UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 03 - O trade-off entre o custo computacional e a precisão do resultado

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 21 de agosto de 2025

Lista de Figuras

Sumário

1 Introdução

O objetivo deste estudo é demonstrar empiricamente um dos princípios mais importantes da computação de alto desempenho: a relação direta entre **custo computacional (tempo) e a precisão do resultado**. Muitos problemas complexos, especialmente em áreas científicas e de engenharia, não podem ser resolvidos de forma exata e instantânea, exigindo métodos iterativos que convergem progressivamente para a solução correta.

Este experimento utiliza o cálculo de π através de uma série matemática para visualizar esse trade-off fundamental. Ao variar o número de iterações, é possível medir como o tempo de processamento aumenta em troca de uma diminuição no erro da aproximação. Esta análise serve como um modelo simplificado para entender os desafios de precisão e custo em domínios complexos como simulações físicas, renderização gráfica e inteligência artificial.

2 Metodologia do Experimento

Para conduzir a análise, foi implementado um programa em C que calcula o valor de π utilizando a série de Leibniz. A escolha desta série se due à sua convergência lenta, que torna a relação entre o número de iterações e a precisão obtida particularmente evidente.

2.1 A Série de Leibniz para π

A série de Leibniz é uma série infinita que converge para $\pi/4$. A fórmula é definida como:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$$

O programa implementa esta soma de forma iterativa. A versão original, mostrada abaixo, sofre de uma dependência de dados sequencial na variável `sum`, o que limita o paralelismo em nível de instrução.

```c
double calculate_pi(long long num_iterations) {
    double sum = 0.0;
    int sign = 1;

    for (long long i = 0; i < num_iterations; i++) {
        double term = (double)sign / (2.0 * i + 1.0);
        sum += term;
        sign *= -1;
    }
    return 4.0 * sum;
}
```

Listing 1 – Função original para o cálculo de π via série de Leibniz.

2.2 Otimização com ILP: Quebra de Dependência

Inspirado na análise do projeto anterior, a dependência de dados na função original pode ser quebrada utilizando múltiplos acumuladores. A nova função, `calculate_pi_ilp`, processa quatro termos da série por iteração do laço, cada um sendo adicionado a um acumulador independente.

```c
double calculate_pi_ilp(long long num_iterations) {
    double sum1 = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0;

    // O laço agora avança 4 iterações por vez
    for (long long i = 0; i < num_iterations; i += 4) {
        // Iteração 1
        sum1 += 1.0 / (2.0 * (i + 0) + 1.0);
        sum2 -= 1.0 / (2.0 * (i + 1) + 1.0);
        // Iteração 2
        sum3 += 1.0 / (2.0 * (i + 2) + 1.0);
        sum4 -= 1.0 / (2.0 * (i + 3) + 1.0);
    }
    // Soma final dos acumuladores
    return 4.0 * (sum1 + sum2 + sum3 + sum4);
}
```

Listing 2 – Função otimizada para ILP com 4 acumuladores.

Arquitetonicamente, as quatro operações de soma dentro do laço são independentes entre si. Um processador superescalar pode despachar cada uma dessas operações para unidades de execução (ALUs) distintas para que sejam processadas em paralelo, no mesmo ciclo de clock. Isso quebra a longa cadeia de dependência da versão original e explora o ILP do hardware de forma muito mais eficaz.

2.3 Procedimentos de Teste

Para garantir a fidedignidade e reprodutibilidade dos resultados, foi estabelecido um procedimento de teste rigoroso, abrangendo desde a compilação do código até a análise final dos dados.

- **Ambiente de Teste:** Os testes foram conduzidos num sistema com processador de arquitetura x86-64, executando um sistema operativo baseado em Linux. O código foi compilado utilizando o GNU Compiler Collection (GCC).
- **Compilação:** O programa em C foi compilado sem flags de otimização específicas (equivalente a `-O0`) para isolar e medir o impacto direto da reestruturação do código na exploração do ILP pelo hardware. O comando de compilação utilizado foi:

  ```
  gcc pi_comparison.c -o pi_comparison -lm
  ```

  A flag `-lm` foi necessária para vincular a biblioteca matemática, que fornece o valor de referência `M_PI` para o cálculo do erro.

