Trabalho 1 - Computação de Alto Desempenho
Semestre 2020/2
Thiago Guimarães - DRE: 118053123

1. A maior dimensão possível encontrada, no caso da máquina sendo utilizada (16 GB de RAM, ~14.2 GB disponível - Manjaro Linux), foi de aproximadamente 42k, seguindo a equação:

   2n + n^2 = bytes disponíveis / 8

   Sendo n = maior dimensão possível.
   Ao rodar os programas criados, foi perceptível que este valor encontrado foi condizente. Ao rodarmos com este valor, a memória RAM da máquina foi quase completamente utilizada.

2. O código para o processo pode ser encontrado aqui: GITHUB.
   O projeto estrutura-se da seguinte forma:
   Temos 3 pastas principais e 1 arquivo (fora o exercício em si):

   c_resolution/
   fortran_resolution/
   plots/
   runner.py

A pasta c_resolution possui o código da solução C no arquivo resolution.c e o arquivo dos dados gerados no arquivo data.csv

A Solução em C Visa Rodar o algoritmo proposto para uma dimensão de matriz fornecida como argumento e como output traz o formato [Dimensão,tempo_de_execucao_ij, tempo_de_execucao_ji], sendo, respectivamente, a dimensão da matriz, o tempo de execução da multiplicação com i externo e o tempo de execução da multiplicação com j externo.

A pasta fortran_resolution possui o código da solução FORTRAN no arquivo resolution.f95 e o arquivo dos dados gerados no arquivo data.csv

A solução em fortran é completamente análoga a em C, funcionando exatamente da mesma forma.

A pasta plots possui os gŕaficos gerados pelas resoluções C e FORTRAN.
O arquivo runner.py é o script python que orquestra a utilização dos códigos via criação de subprocessos para gerar os dados a serem observados.

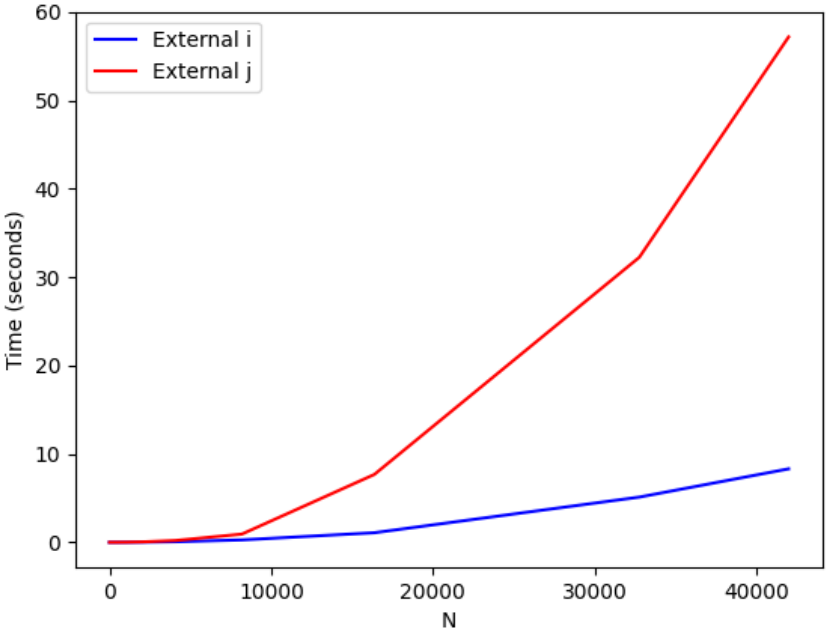
Os códigos em si encontram-se no apêndice do arquivo.

