

UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE FÍSICA
FÍSICA COMPUTACIONAL

**COMPUTAÇÃO DE ALTO DESEMPENHO I:
PROJETO FINAL - EQUAÇÃO DA ONDA EM
2 DIMENSÕES**

Pablo de Deus Silva

Volta Redonda
05 de maio de 2021

Sumário

1	INTRODUÇÃO	4
2	OBJETIVO	5
3	METODOLOGIA	6
3.1	Discretização	6
3.2	O método das Diferenças Finitas	9
4	IMPLEMENTAÇÃO	10
4.1	Fluxograma	10
4.2	Código Serial	10
4.3	Resultados	15
5	BENCHMARK	16
5.1	Laboratório 107	16
5.2	Notebook	20
5.3	Comparação laboratório e notebook	22
6	PROFILE	23
7	TÉCNICAS DE OTIMIZAÇÃO DE SOFTWARE	24
8	COMPARAÇÃO DO PROGRAMA BASE COM O OTIMIZADO	25
9	IMPLEMENTAÇÃO DO PROGRAMA COM OPENMP	26
9.1	Análise do código	26
9.2	Implementação com openMP	26
10	BENCHMARK DO PROGRAMAS COM A COMPARAÇÃO DE VÁRIAS THREADS	28
11	SUPERCOMPUTADOR SANTOS DUMONT	31
12	VALIDAÇÃO DOS RESULTADOS	34
13	CONCLUSÃO	36

Resumo

O trabalho presente descreve uma solução numérica para a equação de onda em duas dimensões utilizando o método de diferenças finitas, de forma a ser utilizado diversas técnicas e ferramentas da disciplina de computação de alto desempenho com a intenção de aumentar a eficiência do código serão aplicadas técnicas de otimização a nível de software como análise da estrutura do código e também a nível de compilação, como a utilização de diferentes compiladores e flags para aumentar o desempenho. Além dessas ferramentas será utilizado o OpenMP para paralelizar o código, a fim de usar mais de uma thread e fazer uma comparação da eficiência a cada implementação dessas ferramentas.

Abstract

The work presented describes a numerical solution for the wave equation in two dimensions using the finite difference method, in order to use several techniques and tools of the high performance computing discipline with the intention of increasing the code efficiency. Optimization at the software level such as analysis of the code structure and also at the compilation level, such as the use of different compilers and flags to increase performance. In addition to these tools, OpenMP will be used to parallelize the code, in order to use more than one thread and make a comparison of the efficiency with each implementation of these tools.

1 Introdução

Equações diferenciais parciais (EDPs) aparecem em diversos problemas na física e matemática, uma equação hiperbólica utilizada para descrever fenômenos ondulatórios é a equação da onda.

$$\frac{\partial^2 z}{\partial t^2} = v_0^2 \left[\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right], \quad v_0 = \sqrt{\frac{\mu_0}{T_0}} \quad (1.1)$$

Uma equação diferencial parcial linear de segunda ordem que descreve o movimento de uma onda em 3 dimensões espaciais e aparece em diversas áreas da física, como no estudo de ondas eletromagnéticas no eletromagnetismo e dinâmica dos fluidos. Tal equação descreve uma membrana oscilando quando generalizada para mais dimensões.

A equação é formada por uma derivada temporal de segunda ordem igualando a soma de derivadas segundas no espaço multiplicado por um valor que representa a velocidade de propagação da onda.

A EDP pode ser resolvida analiticamente utilizando o método de separação de variáveis em conjunto com as condições iniciais e de contorno. Sua solução numérica pode ser encontrada por meio do método de diferenças finitas, tal método será utilizado nesse trabalho para resolvê-la.

Neste projeto será apresentado um código na linguagem C para a solução da equação de onda com condições específicas, além de uma série de ferramentas para otimizar o desempenho desse programa. Primeiramente serão utilizadas técnicas de otimização a nível de código, utilizando boas maneiras de se escrever um programa, e em seguida a nível de compilação, onde será introduzido as flags para os compiladores GNU e Intel. Por último será utilizado técnicas de paralelismo para o aumento do desempenho do algoritmo.

2 Objetivo

O objetivo do trabalho é resolver a equação de onda

$$\frac{\partial^2 z}{\partial t^2} = v_0^2 \left[\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} \right], \quad v_0 = \sqrt{\frac{\mu_0}{T_0}} \quad (2.1)$$

com as condições iniciais

$$z(x, y, 0) = \sin(\pi x) \quad (2.2)$$

$$\left[\frac{\partial y}{\partial t} \right]_{t=0} = 0 \quad (2.3)$$

e condições de contorno

$$x = 0 \quad \rightarrow \quad z(0, y, t) = z_0, \quad (2.4)$$

$$x = L \quad \rightarrow \quad z(L, y, t) = z_0 \quad (2.5)$$

$$y = 0 \quad \rightarrow \quad z(x, 0, t) = z_0 \quad (2.6)$$

$$y = K \quad \rightarrow \quad z(x, K, t) = z_0 \quad (2.7)$$

utilizando o método das diferenças finitas e ferramentas de otimização para um melhor desempenho do código. Ao fim fazer uma comparação entre os desempenhos para a versão paralelizada com a versão serial e uma comparação dos resultados obtidos com a solução analítica do problema

$$z(x, y, t) = \sum_{n=1}^{\infty} \sum_{m=1}^{\infty} \sin\left(\frac{m\pi}{a}x\right) \sin\left(\frac{n\pi}{b}y\right) B_{mn} \cos\left(v_0 \sqrt{\left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2} t\right) \quad (2.8)$$

onde

$$B_{mn} = \frac{4}{ab} \int_0^a \int_0^b \sin(\pi x) \sin\left(\frac{m\pi}{a}x\right) \sin\left(\frac{n\pi}{b}y\right) dy dx$$

para $a = 600$ e $b = 600$ temos,

$$B_{mn} = \frac{4000 \sin(\pi m) \sin^2\left(\frac{\pi n}{2}\right)}{\pi^2(m^2 - 360000)n} \quad (2.9)$$

3 Metodologia

3.1 Discretização

O método escolhido para solucionar a equação de onda com as condições mencionadas na introdução 1 foi o método das Diferenças finitas. Primeiramente é definido uma grade no domínio da função de forma que o mesmo seja separado por intervalos discretos de tamanha $h = (b - a)/n$ ¹, onde a e b definem o intervalo e n um inteiro que determina o tamanho das partes. Essa discretização do domínio pode ser vista na figura abaixo:

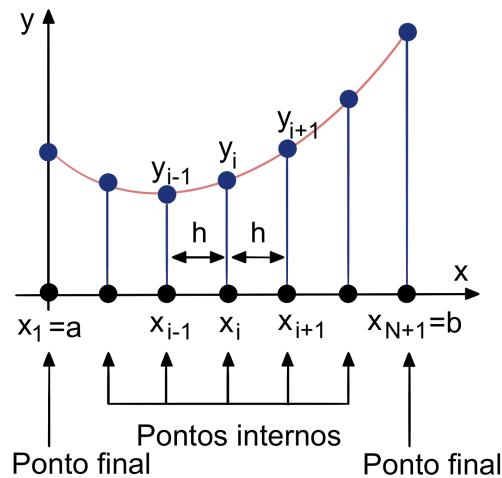


Figura 1 – Elaborado pelo autor

Para a equação da onda 2 dimensões o domínio da solução é:

$$x \in (0, L), \quad y \in (0, K), \quad t \in (0, t_f)$$

Discretizando

$$x_i = (i - 1)\Delta x, \quad i = 1, \dots, N_x \quad \Delta x = \frac{L}{N_x - 1} \quad (3.1)$$

$$y_j = (j - 1)\Delta y, \quad j = 1, \dots, N_y \quad \Delta y = \frac{K}{N_y - 1} \quad (3.2)$$

$$t_k = (k - 1)\Delta t, \quad k = 1, \dots, N_t \quad \Delta t = \frac{t_f}{N_t - 1} \quad (3.3)$$

$$z(x, y, t) = z((i - 1)\Delta x, (j - 1)\Delta y, (k - 1)\Delta t) = z_{i,j}^k \quad (3.4)$$

¹ Intervalo pode ter comprimento diferente para diferentes direções, como $k = (d - c)/m$ na direção y.

Usando diferenças finitas centrais no tempo e na posição temos

$$\frac{\partial^2 z}{\partial t^2} = \frac{z_{i,j}^{k+1} - 2z_{i,j}^k + z_{i,j}^{k-1}}{(\Delta t)^2} \quad (3.5)$$

$$\frac{\partial^2 z}{\partial x^2} = \frac{z_{i+1,j}^k - 2z_{i,j}^k + z_{i-1,j}^k}{(\Delta x)^2} \quad (3.6)$$

$$\frac{\partial^2 z}{\partial y^2} = \frac{z_{i,j+1}^k - 2z_{i,j}^k + z_{i,j-1}^k}{(\Delta y)^2} \quad (3.7)$$

Substituindo na equação da onda 2.1

$$\frac{z_{i,j}^{k+1} - 2z_{i,j}^k + z_{i,j}^{k-1}}{(\Delta t)^2} = v_0^2 \left[\frac{z_{i+1,j}^k - 2z_{i,j}^k + z_{i-1,j}^k}{(\Delta x)^2} + \frac{z_{i,j+1}^k - 2z_{i,j}^k + z_{i,j-1}^k}{(\Delta y)^2} \right] \quad (3.8)$$

rearranjando

$$z_{i,j}^{k+1} = 2z_{i,j}^k - z_{i,j}^{k-1} + \frac{(\Delta t)^2}{(\Delta x)^2} v_0^2 [z_{i+1,j}^k - 2z_{i,j}^k + z_{i-1,j}^k] + \frac{(\Delta t)^2}{(\Delta y)^2} v_0^2 [z_{i,j+1}^k - 2z_{i,j}^k + z_{i,j-1}^k] \quad (3.9)$$

fazendo

$$\lambda^2 = \frac{\Delta t^2}{\Delta x^2} v_0^2 \quad (3.10)$$

$$\delta^2 = \frac{\Delta t^2}{\Delta y^2} v_0^2 \quad (3.11)$$

obtemos

$$z_{i,j}^{k+1} = 2z_{i,j}^k - z_{i,j}^{k-1} + \lambda^2 [z_{i+1,j}^k - 2z_{i,j}^k + z_{i-1,j}^k] + \delta^2 [z_{i,j+1}^k - 2z_{i,j}^k + z_{i,j-1}^k] \quad (3.12)$$

por fim

$$z_{i,j}^{k+1} = 2z_{i,j}^k - z_{i,j}^{k-1} + \lambda^2 z_{i+1,j}^k - \lambda^2 2z_{i,j}^k + \lambda^2 z_{i-1,j}^k + \delta^2 z_{i,j+1}^k - \delta^2 2z_{i,j}^k + \delta^2 z_{i,j-1}^k \quad (3.13)$$

$$z_{i,j}^{k+1} = 2z_{i,j}^k (1 - \lambda^2 - \delta^2) - z_{i,j}^{k-1} + \lambda^2 z_{i+1,j}^k + \lambda^2 z_{i-1,j}^k + \delta^2 z_{i,j+1}^k + \delta^2 z_{i,j-1}^k \quad (3.14)$$

Essa é a equação que dará a evolução temporal da onda gerada pelas condições iniciais e condições de contorno definidas anteriormente.

