Arquitetura de Computadores 2 - Trabalho Final - Programação Paralela

GIOVANNA ALBURQUERQUE, LUCIANO PEREIRA E RYAN MATHEUS

15 de dezembro de 2022

CPU de cada membro

Giovanna: **Intel Core i5-7300HQ**
Luciano: **Intel Core i3-7100**
Ryan: **Intel Core i7-5500U**

1. CODIGO ORIGINAL

Abaixo o código original, contendo 3 loops, o primeiro usando loop para preencher 4 arrays com operação de multiplicação com double e os outros dois loops fazem o papel de realizar multiplicações entre valores dos arrays:

```
#include <chrono>
#include <iostream>

// GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS

static long NUM = 10000;
double step;

using namespace std;

int main () {
    double a[NUM], b[NUM], c[NUM], d[NUM]; // arrays
    int i,j; // loop counters
    auto start = std::chrono::high_resolution_clock::now();


    for (i=0; i < NUM; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
        c[i] = i * 3.0;
        d[i] = i * 4.0;
    }

    for (i=0; i < NUM-2; i++) {
        for (j=0; j < NUM-2; j++) {
            a[i] = b[i] + c[i] * d[j];
        }
    }

    for (i=0; i < NUM; i++) {
        d[i] = a[i] * a[i];
    }

    auto finish = std::chrono::high_resolution_clock::now();

    auto duration =
   std::chrono::duration_cast<std::chrono::microseconds>(finish -
   start).count();

    float time = duration / 1000000.0;
    // print time
    cout << "Time: " << time << " seconds" << endl;
}
```

Tabela de tempo de execução

(Desconsiderado a primeira execução devido ao cache miss e o consequente tempo discrepante aos demais testes)

	Giovanna	Luciano	Ryan
N = 10000	0.274	0.229677 s	0.3566 s
N = 30000	2.34548	2.03315 s	2.46581
N = 50000	6.34386	5.42051 s	6.81111

2. OTIMIZAÇÃO DE MEMÓRIA

Para otimizar o código foi feito a substituição de variáveis de ponto flutuante para inteiros bem como o uso de apenas uma matriz ao invés de 4 arrays

```
#include <chrono>
#include <iostream>

// GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS

static long NUM = 10000;
double step;

using namespace std;

int main () {
    //double a[NUM], b[NUM], c[NUM], d[NUM];
    int effarray[NUM][4];
    int i,j, temp;
    auto start = std::chrono::high_resolution_clock::now();

    for (i=0; i < NUM; i++) {
        temp = i * 1;
        effarray[i][0] = temp;
        effarray[i][1] = temp * 2;
        effarray[i][2] = temp * 3;
        effarray[i][3] = temp * 4;
    }

    for (i=0; i < NUM; i++) {
        for (j=0; j < NUM; j++) {
            effarray[i][0] = effarray[i][1] + effarray[j][2];
        }
    }

    for (i=0; i < NUM; i++) {
        effarray[i][3] = effarray[i][0] * effarray[i][0];
    }

    auto finish = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(finish -
start).count();

    float time = duration / 1000000.0;
    // print time
    cout << "Time: " << time << " segundos" << endl;
}
```

Tabela de tempo de execução

(Desconsiderado a primeira execução devido ao cache miss e o consequente tempo discrepante aos demais testes)

	Giovanna	Luciano	Ryan
N = 10000	0.22633 s	0.191 s	0.32685 s
N = 30000	2.23434	1.88837 s	3.21211 s
N = 50000	5.98658	5.18491 s	8.90605 s

3. OTIMIZAÇÃO NA COMPILAÇÃO

Sendo usado o código original, foram compilados 3 arquivos usando a seguinte argumentação:

- g++ -O1 '.\projeto com otimização de memoria.cpp' -o projetoO1
- g++ -O2 '.\projeto com otimização de memoria.cpp' -o projetoO2
- g++ -O3 '.\projeto com otimização de memoria.cpp' -o projetoO3

A seguir será apresentado os valores de tempo para cada grau de otimização (O1, O2, O3)