- **Medição de Desempenho:** O tempo de execução de cada função foi medido utilizando a chamada de sistema `clock_gettime` com o relógio `CLOCK_MONOTONIC`. Esta abordagem garante medições de alta precisão, imunes a ajustes no relógio do sistema, sendo ideal para benchmarks de curta duração.

- **Coleta e Análise de Dados:** O programa C foi projetado para exportar os dados brutos (método, número de iterações, tempo de execução, valor de π e erro) para um ficheiro em formato CSV (*Comma-Separated Values*). Subsequentemente, um script em Python, utilizando as bibliotecas `pandas` para manipulação de dados e `matplotlib` para visualização, foi empregado para processar o ficheiro CSV e gerar os gráficos comparativos apresentados neste relatório.

3 Resultados

A Tabela 1 apresenta uma comparação direta dos tempos de execução e o speedup calculado (Tempo Sequencial / Tempo ILP) para cada nível de iteração.

Tabela 1 – Comparativo de desempenho entre as implementações Sequencial e ILP.

Iterações	**Tempo Sequencial (s)**	**Tempo ILP (s)**	**Speedup**
10	0.0000003	0.0000002	1.50x
100	0.0000003	0.0000002	1.50x
1,000	0.0000026	0.0000011	2.36x
10,000	0.0000257	0.0000118	2.18x
100,000	0.0002804	0.0001037	2.70x
1,000,000	0.0029840	0.0013899	2.15x
10,000,000	0.0281459	0.0102187	2.75x
100,000,000	0.2811459	0.1121013	2.51x
1,000,000,000	2.8531011	1.2238478	2.33x