Agora discretizando a primeira condição inicial 2.2,

$$z(x, y, 0) = \sin(\pi x) \quad \rightarrow \quad z(x_i, y_j, 0) = z_{i,j,0} = \sin(\pi x_i) \quad (3.15)$$

agora a segunda 2.3,

$$\left[\frac{\partial y}{\partial t} \right]_{t=0} = 0 \quad \rightarrow \quad \frac{z_{i,j}^{k+1} - z_{i,j}^{k-1}}{2\Delta t} = 0 \rightarrow z_{i,j}^{k+1} = z_{i,j}^{k-1}, \quad k = 0 \quad (3.16)$$

Logo

$$z_{i,j}^{+1} = z_{i,j}^{-1} \quad (3.17)$$

Usando a condição acima para $k = 0$

$$z_{i,j}^{+1} = 2z_{i,j}^0 (1 - \lambda^2 - \delta^2) - z_{i,j}^{-1} + \lambda^2 z_{i+1,j}^0 + \lambda^2 z_{i-1,j}^0 + \delta^2 z_{i,j+1}^0 + \delta^2 z_{i,j-1}^0 \quad (3.18)$$

se torna

$$z_{i,j}^{+1} = 2z_{i,j}^0 (1 - \lambda^2 - \delta^2) - z_{i,j}^{+1} + \lambda^2 z_{i+1,j}^0 + \lambda^2 z_{i-1,j}^0 + \delta^2 z_{i,j+1}^0 + \delta^2 z_{i,j-1}^0 \quad (3.19)$$

logo

$$2z_{i,j}^{+1} = 2z_{i,j}^0 (1 - \lambda^2 - \delta^2) + \lambda^2 z_{i+1,j}^0 + \lambda^2 z_{i-1,j}^0 + \delta^2 z_{i,j+1}^0 + \delta^2 z_{i,j-1}^0 \quad (3.20)$$

por fim

$$z_{i,j}^{+1} = \frac{2z_{i,j}^0 (1 - \lambda^2 - \delta^2) + \lambda^2 z_{i+1,j}^0 + \lambda^2 z_{i-1,j}^0 + \delta^2 z_{i,j+1}^0 + \delta^2 z_{i,j-1}^0}{2} \quad (3.21)$$

Discretizando as condições de contorno

$$x = 0 \quad \rightarrow \quad z(0, y_j, t_n) = z_0, \quad (3.22)$$

$$x = L \quad \rightarrow \quad z(L, y_j, t_n) = z_0, \quad (3.23)$$

$$y = 0 \quad \rightarrow \quad z(x_i, 0, t_n) = z_0, \quad (3.24)$$

$$y = K \quad \rightarrow \quad z(x_i, K, t_n) = z_0, \quad (3.25)$$

3.2 O método das Diferenças Finitas

Como resultado da discretização foi

$$z_{i,j}^{k+1} = 2z_{i,j}^k (1 - \lambda^2 - \delta^2) - z_{i,j}^{k-1} + \lambda^2 z_{i+1,j}^k + \lambda^2 z_{i-1,j}^k + \delta^2 z_{i,j+1}^k + \delta^2 z_{i,j-1}^k \quad (3.26)$$

para cada $i = 1, 2, 3, \dots, n$ e $j = 1, 2, 3, \dots, m$ temos a solução da nossa equação diferencial dada por aproximações discretas $z(x, y, t) \approx z(x_i, y_j, t_k)$.

A solução no ponto $z(i+1,j+1,k+1)$ depende dos valores da malha no tempo k e $k-1$ e nas posições i , $i-1$, j e $j-1$ como ilustrado na figura abaixo:

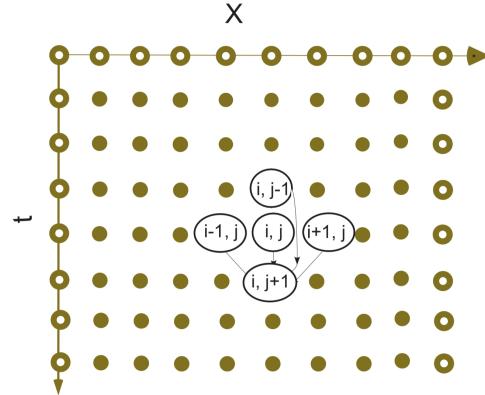
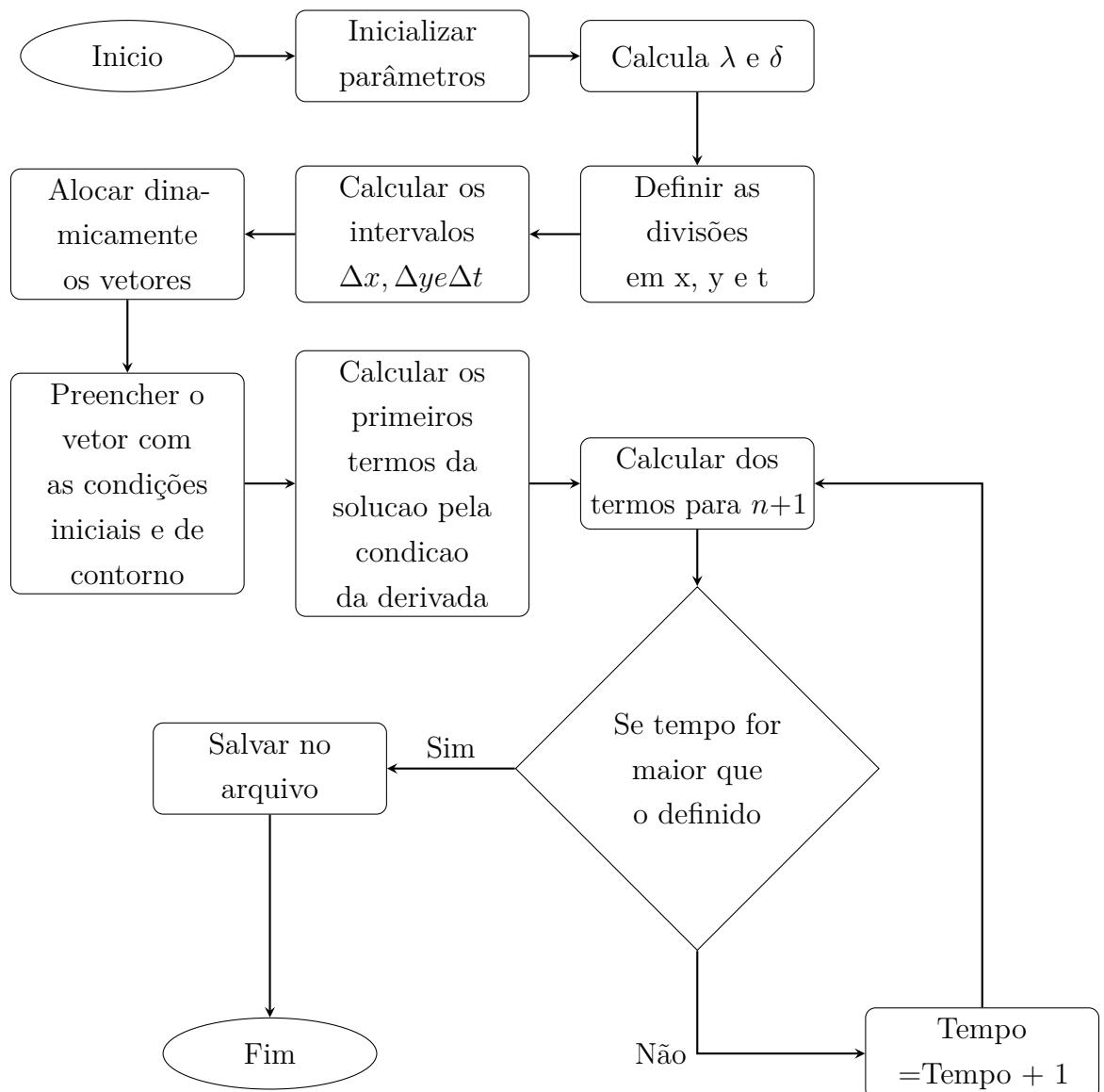


Figura 2 – Dependência dos dados

4 Implementação

4.1 Fluxograma

Para a melhor construção do programa, foi feito um fluxograma ilustrando os funcionamento do código, o fluxograma é o seguinte:



4.2 Código Serial

Para a solução da EDP, foi utilizado o método de diferenças finitas, no qual o domínio e as condições iniciais e de contorno do problema são discretizados.

