
- **calculate_error():** Calcula erro absoluto comparado ao valor real
- **print_results():** Formata resultados em tabela organizada

Testes Automatizados

O programa realiza testes com 6 diferentes números de iterações, variando de 100 a 10,000,000, permitindo análise detalhada da convergência e performance.

📊 Resultados Esperados

Tabela de Performance

O programa gera uma tabela comparativa mostrando:

Método	Iterações	π Aproximado	Tempo (s)	Erro	Precisão
Leibniz	1,000	3.140592653590	0.000015	1.00e-03	99.9682%
Nilakantha	1,000	3.141592653590	0.000012	1.00e-08	99.9999%

Análise Teórica dos Resultados

1. Convergência:

- **Leibniz:** Convergência O(1/n) - lenta
- **Nilakantha:** Convergência O(1/n^3) - rápida

2. Complexidade Temporal:

- Ambos algoritmos: O(n) onde n = número de iterações
- Tempo cresce linearmente com iterações

3. Precisão vs Performance:

- Mais iterações = maior precisão
- Lei dos retornos decrescentes aplicada
- Trade-off entre tempo e acurácia

🌍 O Padrão de Precisão Crescente em Aplicações Reais

O comportamento observado neste projeto - onde maior esforço computacional resulta em precisão incrementalmente melhor - é um padrão fundamental que se repete em diversas aplicações críticas da computação moderna.

1. Simulações Físicas de Alta Fidelidade

Dinâmica de Fluidos Computacional (CFD)

- **10^3 células:** estimativa grosseira do arrasto
- **10^6 células:** captura turbulência básica
- **10^9 células:** resolve detalhes críticos para segurança
- **Custo:** Dias de supercomputador para cada incremento
- **Impacto:** Diferença entre aprovação e rejeição em certificação

Simulações Estruturais (Elementos Finitos)

- **Malha grosseira:** tendências gerais de tensão
- **Malha refinada:** identifica pontos de falha críticos
- **Trade-off:** Cada refinamento dobra o tempo de computação
- **Consequência:** Erro de 1% pode significar colapso estrutural

2. Inteligência Artificial e Aprendizado de Máquina

Treinamento de Modelos de Linguagem

- **GPT-1 (117M parâmetros):** texto básico
- **GPT-3 (175B parâmetros):** capacidades emergentes
- **GPT-4 (~1.7T parâmetros):** raciocínio sofisticado

- **Custo:** Crescimento exponencial de recursos

🎯 Algoritmos de Busca (AlphaGo)

- **1.000 simulações:** jogada razoável
- **100.000 simulações:** nível profissional
- **10.000.000 simulações:** superhumano
- **Escalabilidade:** Cada ordem de magnitude requer 10x mais hardware

3. Computação Científica e Pesquisa

🧬 Descoberta de Medicamentos

- **Nanosegundos:** movimentos locais
- **Microsegundos:** mudanças conformacionais
- **Milisegundos:** dobramento completo de proteínas
- **Desafio:** Cada escala temporal requer ordens de magnitude mais computação

4. Padrões Comuns e Lições Aprendidas

Lei dos Retornos Decrescentes Universais

Precisão $\propto$ log(Recursos Computacionais)

Estratégias de Otimização Emergentes:

- **Computação Adaptativa:** Alocar recursos onde precisão é mais crítica
- **Aproximações Inteligentes:** Modelos substitutos e reduced-order modeling
- **Hardware Especializado:** TPUs, FPGAs, computação quântica

💻 Compilação e Execução

Linux/macOS:

```
gcc -o tarefa3 tarefa3.c -lm ./tarefa3
```

Windows (MinGW/MSYS2):

```
gcc -o tarefa3.exe tarefa3.c -lm tarefa3.exe
```

Windows (Visual Studio):

```
cl tarefa3.c /Fe:tarefa3.exe tarefa3.exe
```

Configuração de Encoding (Windows PowerShell):

```
chcp 65001 # Define codificação UTF-8
```

🔬 Conceitos de Programação Paralela

Este projeto demonstra conceitos fundamentais para programação paralela:

1. Paralelização Potencial:

- Loop principal pode ser dividido entre threads
- Redução paralela para somar termos
- Independência entre iterações

2. Medição de Performance:

- Benchmarking preciso com clock()
- Análise de escalabilidade

- Profiling de algoritmos

3. Trade-offs Computacionais:

- Tempo vs Precisão
- Memória vs Velocidade
- Algoritmo vs Hardware

🚀 Extensões Possíveis

1. Implementação Paralela:

```
#include <omp.h> // Paralelizar com OpenMP #pragma omp parallel for reduction(+:pi_approx) for (long long i = 0; i < iterations; i++) { // Cálculo do termo }
```

2. Outros Algoritmos:

- Método de Monte Carlo
- Série de Machin
- Algoritmo de Chudnovsky
- Algoritmo de Bailey–Borwein–Plouffe

3. Análises Avançadas:

- Gráficos de convergência
- Análise estatística de erro
- Comparação com diferentes precisões (float, double, long double)

🎯 Conclusões

Este projeto ilustra princípios fundamentais da computação científica:

1. Algoritmos diferentes têm características de convergência distintas

2. Existe sempre um trade-off entre precisão e performance

3. A escolha do algoritmo pode ser mais importante que recursos computacionais

4. Medição rigorosa é essencial para otimização

Estes conceitos são aplicáveis em simulações reais, IA, computação financeira e qualquer área que demande cálculos iterativos de alta precisão.

Projeto desenvolvido para demonstrar conceitos de aproximação numérica, análise de performance e fundamentos de programação paralela.