

Pedro Henrique dos Santos Cunha

## **Calculo do potencial eletrostático serial e paralelo utilizando OpenMP**

Universidade Federal Fluminense - UFF

Campus Volta Redonda

Instituto de Ciências Exatas

Brasil

2 de fevereiro de 2022

# Resumo

Alguns problemas de natureza física e matemática precisam ser resolvidos computacionalmente. No entanto, dependendo do tipo de problema, a resolução pode demorar muito ou gastar uma quantidade excessiva de memória. Com o intuito de melhorar o desempenho das resoluções de problemas que utilizam muitos dados, foi desenvolvida uma solução de programação específica para computadores com arquitetura paralela.

Em resumo, foi feita uma comparação dos resultados das duas abordagens para cada sistema físico. O sistema escolhido para ser resolvido foi o problema do cálculo do potencial eletrostático para um sistema físico composto por dois capacitores concêntricos paralelos com um corte angular em sua lateral.

O intuito de escolhermos esse tema se deve ao fato do problema ter uma gama de possíveis implementações computacional, possibilitando diversos tipos de abordagem a respeito do mesmo, além de se tratar de um problema físico de interesse do aluno.

**Palavras-chaves:** Multithreading. Paralelização. Potencial Coulombiano.

# Abstract

Some problems of physical and mathematical nature are required to be solved computationally. However, depending on the type of problem, the resolution can take a long time or spend an unreasonable amount of memory. With the intent of improving the performance of problem resolutions that use a lot of data, a specific programming solution for computers with parallel architecture was developed.

In order to present the advantages of the application of parallel architecture, using OpenMP, in relation to serial computing we choose a problem of electrostatic potential for one physic system composed by two parallel concentric capacitors with a cut by an angle.

The purpose of choosing this theme is due to the fact that the problem has a range of possible computational implementations, allowing different types of approach to it, in addition to being a physical problem of interest to the student.

**Key-words:** Multithreading. Parallelization. Coulombian Potential.

# Sumário

<b>Sumário</b>	4
<b>Lista de ilustrações</b>	6
<b>1 INTRODUÇÃO</b>	7
<b>2 OBJETIVOS</b>	9
<b>3 METODOLOGIA</b>	11
<b>4 IMPLEMENTAÇÃO</b>	13
<b>5 PROFILE</b>	16
<b>5.1 Código em serial</b>	16
5.1.1 Norma infinita da matriz: discussão	17
5.1.2 Utilizando a simetria do problema e boas práticas	17
<b>6 BENCHMARK - OTIMIZAÇÃO A NÍVEL DE COMPILAÇÃO</b>	19
<b>6.1 Flags de otimização</b>	19
<b>6.2 Comparação: Programa Base e Otimizado</b>	21
<b>7 OPENMP</b>	22
<b>7.1 Funcionamento da API</b>	22
<b>7.2 Diretivas</b>	23
<b>7.3 Implementação do programa utilizando OpenMP</b>	24
7.3.1 Primeira tentativa de paralelização	24
7.3.2 Paralelização do método das diferenças finitas	26
<b>7.4 Benchmark - OpenMP</b>	26
7.4.1 Laboratório 107C	26
7.4.2 LNCC - SequanaX	28
<b>8 VALIDAÇÃO DOS RESULTADOS</b>	31
<b>9 CONCLUSÕES</b>	33
<b>REFERÊNCIAS</b>	34
<b>10 APÊNDICES</b>	35
<b>10.1 A - Código serial</b>	35

10.2	B - Código com simetria e boas práticas . . . . .	39
10.3	C - Código paralelizado . . . . .	42

# Listas de ilustrações

Figura 1 – Ilustração do método das diferenças finitas. Clique aqui para acessar a fonte da figura. . . . .	13
Figura 2 – Contorno utilizado. Sobre a circunferência em vermelho, utilizamos um potencial $\phi = 100$ e para a circunferência em azul, utilizamos um potencial $\phi = -50$ . O ângulo do corte foi de $\theta = 0.4$ radianos. . . . .	13
Figura 3 – Fluxograma do programa. . . . .	15
Figura 4 – Tempo de execução do programa serial. . . . .	16
Figura 5 – Profile do programa serial. . . . .	16
Figura 6 – Profile do programa utilizando simetria e boas práticas . . . . .	18
Figura 7 – Comparação do tempo de execução para cada caso de flags com o programa serial. . . . .	20
Figura 8 – Comparação da eficiência para cada caso de flags com o programa serial. . . . .	20
Figura 9 – Diagrama de representação para o modelo Fork-Join. Imagem retirada de [1] . . . . .	23
Figura 10 – Comparação do tempo de execução para cada thread no laboratório 107C. . . . .	26
Figura 11 – Eficiência atingida para cada thread no laboratório 107C. . . . .	27
Figura 12 – Comparação do tempo de execução para cada thread no nó SequanaX. . . . .	29
Figura 13 – Eficiência atingida para cada thread no nó SequanaX. . . . .	29
Figura 14 – Valores do potencial na malha. Calculo realizado em paralelo. . . . .	31
Figura 15 – Ilustração do contorno utilizado. . . . .	32
Figura 16 – Projeção no plano xy dos valores do potencial na malha. Calculo realizado em paralelo. . . . .	32

# 1 Introdução

A presença de uma carga elétrica em determinada região modifica as propriedades do espaço a sua volta gerando um campo elétrico. Se considerarmos uma carga pontual  $q$  colocada na origem do sistema de coordenadas e o vetor  $\vec{r}$  que liga essa carga ao ponto  $p$  no espaço, temos o campo gerado nessa região dado por

$$\vec{E}(\vec{r}) = \frac{q\vec{r}}{4\pi\epsilon_0|r^3|} \quad (1.1)$$

No lugar de uma carga pontual podemos considerar uma região onde temos uma distribuição contínua de cargas:

$$q = \int \rho(\vec{r})dV \quad (1.2)$$

Sabemos que a lei de Gauss na forma diferencial é dada por:

$$\nabla \vec{E} = \frac{\rho(\vec{r})}{\epsilon_0} \quad (1.3)$$

Uma vez que o campo elétrico é um campo conservativo, ele pode ser escrito em função de seu potencial elétrico  $\vec{E} = -\nabla\phi$ . Dessa forma, temos:

$$\nabla^2\phi = \frac{\rho(\vec{r})}{\epsilon_0} \quad (1.4)$$

A equação (1.4) é chamada equação de Poisson e estabelece uma relação entre a distribuição de carga e o potencial gerado por ela. Se na região de interesse a distribuição de cargas for nula, o potencial é gerado por cargas que estão fora da região de interesse e a equação de Poisson se reduz a equação de Laplace:

$$\nabla^2\phi = 0 \quad (1.5)$$

Se considerarmos coordenadas cartesianas em duas dimensões, então a equação de Laplace é dada por:

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = 0 \quad (1.6)$$

Neste trabalho, resolveremos a equação de Laplace computacionalmente e utilizaremos técnicas de otimização de software, além do uso da API OpenMP para reduzir o tempo do programa. Ademais, faremos benchmarks comparando o uso de multithreading

no nosso programa para melhor entender como a paralelização afeta o desempenho do mesmo.

## 2 Objetivos

Queremos resolver a equação de Laplace, pois nosso problema irá se tratar do cálculo do potencial eletrostático em uma certa região. Se considerarmos coordenadas cartesianas em duas dimensões, então a equação de Laplace é escrita como:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \quad (2.1)$$

Uma vez que pretendemos utilizar o método das diferenças finitas, precisaremos definir uma região no espaço para o cálculo desse potencial. Se definirmos uma malha marcada por ponto  $(i, j)$  para encontrar o valor de um determinado ponto  $(i, j)$  usaremos os quatro pontos a sua volta, pois iremos utilizar os sucessores e os antecessores em cada uma das direções. Considerando então a equação de Laplace para o potencial elétrico encontraremos o potencial em um determinado ponto usando o valor do potencial nos contornos.

Expandindo o potencial em série de Taylor para as quatro direções no entorno do ponto desejado  $\phi(x + \Delta x, y)$ ,  $\phi(x - \Delta x, y)$ ,  $\phi(x, y + \Delta y)$  e  $\phi(x, y - \Delta y)$ . Podemos escrever as derivadas parciais de segunda ordem como:

$$\frac{\partial^2 \phi(x, y)}{\partial x^2} = \frac{\phi(x + \Delta x, y) + \phi(x - \Delta x, y) - 2\phi(x, y)}{(\Delta x)^2} \quad (2.2)$$

$$\frac{\partial^2 \phi(x, y)}{\partial y^2} = \frac{\phi(x, y + \Delta y) + \phi(x, y - \Delta y) - 2\phi(x, y)}{(\Delta y)^2} \quad (2.3)$$

Substituindo as equações (2.2) e (2.3) na equação diferencial parcial (2.1) obtemos:

$$\frac{\phi(x + \Delta x, y) + \phi(x - \Delta x, y) - 2\phi(x, y)}{(\Delta x)^2} + \frac{\phi(x, y + \Delta y) + \phi(x, y - \Delta y) - 2\phi(x, y)}{(\Delta y)^2} = 0 \quad (2.4)$$

Assumindo que  $\Delta x = \Delta y = h$  então a equação acima pode ser escrita de uma forma mais simples:

$$\phi(x + h, y) + \phi(x - h, y) + \phi(x, y + h) + \phi(x, y - h) - 4\phi(x, y) = 0 \quad (2.5)$$

Então o valor do potencial  $\phi(x, y)$  no ponto desejado será dado por:

$$\phi(x, y) = \frac{1}{4}[\phi(x + h, y) + \phi(x - h, y) + \phi(x, y + h) + \phi(x, y - h)] \quad (2.6)$$

Esse tipo de solução não representa uma solução direta, mas sim discreta, devendo ser realizado de forma recorrente a fim de se tornar uma solução compatível com o problema físico abordado. Em termos de deslocamentos discretos  $x = x_0 + ih$ ,  $y = y_0 + jh$  onde  $i, j = 0, 1, \dots, N_{max}$  o potencial  $\phi_{i,j}$  é

$$\phi_{i,j} = \frac{1}{4}[\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}] \quad (2.7)$$

Nosso objetivo portanto é resolver computacional a equação (2.7). Após encontrada a resolução, faremos análises quanto a performance (tempo de execução) do programa e validaremos os resultados utilizando visualizações construídas a partir da linguagem Python.