O programa inicializa uma matriz z com 3 dimensões, cada dimensão com seus

respectivos tamanhos, primeira coordenada da matriz representa o x com tamanho 6000, segunda coordenada representa o y com tamanho 6000, logo temos uma **malha 6000 x 6000**, e a última coordenada representa o tempo t com tamanho 3. O tamanho 3 foi definido para o tempo com o intuito de economizar memória e não sobrecarregar o computador, tornando possível a escolha de uma quantidade de passos ou malha maiores. O valor 3 foi escolhido sabiamente, pois sabendo que com a discretização da equação da onda, vamos precisar de um valor no presente (j) e outro no passado vizinho (j-1) para poder achar o elemento do futuro (j+1), como mostra a Figura 2.

Após a criação da matriz solução z, é inicializado todos os parâmetros a serem usados no decorrer do programa, também é calculado os Δx e Δy e então a matriz z é alocada, assim é preenchida com as condições iniciais e de contorno. Usamos a condição da derivada em $t = 0$ para definir os valores para $t = 1$.

Depois desses passos, começa o looping para calcular os termos da matriz z para tempos futuros, no caso foi definido um tempo final $tf = 16000$. Nesse looping os termos são calculados com a equação encontrada na discretização da equação de onda e a divisão modular (%) para reciclar nossa posição do tempo, sempre substituindo a posição do tempo que não será mais utilizada para os cálculos posteriores. Quando chega na interação $tf - 1$ a matriz z é salva em um arquivo chamado Wave.dat, tal arquivo será usado para plotar o gráfico da solução no tempo $tf - 1$, no nosso caso $tf - 1 = 15999$.

O código que realiza os procedimentos descrito acima foi escrito na linguagem C e se apresenta abaixo:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <math.h>
5 #include <time.h>
6
7 //Parametros
8 double mu = 0.01;
9 double tensao = 40;
10 double xi = 0;
11 double xf = 600;
12 double yi = 0;
13 double yf = 600;
14 double ti = 0;
15 double tf = 16000;
16 double lambda = 0.4;
17 double delta = 0.8;
18
19 void cabecalho(){
20     printf("\n");
21     printf("=====\\n");

```

```
22 printf("Programa Pablo-WaveEquation2D-06.c v.06\n");
23 printf("Autor: Pablo de Deus\n");
24 printf("Data : 24/02/21\n");
25 printf("Estado: Estavel\n");
26 printf("=====\\n");
27 printf("\n");
28 }
29
30 double ***alocando(int m, int n){
31     int i, j;
32     double ***z;
33     //Alocando a matriz z
34     z = (double***) malloc(n*sizeof(double));
35     for (i = 0; i < n; ++i)
36     {
37         z[i] = (double**) malloc(m*sizeof(double));
38         for (j = 0; j < m; ++j)
39         {
40             z[i][j] = (double*) malloc(3*sizeof(double));
41         }
42     }
43     return z;
44 }
45
46 double ***alocandoInicial (double ***z, int m, int n, double dx, double
    dy){
47     int i, j;
48     for (i = 0; i < n; ++i)
49     {
50         for (j = 0; j < m; ++j)
51         {
52             z[i][j][0] = sin(M_PI*i*dx/75);
53         }
54     }
55     return z;
56 }
57
58 double ***contorno(double ***z, int m, int n){
59     int i, j, k;
60
61     //Em x = 0 e x = n
62     for (j = 0; j < m; ++j)
63     {
64         for (k = 0; k < 3; ++k)
65         {
66             z[0][j][k] = 0;
67             z[n-1][j][k] = 0;
```

```
68     }
69 }
70
71 //Em y = 0 e y = n
72 for (i = 0; i < n; ++i)
73 {
74     for (k = 0; k < 3; ++k)
75     {
76         z[i][0][k] = 0;
77         z[i][m-1][k] = 0;
78     }
79 }
80 return z;
81 }

82
83 double ***condicaodaderivada(double ***z, int m, int n){
84     int i, j;
85     for (i = 1; i < n-1; ++i)
86     {
87         for (j = 1; j < m-1; ++j)
88         {
89             z[i][j][1] = (2*z[i][j][0]*(1 - lambda*lambda - delta*delta) +
90             lambda*lambda*z[i+1][j][0] + lambda*lambda*z[i-1][j][0] + delta*delta *
91             *z[i][j+1][0] + delta*delta*z[i][j-1][0])/2;
92         }
93     }
94     return z;
95 }
96
97 void salvarArquivo(FILE *arquivo, double***z, int i, int j, int k,
98                     double dx, double dy){
99     fprintf(arquivo, "%lf\t%lf\t%lf\t", i*dx, j*dy, z[i][j][(k+1)%3]);
100    fprintf(arquivo, "\n");
101 }
102
103
104 double ***calculoEquacaoDiscreta(double ***z, int i, int j, int k){
105     z[i][j][(k+1)%3] = 2*z[i][j][k%3]*(1 - lambda*lambda - delta*
106     delta) - z[i][j][(k-1)%3] + lambda*lambda*z[i+1][j][(k%3)] + lambda*
107     lambda*z[i-1][j][k%3] + delta*delta*z[i][j+1][k%3] + delta*delta*z[i]
108     [j-1][k%3];
109
110     return z;
111 }
112
113
114 double ***calculoElementosDaMatriz(double ***z, int m, int n, double dx,
115                                     double dy, FILE *arquivo){
116     int i, j, k;
117     for (k = 1; k < tf; ++k)
118     {
```

```
108     for (i = 1; i < n-1; ++i)
109     {
110         for (j = 1; j < m-1; ++j)
111         {
112             z = calculoEquacaoDiscreta(z, i, j, k);
113             //Salvando no arquivo a matriz quando t = tf-1
114             if (k == tf-1)
115             {
116                 salvarArquivo(arquivo, z, i, j, k, dx, dy);
117             }
118         }
119     }
120 }
122
123 int main(){
124     double dx, dy;
125     double ***z;
126     int n, m, i, j, k;
127     FILE *arquivo;
128
129     //Cabecalho
130     cabecalho();
131
132     //Comecando a contar o tempo
133     clock_t begin = clock ();
134
135     //Tamanho da malha
136     n = 6000;
137     m = 6000;
138
139     //Calculando os elementos dx e dy
140     dx = (xf-xi)/n;
141     dy = (yf-yi)/m;
142
143     //Alocando a matriz z
144     z = alocando(m, n);
145
146     //Colocando a condicao inicial
147     z = alocandoInicial(z, m, n, dx, dy);
148
149     //Colocando a condicao de contorno
150     z = contorno(z, m, n);
151     //Usando a condicao inicial da derivada em t = 0 para o calculo dos
152     //termos da matriz z para o tempo k = 1
153     z = condicaodaderivada(z, m, n);
154     //Abrindo o arquivo que sera salvo a matriz para um tempo especifico
```

