Impacto da Localidade de Dados na Performance

Localidade Temporal e Espacial de Cache

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)
Disciplina de Programação Paralela - DCA3703

20 de agosto de 2025

Introdução

Objetivo do Estudo

Demonstrar e quantificar o impacto da localidade de dados no desempenho de algoritmos computacionais.

Questão Central

Identificar o ponto a partir do qual o padrão de acesso à memória sequencial (linha a linha) versus não sequencial (coluna a coluna) causa uma divergência significativa no tempo de execução.

Mecanismo Investigado

O fenômeno está diretamente ligado à arquitetura e ao funcionamento da hierarquia de memória cache do processador.

Metodologia do Experimento

- **Implementação:** Duas versões da multiplicação de matriz por vetor ($y = A \cdot x$) foram desenvolvidas em C.
- **Diferença Chave:** A única alteração entre as versões é a ordem dos laços de repetição, forçando diferentes padrões de acesso à memória.
- **Ambiente:** O código foi compilado com GCC, usando o nível de otimização `-O2` e a flag `-fopenmp`.
- **Coleta de Dados:**
  - Foram testadas matrizes quadradas ($N \times N$) com N variando de 32 a 16384.
  - O tempo de execução medido foi a **mediana** de múltiplas execuções para garantir robustez.

Padrões de Acesso à Matriz

Função 1: Acesso por Linhas (Cache-Friendly)

O laço interno percorre as colunas, resultando em acessos a endereços de memória contíguos.

```
void multiply_matrix_vector(
    int rows, int cols,
    double **A, double *x, double *y) {
  for (int i = 0; i < rows; i++) {
    y[i] = 0.0;
    // Laço interno percorre colunas (j)
    for (int j = 0; j < cols; j++) {
      y[i] += A[i][j] * x[j];
    }
  }
}
```

Padrões de Acesso à Matriz

Função 2: Acesso por Colunas (Cache-Unfriendly)

O laço interno percorre as linhas, resultando em acessos não sequenciais e distantes na memória.

```
void multiply_matrix_vector_cols_outer(
    int rows, int cols,
    double **A, double *x, double *y) {
  for (int i = 0; i < rows; i++) {
    y[i] = 0.0;
  }
  // Laço externo percorre colunas (j)
  for (int j = 0; j < cols; j++) {
    // Laço interno percorre linhas (i)
    for (int i = 0; i < rows; i++) {
      y[i] += A[i][j] * x[j];
    }
  }
}
```

Hipótese: O Papel da Cache

Hipótese Principal

O acesso por linhas terá um desempenho superior devido ao princípio da **localidade espacial**.

Mecanismo de Cache

A CPU não lê dados da RAM byte a byte. Ela carrega blocos contíguos chamados *cache lines* (tipicamente 64 bytes).

- **Acesso por Linhas (Cache Hit):** Ao acessar `A[i][0]`, os elementos vizinhos (`A[i][1]`, `A[i][2]`, etc.) são carregados juntos na cache. As próximas iterações encontram os dados já disponíveis na memória ultrarrápida.
- **Acesso por Colunas (Cache Miss):** O acesso a `A[i+1][j]` após `A[i][j]` requer um bloco de memória completamente diferente , forçando a CPU a buscar uma nova *cache line* da RAM e causando degradação de performance.

Ambiente de Teste: Hierarquia de Cache

Especificações da Máquina de Teste

Os dados de cache do sistema onde o experimento foi executado são:

- **Cache L1:** 96 KB
- **Cache L2:** 2.5 MB
- **Cache L3:** 6 MB

Estimativa de Uso de Memória por Matriz

- **N=512 (2 MB):** Cabe confortavelmente na cache L2 (2.5 MB).
- **N=1024 (8 MB):** Excede a capacidade total da cache L3 (6 MB).
- **N=2048 (32 MB):** Excede em muito a capacidade de todos os níveis de cache.