### 3 Metodologia

Inicialmente, implementaremos o programa utilizando a linguagem de programação C e utilizaremos os compiladores GNU Compiler (GNU) e iFort (Intel) para compilarmos o programa.

Nossa analise irá se basear em 3 etapas principais:

1. Análise dos gargalos através do profilling do programa.
2. Análise das otimizações feitas a nível de compilação, modificando as flags de compilação do programa e também reescrevendo o código utilizando boas práticas.
3. Análise da implementação das diretivas OpenMP e sua API no nosso programa, possibilitando que usemos multithreading para diminuir o tempo gasto na execução do programa.

Para fazermos a análise dos gargalos, utilizaremos o gprof, extensão do compilador GNU Compiler para estes fins. Ademais, na análise das otimizações e da implementação do OpenMP, utilizaremos de gráficos para comparamos os tempos de execução e eficiência de otimização caso a caso.

Na parte da otimização utilizando OpenMP, utilizaremos duas maquinas para as análises, que se trata de uma máquina disponibilizada no laboratório 107C do ICEx do polo universitário de Volta Redonda da UFF, além do nó SequanaX - cluster SDumont - do Laboratório Nacional de Computação Científica.

As informações das máquinas estão listadas abaixo:

#### Laboratório 107C

- Intel(R) Core(TM) CPU i7-2600, 3.40GHz. Máx: 3.80 GHz, Min: 1.60 GHz
- 4 núcleos, cada núcleo possuindo 2 Threads.
- 12GB RAM

## SequanaX (SDumont)

- 2x Intel Xeon Cascade Lake Gold 6252
- 24 núcleos, onde cada núcleo possui 2 Threads, totalizando 48 Threads.
- 384Gb de memória RAM

Como validação do problema físico resolvido, faremos uma implementação em Python para a visualização da malha após a execução do programa, uma vez que abordamos um problema físico e sabemos o comportamento do sistema em questão.

## 4 Implementação

A malha que será utilizada será uma malha  $N \times N$  onde  $N = 1000$ . Nossa programa se baseia no método das diferenças finitas, um método bem conhecido que foi trabalhado durante a disciplina de Métodos Numéricos II e utiliza pontos adjacentes ao ponto  $(i, j)$  da malha, com espaçamentos definidos pelo usuário para percorrer a malha.

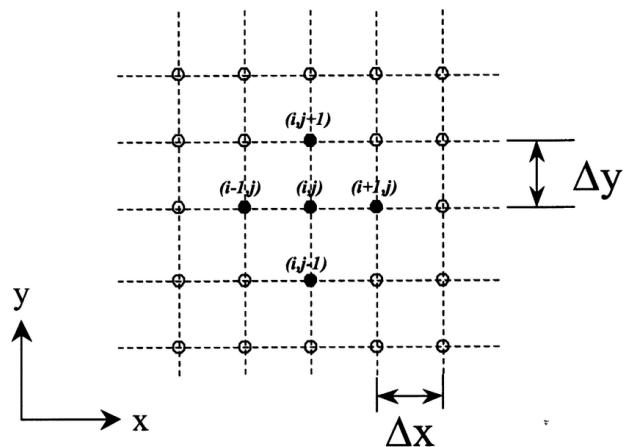


Figura 1 – Ilustração do método das diferenças finitas. [Clique aqui para acessar a fonte da figura.](#)

A condição de contorno utilizada consiste em dois círculos circunscritos com uma abertura de um certo ângulo  $\theta$ , conforme indicado na figura (2).

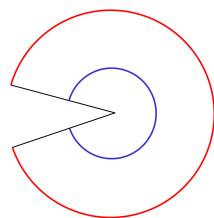


Figura 2 – Contorno utilizado. Sobre a circunferência em vermelho, utilizamos um potencial  $\phi = 100$  e para a circunferência em azul, utilizamos um potencial  $\phi = -50$ . O ângulo do corte foi de  $\theta = 0.4$  radianos.

Após a inicialização da malha e das variáveis, o fluxograma prevê duas etapas principais da nossa implementação, que correspondem à maior área do fluxograma:

- A imposição das condições de contorno
- O cálculo do potencial

Os cálculos das condições de contorno serão feitos em uma função separada do cálculo do potencial.

O cálculo do potencial prevê um looping de interações máximo, operando com uma condição de parada através da norma infinita. Sendo assim, precisaremos também de uma função que calcule a norma infinita de uma matriz, sendo chamada dentro da função de cálculo do potencial.

Após o cálculo do potencial ser realizado, devemos coletar os dados da malha no qual o potencial foi inserido e liberar o fluxo de memória utilizado no programa. O fluxograma está exposto na figura 3.

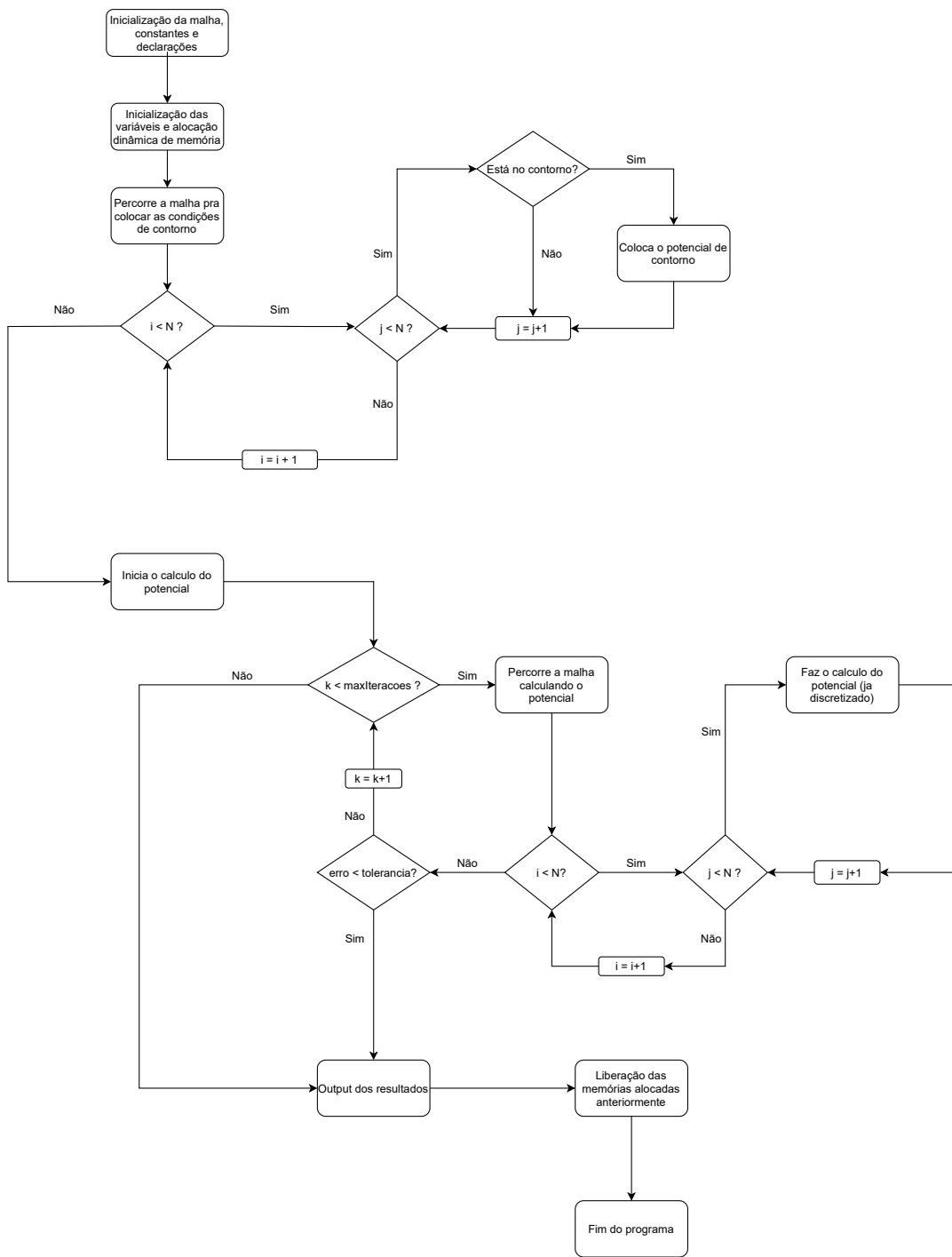


Figura 3 – Fluxograma do programa.

# 5 Profile

## 5.1 Código em serial

Seguindo o fluxograma, implementamos nosso programa em C utilizando uma função para o cálculo do potencial e, também, utilizamos o método da norma infinita para estipularmos uma tolerância de interrupção do loop, como mostrado no apêndice 10.1. No caso do programa serial, o tempo de execução, foi um pouco mais de 5 horas e 28 minutos.

```
real    328m57,925s
user    328m51,453s
sys     0m4,237s
```

Figura 4 – Tempo de execução do programa serial.