```

154     , t = tf-1
155     arquivo = fopen("Wave.dat", "w");
156     //Calculo dos proximos termos da matriz z para k > 1
157     z = calculoElementosDaMatriz(z, m, n, dx, dy, arquivo);
158     //Fechando o arquivo
159     fclose(arquivo);
160
160     //Terminando o tempo
161     clock_t end = clock ();
162     printf("Tempo de execucao: %10.2f segundos \n", (end - begin)/(1.0*
163     CLOCKS_PER_SEC));
163
164     return 0;
165 }
```

Listing 4.1 – Programa serial em C

4.3 Resultados

Para a configuração com uma **malha 6000 x 6000**, variando o **tempo de 0 a 16000s**, o programa demorou **13168,97 segundos**, equivale a **3 horas 39 minutos e 28 segundos**, a superfície obtida foi a seguinte:

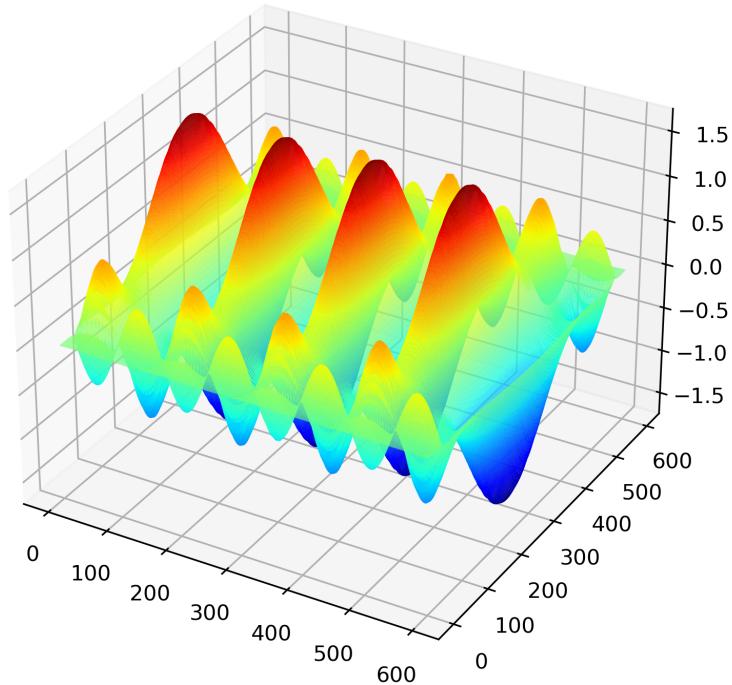


Figura 3 – Solução da equação da onda

5 Benchmark

5.1 Laboratório 107

Os Benchmarks foram feitos com as flags *-fexpensive-optimizations*, *-m64*, *-foptimize-register-move*, *-funroll-loops*, *-ffast-math*, *-mavx*, *-march=native*, e *-mtune=native* e variando *-O0*, *-O1*, *-O2* e *-O3*, os tempos obtidos para o laboratório 107 foram os seguintes:

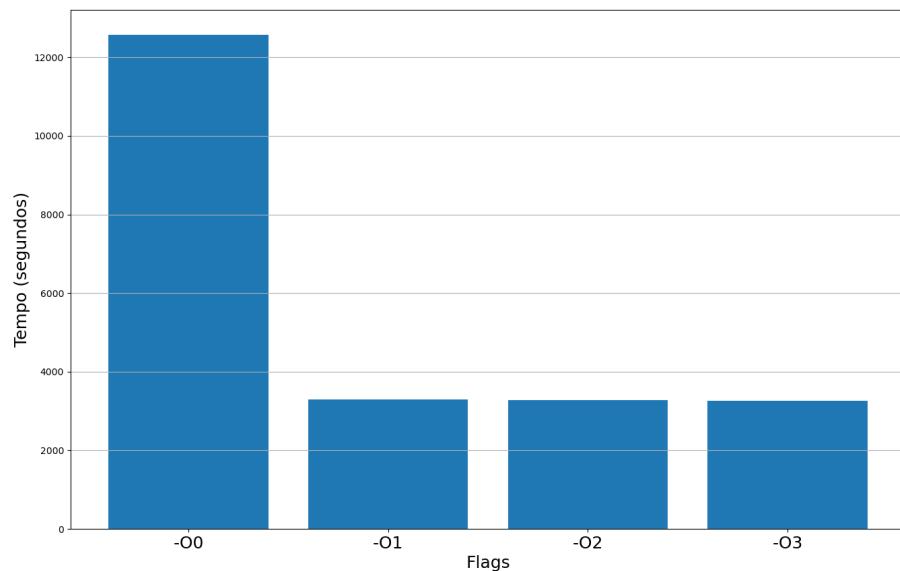


Figura 4 – Tempo para cada flag no laboratório 107

e a eficiência para cada flag em relação a flag -O0:

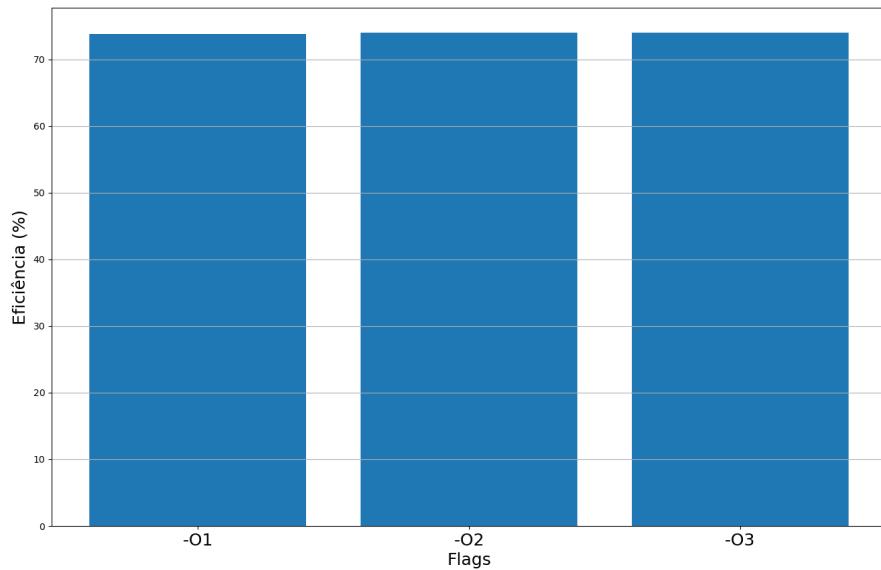


Figura 5 – Eficiência para cada flag no laboratório 107

Para o compilador intel foi utilizado dois conjuntos de flags, o primeiro conjunto foi `-mp1 -axAVX -ipo -OX` e os resultados para essas flags se encontram no gráfico abaixo:

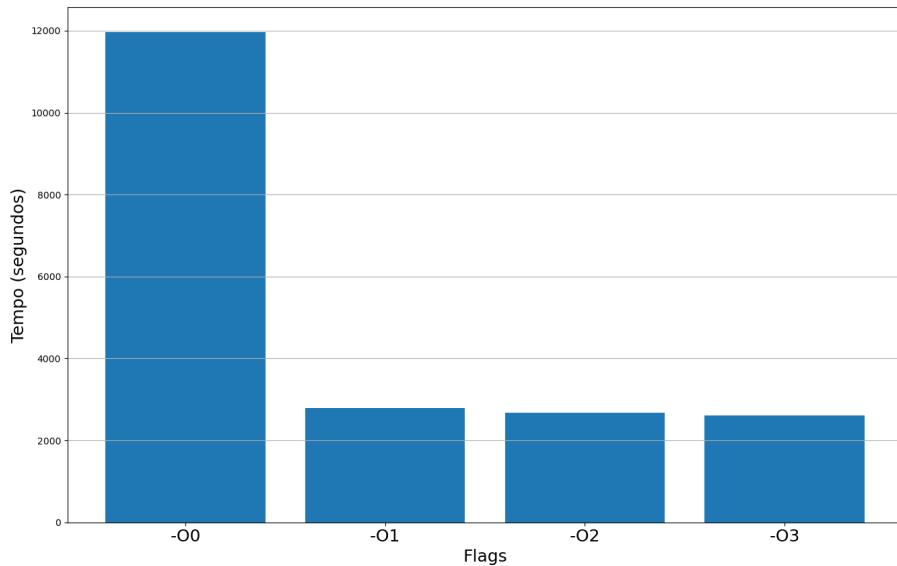


Figura 6 – Tempo para cada flag no laboratório 107

A eficiência para as mudanças dos -OX em relação ao -O0 foi:

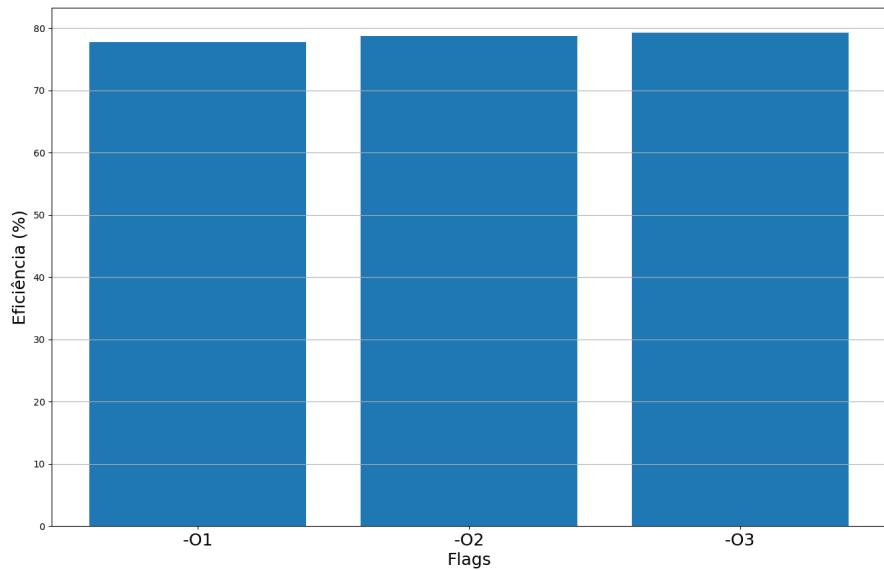


Figura 7 – Eficiência para cada flag no laboratório 107

O segundo conjunto de flags foi $-w -mp1 -xHOST -fast=2 -OX$ e os resultados estão no gráfico abaixo:

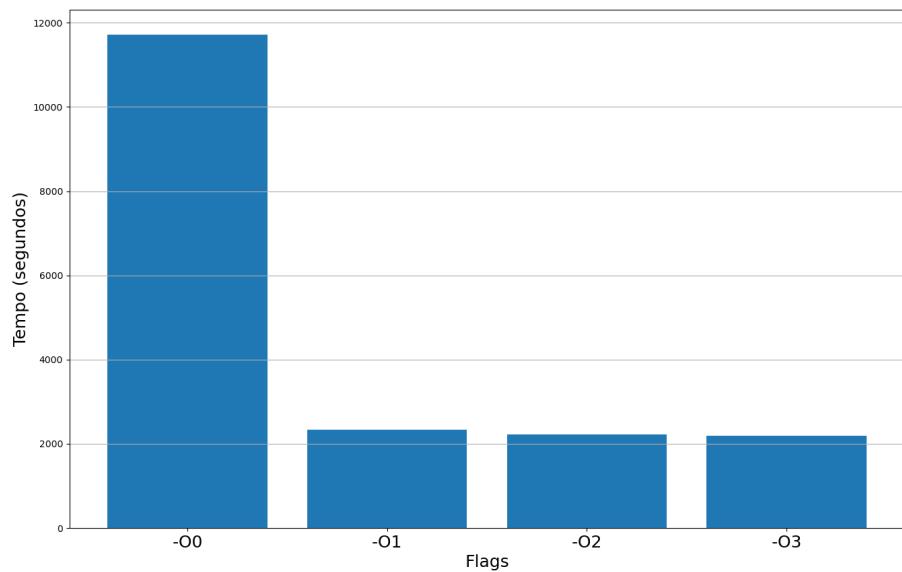


Figura 8 – Tempo para cada flag no laboratório 107

da mesma forma para o primeiro conjunto, a eficiência para o segundo pode ser visualizada no gráfico a seguir:

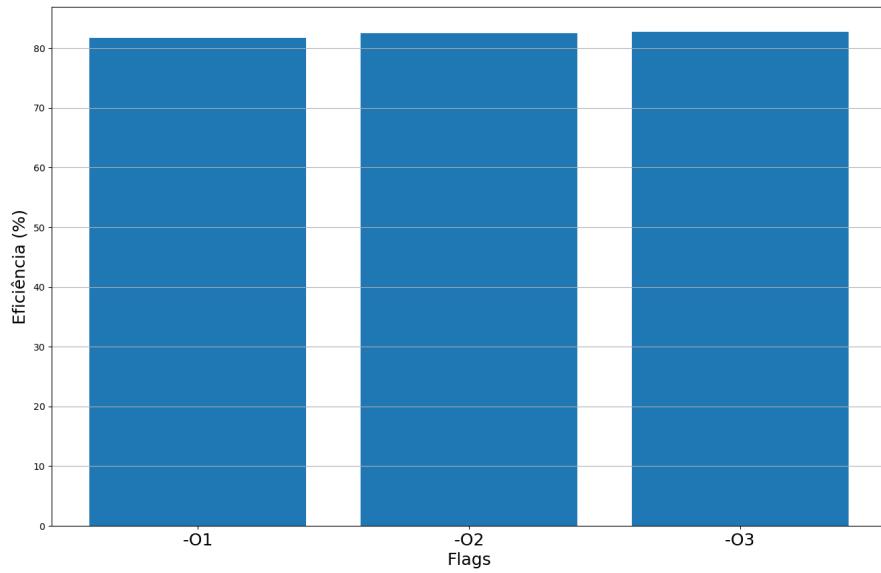


Figura 9 – Eficiência para cada flag no laboratório 107

A comparação entre os tempos obtidos pelo compilador GCC e o Ifort é mostrada adiante:

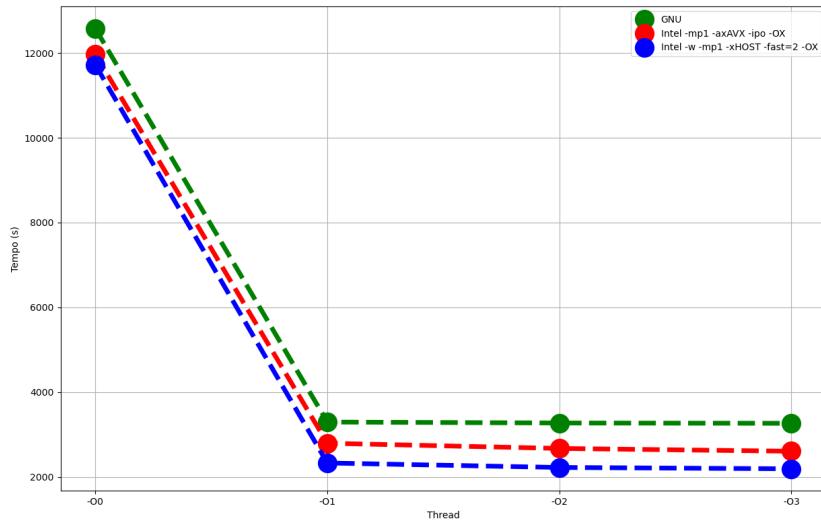


Figura 10 – Comparaçāo do tempo entre o compilador GCC e o Ifort

Comparação entre as eficiências:

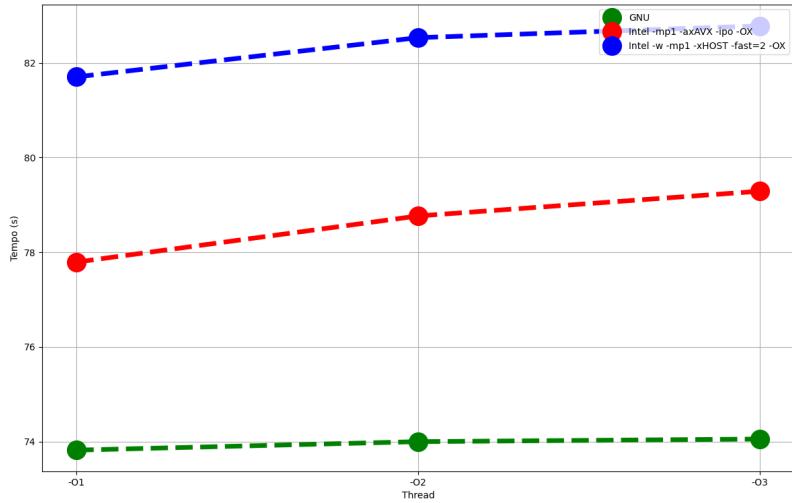


Figura 11 – Comparação das eficiências entre o compilador GCC e o Ifort

Os tempos são próximos, mas é visível a eficiência do o Ifort com as flags `-w -mp1 -xHOST -fast=2 -OX` em relação ao GNU e ao próprio Ifort com outro conjunto de flags.