Como os dados são adquiridos (algoritmo do runner.py):

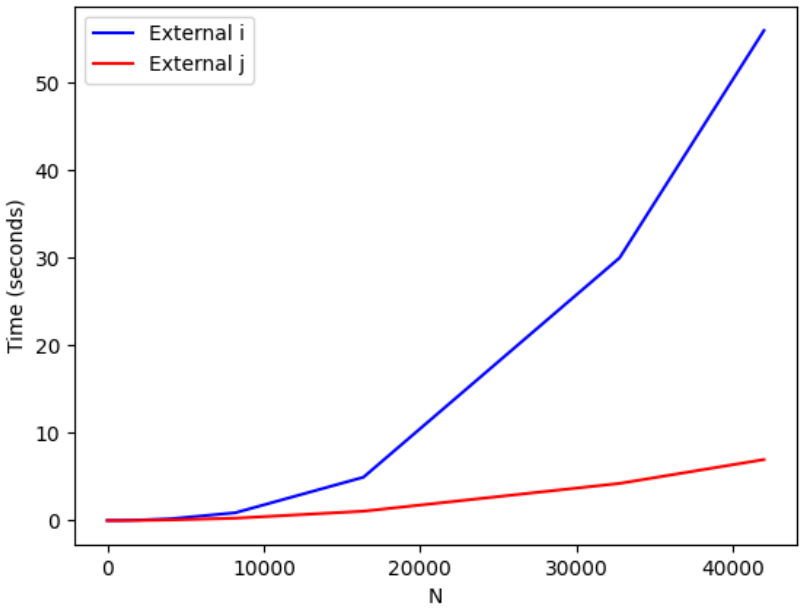
- Compilamos a linguagem em questão
- Criamos um arquivo de dados vazio
- Rodamos o programa da linguagem que armazenamos seu output no arquivo de dados em loop, dobrando o tamanho da matriz a cada iteração, até chegarmos no valor limite
- Rodamos o programa para o valor limite para obtermos o último *datapoint*
- Criamos um gráfico para representar os dados respectivos
- O mesmo é feito para a outra linguagem - no caso, fazemos primeiro com a linguagem C e depois com o Fortran

Resultado Final:

Resultados - Linguagem C



Resultados - Linguagem Fortran



Os dois gráficos possuem 2 linhas: 1 azul, referente a execução da multiplicação matriz X vetor com loop externo em i (linhas) e 1 vermelha, referente a execução da multiplicação matriz X vetor com loop externo em j(colunas).

Observando os dois gráficos, fica em grande evidência a diferença entre as 2 linguagens: o programa C performa muito melhor (dado que performar melhor = executar em um tempo menor) utilizando multiplicação com loop externo em i e o programa fortran performa muito melhor executando uma multiplicação com loop externo em j.

Esta diferença se dá pela forma com que essas diferentes linguagens armazenam seus dados. A linguagem C armazena utilizando *Row-Major order*, o que implica que os cada linha da matriz está armazenada uma ao lado da outra, o que gera um acesso mais rápido aos dados quando estamos com i no loop externo. De maneira oposta, fortran armazena utilizando *Column-Major order,* o que implica que cada coluna da matriz está armazenada uma ao lado da outra, facilitando o acesso para o caso com j no loop externo, de forma análoga ao C com i no loop externo.

APÊNDICE

Arquivos de código

Código python

```python
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
    You, 4 days ago • finishing C code and adding python code to run th…

def process(language: str, compiler_call: str) -> str:
    data_file_path = f"./{language}_resolution/data.csv"
    # create empty data file
    with open(data_file_path, "w") as data_file:
        data_file.write("")
        data_file.close()

    compile = subprocess.call(
        compiler_call,
        shell=True,
    )

    if compile != 0:
        raise RuntimeError(f"Couldnt compile {language} file")

    with open(data_file_path, "a") as data_file:
        size = 1
        while size < 42000:
            ret_code = subprocess.call(
                [f"./{language}_resolution/resolution", str(size)], stdout=data_file
            )
            size = size * 2
        subprocess.call(
            [f"./{language}_resolution/resolution", str(42000)], stdout=data_file
        )
    return data_file_path


def generate_graph(file_path: str, plot_dest: str) -> None:
    dataframe = pd.read_csv(file_path, delimiter=",")
    dataframe.columns = ["Matrix Dimension", "External i", "External j"]
    plt.plot("Matrix Dimension", "External i", data=dataframe, color="blue")
    plt.plot("Matrix Dimension", "External j", data=dataframe, color="red")
    plt.xlabel("N")
    plt.ylabel("Time (seconds)")
    plt.legend()
    plt.savefig(plot_dest)
```

```python
def main():
    # First, run C solution
    data_file_path = process(
        language="c",
        compiler_call="gcc -Wall -o ./c_resolution/resolution ./c_resolution/resolution.c",
    )
    plt.figure(0)
    generate_graph(data_file_path, "./plots/c_resolution.png")
    # Then, run FORTRAN solution
    data_file_path = process(
        language="fortran",
        compiler_call="gfortran ./fortran_resolution/resolution.f95 -o ./fortran_resolution/resolution",
    )
    plt.figure(1)
    generate_graph(data_file_path, "./plots/fortran_resolution.png")


if __name__ == "__main__":
    main()