E o profile pode ser visualizado utilizando a extensão `gprof2dot` do gprof, onde podemos ver que o maior tempo de execução do programa se refere ao cálculo do potencial e no cálculo da norma infinita do potencial para fazer a comparação e verificar se está acima ou abaixo do erro tolerado.

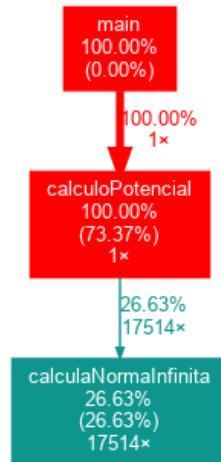


Figura 5 – Profile do programa serial.

Onde o número em porcentagens simboliza, em porcentagens, o tempo gasto em cada parte do código. Repare que o maior tempo gasto para a execução do programa foi utilizado no cálculo do potencial, sendo este essencialmente simples. O tempo gasto se justifica, portanto, a partir do looping de interações máximas utilizado para a repetição do cálculo, previsto no fluxograma. Repare ainda que 26% do tempo utilizado para a execução neste caso se deu no cálculo da norma infinita das matrizes, sendo assim, um

ponto de alto custo computacional e sujeito a gargalos. Portanto, façamos uma discussão sobre o cálculo da norma infinita:

### 5.1.1 Norma infinita da matriz: discussão

A norma infinita de uma matriz é definida como:

**Definição 1.** Seja  $\mathbf{A} = [a_{ij}]_{r \times s}$  uma matriz  $r \times s$ . A norma infinita da matriz, denotada por  $\|\mathbf{A}\|_\infty$  é o número não negativo dado por:

$$\|\mathbf{A}\|_\infty = \max \sum_{i=1}^r |a_{ij}|$$

Isto é, é a maior soma absoluta das linhas.

Tendo isso em mente, observe que a norma infinita no nosso programa é utilizada para impor uma condição de parada no looping que controla quantas iterações faremos no calculo do nosso potencial.

O nosso código é um código com complexidade polinomial  $O(n^3)$ , mas com a condição de parada da norma infinita, podemos interromper o looping antes do mesmo chegar ao fim, resultando em uma complexidade diferente da complexidade  $O(n^3)$ . Isso em programas serial é bom, mas em programas paralelos, você consegue paralelizar um código de complexidade  $O(n^3)$  de forma muito mais eficaz que um código que não tenha complexidade polinomial. Paralelizações de códigos totalmente polinomiais (no sentido de  $O(n^k)$ ) é preferível em termos de repetições longas.

A razão por detrás disto é que, se tratando de termos polinomiais, quanto mais divisões tivermos, mais rápido a execução do código se torna. Por exemplo, em códigos de complexidade  $O(n \log(n))$ , existirá uma área de extrema otimização e outra que causará gargalos e problemas de otimização, necessitando técnicas mais robustas para que a paralelização seja feita com eficiência.[\[2\]](#)

Nesse sentido, como um dos objetivos deste trabalho é a paralelização do código utilizando OpenMP, retiramos, portanto, o calculo da norma infinita como condição de parada do looping.

### 5.1.2 Utilizando a simetria do problema e boas práticas

O nosso problema possui simetria radial. Podemos tirar vantagem do método das diferenças finitas e utilizar a simetria do potencial na malha para colocar as condições de contorno e fazer o cálculo do potencial. Dessa forma, ao invés de percorrermos a malha inteira, isto é, percorrermos os  $N \times N$  pontos da malha, iremos percorrer  $N/2 \times N/2$  e

utilizaremos a simetria para percorrermos o restante da malha. Faremos isto tanto para impor as condições de contorno, quanto para o cálculo do potencial.

Tambem utilizamos as boas práticas de programação para as variáveis (camelCase) e para as operações feitas no código, com o intuito de tornar o código mais legível e mais limpo. Listamos o código no apêndice [10.2](#).

Os resultados encontrados foram satisfatórios, executando o programa com a mesma quantidade de iterações máximas no calculo do potencial, reduzimos o tempo de execução para um pouco mais que 2 horas e 13 minutos, como mostra a figura abaixo.

index	% time	self	children	called	name
[1]	100.0	0.00	8035.59		<spontaneous>
		8035.58	0.00	1/1	main [1]
		0.01	0.00	1/1	finiteDifference [2]
		0.00	0.00	1/1	getResults [3]
		0.00	0.00	1/1	setContour [5]
		0.00	0.00	1/1	setAngleCut [4]
-----					
[2]	100.0	8035.58	0.00	1/1	main [1]
		8035.58	0.00	1	finiteDifference [2]
-----					
[3]	0.0	0.01	0.00	1/1	main [1]
		0.01	0.00	1	getResults [3]
-----					
[4]	0.0	0.00	0.00	1/1	main [1]
		0.00	0.00	1	setAngleCut [4]
-----					
[5]	0.0	0.00	0.00	1/1	main [1]
		0.00	0.00	1	setContour [5]
-----					

Figura 6 – Profile do programa utilizando simetria e boas práticas

**Observação.** Este profile foi refeito para esse trabalho final. No profile feito anteriormente na disciplina, havíamos efetuado a divisão do programa em mais funções para o calculo do potencial. Isso se justificava uma vez que precisavamos também chamar a função para o calculo da norma infinita, isto é, uma vez que precisavamos chamar a função do cálculo do potencial e da norma infinita, fazia sentido segmentar o código em funções diferentes para torná-lo mais organizado. Mas agora que retiramos a função da norma infinita, podemos realizar o cálculo do potencial na propria função do método de diferenças finitas sem problemas de organização.

# 6 Benchmark - Otimização a nível de compilação

## 6.1 Flags de otimização

Uma das primeiras técnicas de otimização que podem ser utilizadas, são as técnicas baseadas na mudança das flags de compilação do programa.

Um compilador é um programa que lê um conjunto de instruções de um código fonte e traduz para a linguagem de máquina. Para que isso ocorra, o código passa por diversas fases, como pelo analisador semântico e pelo analisador sintático. Podemos citar como exemplo os compiladores GNU Compiler e iFort da Intel.

Na compilação de um código, existem vários níveis nos quais o compilador pode intervir para otimizar o código  $-O0$ ,  $-O1$ ,  $-O2$ ,  $-O3$ . Cada um desse níveis habilita uma série de flags de otimização. As flags habilitadas por cada nível dependem do compilador e do próprio computador.

Usaremos o compilador GNU Compiler e iFort da Intel, onde analisaremos o programa sendo executado habilitando as seguintes flags nos 5 casos:

1. Flags of non-optimized case: `-OX`
2. Flags I: `-OX -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native`
3. Flags II: `-OX -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=corei7-avx (Lab 107) -march=corei7-avx (Lab 107)`
4. Flags III: `-OX -w -mp1 -zero -xHOST`
5. Flags IV: `-OX -w -mp1 -zero -xHOST -fast=2`

Onde o termo `-OX` se refere à variação  $-O0$ ,  $-O1$ ,  $-O2$ ,  $-O3$ .

Utilizamos para esse benchmark, a maquina disponibilizada no laboratório 107C, cujas especificações foram listadas no capítulo 3.

Os resultados encontrados tanto para o tempo quanto para a eficiência foram, respectivamente para cada caso:

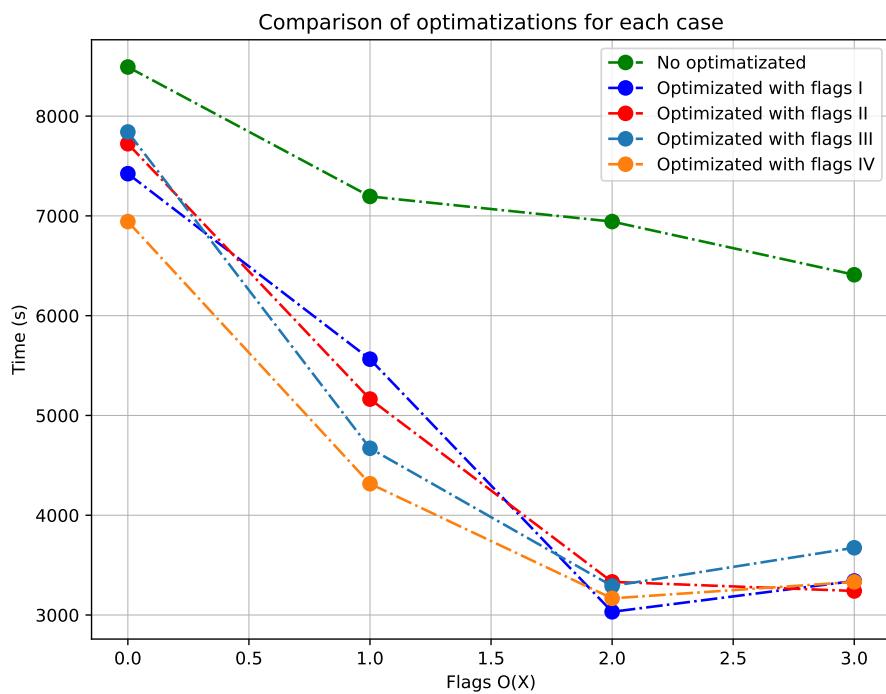


Figura 7 – Comparaçāo do tempo de execuāo para cada caso de flags com o programa serial.

