Computação de Alto Desempenho COC472 - Trabalho 1

Bruno Dantas de Paiva
DRE: 118048097

May 2, 2021

1 Questão 1

Para determinar a maior dimensão possível para alocar a matriz e os vetores gerados, foi utilizado o próprio código para calcular o produto matriz-vetor junto da seguinte fórmula: $2^n + 2n = 3^{31} * (quantidadederamemgb) * 8$, onde 2^n representa o tamanho armazenado pela matriz, 2n representa o tamanho armazenado pelos 2 vetores, 2^{31} é o valor que representa 1gb e 8 é o tamanho em bytes necessários para armazenar um double, deste modo, para 16gb (quantidade de ram utilizada na máquina de teste) teriamos um valor próximo de 46000. Porém, levando em consideração o fato do sistema operacional ainda consumir uma parte da ram, foi necessário obter um valor mais exato executando o próprio código desenvolvido. Para tal, foi fixado o valor máximo de 42000 e foi incrementado até o valor máximo onde o próprio sistema operacional daria kill no processo. Após isso, foi aumentado o valor máximo de 42000 de 1000 em 1000, a fim de fazer uma busca sobre qual seria o valor máximo e repetido o processo de 100 em 100. Repetindo este procedimento até o máximo, foi obtido um valor de 44400, onde este seria o valor apróximado máximo que o sistema não daria kill no processo.

2 Questão 2

Para esta questão, foi optado por colocar todos os códigos utilizados ao fim do pdf, seguido de uma explicação de como foi executado o processo, deste modo se tornaria algo mais limpo para a leitura.

Quanto aos resultados, é importante observarmos o gráfico primeiro, como é possível verificar abaixo:

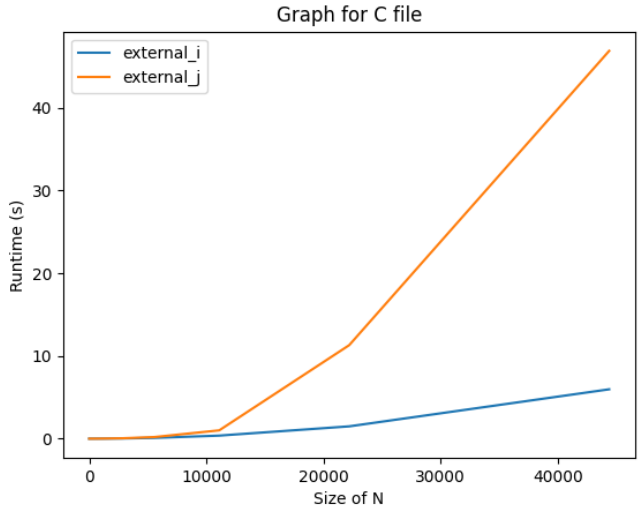

Figure 1: Imagem contendo o gráfico de N x Tempo(s) para os cálculos feitos em C

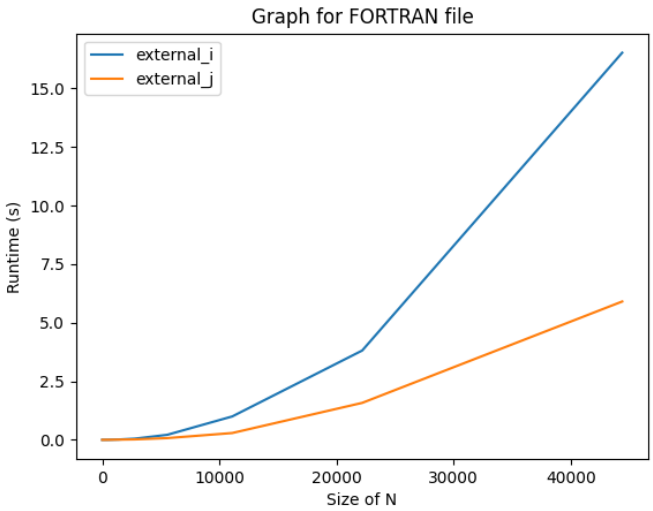

Figure 2: Imagem contendo o gráfico de N x Tempo(s) para os cálculos feitos em Fortran

Para elucidar o entendimento do gráfico, external_i representa o gráfico obtido quando o "i" está localizado no loop externo e, external_j quando o "j" está localizado no loop externo.