Tabela de tempo de execução com N = 10000

	Giovanna	Luciano	Ryan
O1	0.034288	0.029999	0.039558
O2	0.000021	0.000426	0.000492
O3	0.000019	0.000386	0.000449

Tabela de tempo de execução com N = 30000

	Giovanna	Luciano	Ryan
O1	0.310695	0.23868	0.305326
O2	0.000113	0.001026	0.001685
O3	0.000087	0.000976	0,001283

Tabela de tempo de execução com N = 50000

	Giovanna	Luciano	Ryan
O1	0.827625	0.669046	0.845524
O2	0.000256	0.001001	0.002458
O3	0.000138	0.001000	0.002559

Conclusão

Nota-se uma grande melhoria ao aplicar uma otimização, sendo que do primeiro nível para o segundo nível ocorre uma melhoria radical, chegando a melhorar de 300 a 3000 vezes

4. OTIMIZAÇÃO DE PARALELIZAÇÃO

Sendo usado o código otimizado em memoria acima com novas adições abaixo, envolvendo o OPENMP:

```
#include <stdio.h>
#include <omp.h>
#include <chrono>
#include <iostream>

// GRUPO : LUCIANO PEREIRA, GIOVANNA ALBURQUERQUE, RYAN MATHEUS

static long NUM = 10000;
using namespace std;

int main()
{
    int effarray[NUM][4];
    int i, j, temp;
    auto start = std::chrono::high_resolution_clock::now();

omp_set_num_threads(4);
#pragma omp parallel default(none) shared(NUM, effarray, temp) private(i, j)
    {
#pragma omp for nowait
        for (i=0; i < NUM; i++) {
        temp = i * 1;
        effarray[i][0] = temp;
        effarray[i][1] = temp * 2;
        effarray[i][2] = temp * 3;
        effarray[i][3] = temp * 4;
    }

#pragma omp for nowait
        for (i=0; i < NUM; i++) {
            for (j=0; j < NUM; j++) {
                effarray[i][0] = effarray[i][1] + effarray[j][2];
        }
    }

#pragma omp for nowait
        for (i=0; i < NUM; i++) {
            effarray[i][3] = effarray[i][0] * effarray[i][0];
        }
    }
    auto finish = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(finish -
start).count();
    float time = duration / 1000000.0;
    cout << "Time: " << time << " seconds" << endl;
}
```

Tabela de tempo de execução com N = 10000

	Giovanna	Luciano	Ryan
1 Thread	0.2411	0.202995	0.290995
2 Threads	0.12525	0.125032	0.161659
3 Threads	0.095394	0.104511	0.158534
4 Threads	0.109656	0.098022	0.143941

Tabela de tempo de execução com N = 30000

	Giovanna	Luciano	Ryan
1 Thread	2.30488	1.93231	2.57518
2 Threads	1.22573	1.12519	1.64952
3 Threads	0.878431	0.91599	1.58673
4 Threads	0.7526	0.80757	1.43625

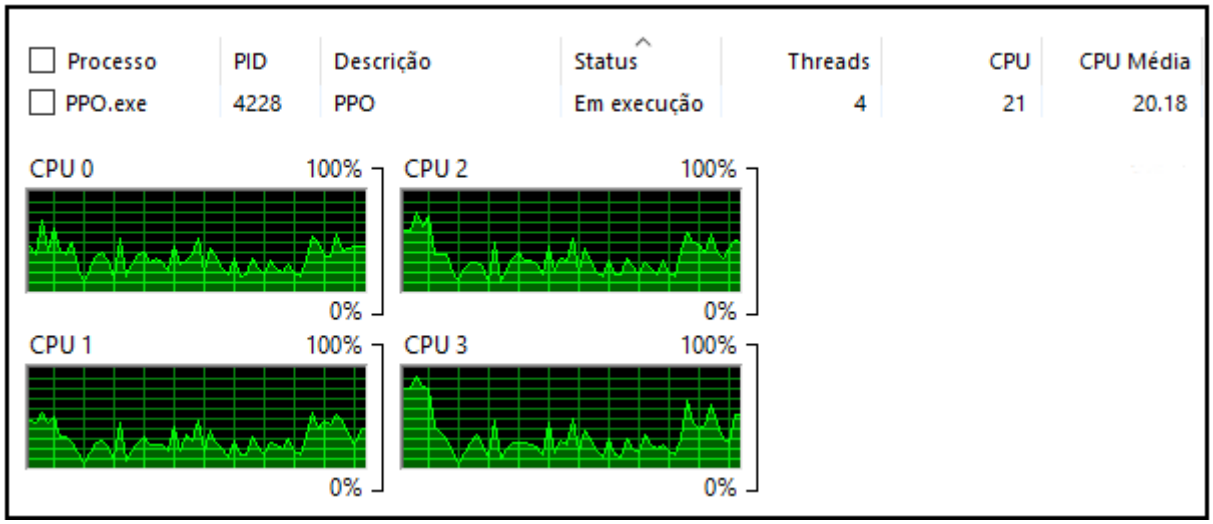
Tabela de tempo de execução com N = 50000