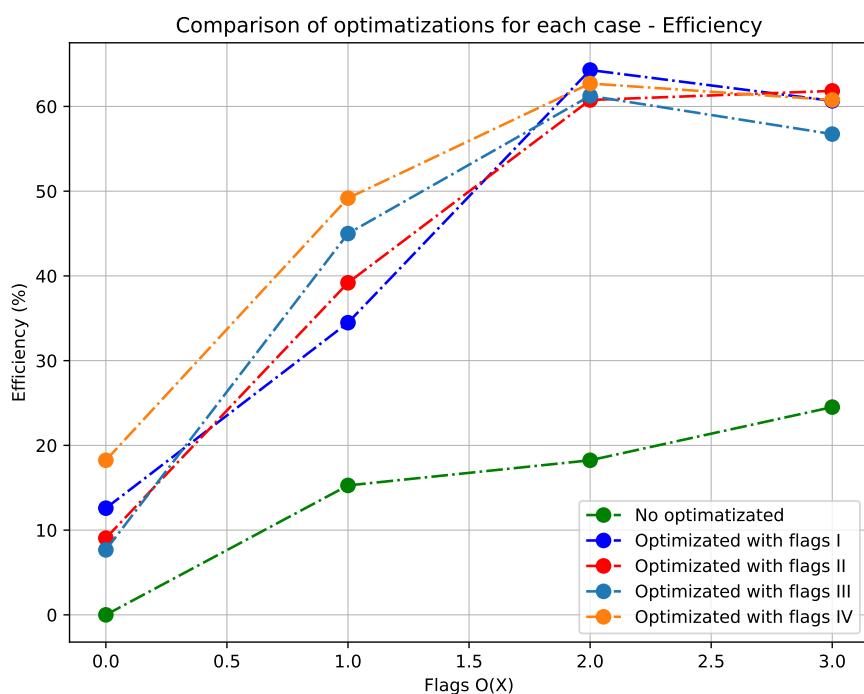


Figura 8 – Comparaçāo da eficiêncāa para cada caso de flags com o programa serial.

A eficiência foi calculada seguindo a fórmula:

$$\text{Eficiência} = \left(1 - \frac{T[n]}{T_0}\right) \times 100$$

Onde o termo  $T_0$  representa o pior caso da análise do benchmark, sendo utilizado o -O0.

## 6.2 Comparação: Programa Base e Otimizado

Repare que o caso onde utilizamos somente a flag -O0 (8492.13 segundos) se mostrou ainda pior do que o caso sem a utilização das flags. O melhor caso de otimização ocorreu com o uso das flags:

- -O2 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native

O programa base utilizado no início do trabalho teve como tempo de execução 5 horas, 28 minutos e 57 segundos. Utilizando as otimizações descritas anteriormente e a flag citada acima, reduzimos o tempo de execução do programa para 3032.94 segundos, totalizando aproximadamente 50 minutos e 33 segundos de tempo de execução. Nesse sentido, a eficiência atingida foi de 84.63% com relação ao programa base, resultados bastante satisfatórios que justificam as análises feitas até o momento.

# 7 OpenMP

Quando tentamos resolver algum problema de natureza física ou matemática, notamos que alguns deles requerem um nível de esforço computacional muito grande. Códigos de simulações e modelagens de sistemas físicos por exemplo, que lidam com a manipulação de muitos dados em arrays podem demorar muito tempo para serem executados. Com o intuito de melhorar a performance de resolução desse tipo de problema, começou-se a pensar em uma maneira de agregar poder computacional através do uso de vários computadores e CPUs trabalhando em conjunto. Cada CPU executa uma tarefa, ou uma parte da tarefa, simultaneamente com as demais. Esse processo é o que chamamos de paralelização.

Podemos identificar dois tipos principais de arquiteturas de computadores paralelos. Sistemas de memória compartilhada representam um tipo de arquitetura na qual os processadores compartilham uma mesma memória, e sistemas de memória distribuída nos quais cada processador tem sua memória privada.

Para a implementação deste paradigma na programação, fez-se necessário o desenvolvimento de uma interface de programação (API) capaz de executar códigos paralelizados, ou seja, códigos nos quais as operações são divididas entre os diversos processadores. A interface padrão utilizada para programação em computadores de memória compartilhada é o Open Multi-Processing, conhecido popularmente como OpenMP<sup>[3]</sup>.

O OpenMP é uma API padrão utilizada para desenvolvimento de códigos paralelos executados em arquiteturas de memória compartilhada. Sua estrutura é baseada em diretivas de compilação que podem ser utilizadas em códigos de linguagem C, C++ e Fortran.

## 7.1 Funcionamento da API

Um programa em OpenMP é executado por diversas threads que consistem em subsistemas que irão executar tarefas de forma conjunta. Além disso o código pode ser formado por regiões que serão executados em serial por uma única thread e regiões executadas em paralelo por um conjunto de threads.

As threads trabalham num modelo chamado Fork-Join. O programa é executado em serial por uma única thread, chamada de master-thread até encontrar a primeira diretiva OpenMP que representa uma região paralela. Nessa momento acontece o processo chamado de Fork, no qual a master-thread divide o segmento em varias threads para executar o código na região paralela. Ao executar o bloco de tarefas e chegar ao final da

diretiva, ocorre o processo de Join, neste processo as threads são finalizadas e sincronizadas e voltam a se unir permanecendo apenas a master-thread.

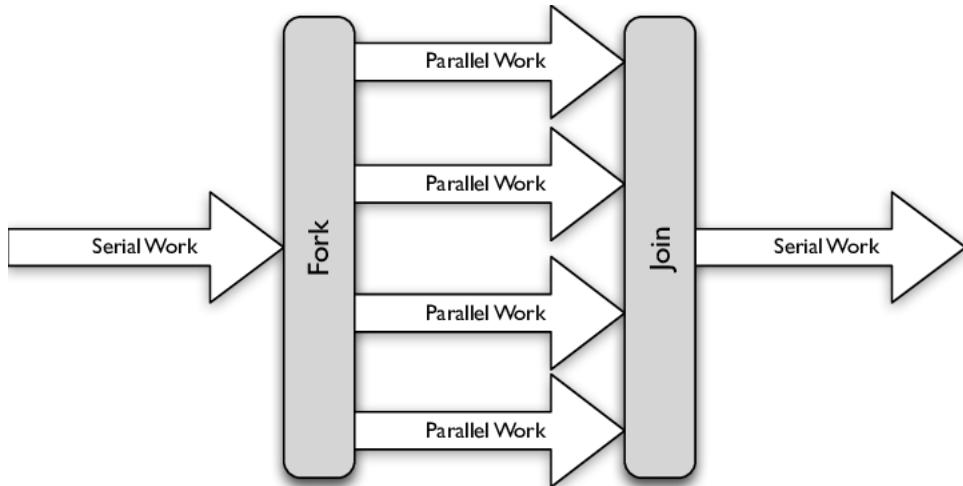


Figura 9 – Diagrama de representação para o modelo Fork-Join. Imagem retirada de [1]

## 7.2 Diretivas

Durante o curso, fomos apresentados à diversas diretivas que podem ser utilizadas para a paralelização dos códigos utilizando OpenMP. Na linguagem C, uma diretiva openMP é indicada por "#pragma omp" em sua inicialização.

Nesse trabalho utilizamos apenas uma delas que é a diretiva "#pragma omp parallel for", uma vez que trabalhamos com muitos loops no nosso código.

Se tratando de regiões paralelas, as variáveis também precisam ser declaradas para seu pleno funcionamento nessas regiões, nesse sentido, utilizamos normalmente a *shared* para as variáveis utilizadas por todas as threads como meios operativos e *private* que são aquelas que cada thread acessa. Esse tipo de diretivas são chamadas cláusulas.

Uma das cláusulas também utilizadas é a cláusula *schedule*. Essas cláusulas são especiais pois nela utilizamos uma quantidade denominada *chunk*.

As chunks são regiões de memória de tamanho pré-definido que são distribuídas para as threads. A distribuição pode ser feita de forma dinâmica, estáticas ou guiadas. Neste trabalho, fizemos a distribuição de forma dinâmica e estática.

A *schedule* estática distribui de forma sequencial e estática - por isso o nome - a memória para cada thread utilizada, caso essa divisão seja feita de maneira irregular (ocasião que pode ocorrer tanto pelo número de threads utilizado quanto pelo número do chunk utilizado), encontraremos quedas de performances na execução do código.

A *schedule* dinâmica distribui de forma mais flexível essa região de memória para

evitar irregularidades nessa divisão. No entanto, caso ocorra da divisão também ser feita de forma irregular, quedas de performance maiores acontecerão.

## 7.3 Implementação do programa utilizando OpenMP

Como mencionado anteriormente, utilizamos as diretivas "**#pragma omp parallel for**". A schedule escolhida foi a schedule dinâmica para tornar a distribuição mais flexível para as threads.

Além disso, utilizamos o programa compilado com a melhor flag da análise no caso do programa serial, que se trata da flag:

- -O2 -fexpensive-optimizations -m64 -foptimize-register-move -funroll-loops -ffast-math -mtune=native -march=native

Havíamos discutido anteriormente no profile o método da norma infinita e nesta etapa do trabalho podemos validar a decisão de retirá-la do código. O argumento por detrás dessa decisão se baseia no fato de que o método das diferenças finitas considera um bloco de memória (no caso as linhas) para fazer a comparação. Uma tentativa de paralelização nesta etapa possui muitas chances de falha uma vez que o cálculo da norma infinita era realizado dentro do looping principal de iteração do método de diferenças finitas. Isto é, uma vez que as tarefas eram divididas para as threads, uma das threads poderia atingir a tolerância desejada e automaticamente retornaria para a função main, interrompendo o looping e não sincronizando com as outras threads.