```

Código C

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

double *spawn_random_vector(int size)
{
    double *vector = (double *)malloc((size) * sizeof(double));
    for (int i = 0; i <= size; i++)
    {
        vector[i] = rand() % size;
    }
    return vector;
}

double **spawn_random_matrix(int size)
{
    double **matrix = (double **)malloc((size) * sizeof(double *));
    for (int i = 0; i <= size; i++)
    {
        matrix[i] = spawn_random_vector(size);
    }
    return matrix;
}
double *multiply_first_i(double **matrix, double *vector, int size)
{
    double *result_vector = (double *)malloc((size) * sizeof(double));
    for (int i = 0; i <= size; i++)
    {
        result_vector[i] = 0;
        for (int j = 0; j <= size; j++)
        {
            result_vector[i] = result_vector[i] + vector[j] * matrix[i][j];
        }
    }
    return result_vector;
}

```

```c
double *multiply_first_j(double **matrix, double *vector, int size)
{
    double *result_vector = (double *)malloc((size) * sizeof(double));
    for (int j = 0; j <= size; j++)
    {
        result_vector[j] = 0;
        for (int i = 0; i <= size; i++)
        {
            result_vector[i] = result_vector[i] + vector[j] * matrix[i][j];
        }
    }
    return result_vector;
}

int main(int argc, char *argv[])
{

    srand(time(NULL));
    clock_t before, after;
    int number_of_rows = atoi(argv[1]);
    double **matrix = spawn_random_matrix(number_of_rows);
    double *vector = spawn_random_vector(number_of_rows);

    before = clock();
    double *result_vector = multiply_first_i(matrix, vector, number_of_rows);
    after = clock();
    free(result_vector);
    double difference_ij = ((double)(after - before)) / CLOCKS_PER_SEC;

    before = clock();
    result_vector = multiply_first_j(matrix, vector, number_of_rows);
    after = clock();
    free(result_vector);
    double difference_ji = ((double)(after - before)) / CLOCKS_PER_SEC;
    printf("%d,%.8f,%.8f\n", number_of_rows, difference_ij, difference_ji);
    return 0;
}
```

Código Fortran

```fortran
program multiply_matrix_vector
    implicit none
      You, 4 days ago • adding new data files, new plots and final versio…
    character(len=32) :: argument
    integer :: size
    real(8) :: before, after, diff_ij, diff_ji
    real(8), dimension(:,:), allocatable :: matrix
    real(8), dimension (:), allocatable :: vector
    real(8), dimension (:), allocatable :: result_vector

    call getarg(1, argument)
    read(argument, "(I10)") size
    call increment(size, 1)
    call random_seed()

    allocate(matrix(size, size))
    allocate(vector(size))
    allocate(result_vector(size))
    call spawn_random_matrix(matrix, size)
    call spawn_random_vector(vector, size)

    call cpu_time(before)
    call multiply_first_i(matrix, vector, size, result_vector)
    call cpu_time(after)
    diff_ij = after - before

    call cpu_time(before)
    call multiply_first_j(matrix, vector, size, result_vector)
    call cpu_time(after)
    diff_ji = after - before