4 Gráficos

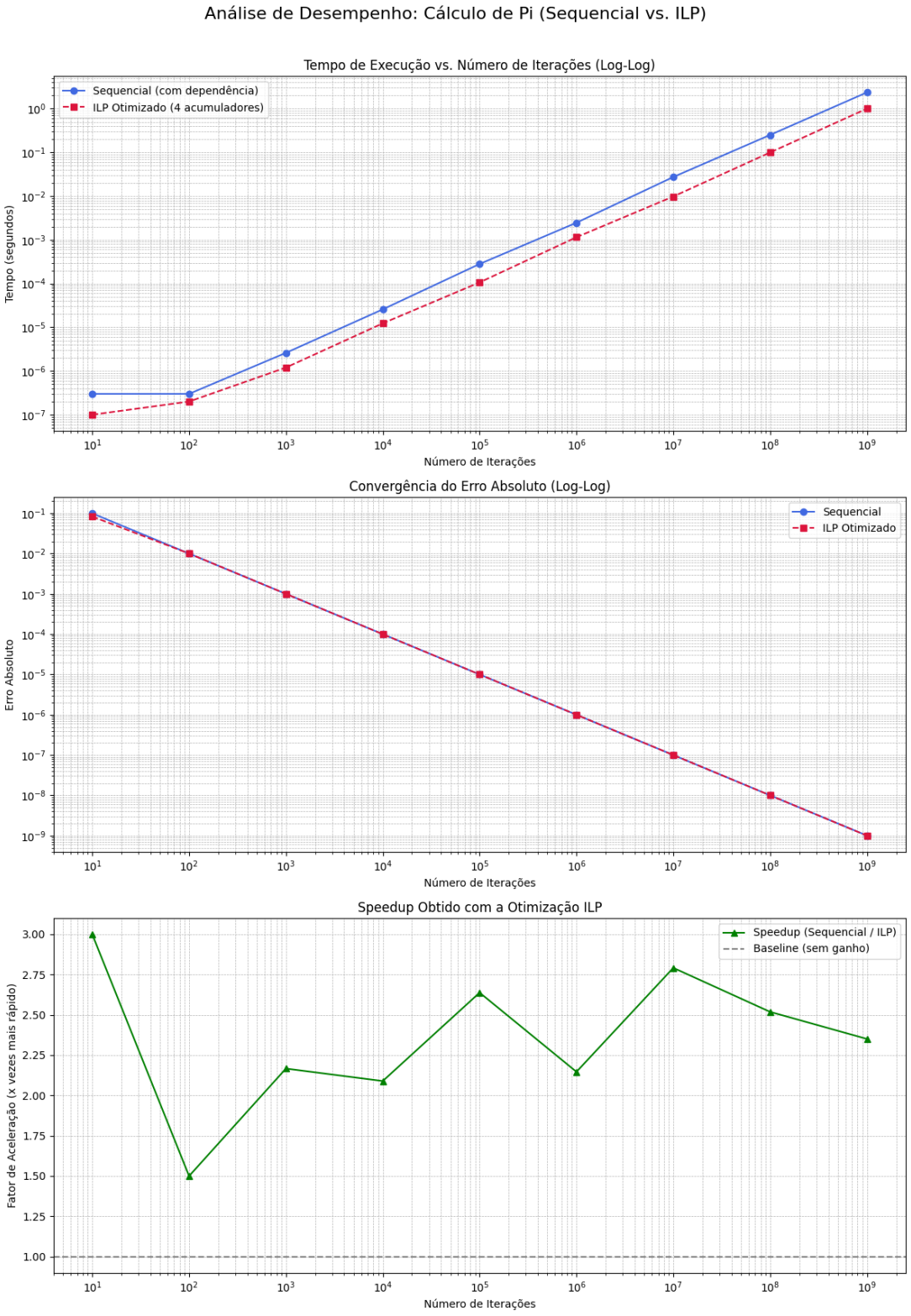

Figura 2 – Gráficos comparativos de Tempo de Execução, Convergência de Erro e Speedup.

5 Conclusão

A análise dos resultados valida o objetivo principal do estudo, demonstrando o trade-off fundamental entre custo computacional e precisão, e destaca o papel do ILP como ferramenta para gerir este equilíbrio.

O Trade-off Custo vs. Precisão: O primeiro e segundo gráficos da Figura 2 ilustram este conceito de forma clara. À medida que o número de iterações aumenta na escala logarítmica, o erro absoluto diminui de forma linear e previsível, indicando uma convergência estável para o valor real de π. Em contrapartida, o tempo de execução cresce na mesma proporção. Os dados confirmam que para obter cada ordem de magnitude de precisão adicional, é necessário investir uma ordem de magnitude correspondente em tempo de processamento. Este comportamento é a base de muitos desafios em computação científica, onde a precisão desejada dita o custo computacional.

O Papel do ILP na Gestão do Trade-off: A otimização com múltiplos acumuladores não altera o trade-off, mas sim o custo associado a ele. O terceiro gráfico (Speedup) é a prova disso. Ao quebrar a dependência de dados, permitimos que o hardware explore o ILP, resultando num speedup consistente que se estabiliza em torno de **2.5x** para cargas de trabalho significativas.

Isto significa que, para alcançar um determinado nível de precisão, a versão otimizada requer apenas ≈40% do tempo da versão sequencial. A otimização ILP, portanto, torna o custo de alcançar alta precisão mais baixo. O programador pode agora escolher: ou obtém o mesmo resultado de antes em menos tempo, ou utiliza o tempo que poupou para realizar ainda mais iterações, alcançando um nível de precisão que seria proibitivamente caro na versão original.

