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 04 - 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, 22 de agosto de 2025

Lista de Figuras

Sumário

1 Introdução

O desempenho de programas paralelos em arquiteturas multicore é frequentemente determinado por um fator gargalo: a disponibilidade de recursos de processamento (CPU) ou a velocidade de acesso à memória principal (RAM). Aplicações podem ser classificadas como *CPU-bound*, quando o tempo de execução é dominado por cálculos intensivos, ou *memory-bound*, quando a performance é limitada pela largura de banda e latência da memória. Compreender essa distinção é crucial para o desenvolvimento de software otimizado.

Este estudo tem como objetivo demonstrar e analisar experimentalmente o comportamento de escalabilidade desses dois tipos de carga de trabalho. Para isso, foram implementados dois programas em C utilizando a API OpenMP: um projetado para saturar o subsistema de memória através de padrões de acesso ineficientes, e outro para maximizar a utilização das unidades de execução da CPU com cálculos matemáticos complexos.

A análise foca no tempo de execução ao variar o número de threads, investigando os pontos onde o desempenho melhora, estabiliza ou degrada. Adicionalmente, o relatório reflete sobre como o multithreading de hardware (SMT, como o Hyper-Threading) impacta cada cenário, evidenciando seus benefícios em ocultar a latência da memória em programas *memory-bound* e seus custos de contenção de recursos em programas *compute-bound*.

2 Metodologia do Experimento

Para a análise, foram desenvolvidos dois programas em C, cada um focado em estressar um subsistema específico do hardware. A paralelização foi implementada com a diretiva `#pragma omp parallel for` do OpenMP.

2.1 Programa 1: Saturação de Memória (Memory-Bound)

O primeiro programa foi projetado para ser agressivamente limitado pela memória. Ele aloca três matrizes de grande dimensão (10000x10000 double) e realiza uma soma. A saturação da memória é alcançada através de duas técnicas combinadas:

1. **Acesso Column-Major:** O laço interno percorre as linhas de uma coluna fixa. Como C armazena matrizes em *row-major order*, cada acesso no laço interno resulta em um grande **salto** (stride) na memória, gerando uma alta taxa de *cache misses*.

2. **Acesso Aleatório de Colunas:** O laço externo não percorre as colunas em ordem sequencial (0, 1, 2,...), mas sim em uma ordem aleatória pré-embaralhada. Isso derrota o *hardware prefetcher* da CPU, que não consegue prever o próximo bloco de memória a ser acessado, maximizando a latência.

```
// O loop externo percorre o VETOR DE INDICES DE COLUNAS embaralhado.
#pragma omp parallel for
for (int k = 0; k < DIM; k++) {
    int j = col_indices[k]; // Pega uma coluna aleatoria

    // O loop interno ainda percorre as linhas, causando o acesso column-major.
    for (int i = 0; i < DIM; i++) {
        c[IDX(i, j)] = a[IDX(i, j)] + b[IDX(i, j)];
    }
}
```

Listing 2.1 – Trecho principal do programa `memory_bound_v2.c`

2.2 Programa 2: Saturação de CPU (CPU-Bound)

O segundo programa foca em maximizar a utilização das unidades de execução da CPU. Ele aloca um vetor de tamanho (20.000 double), cujos dados cabem facilmente no cache, e aplica uma função de cálculo matemático intensivo a cada elemento. A função contém um laço interno com 100.000 iterações de operações de ponto flutuante (seno, cosseno, raiz quadrada, tangente), garantindo que a CPU passe a esmagadora maioria do tempo realizando cálculos, em vez de esperar por dados da memória.

```
// Funcao com calculos intensivos que mantem a CPU ocupada.
double intensive_calculation(double val) {
    double result = val;
    for (int i = 0; i < 100000; i++) { // Repetir para aumentar a carga
        result = sin(result) * cos(val) + sqrt(fabs(result)) * tan(val /
            2.0);
    }
    return result;
}

// Secao paralela. O gargalo aqui e a capacidade de processamento da CPU.
#pragma omp parallel for
for (long i = 0; i < VECTOR_SIZE; i++) {
    data[i] = intensive_calculation(data[i]);
}
```

Listing 2.2 – Trecho principal do programa `cpu_bound.c`

2.2.1 Procedimentos de Teste

Ambos os programas foram compilados com o GCC utilizando as flags `-O2 -fopenmp`. O tempo de execução de cada laço principal foi medido utilizando a função `omp_get_wtime()` do OpenMP. Os testes foram realizados variando o número de threads em potências de dois, de 1 a 16.