Sabendo isto, é possível chegar a conclusão de que para o processo executado na Linguagem C, quando o "j" está no loop externo, nós temos uma perda de desempenho com relação ao "i".

Já na linguagem Fortran, o processo ocorre de maneira inversa, onde a execução ocorre de forma mais rápida quando o "j" está localizado no loop externo.

Tal fato ocorre pois as linguagens possuem maneiras diferentes de armazenar arrays multidimensionais

(Column-Major Order e Row-Major Order).

A Linguagem Fortran, armazena os dados na forma Column-Major Order, ou seja, os valores de cada coluna são armazenado de forma sequencial em memória, deste modo é possível ter um acesso mais rápido, como podemos observar no gráfico gerado por esta linguagem quando o "j" está localizado no loop externo.

Contrariamente, a Linguagem C, armazena os dados na forma Row-Major Order, ou seja, os valores de cada linha serão armazenados de forma sequencial em memória, onde o acesso neste caso será mais rápido caso seja acessado linha a linha, como pode ser observado no gráfico gerado por esta linguagem quando "i" está localizado no loop externo.

3 Códigos

3.1 Bash

```
#!/bin/bash

TIME_DIR=time_files
IMAGE_DIR=image_files
IJ_FILENAME=external_i
JI_FILENAME=external_j
IMAGE_FILENAME=matrix
DATASET=True #Needs to be true or false

C_IJ_PATH=$TIME_DIR/C/$IJ_FILENAME.csv
C_JI_PATH=$TIME_DIR/C/$JI_FILENAME.csv
C_IMAGE_PATH=$IMAGE_DIR/C/$IMAGE_FILENAME.png

F_IJ_PATH=$TIME_DIR/Fortran/$IJ_FILENAME.csv
F_JI_PATH=$TIME_DIR/Fortran/$JI_FILENAME.csv
F_IMAGE_PATH=$IMAGE_DIR/Fortran/$IMAGE_FILENAME.png

#Initializing directories
mkdir -p $TIME_DIR/C
mkdir -p $IMAGE_DIR/C
mkdir -p $TIME_DIR/Fortran
mkdir -p $IMAGE_DIR/Fortran

#Initializing csv files
echo 'Number;Time(s)' > $C_IJ_PATH
echo 'Number;Time(s)' > $C_JI_PATH
echo 'Number;Time(s)' > $F_IJ_PATH
echo 'Number;Time(s)' > $F_JI_PATH

#Generating CSV Files
echo 'Generating CSV Files'

gcc c_files/matrix.c -o c_matrix
gfortran fortran_files/matrix.f95 -o fortran_matrix
for value in $(python python_files/values_generator.py 42000 $DATASET)
do
    echo $(./c_matrix $value 1) >> $C_IJ_PATH
    echo $(./c_matrix $value 0) >> $C_JI_PATH
    echo $(./fortran_matrix $value 1) >> $F_IJ_PATH
    echo $(./fortran_matrix $value 0) >> $F_JI_PATH
done

#Generating Graph images
echo $(python python_files/graph_generator.py $C_IMAGE_PATH $C_IJ_PATH $C_JI_PATH)
echo $(python python_files/graph_generator.py $F_IMAGE_PATH $F_IJ_PATH $F_JI_PATH)
```

```
#Removing unnecessary files
rm c_matrix
rm fortran_matrix
rm -rf $TIME_DIR
```

3.2 C

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

char *lower(char *string)
{
    for (int i = 0; string[i]; i++)
        string[i] = tolower(string[i]);

    return string;
}

double get_random_value(int size)
{
    return rand() % size + 1;
}

double *instantiate_vector(int vector_size)
{
    return (double *)malloc((vector_size + 1) * sizeof(double));
}

double *zero_vector(int vector_size)
{
    double *vector = instantiate_vector(vector_size);
    for (int i = 0; i <= vector_size; i++)
        vector[i] = 0;

    return vector;
}

double *generate_random_vector(int vector_size)
{
    double *vector = instantiate_vector(vector_size);
    for (int i = 0; i <= vector_size; i++)
        vector[i] = get_random_value(vector_size);

