Computação de Alto Desempenho - COC472

Trabalho Prático 1

Antonny Victor da Silva, DRE: 120031917

14 de maio de 2023

1 Introdução

Este relatório consiste na apresentação da implementação do Trabalho Prático 1 da disciplina Computação de Alto Desempenho (CAD), contendo seções explicativas sobre as tomadas de decisão, casos de teste e explicações de partes centrais do código que pode ser encontrado no link abaixo.
Código-Fonte: `https://github.com/antonnyvictor18/Computaca-de-Alto-Desempenho/tree/main/TP1`

2 Descrição Geral do Trabalho Prático 1

O primeiro trabalho consiste na implementação de um código que realize as devidas operações para realização do produto matriz - vetor, $A \times x = b$. A implementação foi feita nas linguagens C e Fortran com o intuito de investigar como a performance do código é afetada ao mudar a ordem com que os loops aninhados são realizados nas duas linguagens.

3 Implementação em C

Para realizar a tarefa, criei duas funções para calcular o sistema linear: uma da forma padrão (acessando linha por linha da matriz A e multiplicando pelos elementos do vetor x) e da forma invertida (acessando coluna por coluna da matriz A e multiplicando pelos elementos do vetor x). Segue o exemplo abaixo:

```c
void matrix_vector_produto(double** A, double *x, double *b, int *n) {
    for (int i = 0; i < (*n); i++) {
        for (int j = 0; j < (*n); j++) {
            b[i] += A[i][j] * x[j];
        }
    }
}

void matrix_vector_produto_invertido(double** A, double *x, double *b,
    int *n) {
    for (int j = 0; j < (*n); j++) {
        for (int i = 0; i < (*n); i++) {
            b[i] += A[i][j] * x[j];
        }
    }
}
```

Além disso, para estimar a quantidade de memória livre do meu sistema, eu consultei essa informação com a ajuda de dos métodos diponibilizados pela biblioteca sys/sysinfo.h. Em seguida, usei a seguinte relação: $n^2 \times 8bytes + 2n \times 8bytes = (n^2 + 16n)bytes$ para estimar o tamanho máximo que o meu sistema pode ter, assumindo que um dado double possui

tamanho de 8 bytes.
Resolvendo essa equação, teremos que o tamanho máximo n será: $n = -8 + \frac{\sqrt{64+mem\_livre}}{2}$

```
struct sysinfo info;
sysinfo(&info);
long long int max_size = info.freeram;
printf("Quantidade de RAM livre: %lld\n",max_size);
srand(time(NULL));
int n_max = sqrt((max_size / 4) / (sizeof(double)));
```

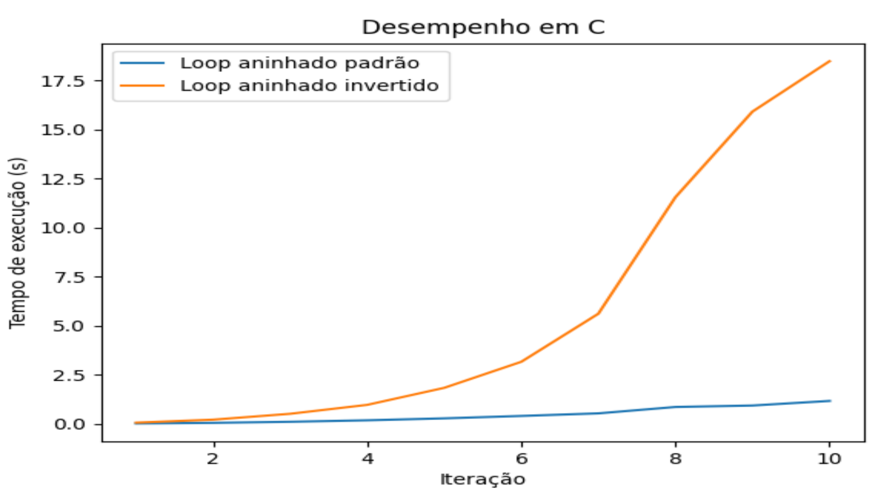
Com isso, dividi o n em 10 pedaços para caber em 10 iterações, marquei o tempo de execução de cada iteração, gerei um arquivo .txt com os resultados e, por fim, usei a linguagem Python para gerar as curvas de desempenho das duas formas de resolver o sistema linear.