### 7.3.1 Primeira tentativa de paralelização

Inicialmente, havíamos buscados tentativas de paralelizar os loopings da malha, isto é, os loopings que percorrem a malha e atribuem o potencial a cada ponto da malha. Pelo profile, sabíamos que o tempo do programa era exclusivamente gasto no método das diferenças finitas e portanto ali seria a região que deveria ser paralelizada, como mostramos no código abaixo:

```
1 void finiteDifference(double **v, double **v_old, double dx, double dy,
2                         int chunk, int numberThreads){
3     int a, b, i, j, k;
4     double x, y, r;
5
6     omp_set_num_threads(numberThreads);
7
8     #pragma omp parallel private(a, b, x, y, r, i, j, k) shared(v, v_old)
9 }
```

```

9      {
10         for(k = 0; k < maxIt; k++){
11
12             for(a = 0; a < N; a++)
13                 for(b = 0; b < N; b++)
14                     v_old[a][b] = v[a][b];
15
16             #pragma omp for schedule(dynamic, chunk) collapse(2)
17             for(i = 1; i < N-1; i++){
18                 for(j = N/2; j < N-1; j++){
19
20                     x = xInicial + i*dx;
21                     y = yInicial + j*dy;
22                     r = sqrt(x*x + y*y);
23                     if(fabs(raioExterno - r) < tolerance && (v[i][j] ==
potencialExterno )){
24                         v[i][j] = potencialExterno;
25                     }
26                     else{
27                         if(fabs(raioInterno - r) < tolerance && (v[i][j]
== potencialInterno)){
28                             v[i][j] = potencialInterno;
29                         }
30                         else{
31                             v[i][j] = (1.0/4.0)*(v_old[i+1][j] + v_old[i
-1][j] + v_old[i][j+1] + v_old[i][j-1]);
32                             v[i][N-j-1] = v[i][j];
33                         }
34                     }
35                 }
36             }
37         }
38     }
39 }
```

Listing 7.1 – Primeira tentativa de paralelização do método das diferenças finitas.

No entanto, após terem sido feitas as devidas declarações de diretivas e feita a paralelização, o código não apresentava melhorias em relação ao tempo de execução, oscilando entre o tempo do código de execução serial.

Partindo de uma análise mais minuciosa no profile mostrado anteriormente, podemos perceber que os cálculos do potencial na malha também é feito nas funções que impõem as condições de contorno ao problema e são feitos de forma quase que instantânea. Por conta disso, partimos da prerrogativa de que os loopings da malha era percorrido rápido o suficiente para não precisar de uma paralelização, sendo assim, o nosso alvo deveria ser o looping exterior, que controla o número máximo de iterações.

### 7.3.2 Paralelização do método das diferenças finitas

Ao aplicarmos a diretiva "#pragma omp parallel for" para o looping exterior que controle o número maximo de iterações, a melhoria em termos de tempo de execução foi atingida. Podemos perceber melhorias substanciais utilizando apenas 2 threads, sendo assim, o código foi paralelizado como apresentado no apêndice 10.3.

## 7.4 Benchmark - OpenMP

### 7.4.1 Laboratório 107C

No laboratório 107C, implementamos o código paralelo com o mesmo número de iterações do código serial para que seja feita uma análise mais concreta. Os tempos de execução atingidos estão listados em forma tabular abaixo:

Tempo de execução (s)	Threads
2992.94	1
1683.12	2
1165.86	3
891.98	4
859.40	5
742.43	6
681.61	7
609.05	8

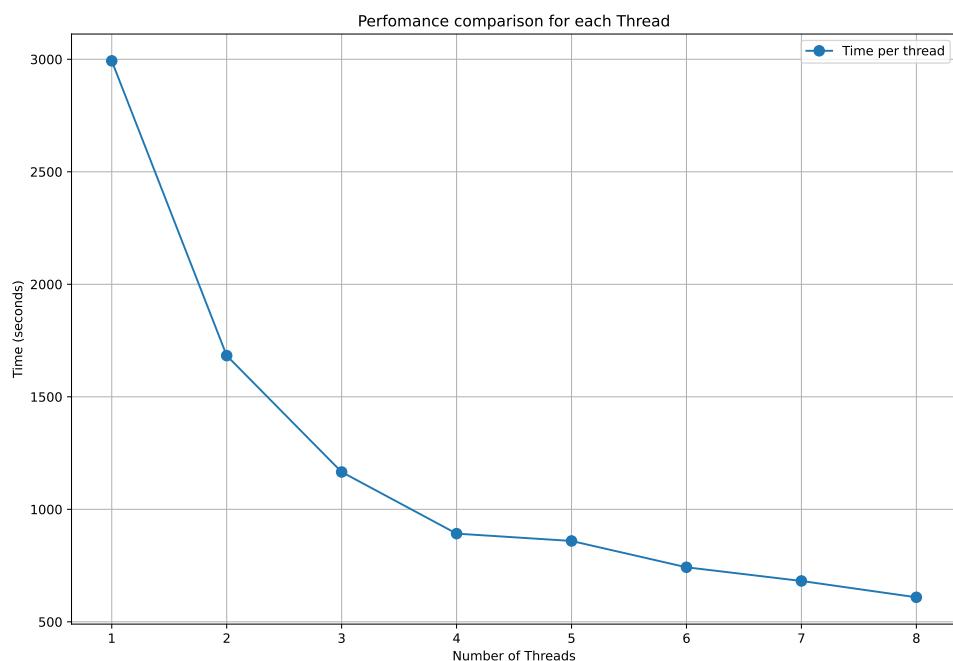


Figura 10 – Comparaçāo do tempo de execuāo para cada thread no laborat rio 107C.

Novamente, calculamos a eficiência utilizando a formula

$$\text{Eficiência} = \left(1 - \frac{T[n]}{T_0}\right) \times 100$$

Onde o termo  $T_0$  representa o pior caso da análise do benchmark, o caso com 1 thread. Sendo assim, encontramos:

Eficiência (%)	Thread
0.0	1
43.76	2
61.04	3
70.19	4
71.28	5
75.19	6
77.22	7
79.65	8

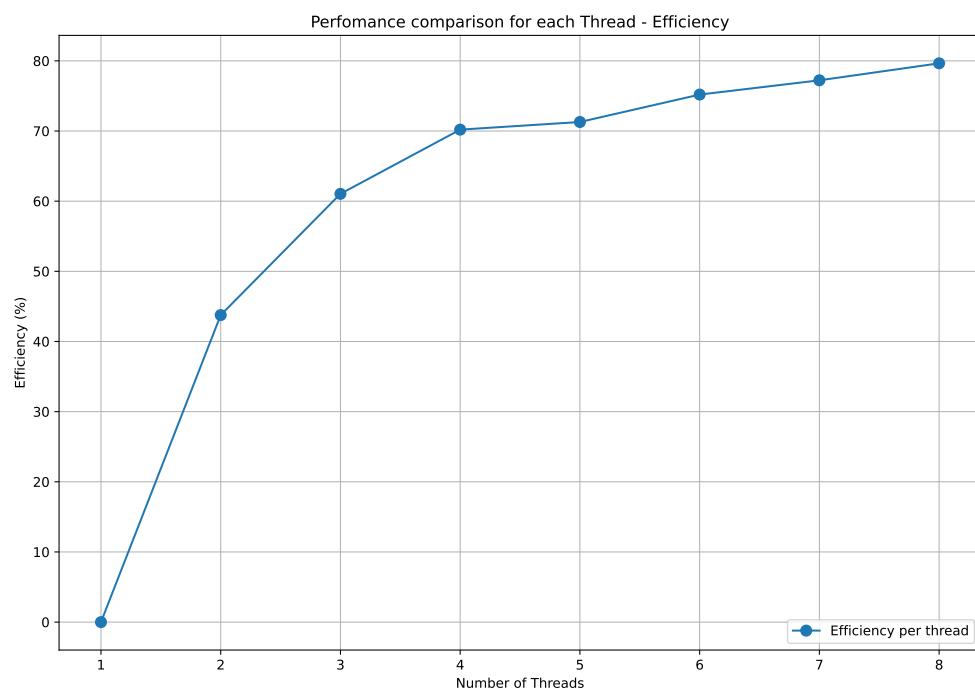


Figura 11 – Eficiência atingida para cada thread no laboratório 107C.

Onde utilizamos, em todos os dados, duas casas decimais sendo a ultima casa o algarismo duvidoso.

#### 7.4.2 LNCC - SequanaX

Para a utilização da maquina do nó Sequana, houve a necessidade de diminuir o tempo de execução do código em serial para no máximo 20 minutos, uma vez que a fila dos jobs do nó tem limite máximo de 20 minutos. Sendo assim, diminuimos o tamanho do looping que controla as iterações do potencial.

O tempo serial atingido no laboratório foi de 20 minutos e 41 segundos exatamente nessas configurações, ao transpor para o Sequana, o tempo de execução com 1 thread foi de 19 minutos e 13 segundos.