5.2 Notebook

Foi feito um benchmark do código para o notebook pessoal, segue as configurações **Processador**

Architecture: 32-bit, 64-bit

Byte Order: Little Endian

CPU(s): 4

Thread(s) per core: 2

Core(s) per socket: 2

Socket(s): 1

Model name: Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz

CPU MHz: 1404.083

CPU max MHz: 3000,0000

CPU min MHz: 500,0000

L1d e L1i cache: 64 KiB

L2 cache: 512 KiB

L3 cache: 4 MiB

Memória Ram

16 GBs 1600 MHz

Placa de Vídeo

Nvidia GTX 960M GDDR5 128 Bit

os resultados obtidos para o laptop foram:

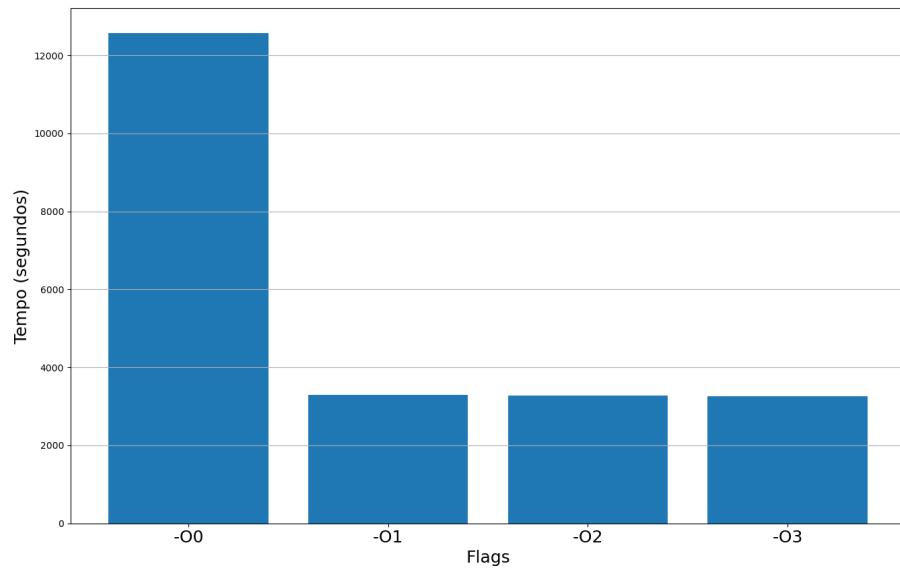


Figura 12 – Tempo para cada flag no Notebook

e a eficiência para cada flag em relação a -O0 foi:

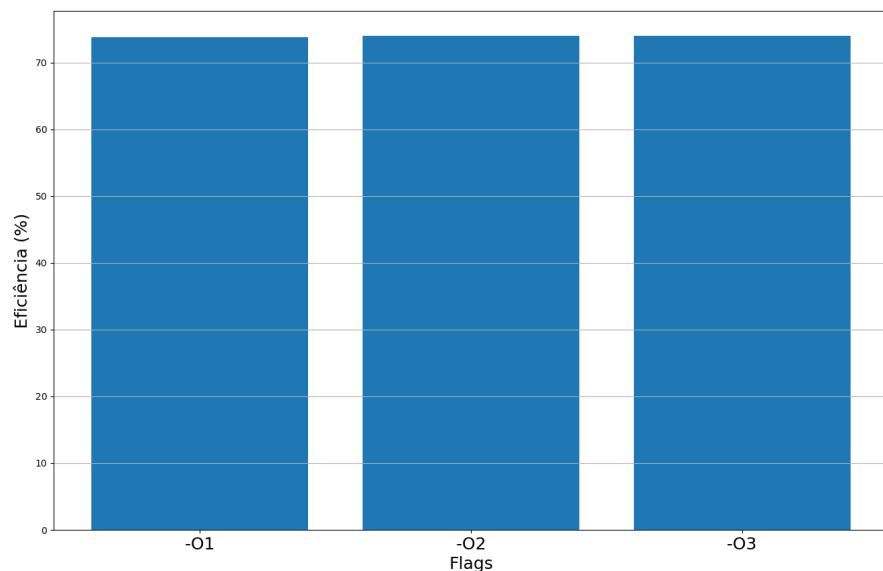


Figura 13 – Eficiência para cada flag no Notebook

5.3 Comparação laboratório e notebook

Os resultados obtidos no laboratório são muito semelhantes ao encontrado pelo notebook como pode ser visto no gráfico abaixo:

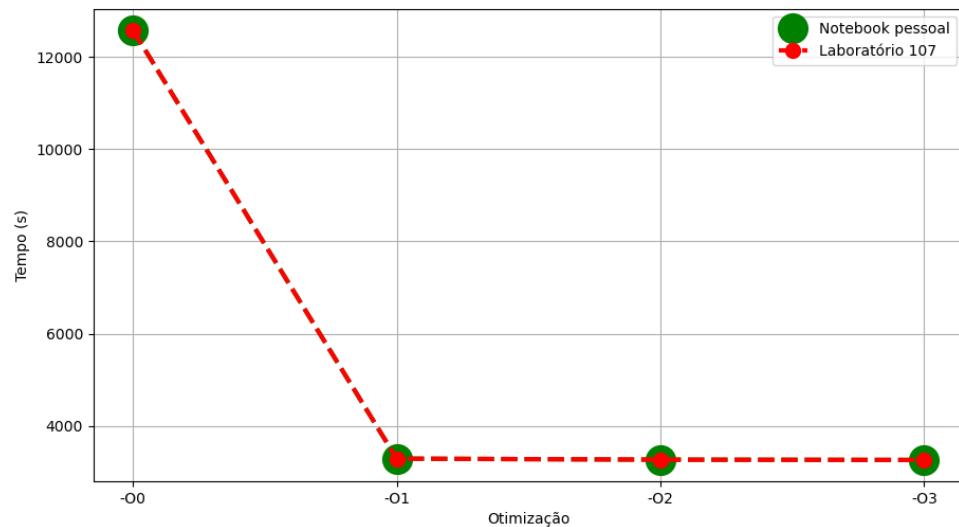


Figura 14 – Comparação do tempo entre laboratório e notebook

da mesma forma que os gráficos 4 e 12 mostram.

6 Profile

No laboratório 107 o programa demorou 3226.30 segundos para uma malha de 6000 x 6000 pontos com o tempo variando de 1 até 16000 como mostra na figura ??, é possível ver também que a função mais chamada é a calculoProximosElementos, que consiste no cálculo dos elementos $z_{i,j}^{k+1}$, ou seja, o elemento do próximo tempo.

	Each sample counts as 0.01 seconds.						
	%	cumulative	self	self	total		
time	seconds	seconds	calls	Ks/call	Ks/call	name	
91.95	13488.07	13488.07	54470332	0.00	0.00	calculoEquacaoDiscreta	
7.49	14586.79	1098.71	1	1.10	14.67	calculoProximosElementos	
0.54	14665.71	78.93	35976004	0.00	0.00	salvarArquivo	
0.00	14666.17	0.46	1	0.00	0.00	condicaodaderivada	
0.00	14666.41	0.24	1	0.00	0.00	alocandoInicial	
0.00	14666.65	0.24	1	0.00	0.00	alocando	
0.00	14666.65	0.00	1	0.00	0.00	cabecalho	
0.00	14666.65	0.00	1	0.00	0.00	contorno	

Figura 15 – Profile da execução do programa com o cálculo e o salvamento dos dados feitos por funções a parte.

Através do *profile* foi possível ver que a função que tem o maior gasto computacional é a responsável pelos cálculos dos elementos, essa será a função que terá uma maior atenção posteriormente, a qual será paralelizada.

7 Técnicas de otimização de software

Algumas alterações foram feitas no programa com o intuito de otimizá-lo, a principal delas foi a alteração das funções *calculoEquacaoDiscreta* e *salvarArquivo*, esses métodos foram removidas e implementadas dentro da função *calculoElementosDaMatriz*, pois são funções que constantemente são chamadas como mostrado no capítulo 6. Essa mudança pode ser vista abaixo