	Giovanna	Luciano	Ryan
1 Thread	6.73661	4.91612	8.90576
2 Threads	4.16616	3.03005	4.47458
3 Threads	2.7001	2.37024	4.35857
4 Threads	2.10147	2.2135	4.01038

Quando o número de threads ultrapassa o número de CPUs de uma máquina a otimização se torna irrelevante pois as CPU de todos os membros só tem 2 núcleos com 2 pipelines, ou seja, a partir de 5 threads a eficiência é basicamente a mesma de 4 threads.

Uso nas CPUs (N=120000)

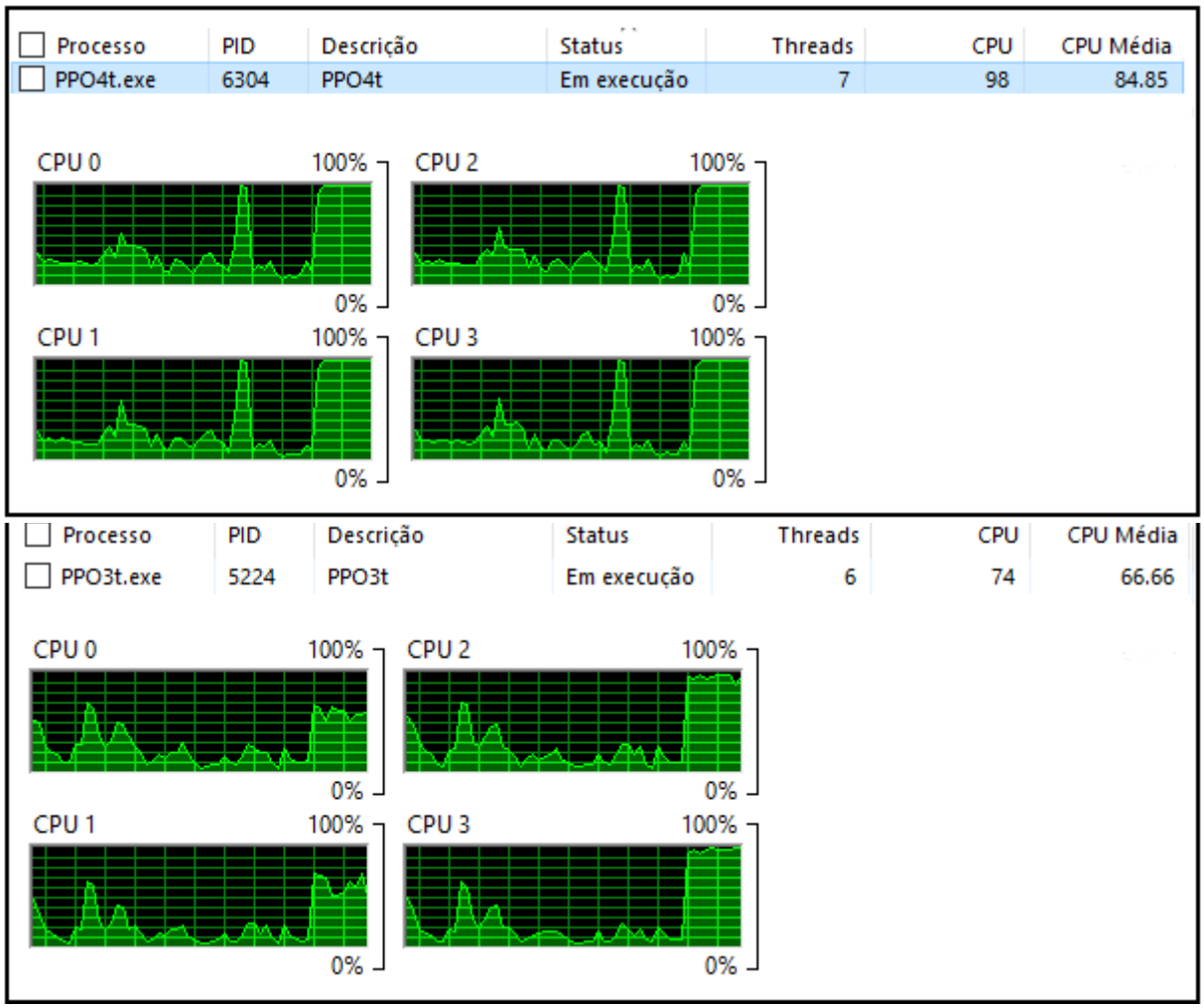
Uma Thread no Código



Duas Threads no Código