Utilizando as 48 threads, atingimos os resultados listados na tabela abaixo:

<b>Tempo de execução (s)</b>	<b>Threads</b>
1153.17	1
532.36	2
855.27	4
475.65	6
718.22	8
316.12	10
330.77	12
234.98	14
236.46	16
201.28	18
184.64	20
166.67	22
159.04	24
147.34	26
143.28	28
135.12	30
124.16	32
121.85	34
111.94	36
114.95	38
109.38	40
98.42	42
92.37	44
90.44	46
96.86	48

Tais resultados refletem de forma explicita a eficiência da paralelização, atingido um tempo minimo de 1 minuto e 30 segundos para executar o programa completo. Podemos visualizar os dados expostos na tabela em um gráfico, como mostrado a seguir:

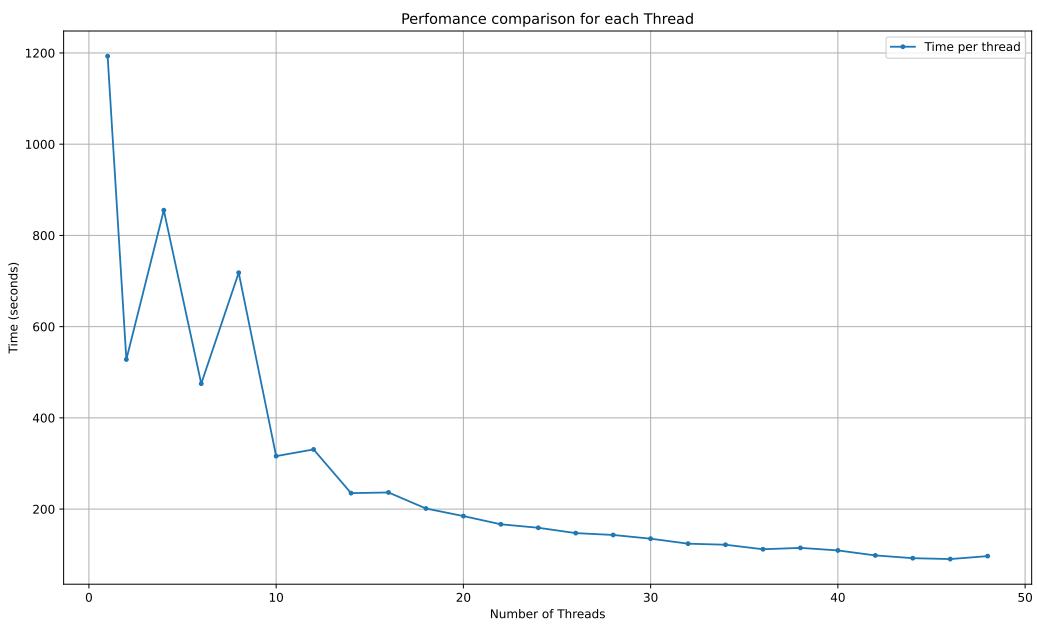


Figura 12 – Comparaçāo do tempo de execuāo para cada thread no nō SequanaX.

Repare que a curva para as 10 primeiras threads oscila bastante, principalmente entre as threads 4 e 8, seguido por uma oscilação menor nas threads 12 até 16. Ao analisar melhor o gráfico, salvo os pontos de oscilação, podemos reparar que se trata de um decaimento exponencial, atingindo eficiências mostradas no gráfico abaixo:

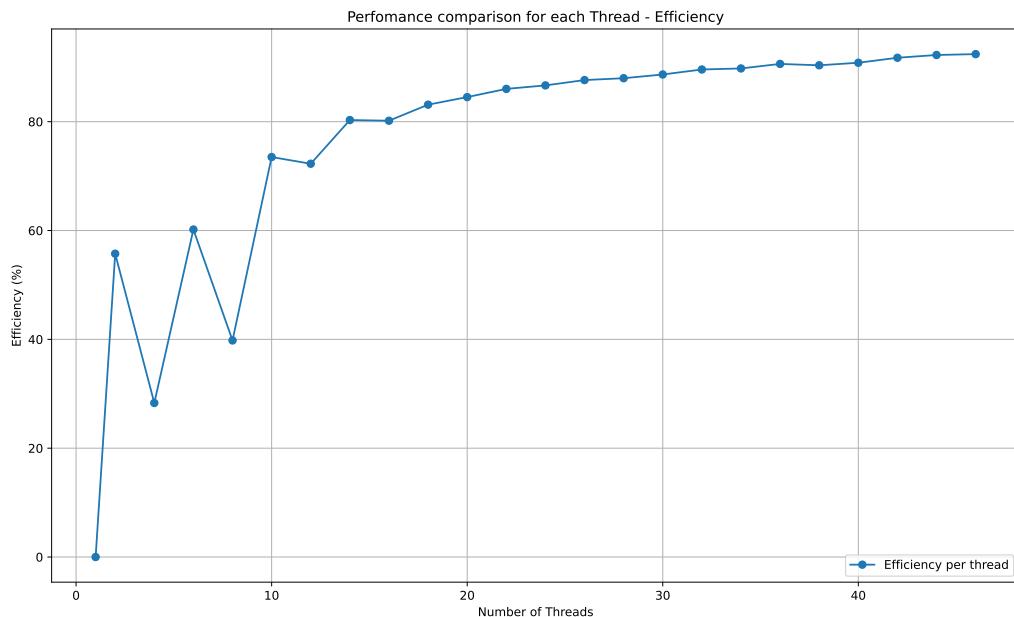


Figura 13 – Eficiēncia atingida para cada thread no nō SequanaX.

Os dados referentes à eficiência são expostos na tabela abaixo:

Eficiência (%)	Threads
0.00	1
55.74	2
28.30	4
60.18	6
39.79	8
73.50	10
72.27	12
80.30	14
80.17	16
83.12	18
84.52	20
86.02	22
86.66	24
87.64	26
87.98	28
88.67	30
89.59	32
89.78	34
90.61	36
90.36	38
90.83	40
91.74	42
92.25	44
92.41	46
91.89	48

Onde, novamente, utilizamos duas casas decimais sendo a ultima o algarismo duvidoso.

O decaimento exponencial de tempo, conforme a figura 12, nos mostra o poder da paralelização e da utilização do OpenMP no nosso problema. O comportamento entre as threads 4 e 10 revela um caráter curioso, nos mostrando que a distribuição em mais threads não necessariamente otimiza o tempo de execução do programa, podendo ainda aumentá-lo. Tal comportamento acontece novamente para as threads 46 e 48. No entanto, os resultados atingidos de forma geral foram muito satisfatórios e o comportamento da curva demonstra um caráter de decaimento, dessa maneira, podemos presumir que com a utilização de ainda mais threads o tempo de execução poderia diminuir ainda mais.

## 8 Validação dos resultados

Após efetuar a paralelização, o leitor pode se perguntar: Uma vez que o problema abordado se trata de um problema físico, o resultado encontrado (em termos do potencial) está correto?

Sabemos de estudos anteriores sobre a teoria eletromagnética que o potencial elétrico, seja ele imposto sob uma casca esférica ou outra geometria, deve tender a zero no infinito, ou seja, a medida que nos distanciamos da fonte do potencial elétrico, o potencial elétrico diminui. Em outras palavras, o potencial elétrico decai com a distância.

O contorno utilizado se trata de duas regiões circulares concêntricas com uma abertura de um determinado arco em suas laterais. Dessa forma, a intuição física nos dias que a malha mostrará um degradê de cores a medida que os cálculos do potencial forem sendo realizados, relacionado à intensidade do potencial imposta sob o contorno utilizado, isto é, a partir do contorno, a região da malha estará sendo preenchida com potencial elétrico e a projeção bidimensional da malha deve reproduzir tal comportamento.

Uma vez que o cálculo do potencial no programa serial é feito de forma sequencial em relação ao looping, ou seja, o looping é executado até o seu fim de forma sequencial para cada entrada, o potencial pode ser calculado pra malha sem muitos problemas. No entanto, se tratando do código do programa em paralelo, uma das preocupações seria a divisão do cálculo entre as threads poder gerar conflitos em relação ao valor do potencial na malha, resultando em regiões cujo o valor do potencial estaria errado. Felizmente, nosso código não apresentou esse tipo de problema, como mostrado a seguir:

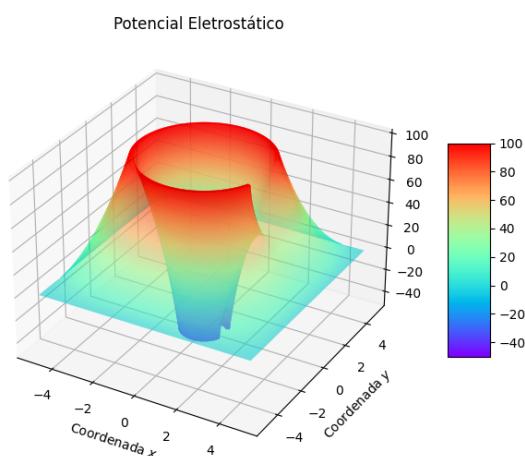


Figura 14 – Valores do potencial na malha. Cálculo realizado em paralelo.

Uma melhor forma de visualização do resultado de forma a validar o cálculo realizado em relação ao seu sentido físico, se trata da projeção no plano xy da imagem 14, revelando as características do contorno em relação ao contorno utilizado como referência.

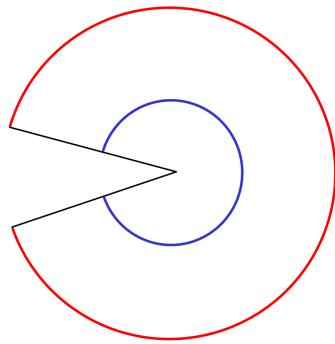


Figura 15 – Ilustração do contorno utilizado.

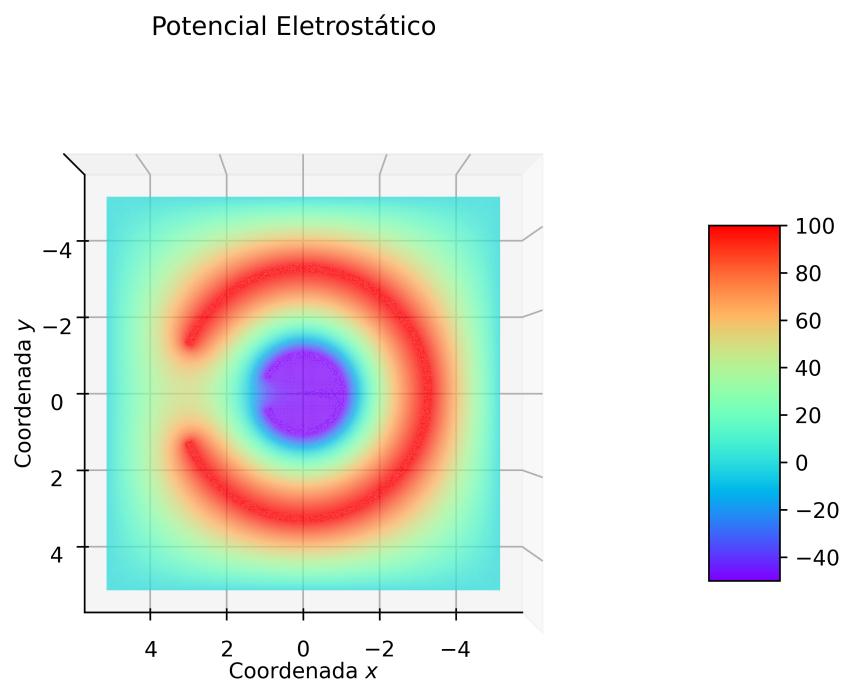


Figura 16 – Projeção no plano xy dos valores do potencial na malha. Cálculo realizado em paralelo.