```

1 double ***calculoElementosDaMatriz(double ***z, int m, int n, double dx,
2   double dy, FILE *arquivo){
3   int i, j, k;
4   for (k = 1; k < tf; ++k)
5   {
6     for (i = 1; i < n-1; ++i)
7     {
8       for (j = 1; j < m-1; ++j)
9       {
10      z[i][j][(k+1)%3] = 2*z[i][j][k%3]*(1 - lambda*lambda - delta*
11        delta) - z[i][j][(k-1)%3] + lambda*lambda*z[i+1][j][(k%3)] + lambda*
12        lambda*z[i-1][j][k%3] + delta*delta*z[i][j+1][k%3] + delta*delta*z[i]
13        ][j-1][k%3];
14
15      //Salvando no arquivo a matriz quando t = tf-1
16      if (k == tf-1)
17      {
18        fprintf(arquivo, "%lf\t%lf\t%lf\t%lf\n", i*dx, j*dy, z[i][j]
19        ][(k+1)%3]);
15      }
16    }
17  }
18}

```

Listing 7.1 – Otimização de software no código serial

Após as modificações o programa foi novamente executado e obtido o seguinte profile

Each sample counts as 0.01 seconds.						
%	cumulative	self	self	total		
time	seconds	seconds	calls	Ks/call	Ks/call	name
100.16	12890.60	12890.60	1	12.89	12.89	calculoProximosElementos
0.00	12891.06	0.46	1	0.00	0.00	condicaodaderivada
0.00	12891.32	0.26	1	0.00	0.00	alocando
0.00	12891.45	0.13	1	0.00	0.00	alocandoInicial
0.00	12891.45	0.00	1	0.00	0.00	cabecalho
0.00	12891.45	0.00	1	0.00	0.00	contorno

8 Comparação do programa base com o otimizado

Para maior desempenho no programa serial, foi utilizado alguns flags do compilador GNU e no da Intel. Para o GCC foi utilizado as seguintes flags *-fexpensive-optimizations*, *-m64*, *-foptimize-register-move*, *-funroll-loops*, *-ffast-math*, *-mavx*, *-march=native*, *-mtune=nativee* e *-OX*, com X variando de 0 a 3. Para o Ifort foi utilizado as seguintes flags *-mp1 -axAVX -ipo -OX* para uma execução e *-w -mp1 -xHOST -fast=2 -OX* para outra, com X variando da mesma forma.

Como pode se visualizar nos gráficos [4](#) e [5](#), para o laboratório o programa passou de 13168,97 segundos para 3270.359 segundos apenas utilizando as flags, uma eficiência de 75,16%. Para o computador pessoal, utilizando as mesmas flags o tempo serial passou de 12578.02 segundos para 3263.825 segundos como mostrado nos gráficos [12](#) e [13](#), uma eficiência de 74,05%.

9 Implementação do programa com openMP

Tendo realizado todas otimizações no programa serial, utilizando técnicas de otimização de software e de compilação com as flags, agora o código será paralelizado para que faça uma comparação das eficiências.

9.1 Análise do código

Durante a análise do profiling, capítulo 6, foi visto que a função que mais consome recursos computacionais é o método que efetua os cálculos dos elementos da matriz *calculoElementosDaMatriz*. O grande problema dessa função é a necessidade de uma sequência de três *for*, onde um for está dentro do outro, sendo que o primeiro for desloca no tempo, partindo de 1 até 16000, o segundo anda na direção x da malha e o terceiro na direção y, a consequência disso é um programa de ordem $O(n^3)$ de complexidade.

9.2 Implementação com openMP

O tamanho da malha foi definido como 6000 x 6000 e a implementação do paralelismo com openMP será feito na função que mais consome os recursos, *calculoElementosDaMatriz*, essa função pois três *for* em sequência. A melhor forma encontrada para distribuir os cálculos entre as 8 threads do computador do laboratório 107 foi aplicar o paralelismo no *for* do deslocamento da primeira coordenada espacial que no nosso código é representado pelo segundo *for* da sequência. A implementação foi feita de forma que cada thread fique responsável por 3 passos.

A implementação no código foi feito pela diretiva *#pragma omp parallel for* que pode ser vista a seguir:

```

1 double ***calculoProximosElementos(double ***z, int m, int n, double dx,
2                                     double dy, FILE *arquivo, int chunck){
3     int i, j, k, p;
4     p = (int)tf;
5     for (k = 1; k < p; ++k)
6     {
7         #pragma omp parallel for default(shared) private (i, j) schedule(
8             static, chunck)
9         for (i = 1; i < n-1; ++i)
10        {
11            for (j = 1; j < m-1; ++j)
12            {
13                z[i][j] = z[i][j] + dx * dy * tf;
14            }
15        }
16    }
17 }
```