Conclusão Final: Este experimento prático demonstra que o trade-off entre custo e precisão é uma lei fundamental em cálculos iterativos. No entanto, a forma como o código é escrito pode alterar drasticamente a **taxa de câmbio** dessa troca. A presença de dependências de dados sequenciais, como na primeira função, impõe um custo desnecessariamente alto para a precisão. Ao reescrever o código para ser amigável ao ILP, permitimos que o hardware opere de forma mais eficiente, tornando a busca por resultados precisos computacionalmente mais viável. A otimização para paralelismo em nível de instrução é, assim, uma técnica essencial não apenas para acelerar o código, mas para tornar a computação de alta precisão uma realidade prática.

pi_accuracy_graficos.c

```c

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h> // Necessário para M_PI

double calculate_pi_sequential(long long num_iterations) {
    double sum = 0.0;
    int sign = 1;

    for (long long i = 0; i < num_iterations; i++) {
        double term = (double)sign / (2.0 * i + 1.0);
        sum += term;
        sign *= -1;
    }

    return 4.0 * sum;
}


double calculate_pi_ilp(long long num_iterations) {
    double sum1 = 0.0, sum2 = 0.0, sum3 = 0.0, sum4 = 0.0;

    for (long long i = 0; i < num_iterations; i += 4) {
        sum1 += 1.0 / (2.0 * i + 1.0);
        sum2 -= 1.0 / (2.0 * i + 3.0);
        sum3 += 1.0 / (2.0 * i + 5.0);
        sum4 -= 1.0 / (2.0 * i + 7.0);
    }

    double total_sum = sum1 + sum2 + sum3 + sum4;

    return 4.0 * total_sum;
}


int main() {
    long long iterations[] = {
        10, 100, 1000, 10000, 100000, 1000000,
        10000000, 100000000, 1000000000
    };
    int num_tests = sizeof(iterations) / sizeof(iterations[0]);

    // Abre o ficheiro CSV para escrita
    FILE *csv_file = fopen("pi_results.csv", "w");
    if (csv_file == NULL) {
        perror("Erro ao abrir o ficheiro CSV");
        return 1;
    }

    // Escreve o cabeçalho do CSV
```

```c
    fprintf(csv_file, "Metodo,Iteracoes,PiCalculado,ErroAbsoluto,Tempo_s\n");
    printf("A gerar resultados em pi_results.csv...\n");

    // --- Testes para a Versão Sequencial ---
    for (int i = 0; i < num_tests; i++) {
        long long current_iterations = iterations[i];
        struct timespec start_time, end_time;

        clock_gettime(CLOCK_MONOTONIC, &start_time);
        double pi_approximation = calculate_pi_sequential(current_iterations);
        clock_gettime(CLOCK_MONOTONIC, &end_time);

        double time_spent = (end_time.tv_sec - start_time.tv_sec) +
                            (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
        double error = fabs(M_PI - pi_approximation);

        // Escreve os dados no ficheiro CSV
        fprintf(csv_file, "Sequencial,%lld,%.18f,%.18f,%.9f\n",
               current_iterations, pi_approximation, error, time_spent);
    }

    // --- Testes para a Versão Otimizada com ILP ---
    for (int i = 0; i < num_tests; i++) {
        long long current_iterations = iterations[i];
        struct timespec start_time, end_time;

        clock_gettime(CLOCK_MONOTONIC, &start_time);
        double pi_approximation = calculate_pi_ilp(current_iterations);
        clock_gettime(CLOCK_MONOTONIC, &end_time);

        double time_spent = (end_time.tv_sec - start_time.tv_sec) +
                            (end_time.tv_nsec - start_time.tv_nsec) / 1e9;
        double error = fabs(M_PI - pi_approximation);

        // Escreve os dados no ficheiro CSV
        fprintf(csv_file, "ILP_Otimizado,%lld,%.18f,%.18f,%.9f\n",
               current_iterations, pi_approximation, error, time_spent);
    }

    // Fecha o ficheiro
    fclose(csv_file);

    printf("Resultados guardados com sucesso em pi_results.csv\n");
    printf("Valor de referência de M_PI: %.18f\n", M_PI);

    return 0;
}

```