## 9 Conclusões

Os processos de paralelização são uma ferramenta computacional muito utilizada para extensos cálculos e métodos que requerem um poder computacional mais distribuído e otimizado do que propriamente na natureza serial que encontramos habitualmente.

Em termos físicos, em que estamos interessados na modelagem de fenômenos da natureza, percebemos que muitos dos processos que requerem explicitações requerem uma alta gama de dados e nuances que as fórmulas fornecem em pequenos intervalos. Por conta disso, a visualização e a exploração do caráter teórico da Física é, muitas vezes, limitado pela possibilidade de se realizar esses processos e em alguns não possuindo solução analítica.

Um desses processos, por exemplo, é o cálculo do potencial elétrico, que pode se tornar demasiadamente complicado a depender de suas condições de contorno. A utilização de Computação de Alto Desempenho portanto, pode se tornar o caminho para tal, principalmente através da utilização de OpenMP que nos permite uma maior flexibilidade no uso da memória e dos processos de nossa máquina.

Tendo isso em mente, o apresentado anteriormente está de acordo tanto com a necessidade, como com o resultado esperado e que provém como resposta a uma exploração de teorias físicas em termos numéricos. Além de podermos visualizar o mundo físico, ainda fomos capazes de realizar tal de uma maneira muito mais rápida e eficiente, chegando a melhorias de até 90 % de nosso código feito em serial e sem afetar e corromper os dados, e consequentemente, o significado físico. Portanto, podemos perceber que o trabalho supriu uma necessidade e além disso, ofereceu ferramentas efetivas a sua melhora e aprimoramento, onde ao utilizarmos como pior caso o tempo mostrado na figura 4 e o melhor caso como sendo o caso do calculo utilizando 8 threads no laboratório 107C, alcançamos uma melhoria (eficiência) de 96,91% em termos de tempo de execução.

# Referências

- [1] David Alexander Beckingsale. *Towards scalable adaptive mesh refinement on future parallel architectures*. PhD thesis, University of Warwick, 2015. Citado 2 vezes nas páginas [6](#) e [23](#).
- [2] Otavio Migliavacca Madalossoa, Andrea Schwertner Charaoa, Haroldo Fraga de Campos Velhob, Renata Sampaio da Rocha Ruizb, and Joao Vicente Ferreira Limaa. Multicore and many-core architectures in a parallel  $O(n \log n)$  friends-of-friends algorithm for astronomical object classification. In *Proceedings of the 4th Conference of Computational Interdisciplinary Science. Pan-American Association of Computational Interdisciplinary Sciences*, 2016. Citado na página [17](#).
- [3] The OpenMP api specification for parallel programming. <https://www.openmp.org/>. Accessed: 2022-01-31. Citado na página [22](#).

# 10 Apêndices

## 10.1 A - Código serial

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 // Declara es e constantes
6
7 #define N 1002
8 #define tolerance 0.01
9 #define maxIt 1000000
10 #define errorTolerance 1e-16
11 #define potencialInterno -50.0
12 #define potencialExterno 100.0
13 #define raioInterno 1.0
14 #define raioExterno 3.0
15 #define theta 0.4
16
17 // Dimens es da malha
18
19 #define xInicial -5.0
20 #define xFinal 5.0
21 #define yInicial -5.0
22 #define yFinal 5.0
23
24 void setCondicoesContorno(double **v, double dx, double dy){
25     double x, y, r;
26
27     for(int i = 1; i < N-1; i++){
28         for(int j = 1; j < N-1; j++){
29             x = xInicial + i*dx;
30             y = yInicial + j*dy;
31             r = sqrt(x*x + y*y);
32
33             if(fabs(raioExterno - r) < tolerance){
34                 v[i][j] = potencialExterno;
35             }
36             else{
37
38                 if(fabs(raioInterno - r) < tolerance){
39                     v[i][j] = potencialInterno;
40                 }
41             }
42         }
43     }
44 }
```

```

41         }
42     }
43 }
44 }
45
46 void setCorteAngulo(double **v, double dx, double dy){
47     double x, y, r, ang;
48
49     for(int i = 1; i < N-1; i++){
50         for(int j = 1; j < N-1; j++){
51             x = xInicial + i*dx;
52             y = yInicial + j*dy;
53             r = sqrt(x*x + y*y);
54             ang = acos(x/r);
55
56             if(fabs(raioExterno - r) < tolerance && ang < theta){
57                 v[i][j] = 0;
58             }
59             else{
60                 if(fabs(raioInterno - r) < tolerance && ang < theta){
61                     v[i][j] = 0;
62                 }
63             }
64         }
65     }
66 }
67
68 double calculaNormaInfinita(double **M, int nL, int nC){
69
70     double norma = 0, soma = 0;
71
72     for(int i = 0; i < nL; i++){
73         for(int j = 0; j < nC; j++){
74             soma = soma + fabs(M[i][j]);
75         }
76         if(norma < soma){
77             norma = soma;
78         }
79         soma = 0;
80     }
81
82     return norma;
83 }
84
85 void getResultados(double **v, double dx, double dy){
86     double x, y;

```

```

88 FILE *arquivo;
89
90 arquivo = fopen("dadosLaplace.dat", "w");
91
92 fprintf(arquivo, "x\ty\tz\n");
93
94 for(int i = 0; i < N; i++){
95     for(int j = 0; j < N; j++){
96         x = xInicial + i*dx;
97         y = yInicial + j*dy;
98         fprintf(arquivo, "%g %g %g\n", x, y, v[i][j]);
99     }
100 }
101
102 fclose(arquivo);
103 }
104
105 void calculoPotencial(double **v, double **v_old, double dx, double dy){
106
107     double x, y, r;
108     long double normaV, normaV_old, normaDif;
109
110     for(int k = 0; k < maxIt; k++){
111
112         for(int a = 0; a < N; a++)
113             for(int b = 0; b < N; b++)
114                 v_old[a][b] = v[a][b];
115
116         for(int i = 1; i < N-1; i++){
117             for(int j = 1; j < N-1; j++){
118                 x = xInicial + i*dx;
119                 y = yInicial + j*dy;
120                 r = sqrt(x*x + y*y);
121
122                 if(abs(raioExterno - r) && (v[i][j] == potencialExterno)
123 )) {
124                     v[i][j] = potencialExterno;
125
126                 }
127                 else{
128                     if(abs(raioInterno - r) < tolerance){
129                         v[i][j] = potencialInterno;
130                     }
131                     else{
132                         v[i][j] = (1.0/4.0)*(v_old[i+1][j] + v_old[i-1][
133                             j] + v_old[i][j+1] + v_old[i][j-1]);
134                     }
135                 }
136             }
137         }
138     }
139 }
```

```

133         }
134     }
135 }
136
137     normaV = calculaNormaInfinita(v, N, N);
138     normaV_old = calculaNormaInfinita(v_old, N, N);
139     normaDif = fabs((normaV - normaV_old)/normaV_old);
140
141     if(normaDif < errorTolerance){
142         printf ("\nRetornando\n");
143         return;
144     }
145 }
146 }
147
148 int main(int argc, char *argv[]){
149
150     int i, j, k;
151     double dx, dy, d, x, y, r;
152     double **v, **v_old;
153
154     dx = (xFinal - xInicial)/N;
155     dy = (yFinal - yInicial)/N;
156
157     v = malloc(N*sizeof(double *));
158     for(int i=0;i<N;i++){
159         v[i] = malloc(N*sizeof(double));
160     }
161
162     v_old = malloc(N*sizeof(double *));
163     for(int i=0;i<N;i++){
164         v_old[i] = malloc(N*sizeof(double));
165     }
166
167     setCondicoesContorno(v, dx, dy);
168     setCorteAngulo(v, dx, dy);
169     calculoPotencial(v, v_old, dx, dy);
170     getResultados(v, dx, dy);
171
172     free(v);
173     free(v_old);
174 }
```

Listing 10.1 – Programa Serial

## 10.2 B - Código com simetria e boas práticas

```
1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5
6 // Declara es e constantes
7
8 #define N 1000
9 #define tolerance 0.1
10#define maxIt 1000000
11#define potencialInterno -50.0
12#define potencialExterno 100.0
13#define raioInterno 1.0
14#define raioExterno 3.0
15#define theta 0.4
16
17// Dimens es da malha
18
19#define xInicial -5.0
20#define xFinal 5.0
21#define yInicial -5.0
22#define yFinal 5.0
23
24 void setContour(double **v, double dx, double dy, int chunk, int
    numberThreads){
25     int i, j;
26     double x, y, r;
27
28     for(i = 0; i < N/2; i++){
29         for(j = 0; j < N/2; j++){
30             x = xInicial + i*dx;
31             y = yInicial + j*dy;
32             r = sqrt(x*x + y*y);
33
34             if(fabs(raioExterno - r) < tolerance){
35                 v[i][j] = potencialExterno;
36                 v[N-i-1][j] = potencialExterno;
37                 v[i][N-j-1] = potencialExterno;
38                 v[N-i-1][N-j-1] = potencialExterno;
39             }
40             else{
41
42                 if(fabs(raioInterno - r) < tolerance){
43                     v[i][j] = potencialInterno;
44                     v[N-i-1][j] = potencialInterno;
45                     v[i][N-j-1] = potencialInterno;
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
```