```
11     z[i][j][(k+1)%3] = 2*z[i][j][k%3]*(1 - lambda*lambda - delta*
12     delta) - z[i][j][(k-1)%3] + lambda*lambda*z[i+1][j][(k%3)] + lambda*
13     lambda*z[i-1][j][k%3] + delta*delta*z[i][j+1][k%3] + delta*delta*z[i]
14     ][j-1][k%3];
15
16     //Salvando no arquivo a matriz quando t = tf-1
17     if (k == tf-1)
18     {
19         fprintf(arquivo, "%lf\t%lf\t%lf\n", i*dx, j*dy, z[i][j][(k+1)
20         %3]);
21     }
22 }
```

Listing 9.1 – Programa em C

Os parâmetros utilizados foram:

- Default: As variáveis não especificadas são definidas como shared;
- Private: Variáveis não compartilhadas com as outras threads, são elas i e j;
- Schedule: O intervalo do *for* é dividido de acordo com o chunk e estaticamente por causa do *static*, o tamanho do chunk foi definido como 3.

10 Benchmark do programas com a comparação de várias threads

Com a utilização das flags foi possível obter um grande desempenho em relação ao programa serial sem flags, agora com a implementação do paralelismo foi obtido ainda mais desempenho. As threads foram variadas de 1 até 8 para o laboratório e para o notebook pessoal.

Os tempos para o computador do laboratório 107 são dados a seguir

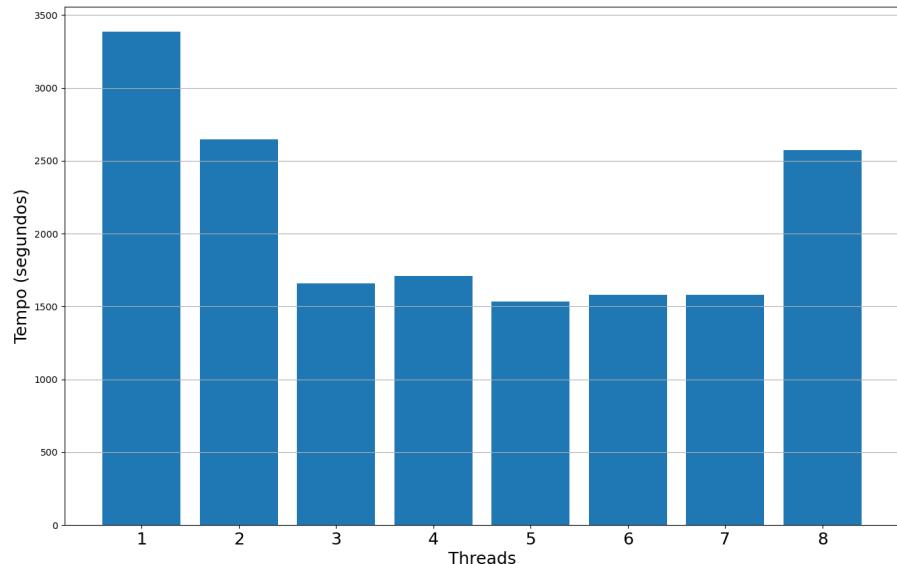


Figura 16 – Tempo por Threads para o laboratório 107

A eficiência dos resultados encontrados para o código em paralelo com as flags de otimização `-mp1 -axAVX -ipo -OX -w -mp1-xHOST -fast=2 -OX` em relação ao tempo encontrado pelo código serial sem nenhuma flag chegou a 88.3% para 5 threads, os resultados estão no gráfico a seguir:

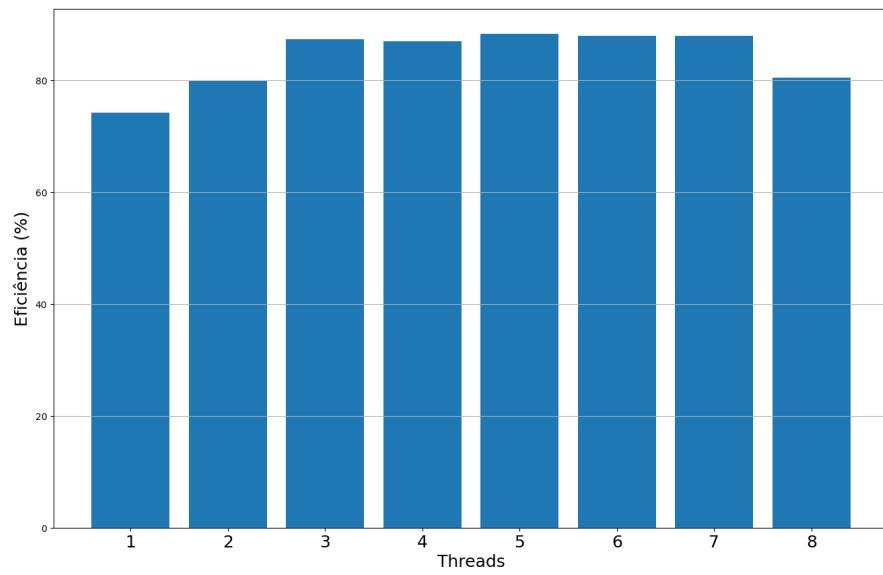


Figura 17 – Eficiência por Threads para o laboratório 107

Da mesma forma que para o laboratório 107 foi feito para o computador pessoal e o resultado se encontra no gráfico abaixo:

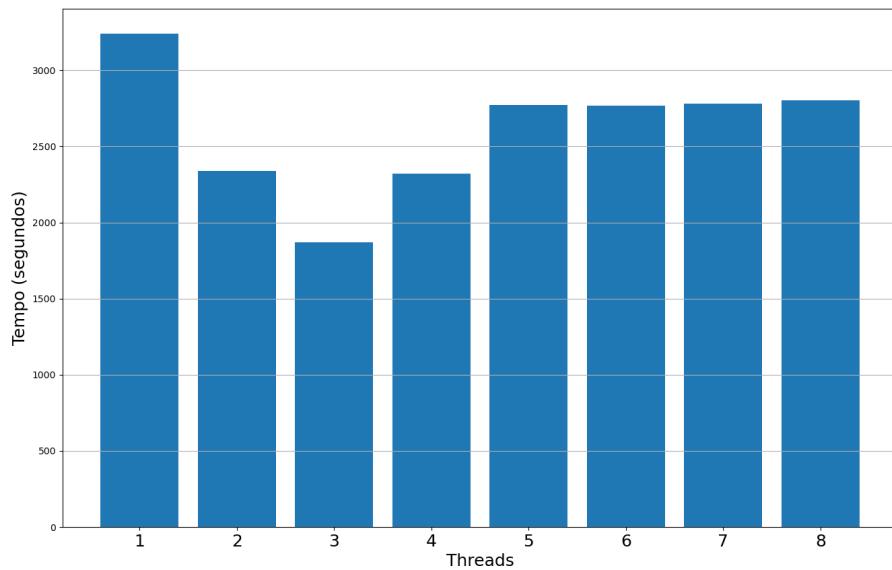


Figura 18 – Tempo por Threads para o computador pessoal

Diferente do computador do laboratório que teve maior desempenho para 5 threads, o notebook teve o melhor resultado para 3 threads conseguindo de 85.79% eficiência

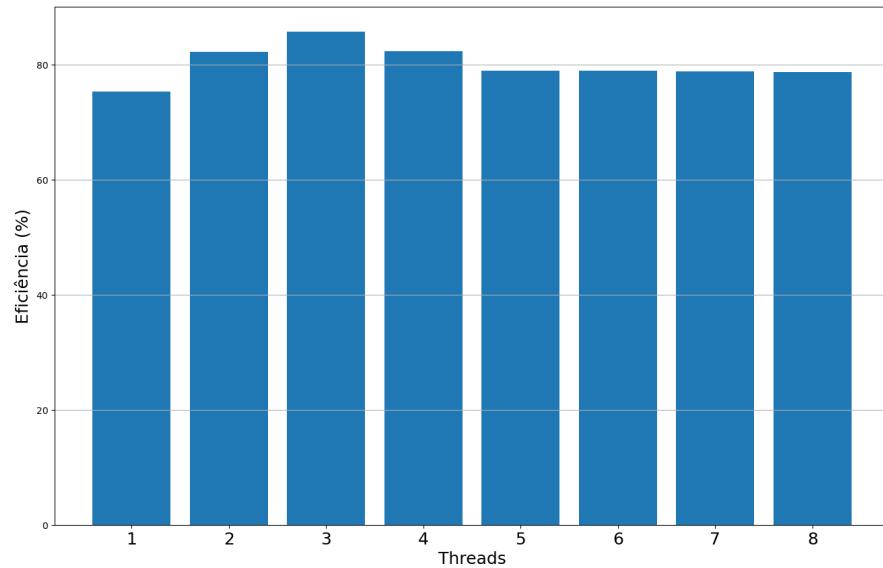


Figura 19 – Eficiência por Threads para o computador pessoal

A comparação entre os tempos do laboratório 107 com o notebook pessoal são dados no gráfico abaixo, no qual é possível ver que o notebook pessoal demonstrou ser mais eficiente para o uso de poucas threads e a partir desse número o computador do laboratório mostrou maior desempenho

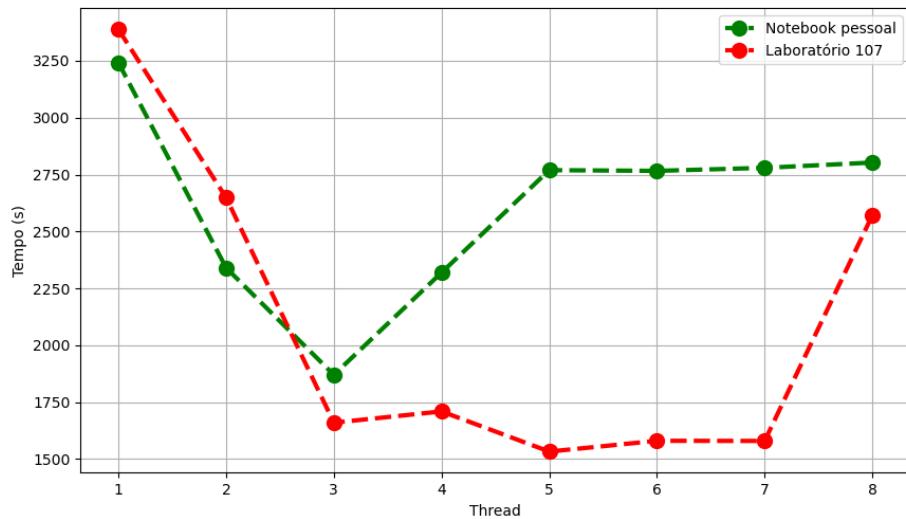


Figura 20 – Comparaçāo entre os tempos por Threads do laboratório 107 com o computador pessoal

11 Supercomputador Santos Dumont

Apesar de várias otimizações realizadas e um grande desempenho alcançado, todos os resultados foram adquiridos de computadores que estão fazendo outras tarefas simultaneamente e que possuem processadores populares, com a intenção de fazer um teste mais apurado e abrangendo um nível de desempenho maior, foi disponibilizado o Supercomputador Santos Dumont (LNCC) para a realização de testes no código para a solução da equação da onda.

Processador

Model name: Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz

Byte Order: Little Endian

CPU(s): 24

On-line CPU(s) list: 0-23

Thread(s) per core: 1

Core(s) per socket: 12

Socket(s): 2

NUMA node(s): 2

CPU family: 6

L1d cache: 32K

L1i cache: 32K

L2 cache: 256K

L3 cache: 30720K

NUMA node0 CPU(s): 0-11

NUMA node1 CPU(s): 12-23

Os valores encontrados para a solução em um nó do LNCC com 24 Threads ¹ a uma malha de 6000 x 6000 com tempo final reduzido a 50 ² pode ser visto no gráfico abaixo:

¹ Não foi possível executar o código para o computador sequana de 48 Threads.

² O tempo foi reduzido pois o computador do LNCC estava limitado a executar o código a no máximo 20 minutos.

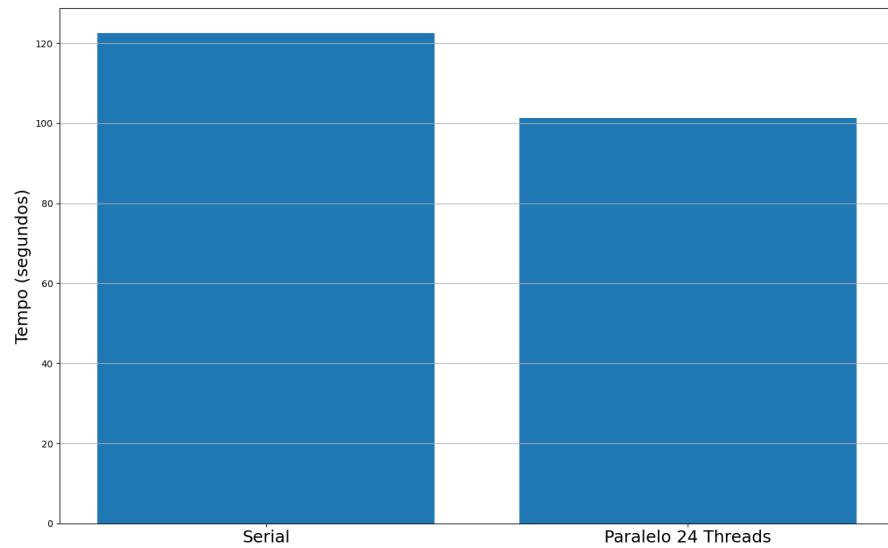


Figura 21 – Comparaçāo para o código serial e paralelo executado no LNCC