Impacto do Padrão de Acesso à Memória (Escala Logarítmica)

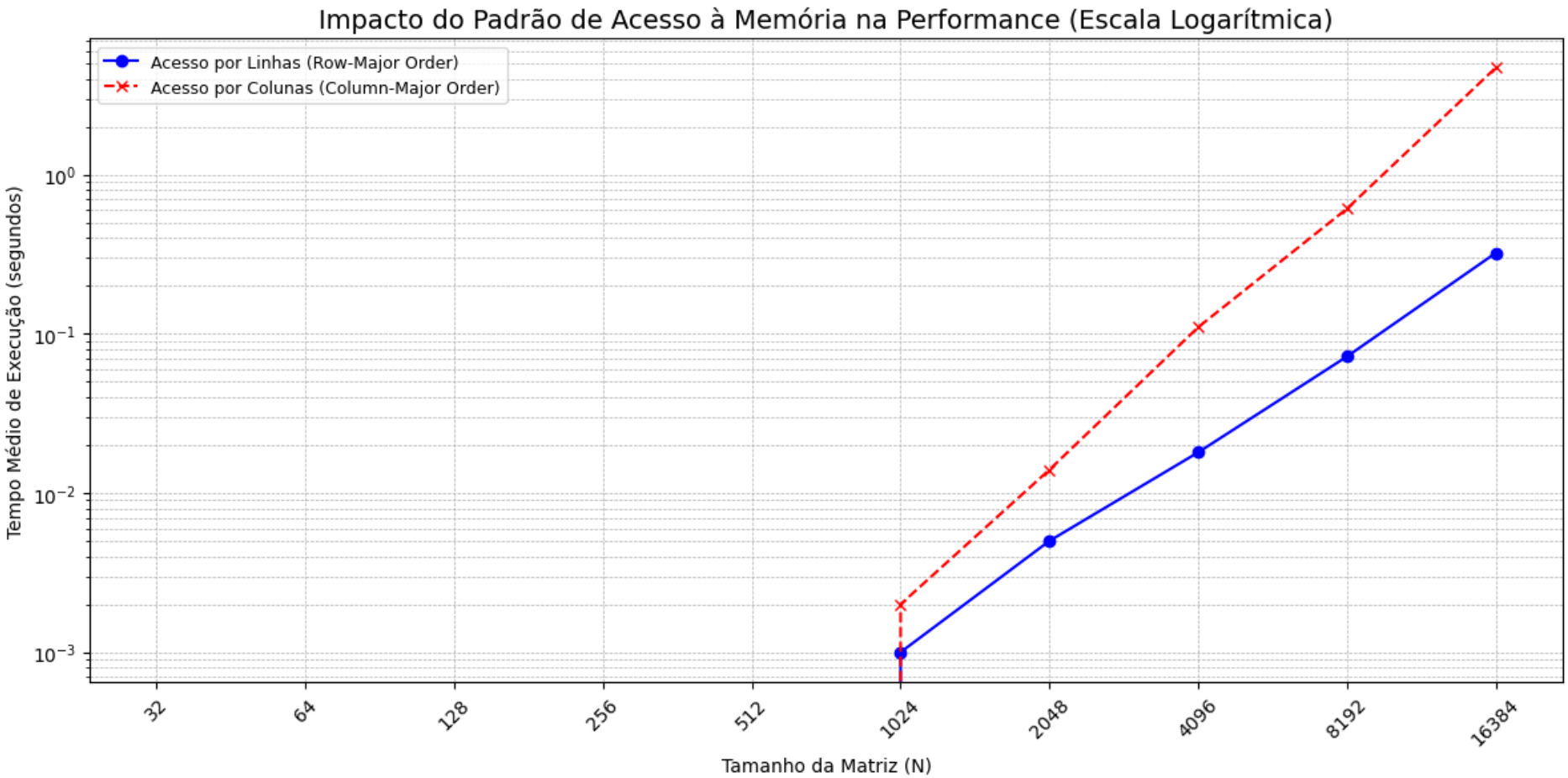

Figure: Comparação do tempo de execução mediano para acesso por linhas e colunas.

Crescimento do Fator de Lentidão vs. Tamanho da Matriz

Métrica: Fator de Lentidão

Uma forma de normalizar os resultados para análise, calculada como:

$$\text{Fator de Lentidão} = \frac{\text{Tempo do método por Colunas}}{\text{Tempo do método por Linhas}}$$

Um fator de 5.0, por exemplo, significa que o acesso por colunas foi 5 vezes mais lento.