Três Threads no Código

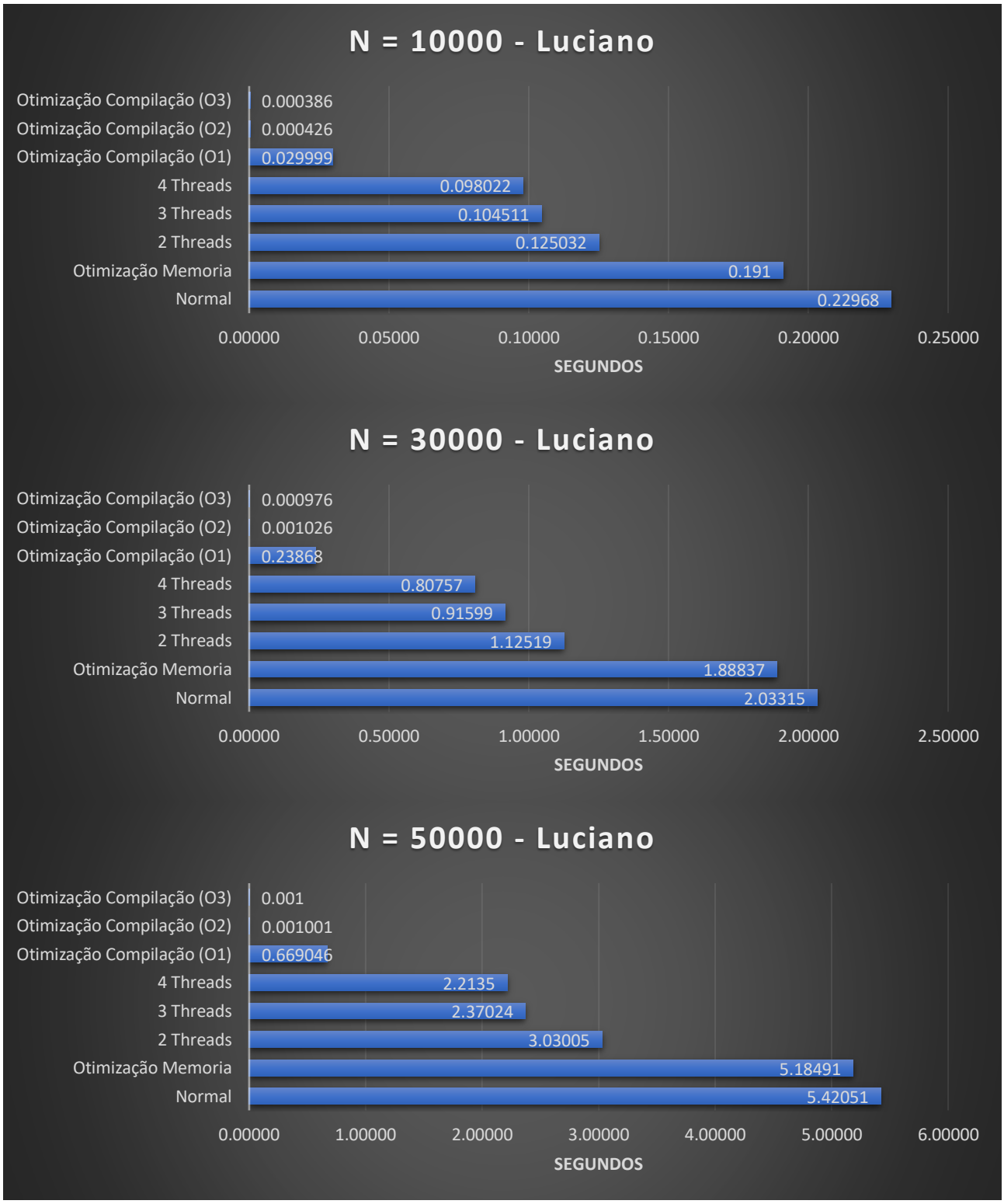
Quatro Threads no Código

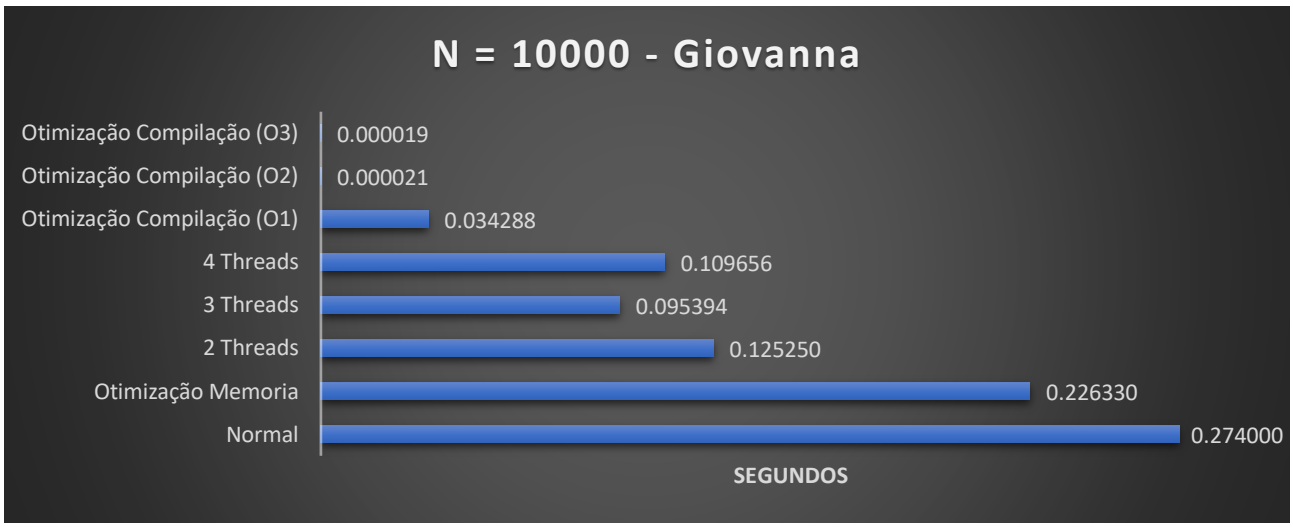


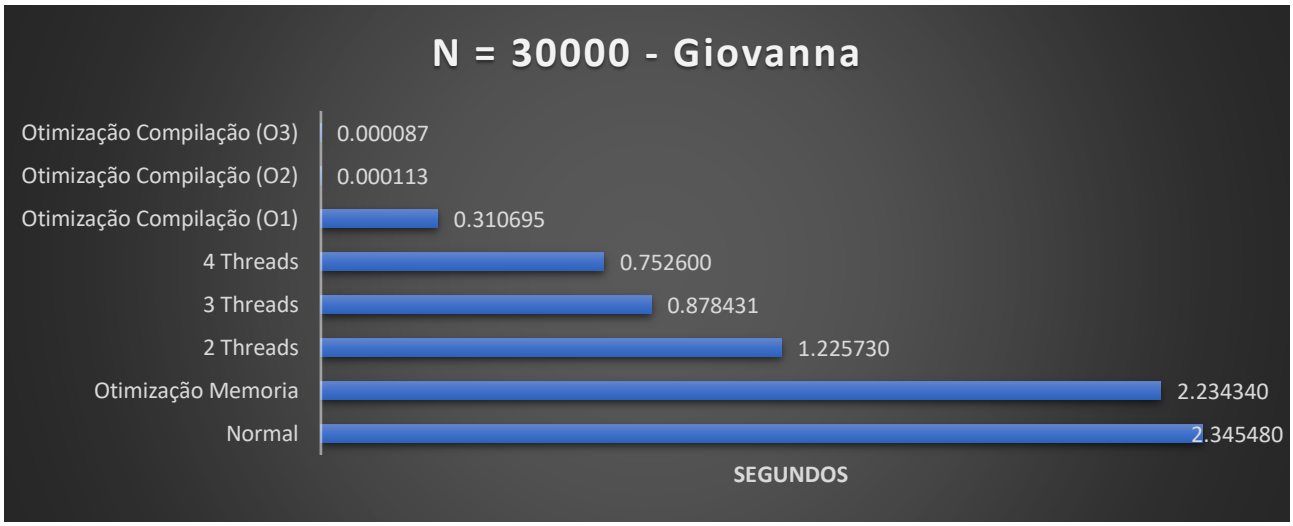
Conclusões

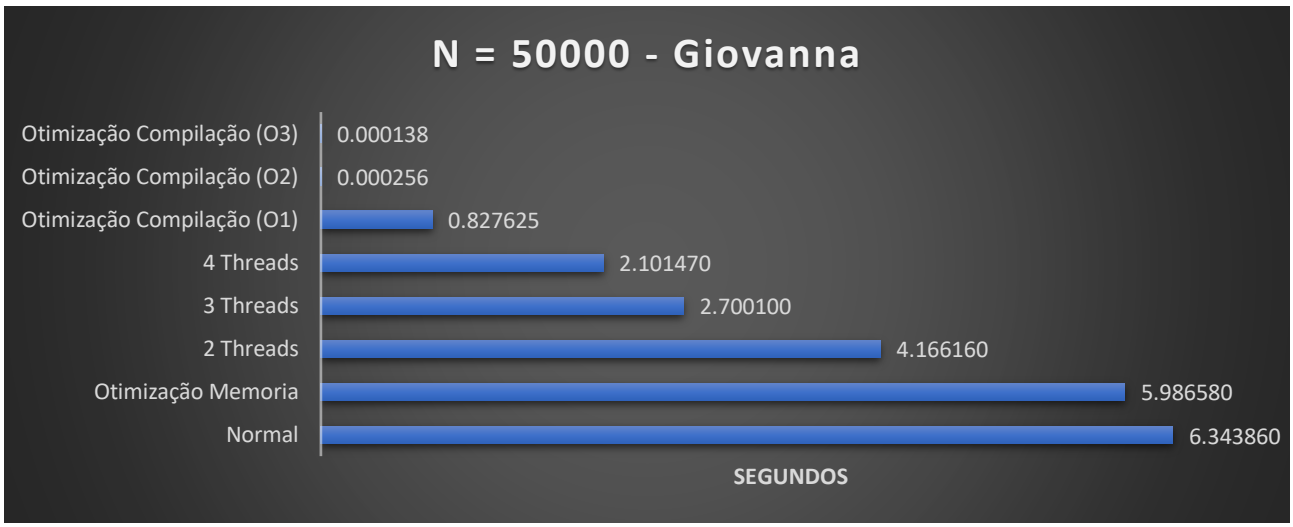
Após analise das imagens consta-se que à medida que o OPENMP inicializa mais threads, mais consome-se a CPU em geral, em 4 threads chega perto de alcançar os 100% de uso, enquanto uma thread em média só consome 20%, com isso podemos entender um dos motivos pelo qual a execução se torna mais rápida