    return vector;
}
```

```
double **instantiate_matrix(int matrix_size)
{
    double **matrix = (double **)malloc((matrix_size + 1) * sizeof(double *));
    for (int i = 0; i <= matrix_size; i++)
        matrix[i] = instantiate_vector(matrix_size);

    return matrix;
}

void fill_matrix_line(int current_line, int matrix_size, double **matrix)
{
    for (int j = 0; j <= matrix_size; j++)
        matrix[current_line][j] = get_random_value(matrix_size);
}

double **generate_random_matrix(int matrix_size)
{
    double **matrix = instantiate_matrix(matrix_size);
    for (int i = 0; i <= matrix_size; i++)
        fill_matrix_line(i, matrix_size, matrix);

    return matrix;
}

void line_product_ij(int size, int line_index,
double *result, double **matrix, double *vector)
{
    for (int j = 0; j <= size; j++)
        result[line_index] += matrix[line_index][j] * vector[j];
}

double *matrix_vector_product_ij(int size, double **matrix, double *vector)
{
    double *result = zero_vector(size);

    for (int i = 0; i <= size; i++)
        line_product_ij(size, i, result, matrix, vector);

    return result;
}

void line_product_ji(int size, int column_index,
double *result, double **matrix, double *vector)
{
    for (int i = 0; i <= size; i++)
        result[i] += matrix[i][column_index] * vector[column_index];
}
```

```
double *matrix_vector_product_ji(int size, double **matrix, double *vector)
{
    double *result = zero_vector(size);
    for (int j = 0; j <= size; j++)
        line_product_ji(size, j, result, matrix, vector);

    return result;
}

void free_matrix(double **matrix, int matrix_size)
{
    for (int i = 0; i <= matrix_size; i++)
        free(matrix[i]);

    free(matrix);
}
```

```
int main(int argc, char *argv[])
{
    clock_t antes, depois;
    double *result;
    if (argc != 3 || (atoi(argv[2]) != 0 && atoi(argv[2]) != 1))
    {
        printf("Invalid Arguments.\n");
        return 0;
    }

    int size = atoi(argv[1]);
    int type_of_function = atoi(argv[2]);

    srand(time(NULL));

    double **matrix = generate_random_matrix(size);
    double *vector = generate_random_vector(size);

    if (type_of_function == 1)
    {
        antes = clock();
        result = matrix_vector_product_ij(size, matrix, vector);
        depois = clock();
    }

    else
    {
        antes = clock();
        result = matrix_vector_product_ji(size, matrix, vector);
        depois = clock();
    }

    free_matrix(matrix, size);
    free(vector);
    free(result);

    printf("%d;%.6f\n", size, ((double)(depois - antes)) / CLOCKS_PER_SEC);

    return 0;
}
```

3.3 Fortran

```
program matrix_vector
    implicit none

    !Variable declaration
    character(len=32) :: arg
    integer :: size, func
    real(8) :: start, finish
    real(8), dimension(:,:), allocatable :: matrix
    real(8), dimension (:), allocatable :: vector
    real(8), dimension (:), allocatable :: result

    !Getting argv unique argument (size) and casting it to integer
    call getarg(1, arg)
    read(arg, "(I10)") size

    !Check which function will run
    call getarg(2, arg)
    read(arg, "(I1)") func

    !Allocating variables
    allocate(matrix(size, size))
    allocate(vector(size))
    allocate(result(size))

    call random_seed()

    call generate_random_vector(vector, size)
    call generate_random_matrix(matrix, size)

    if (func == 1) then
        call cpu_time(start)
        call matrix_vector_product_ij(vector, matrix, result, size)
        call cpu_time(finish)
    else if (func == 0) then
        call cpu_time(start)
        call matrix_vector_product_ji(vector, matrix, result, size)
        call cpu_time(finish)
    else
       CALL EXIT(0)
    end if