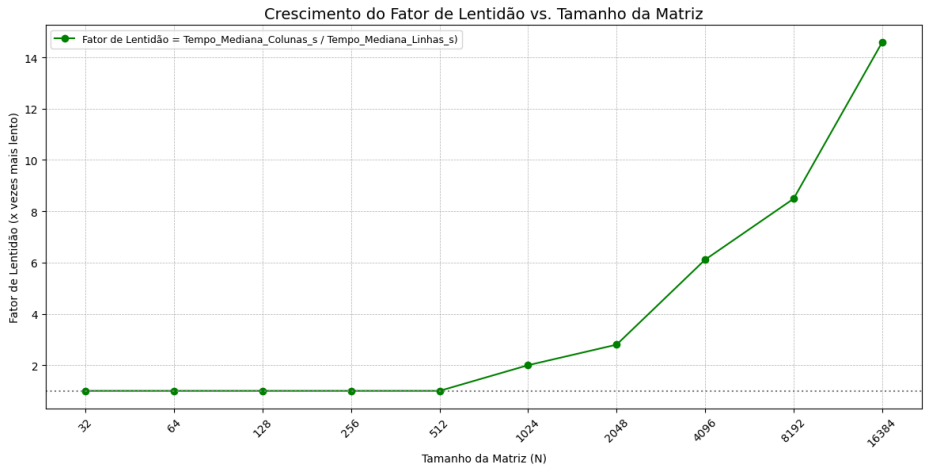

Figure: O fator de lentidão demonstra o custo crescente do acesso não otimizado à memória.

Conclusão

Hipótese Validada

O estudo validou experimentalmente que o desempenho é profundamente influenciado pela maneira como os dados são acessados na memória.

Correlação Direta com a Hierarquia de Cache

A análise dos dados revelou a causa da divergência de performance:

- **Início da Divergência:** Ocorre de forma mensurável quando a matriz excede a capacidade do **cache L2**.
- **Agravamento Drástico:** A degradação se agrava drasticamente quando o volume de dados ultrapassa a capacidade do **cache L3**.
- **Colapso da Performance:** Neste ponto, o acesso não sequencial resulta em falhas de cache constantes, tornando a alta latência da **memória RAM** o principal gargalo de velocidade.

mxv_teste_grafico_v1.c

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

// Função de comparação para usar com qsort em um array de doubles.
int compare_doubles(const void *a, const void *b) {
    double da = *(const double *)a;
    double db = *(const double *)b;
    if (da > db) return 1;
    if (da < db) return -1;
    return 0;
}

//Aloca dinamicamente uma matriz 2D.

double **create_matrix(int rows, int cols) {
    // Aloca um array de ponteiros (um para cada linha)
    double **matrix = (double **)malloc(rows * sizeof(double *));
    if (matrix == NULL) return NULL;

    // Para cada ponteiro de linha, aloca a memória para as colunas
    for (int i = 0; i < rows; i++) {
        matrix[i] = (double *)malloc(cols * sizeof(double));
        if (matrix[i] == NULL) {
            // Se falhar no meio, libera o que já foi alocado
            for(int k = 0; k < i; k++) free(matrix[k]);
            free(matrix);
            return NULL;
        }
    }
    return matrix;
}

// Libera a memória de uma matriz alocada dinamicamente.

void free_matrix(int rows, double **matrix) {
    if (matrix == NULL) return;
    // Primeiro, libera a memória de cada linha
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    // Finalmente, libera o array de ponteiros
    free(matrix);
}

// Preenche uma matriz e um vetor com valores aleatórios.

void fill_random_data(int rows, int cols, double **matrix, double *vector) {
    // Usa o tempo atual como semente para o gerador de números aleatórios
    srand(time(NULL));
```

```c
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = (double)(rand() % 10); // Números aleatórios de 0 a 9
        }
    }
    for (int i = 0; i < cols; i++) {
        vector[i] = (double)(rand() % 10);
    }
}

// Executa a multiplicação de matriz por vetor (y = A * x).