    deallocate(matrix)
    deallocate(vector)
    deallocate(result_vector)

    print *,argument,",",diff_ij,",",diff_ji

    contains

    elemental subroutine increment(var, incr)
        implicit none
        integer,intent(inout) :: var
        integer,intent(in)    :: incr

        var = var + incr
    end subroutine

    subroutine spawn_random_vector(vector, size)
        implicit none
        real(8), dimension(:) :: vector
        integer :: size, i
        real(8) :: value
        do i=1, size
            call random_number(value)
            vector(i) = value * (size)
        end do
    end

    subroutine spawn_random_matrix(matrix, size)
        implicit none
        real(8), dimension(:,:) :: matrix
        integer :: size, i, j
        real(8) :: value

        do i=1, size
            do j=1, size
                call random_number(value)
                matrix(i, j) = value * (size)
            end do
        end do
    end
```

```
72
73      subroutine multiply_first_i(matrix, vector, size, result_vector)
74          implicit none
75          real(8), dimension (:) :: vector
76          real(8), dimension (:, :) :: matrix
77          real(8), dimension (:) :: result_vector
78          integer :: i, j, size
79
80          do i=1, size
81              result_vector(i) = 0
82              do j=1, size
83                  result_vector(i) = result_vector(i) +  vector(j)*matrix(i, j);
84              end do
85          end do
86      end
87
88      subroutine multiply_first_j(matrix, vector, size, result_vector)
89          implicit none
90          real(8), dimension (:) :: vector
91          real(8), dimension (:, :) :: matrix
92          real(8), dimension (:) :: result_vector
93          integer :: i, j, size
94
95          do j=1, size
96              result_vector(j) = 0
97              do i=1, size
98                  result_vector(i) = result_vector(i) +  vector(j)*matrix(i, j);
99              end do
00          end do
01      end
02  end program multiply_matrix_vector
03
04
```

Arquivos de Dados

c_resolution/data.csv

```
You, 2 hours ago | 1 author (You)
1,0.00000200,0.00000100
2,0.00000200,0.00000100
4,0.00000200,0.00000100
8,0.00000200,0.00000100
16,0.00000300,0.00000200
32,0.00000700,0.00000700
64,0.00001300,0.00001600
128,0.00008200,0.00008000
256,0.00022400,0.00035400
512,0.00146100,0.00222500
1024,0.00424900,0.01033300
2048,0.01734000,0.04896800
4096,0.06801500,0.20393000
8192,0.27057900,0.92839600
16384,1.08339000,7.68180200
32768,5.11627400,32.25879800
42000,8.31341400,57.20231000
```

fortran_resolution/data.csv

```
1     ,   1.9999999999998318E-006 ,   1.0000000000001327E-006
2     ,   1.9999999999998318E-006 ,   1.0000000000001327E-006
4     ,   1.9999999999998318E-006 ,   1.0000000000001327E-006
8     ,   2.0000000000002655E-006 ,   1.0000000000001327E-006
16    ,   2.9999999999999645E-006 ,   1.9999999999998318E-006
32    ,   7.9999999999997608E-006 ,   7.0000000000000617E-006
64    ,   1.9000000000000353E-005 ,   1.5999999999999955E-005
128   ,   7.2999999999999714E-005 ,   6.7999999999999918E-005
256   ,   3.3099999999999970E-004 ,   2.4900000000000009E-004
512   ,   1.5219999999999999E-003 ,   9.8500000000000063E-004
1024  ,   9.7779999999999985E-003 ,   3.9700000000000013E-003
2048  ,   4.7475999999999990E-002 ,   1.5739000000000003E-002
4096  ,  0.19920399999999994      ,   6.6775999999999947E-002
8192  ,  0.88490399999999991      ,  0.26273200000000019
16384 ,   4.9427100000000008      ,   1.0596029999999992
32768 ,   30.004511000000001      ,   4.2344369999999998
42000 ,   55.966262999999998      ,   6.9592529999999897
```