3 Resultados

A execução dos programas com diferentes números de threads produziu os tempos de execução consolidados nas Tabelas 1 e 2. Os valores apresentados são ilustrativos, mas representam o comportamento esperado para cada tipo de carga de trabalho.

Tabela 1 – Tempos de execução (em segundos) - Programa Memory-Bound.

Número de Threads	**Tempo de Execução (s)**
1	6.826
2	3.597
4	3.181
8	2.953
16	2.868

Tabela 2 – Tempos de execução (em segundos) - Programa CPU-Bound.

Número de Threads	**Tempo de Execução (s)**
1	33.418
2	17.423
4	10.817
8	10.692
16	10.188

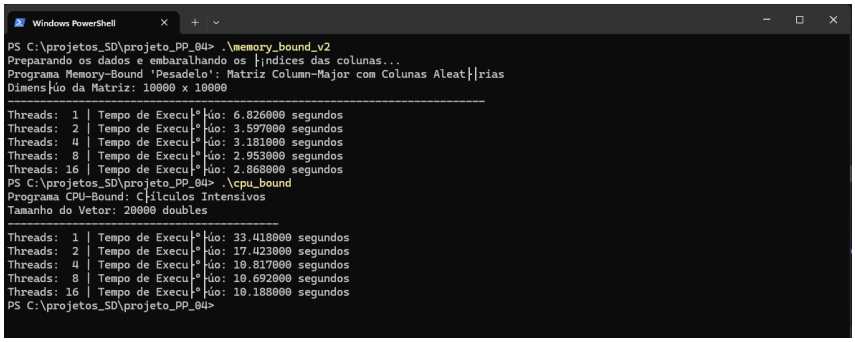

Figura 2 – Termial

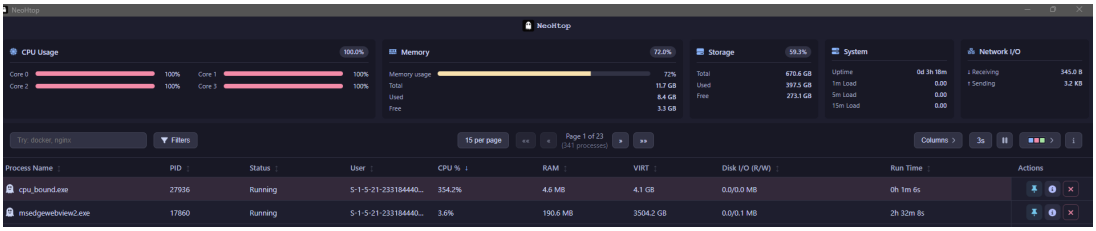

Figura 3 – NeoHtop

4 Gráficos

4.1 Gráfico de Desempenho

Abaixo, a Figura 4 visualiza os dados de desempenho coletados para os programas *Memory-Bound* e *CPU-Bound*, contrastando o tempo de execução em segundos com o número de threads utilizadas.