void multiply_matrix_vector(int rows, int cols, double **A, double *x, double *y) {
    for (int i = 0; i < rows; i++) {
        y[i] = 0.0; // Garante que o valor inicial seja zero
        for (int j = 0; j < cols; j++) {
            y[i] += A[i][j] * x[j];
        }
    }
}

void multiply_matrix_vector_cols_outer(int rows, int cols, double **A, double *x, double *y) {
    for (int i = 0; i < rows; i++) {
        y[i] = 0.0;
    }

    // 2. O laço externo agora itera sobre as COLUNAS (j) da matriz A.
    for (int j = 0; j < cols; j++) {
        // O laço interno itera sobre as LINHAS (i).
        for (int i = 0; i < rows; i++) {
            y[i] += A[i][j] * x[j];
        }
    }
}

int main() {
    int n_inicial = 32;
    int n_final = 16384;
    int n_passo = 2;

    double tempo_mediana_linhas, tempo_mediana_colunas;
    double fator_lentidao;

    // Cabeçalho do CSV
    printf("Tamanho_N,Tempo_Mediana_Linhas_s,Tempo_Mediana_Colunas_s,Fator_Lentidao\n");

    for (int N = n_inicial; N <= n_final; N *= n_passo) {

        int M = N;

        double **A = create_matrix(M, N);
        double *x = (double *)malloc(N * sizeof(double));
        double *y = (double *)malloc(M * sizeof(double));
```

```
105
106         if (A == NULL || x == NULL || y == NULL) {
107             fprintf(stderr, "Falha ao alocar para N=%d\n", N);
108             continue;
109         }
110
111         fill_random_data(M, N, A, x);
112
113         int repeticoes = 1001;
114         if (N >= 512) repeticoes = 51;
115         if (N >= 1024) repeticoes = 11;
116         if (N >= 2048) repeticoes = 5;
117
118         //Alocar arrays para armazenar os tempos de cada repetição ---
119         double *tempos_linhas = (double *)malloc(repeticoes * sizeof(double));
120         double *tempos_colunas = (double *)malloc(repeticoes * sizeof(double));
121         if (tempos_linhas == NULL || tempos_colunas == NULL) {
122              fprintf(stderr, "Falha ao alocar arrays de tempo para N=%d\n", N);
123              free_matrix(M, A); free(x); free(y);
124              continue;
125         }
126
127         // Teste 1: Acesso por Linhas (Coletando tempos individuais)
128         for(int r = 0; r < repeticoes; r++) {
129             double start_time = omp_get_wtime();
130             multiply_matrix_vector(M, N, A, x, y);
131             double end_time = omp_get_wtime();
132             tempos_linhas[r] = end_time - start_time;
133         }
134
135         // Teste 2: Acesso por Colunas (Coletando tempos individuais)
136         for(int r = 0; r < repeticoes; r++) {
137             double start_time = omp_get_wtime();
138             multiply_matrix_vector_cols_outer(M, N, A, x, y);
139             double end_time = omp_get_wtime();
140             tempos_colunas[r] = end_time - start_time;
141         }
142
143         // Ordena os tempos do acesso por linhas
144         qsort(tempos_linhas, repeticoes, sizeof(double), compare_doubles);
145
146         // Ordena os tempos do acesso por colunas
147         qsort(tempos_colunas, repeticoes, sizeof(double), compare_doubles);
148
149         // Calcula a mediana (pegando o elemento do meio do array ordenado)
150         tempo_mediana_linhas = tempos_linhas[repeticoes / 2];
151         tempo_mediana_colunas = tempos_colunas[repeticoes / 2];
152
153         // Calcula o fator de lentidão
154         if (tempo_mediana_linhas > 0) {
155             fator_lentidao = tempo_mediana_colunas / tempo_mediana_linhas;
156         } else {
157             fator_lentidao = 1.0;
158         }
```

```
159
160         // Imprime a linha de dados CSV
161         printf("%d,%.12f,%.12f,%.3f\n", N, tempo_mediana_linhas, tempo_mediana_colunas,
    fator_lentidao);
162
163         // --- MODIFICAÇÃO 3: Liberar a memória dos arrays de tempo ---
164         free_matrix(M, A);
165         free(x);
166         free(y);
167         free(tempos_linhas);
168         free(tempos_colunas);
169     }
170
171     return 0;
172 }
```