Uma comparação entre o tempo para a execução do código em serial com tempo final igual 50 entre o LNCC, laboratório 107 e notebook pessoal pode ser visto a seguir:

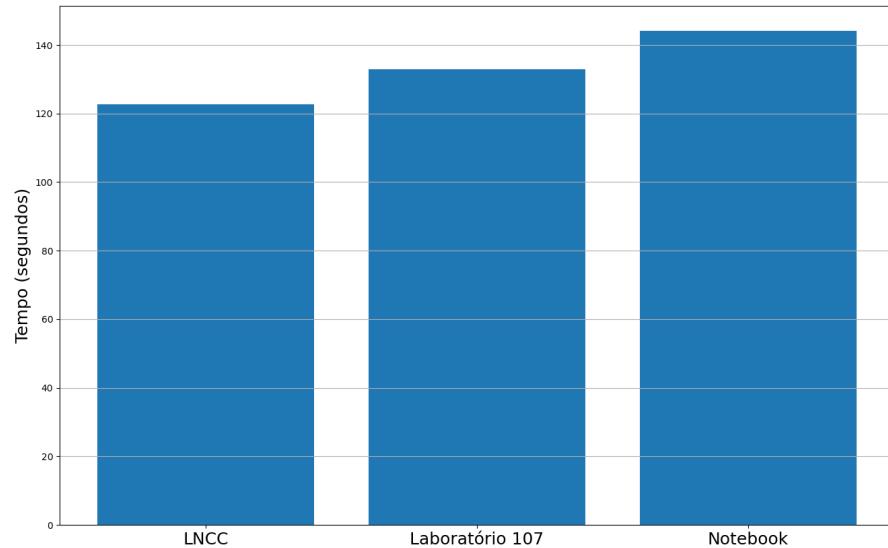


Figura 22 – Comparaçāo para o código serial

Uma mesma comparação para o código em paralelo

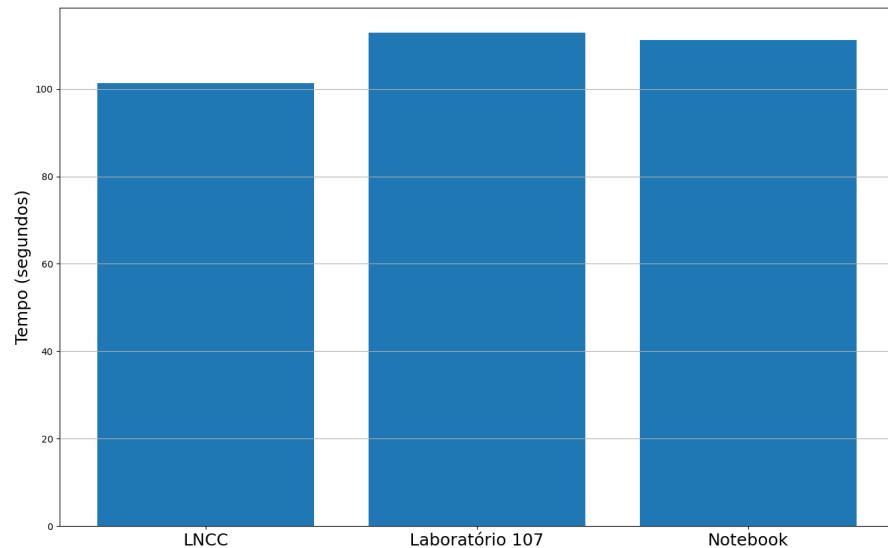


Figura 23 – Comparação para o código paralelo

Pode ser visto que o LNCC teve um desempenho maior, mesmo que pareça pequeno, para grandes escalas essa diferença se torna considerável.

12 Validação dos resultados

O resultado para o código serial pode ser visto no plote da superfície para o tempo 1580 segundos mostrado na figura abaixo:

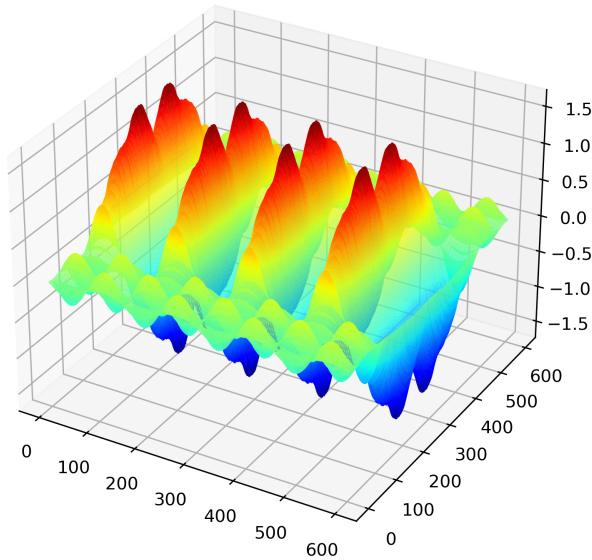


Figura 24 – Superfície gerada pelo código em serial

De forma semelhante o plote foi efetuado para o código em paralelo, é possível ver que não há diferença na parte do plote da superfície, segue o plot a partir dos dados gerados pelo programa com openMP:

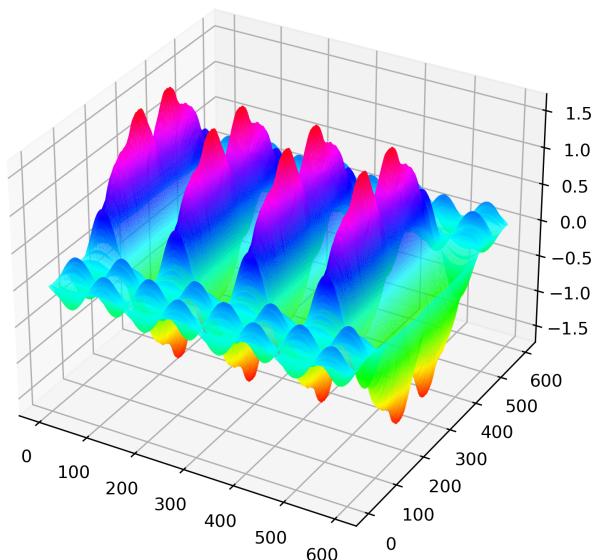


Figura 25 – Superfície gerada pelo código em paralelo

Agora para a solução analítica utilizando as equações 2.8 e 2.9:

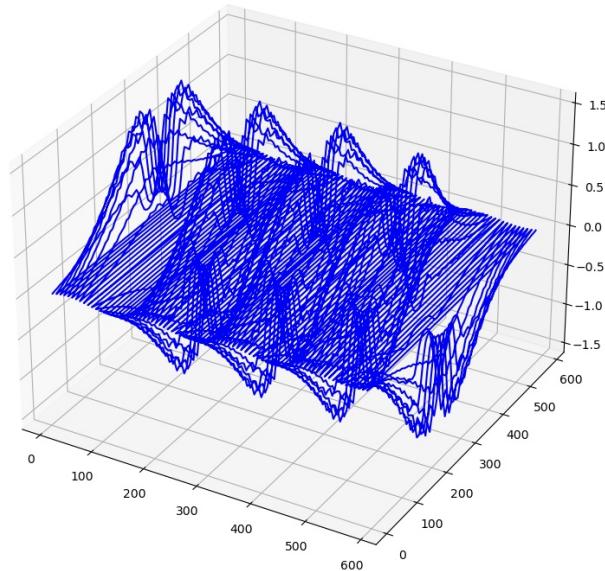


Figura 26 – Superficie gerada pela solução analítica

A solução analítica se difere da solução numérica em razão da aproximação nos cálculos dos B_{mn} , pois m e n vão de 0 até infinito e na prática só é possível variar até um número finito, logo houve uma diferença entre as soluções.

13 Conclusão

O trabalho trouxe resultados importantes para demonstrar a eficiência das técnicas de otimização e paralelismo apresentadas no curso de Programação de Alto Desempenho I, essas ferramentas se tornaram essenciais para a execução de um código que demanda grande recurso computacional.

As ferramentas aprendidas no curso mostrou grande eficaz, partindo das boas maneiras de se estruturar um código, seguindo para as otimizações ao nível de compilação utilizando flags, para a execução do programa em serial foi obtido uma eficiência de 76,19% para o laboratório 107 e 74,05% para o notebook pessoal, mostrando que as flags empenham grande potencial na eficiência de um programa.

Utilizando a ferramenta de *profiling* foi possível identificar em qual parte estava sendo demandado o maior tempo de execução e processamento do computador, com a análise da região foi possível paralelizar de forma que não altere a conexão entre os dados, mantendo a integridade da solução, com o openMP foi possível obter um eficiente em relação ao código em serial sem flags máxima de 88,30% para o laboratório 107 e 85,79% para o computador pessoal, demonstrando novamente a eficiência da paralelização. Vale ressaltar que quanto maior a quantidade de flags não significa maior eficiência como mostra os gráficos [17](#) e [19](#).

Os códigos também foram executados no Supercomputador Santos Dumont em uma máquina de 24 Threads, porém com um tempo reduzido de 16000 para 50 segundos, pois há um tempo limite de execução de 20 minutos por job para a fila `cpu_dev`. Para o código em serial o LNCC obteve uma eficiência de 6% em relação ao laboratório e 14.68% em relação ao notebook, já para o código utilizando openMP a eficiência do Santos Dumont foi de 13.43% em comparação laboratório e 12,93% em comparação ao notebook. Os valores aparentam ser pequenos porém para maiores escalas essa diferença se torna considerável.

Por fim, é visível que as técnicas aprendidas na disciplina são de suma importância para o meio científico, podendo resolver problemas com maior eficiência sem perder a confiabilidade nos dados, tornando possível soluções de problemas de forma mais eficaz e abrindo novas possibilidades para o conhecimento.

Referências

GILAT, A.; SUBRAMANIAM, V. *Numerical methods for engineers and scientists: An introduction with applications using matlab.* 01 2011.