```

46                     v[N-i-1][N-j-1] = potencialInterno;
47                 }
48             }
49         }
50     }
51 }
52
53 void setAngleCut(double **v, double dx, double dy, int chunk, int
54 numberThreads){
55     int i, j;
56     double x, y, r, ang;
57
58     for(i = 0; i < N/2; i++){
59         for(j = N/2; j < N; j++){
60             x = xInicial + i*dx;
61             y = yInicial + j*dy;
62             r = sqrt(x*x + y*y);
63             ang = acos(x/r);
64
65             if(fabs(raioExterno - r) < tolerance && ang < theta){
66                 v[i][j] = 0;
67                 v[i][N-j-1] = 0;
68             }
69             else{
70                 if(fabs(raioInterno - r) < tolerance && ang < theta){
71                     v[i][j] = 0;
72                     v[i][N-j-1] = 0;
73                 }
74             }
75         }
76     }
77 }
78
79 void getResults(double **v, double dx, double dy){
80
81     int i, j;
82     double x, y;
83     FILE *arquivo;
84
85     arquivo = fopen("dadosLaplaceOpenMP.dat", "w");
86
87     fprintf(arquivo, "x\ty\tz\n");
88
89     for(i = 0; i < N; i++){
90         for(j = 0; j < N; j++){
91             x = xInicial + i*dx;

```

```

92         y = yInicial + j*dy;
93         fprintf(arquivo, "%g %g %g\n", x, y, v[i][j]);
94     }
95 }
96
97 fclose(arquivo);
98 }
99
100 void finiteDifference(double **v, double **v_old, double dx, double dy,
101                      int chunk, int numberThreads){
102
103     int a, b, i, j, k;
104     double x, y, r;
105
106     for(k = 0; k < maxIt; k++){
107
108         for(a = 0; a < N; a++)
109             for(b = 0; b < N; b++)
110                 v_old[a][b] = v[a][b];
111
112         for(i = 1; i < N-1; i++){
113             for(j = N/2; j < N-1; j++){
114
115                 x = xInicial + i*dx;
116                 y = yInicial + j*dy;
117                 r = sqrt(x*x + y*y);
118
119                 if(fabs(raioExterno - r) < tolerance && (v[i][j] ==
120
121                     potencialExterno )) {
122                     v[i][j] = potencialExterno;
123                 }
124                 else{
125                     if(fabs(raioInterno - r) < tolerance && (v[i][j]
126
127                     == potencialInterno)) {
128                         v[i][j] = potencialInterno;
129                     }
130                     else{
131                         v[i][j] = (1.0/4.0)*(v_old[i+1][j] + v_old[i
132
133                         -1][j] + v_old[i][j+1] + v_old[i][j-1]);
134                         v[i][N-j-1] = v[i][j];
135                     }
136                 }
137             }
138         }
139     }
140 }
141
142 }
143
144 int main(int argc, char *argv[]){

```

```

135
136     int i, j, k, chunk, numberThreads;
137     double dx, dy, d, x, y, r;
138     double **v, **v_old;
139
140     numberThreads = atoi(argv[1]);
141     chunk = maxIt/numberThreads;
142
143     dx = (xFinal - xInicial)/N;
144     dy = (yFinal - yInicial)/N;
145
146     v = malloc(N*sizeof(double *));
147     for(i=0;i<N;i++){
148         v[i] = malloc(N*sizeof(double));
149     }
150
151     v_old = malloc(N*sizeof(double *));
152     for(i=0;i<N;i++){
153         v_old[i] = malloc(N*sizeof(double));
154     }
155
156     setContour(v, dx, dy, chunk, numberThreads);
157     setAngleCut(v, dx, dy, chunk, numberThreads);
158     finiteDifference(v, v_old, dx, dy, chunk, numberThreads);
159     getResults(v, dx, dy);
160
161     free(v);
162     free(v_old);
163 }
```

Listing 10.2 – Programa Serial

### 10.3 C - Código paralelizado

```

1 #include <omp.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <math.h>
5
6 // Declara es e constantes
7
8 #define N 1000
9 #define tolerance 0.1
10#define maxIt 1000000
11#define potencialInterno -50.0
12#define potencialExterno 100.0
13#define raioInterno 1.0
14#define raioExterno 3.0
```

```

15 #define theta 0.4
16
17 // Dimens es da malha
18
19 #define xInicial -5.0
20 #define xFinal 5.0
21 #define yInicial -5.0
22 #define yFinal 5.0
23
24 void setContour(double **v, double dx, double dy, int chunk, int
      numberThreads){
25     int i, j;
26     double x, y, r;
27
28     for(i = 0; i < N/2; i++){
29         for(j = 0; j < N/2; j++){
30             x = xInicial + i*dx;
31             y = yInicial + j*dy;
32             r = sqrt(x*x + y*y);
33
34             if(fabs(raioExterno - r) < tolerance){
35                 v[i][j] = potencialExterno;
36                 v[N-i-1][j] = potencialExterno;
37                 v[i][N-j-1] = potencialExterno;
38                 v[N-i-1][N-j-1] = potencialExterno;
39             }
40             else{
41
42                 if(fabs(raioInterno - r) < tolerance){
43                     v[i][j] = potencialInterno;
44                     v[N-i-1][j] = potencialInterno;
45                     v[i][N-j-1] = potencialInterno;
46                     v[N-i-1][N-j-1] = potencialInterno;
47                 }
48             }
49         }
50     }
51 }
52
53 void setAngleCut(double **v, double dx, double dy, int chunk, int
      numberThreads){
54     int i, j;
55     double x, y, r, ang;
56
57     for(i = 0; i < N/2; i++){
58         for(j = N/2; j < N; j++){
59

```

```

60             x = xInicial + i*dx;
61             y = yInicial + j*dy;
62             r = sqrt(x*x + y*y);
63             ang = acos(x/r);
64
65             if(fabs(raioExterno - r) < tolerance && ang < theta){
66                 v[i][j] = 0;
67                 v[i][N-j-1] = 0;
68             }
69             else{
70                 if(fabs(raioInterno - r) < tolerance && ang < theta){
71                     v[i][j] = 0;
72                     v[i][N-j-1] = 0;
73                 }
74             }
75         }
76     }
77 }
78
79 void getResults(double **v, double dx, double dy){
80
81     int i, j;
82     double x, y;
83     FILE *arquivo;
84
85     arquivo = fopen("dadosLaplaceOpenMP.dat", "w");
86
87     fprintf(arquivo, "x\ty\tz\n");
88
89     for(i = 0; i < N; i++){
90         for(j = 0; j < N; j++){
91             x = xInicial + i*dx;
92             y = yInicial + j*dy;
93             fprintf(arquivo, "%g %g %g\n", x, y, v[i][j]);
94         }
95     }
96
97     fclose(arquivo);
98 }
99
100 void finiteDifference(double **v, double **v_old, double dx, double dy,
101                         int chunk, int numberThreads){
102
103     int a, b, i, j, k;
104     double x, y, r;
105
106     omp_set_num_threads(numberThreads);

```

```

106
107     #pragma omp parallel private(a, b, x, y, r, i, j, k) shared(v, v_old)
108 }
109 {
110     #pragma omp for schedule(dynamic, chunk)
111
112     for(k = 0; k < maxIt; k++){
113
114         for(a = 0; a < N; a++)
115             for(b = 0; b < N; b++)
116                 v_old[a][b] = v[a][b];
117
118         for(i = 1; i < N-1; i++){
119             for(j = N/2; j < N-1; j++){
120
121                 x = xInicial + i*dx;
122                 y = yInicial + j*dy;
123                 r = sqrt(x*x + y*y);
124                 if(fabs(raioExterno - r) < tolerance && (v[i][j] ==
125                     potencialExterno)){
126
127                     v[i][j] = potencialExterno;
128                 }
129                 else{
130
131                     if(fabs(raioInterno - r) < tolerance && (v[i][j] ==
132                         potencialInterno)){
133
134                         v[i][j] = potencialInterno;
135                     }
136                     else{
137
138                         v[i][j] = (1.0/4.0)*(v_old[i+1][j] + v_old[i-
139                         1][j] + v_old[i][j+1] + v_old[i][j-1]);
140                         v[i][N-j-1] = v[i][j];
141                     }
142                 }
143             }
144         }
145     }
146
147     int main(int argc, char *argv[]){
148
149         int i, j, k, chunk, numberThreads;
150         double dx, dy, d, x, y, r;
151         double **v, **v_old;
152
153         numberThreads = atoi(argv[1]);
154         chunk = maxIt/numberThreads;

```

```
149
150     dx = (xFinal - xInitial)/N;
151     dy = (yFinal - yInitial)/N;
152
153     v = malloc(N*sizeof(double *));
154     for(i=0;i<N;i++){
155         v[i] = malloc(N*sizeof(double));
156     }
157
158     v_old = malloc(N*sizeof(double *));
159     for(i=0;i<N;i++){
160         v_old[i] = malloc(N*sizeof(double));
161     }
162
163     setContour(v, dx, dy, chunk, numberThreads);
164     setAngleCut(v, dx, dy, chunk, numberThreads);
165     finiteDifference(v, v_old, dx, dy, chunk, numberThreads);
166     getResults(v, dx, dy);
167
168     free(v);
169     free(v_old);
170 }
```

Listing 10.3 – Programa Serial