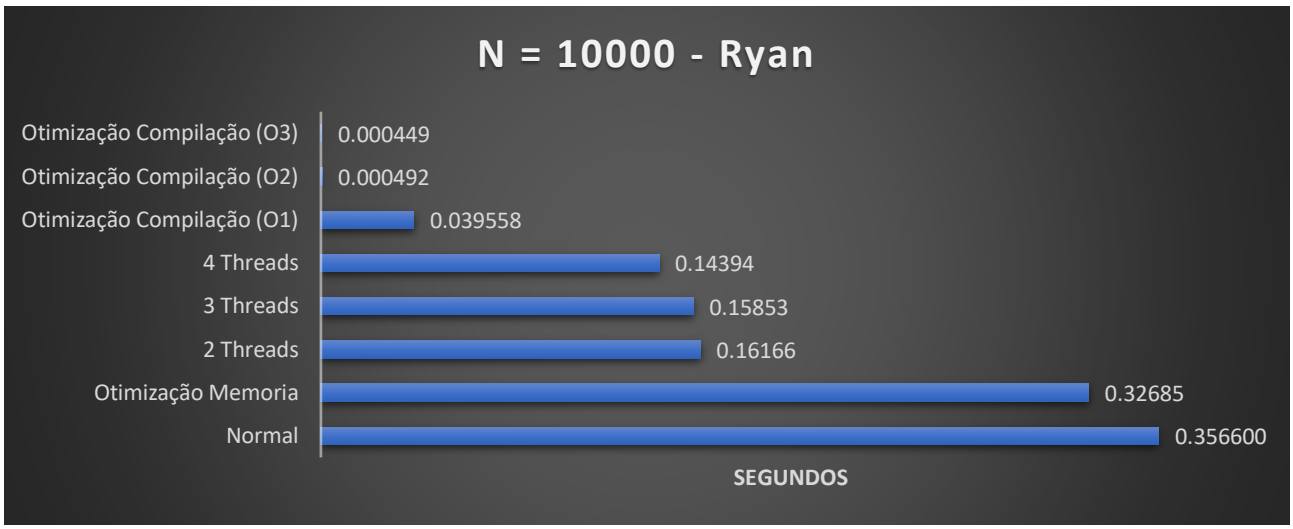
5. COMPARAÇÃO

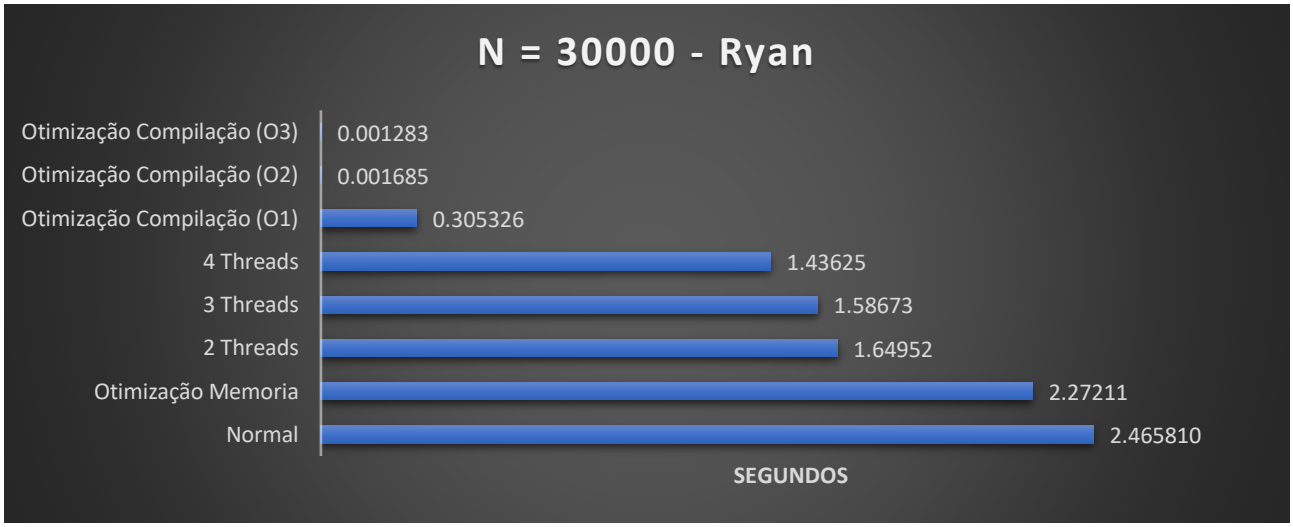


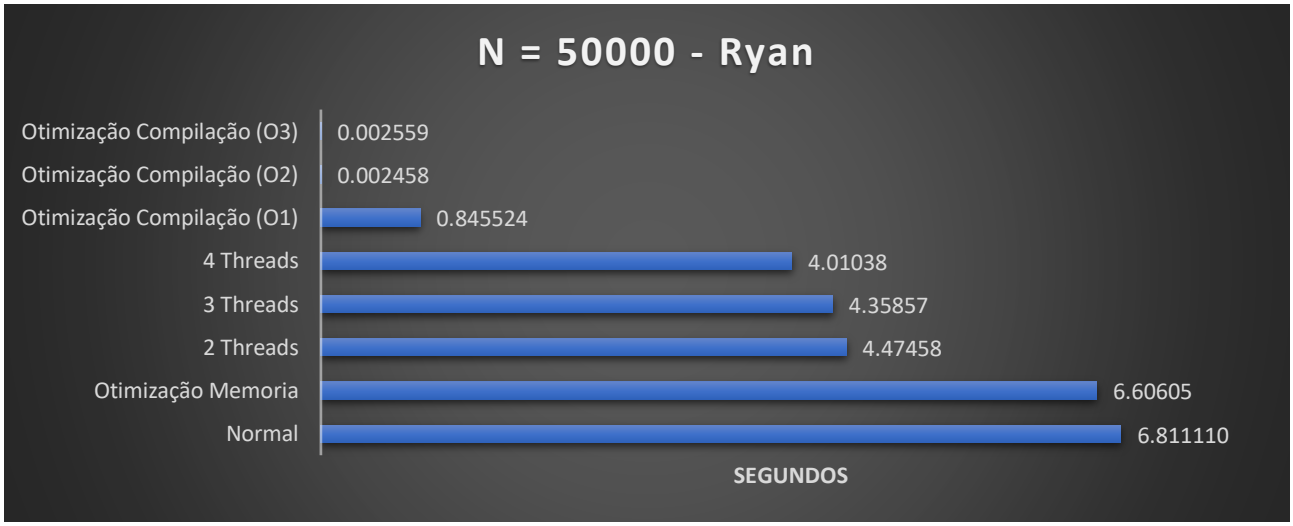

N = 10000 - Giovanna
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.000019
0.000021
0.034288
0.109656
0.095394
0.125250
0.226330
0.274000
SEGUNDOS


N = 30000 - Giovanna
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.000087
0.000113
0.310695
0.752600
0.878431
1.225730
2.234340
2.345480
SEGUNDOS


N = 50000 - Giovanna
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.000138
0.000256
0.827625
2.101470
2.700100
4.166160
5.986580
6.343860
SEGUNDOS


N = 10000 - Ryan
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.000449
0.000492
0.039558
0.14394
0.15853
0.16166
0.32685
0.356600
SEGUNDOS


N = 30000 - Ryan
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.001283
0.001685
0.305326
1.43625
1.58673
1.64952
2.27211
2.465810
SEGUNDOS


N = 50000 - Ryan
Otimização Compilação (O3)
Otimização Compilação (O2)
Otimização Compilação (O1)
4 Threads
3 Threads
2 Threads
Otimização Memoria
Normal
0.002559
0.002458
0.845524
4.01038
4.35857
4.47458
6.60605
6.811110
SEGUNDOS

CONCLUSÃO FINAL

Diante dos fatos apresentados, em todos os 3 participantes ocorreu mudanças positivas em relação a eficiência de execução, a otimização em memória produziu uma leve melhora, mas ao ser colocado a otimização de compilação gerou resultados piores que ao código original (não ocorrendo melhorias a partir do primeiro nível), a otimização por threads foi razoavelmente eficiente, porem com pico de uso da CPU, a melhor otimização ocorreu pelas diretivas de compilação, sendo bastante rápida e com pouco uso de CPU comparada a otimização paralela.