3.1 Gráfico de Resultados

Dada as explicações acima, foram obtidos os seguintes resultados:

Tabela 1: Resultados da Implementação C

nxn	Iteração	Tempo1 (s)	Tempo2 (s)
1738x1738	1	0.011527	0.053670
3476x3476	2	0.042730	0.205228
5214x5214	3	0.097089	0.504318
6952x6952	4	0.172444	0.959803
8690x8690	5	0.271504	1.827526
10428x10428	6	0.394379	3.154277
12166x12166	7	0.525586	5.603216
13904x13904	8	0.851192	11.538181
15642x15642	9	0.925319	15.900891
17380x17380	10	1.159064	18.468200



4 Implementação Fortran

Assim como em C, criei duas funções para calcular o sistema linear: uma da forma padrão (acessando linha por linha da matriz A e multiplicando pelos elementos do vetor x) e da forma invertida (acessando coluna por coluna da matriz A e multiplicando pelos elementos do vetor x). Segue o exemplo abaixo:

```
    subroutine matrix_vector_produto(A, x, b,n)
    implicit none
    integer :: n, i, j
    double precision, intent(in) :: A(n,n), x(n)
    double precision, intent(out) :: b(n)
    do i = 1, n
        do j = 1, n
            b(i) = b(i) + A(i,j) * x(j)
        end do
    end do
    end subroutine

subroutine matrix_vector_produto_invertido(A, x, b, n)
    implicit none
    integer :: n, i, j
    double precision, intent(in) :: A(n,n), x(n)
    double precision, intent(out) :: b(n)
    do j = 1, n
        do i = 1, n
            b(i) = b(i) + A(i,j) * x(j)
        end do
    end do
    end subroutine
```

Contudo, eu resolvi usar, de uma forma direta, a mesma quantidade de memória que usei na implementação em C para que a comparação fosse justa.

```
        program main
        implicit none
        integer, parameter :: n_max = 17380
```

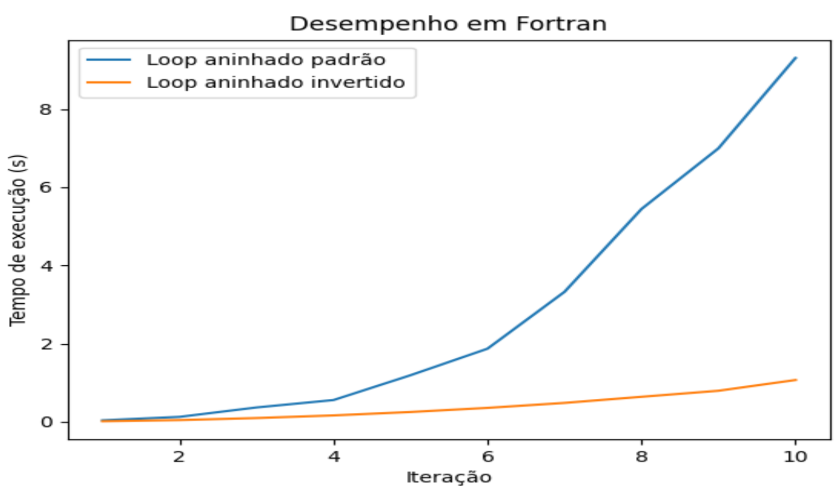
Com isso, segui o mesmo padrão da implementação anterior.

4.1 Gráfico de Resultados

Dada as explicações acima, foram obtidos os seguintes resultados:

Tabela 2: Implementação Fortran

nxn	Iteração	Tempo1 (s)	Tempo2 (s)
1738x1738	1	0.036	0.012
3476x3476	2	0.124	0.042
5214x5214	3	0.365	0.094
6952x6952	4	0.554	0.161
8690x8690	5	1.189	0.248
10428x10428	6	1.868	0.354
12166x12166	7	3.322	0.481
13904x13904	8	5.441	0.637
15642x15642	9	6.994	0.792
17380x17380	10	9.305	1.067



5 Discussão de Resultados

À princípio, sabemos que ambos os algoritmos possuem complexidade $O(n^2)$, isto é, o tempo de execução do algoritmo aumenta proporcionalmente ao quadrado do tamanho da entrada (n). Ou seja, temos dois algoritmos que, apesar de ter loops aninhados de formas diferentes, possuem o mesmo comportamento gráfico

Entretanto, percebemos que um método de aninhamento teve uma curva mais achatada em relação a outra. Na implementação C, o tempo de execução do aninhamento padrão foi melhor. Em Fortran, o aninhamento invertido foi mais elegante.

Isso aconteceu devido a forma como essas linguagens armazenam os dados na memória: A linguagem C armazena arrays em forma de linhas enquanto Fortran armazena em matrizes de coluna principal (column-major order).

Sendo assim, na linguagem C, acessar uma linha de uma matriz é muito mais fácil do que acessar uma coluna. Já no Fortran, é mais fácil acessar a coluna.