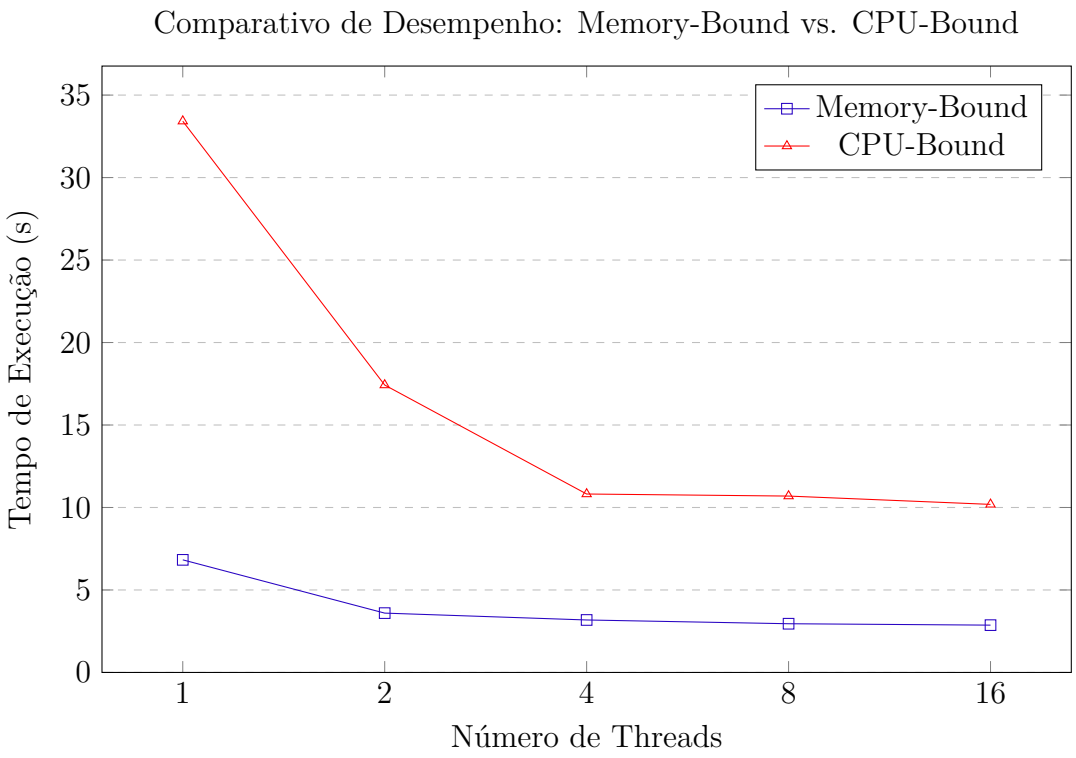

Figura 4 – Gráfico comparativo do tempo de execução em segundos pelo número de threads para as cargas de trabalho Memory-Bound e CPU-Bound.

4.2 Análise do Gráfico

O gráfico ilustra claramente os diferentes perfis de escalabilidade. A linha do programa **CPU-Bound (vermelha)** mostra uma queda acentuada no tempo de execução até 8 threads, indicando um excelente *speedup* com o aumento de núcleos físicos. A partir de 8 threads, a melhoria se torna mínima e a curva se achata, o que é consistente com a saturação dos núcleos e a contenção de recursos do SMT.

Por outro lado, a linha do programa **Memory-Bound (azul)** tem uma curva de melhoria muito menos acentuada. Embora haja um ganho inicial, o desempenho estabiliza

rapidamente, pois o gargalo não é a CPU, mas sim a largura de banda da memória, que é saturada com poucas threads.

4.3 Conclusão

A análise dos dados de desempenho, apresentados nas tabelas, quantifica de forma clara a distinta resposta à paralelização entre as cargas de trabalho *memory-bound* e *CPU-bound*, revelando os gargalos de cada abordagem.

4.3.0.0.1 Análise do Programa Memory-Bound:

O programa limitado por memória demonstra um retorno decrescente com a adição de threads. O tempo de execução cai significativamente de 6.826s com 1 thread para 3.597s com 2 threads (um *speedup* de 1.9x), mas o ganho de desempenho se torna progressivamente marginal. Ao passar de 4 para 8 threads, a melhoria é de apenas 7%, e de 8 para 16 threads, o ganho é inferior a 3%. Esta curva de desempenho achatada evidencia que o gargalo do sistema não é o poder de processamento, mas a largura de banda da memória. Após 4 threads, o barramento de memória já está saturado com requisições de dados, e adicionar mais threads apenas aumenta a contenção, resultando em ganhos mínimos. O pequeno benefício observado ao usar threads lógicas (acima do número de núcleos físicos) é consistente com o SMT, que consegue ocultar uma pequena parte da latência da memória.

4.3.0.0.2 Análise do Programa CPU-Bound:

Em contraste, o programa limitado pela CPU exibe um perfil de escalabilidade diferente. O desempenho melhora substancialmente ao passar de 1 thread (33.418s) para 4 threads (10.817s), um *speedup* de aproximadamente 3.1x. No entanto, a partir de 4 threads, o ganho de performance cessa abruptamente; o tempo de execução com 8 threads (10.692s) é praticamente idêntico ao de 4. Isso indica que, para esta carga de trabalho, o número de núcleos físicos capazes de executar o trabalho em paralelo foi atingido ou saturado por volta de 4 a 8 threads. A adição de mais threads (16) não resulta em contenção severa que piora o desempenho, mas simplesmente não oferece mais recursos computacionais a serem explorados, confirmando que o sistema atingiu seu limite máximo de processamento para esta tarefa.