    !Deallocating variables
    deallocate(matrix)
    deallocate(vector)
    deallocate(result)

    print *, size, ";", (finish - start)
```

```
contains

subroutine generate_random_vector(vector, size)
    implicit none

    real(8), dimension(:) :: vector
    integer :: size, i
    real(8) :: number
    do i = 1, size
        call random_number(number)
        vector(i) = number * (size + 1)
    end do
end

subroutine generate_random_matrix(matrix, size)
    implicit none

    real(8), dimension(:,:) :: matrix
    integer :: size, i, j
    real(8) :: number

    do i = 1, size
        do j = 1, size
            call random_number(number)
            matrix(i, j) = number * (size + 1)
        end do
    end do
end

subroutine generate_vector_zero(vector, size)
    implicit none

    real(8), dimension(:) :: vector
    integer :: size, i
    do i = 1, size
        vector(i) = 0
    end do
end
```

```
    subroutine matrix_vector_product_ji (vector, matrix, result, size)
        implicit none

        real(8), dimension (:) :: vector
        real(8), dimension (:, :) :: matrix
        real(8), dimension (:) :: result
        integer :: i, j, size

        call generate_vector_zero(result, size)

        do j =1, size
            do i=1, size
                result(i) = result(i) +  matrix(i, j) * vector(j);
            end do
        end do
    end

    subroutine matrix_vector_product_ij (vector, matrix, result, size)
        implicit none

        real(8), dimension (:) :: vector
        real(8), dimension (:, :) :: matrix
        real(8), dimension (:) :: result
        integer :: i, j, size

        call generate_vector_zero(result, size)

        do i=1, size
            do j=1, size
                result(i) = result(i) +  matrix(i, j) * vector(j);
            end do
        end do
    end

end program matrix_vector
```

3.4 Python

3.4.1 graph_generator.py

```
import matplotlib.pyplot as plt
import csv
import sys


def get_title(archive_name):
    return [value.lower().replace('.csv', '') for value in archive_name.split('/')]


def read_csv(archive_name: str) -> tuple:
    x = []
    y = []
    with open(archive_name, 'r') as csv_file:
        dataset = csv.reader(csv_file, delimiter=';')
        next(dataset)
        for line in dataset:
            number, runtime = line
            x.append(int(number.strip()))
            y.append(float(runtime.strip()))

    return x, y, get_title(archive_name)[-1]


def plot_scatter(data: list, save_path: str) -> None:
    legend = []
    plt.ylabel("Runtime (s)")
    plt.xlabel("Size of N")

    splitted_path = get_title(save_path)
    language = splitted_path[len(splitted_path)-2].upper()
    plt.title(f"Graph for {language} file")

    for x, y, archive_name in data:
        plt.plot(x, y)
        legend.append(archive_name)

    plt.legend(legend)
    plt.savefig(save_path)
    print(f"Graph generated for the language: {language}")
```

```
if __name__ == "__main__":
    try:
        if(len(sys.argv) != 4):
            print('Could not save the file. Check your arguments')
            exit()
    except:
        print('Could not save the file. Check your arguments')
        exit()

    data = []
    for path in sys.argv[2:]:
        data.append(read_csv(path))

    plot_scatter(data, sys.argv[1])

    return result
```

3.4.2 values_generator.py

```
import sys


def generate_power_values(n: int = 38000) -> None:
    result = []
    for i in range(0, n+1):
        power = int((2**i)*1.355)
        if(power > n):
            break

        result.append(str(power))

    print(' '.join(result))


def generate_hundred_values(n: int = 38000) -> None:
    result = ['1']
    for i in range(100, n+1, 100):
        result.append(str(i))

    print(' '.join(result))


if __name__ == "__main__":
    try:
        if(len(sys.argv) != 3 or (type(int(sys.argv[1])) != int)
        or (sys.argv[2].lower() != "true" and sys.argv[2].lower() != "false")):
            print('')
            exit()
    except:
        print('')
        exit()

    if(sys.argv[2].lower() == "true"):
        generate_power_values(int(sys.argv[1]))
    else:
        generate_hundred_values(int(sys.argv[1]))
```

3.5 Execução

Para executar o programa, foi criado um arquivo em bash, deste modo seria possível automatizar a geração dos gráficos para cada tipo de arquivo de uma maneira mais simples.

Além disso, foi utilizada a linguagem de programação python para criar os valores utilizados para gerar o gráfico e também para gerar o próprio gráfico.

Todo o trabalho desenvolvido com os códigos também se encontra neste repositório: Github.