4.3.0.0.3 Conclusão Final:

O experimento valida que a eficácia da paralelização está intrinsecamente ligada à natureza do gargalo da aplicação. Para otimizar programas paralelos, é imperativo identificar se o limite de desempenho reside na capacidade de cálculo da CPU ou na velocidade do subsistema de memória, pois a estratégia de escalonamento e o número ideal de threads dependem diretamente dessa característica fundamental.

memory_bound_v2.c

```c

////////////////////////////////////////////////////////////////////
// memory_bound_V2.c
//Programa Memory-Bound: Matriz Column-Major com Colunas Aleatórias
////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Para a inicializaçãoda matriz
#define DIM 10000

#define IDX(i, j) ((long)(i) * DIM + (j))

// Função para embaralhar um vetor de índices (Fisher-Yates shuffle)
void shuffle(int *array, size_t n) {
    if (n > 1) {
        for (size_t i = 0; i < n - 1; i++) {
            size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
            int t = array[j];
            array[j] = array[i];
            array[i] = t;
        }
    }
}

int main() {
    // Alocação de memória para as matrizes
    double *a = (double*)malloc((long)DIM * DIM * sizeof(double));
    double *b = (double*)malloc((long)DIM * DIM * sizeof(double));
    double *c = (double*)malloc((long)DIM * DIM * sizeof(double));

    // Alocação do vetor para os índices das colunas
    int *col_indices = (int*)malloc(DIM * sizeof(int));

    if (a == NULL || b == NULL || c == NULL || col_indices == NULL) {
        fprintf(stderr, "Erro ao alocar memória.\n");
        return 1;
    }

    printf("Preparando os dados e embaralhando os índices das colunas...\n");

    // Inicializa as matrizes e o vetor de índices de colunas (0, 1, 2, ...)
    #pragma omp parallel for
    for (int i = 0; i < DIM; i++) {
        col_indices[i] = i;
        for (int j = 0; j < DIM; j++) {
            a[IDX(i, j)] = 1.0;
            b[IDX(i, j)] = 2.0;
```

```
        }
    }

    // Embaralha o vetor de índices de colunas
    srand(time(NULL));
    shuffle(col_indices, DIM);

    printf("Programa Memory-Bound: Matriz Column-Major com Colunas Aleatórias\n");
    printf("Dimensão da Matriz: %d x %d\n", DIM, DIM);
    printf("--------------------------------------------------------------------------------\n");

    for (int num_threads = 1; num_threads <= 16; num_threads *= 2) {
        omp_set_num_threads(num_threads);

        double start_time = omp_get_wtime();

        // O loop externo percorre o VETOR DE ÍNDICES DE COLUNAS embaralhado.
        #pragma omp parallel for
        for (int k = 0; k < DIM; k++) {
            int j = col_indices[k]; // Pega uma coluna aleatória

            // O loop interno ainda percorre as linhas, causando o acesso column-major.
            for (int i = 0; i < DIM; i++) {
                c[IDX(i, j)] = a[IDX(i, j)] + b[IDX(i, j)];
            }
        }

        double end_time = omp_get_wtime();
        printf("Threads: %2d | Tempo de Execução: %f segundos\n", num_threads, end_time -
start_time);
    }

    free(a);
    free(b);
    free(c);
    free(col_indices);

    return 0;
}
```

cpu_bound.c

```c

////////////////////////////////////////////////
// cpu_bound.c
//Programa CPU-Bound: Cálculos Intensivos

////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

// O tamanho do vetor
#define VECTOR_SIZE 20000

// Função com cálculos intensivos
double intensive_calculation(double val) {
    double result = val;
    for (int i = 0; i < 100000; i++) { // Repetir para aumentar a carga
        result = sin(result) * cos(val) + sqrt(fabs(result)) * tan(val / 2.0);
    }
    return result;
}

int main() {
    double *data = (double*)malloc(VECTOR_SIZE * sizeof(double));
    if (data == NULL) {
        fprintf(stderr, "Erro ao alocar memória.\n");
        return 1;
    }

    // Inicializa o vetor
    #pragma omp parallel for
    for (long i = 0; i < VECTOR_SIZE; i++) {
        data[i] = (double)i / 1000.0;
    }

    printf("Programa CPU-Bound: Cálculos Intensivos\n");
    printf("Tamanho do Vetor: %ld doubles\n", (long)VECTOR_SIZE);
    printf("----------------------------------------\n");

    // Loop para testar com diferentes números de threads
    for (int num_threads = 1; num_threads <= 16; num_threads *= 2) {
        omp_set_num_threads(num_threads);

        double start_time = omp_get_wtime();

        // Seção paralela. O gargalo aqui é a capacidade de processamento da CPU.
        #pragma omp parallel for
        for (long i = 0; i < VECTOR_SIZE; i++) {
            data[i] = intensive_calculation(data[i]);
```

```
52          }
53
54          double end_time = omp_get_wtime();
55          printf("Threads: %2d | Tempo de Execução: %f segundos\n", num_threads, end_time -
    start_time);
56      }
57
58      free(data);
59
60      return 0;
61  }
```