APOLO ALVES FERREIRA DA SILVA

ANÁLISE COMPARATIVA DO DESEMPENHO: IMPLEMENTAÇÃO SERIAL E PARALELA DO MÉTODO DAS DIFERENÇAS FINITAS PARA A TRANSFERÊNCIA DE CALOR EM UMA CHAPA BIDIMENSIONAL

Resende - RJ

2025

ASSOCIAÇÃO EDUCACIONAL DOM BOSCO
FACULDADE DE ENGENHARIA DE RESENDE

APOLO ALVES FERREIRA DA SILVA

ANÁLISE COMPARATIVA DO DESEMPENHO: IMPLEMENTAÇÃO SERIAL E PARALELA DO MÉTODO DAS DIFERENÇAS FINITAS PARA A TRANSFERÊNCIA DE CALOR EM UMA CHAPA BIDIMENSIONAL

Trabalho de Graduação apresentado à Associação Educacional Dom Bosco, Faculdade de Engenharia de Resende, Curso de Engenharia Mecânica, como requisito parcial para obtenção do diploma de Bacharel em Engenharia Mecânica.

Resende - RJ
2025

Catalogação na fonte
Biblioteca Central da Associação Educacional Dom Bosco – Resende-RJ

S586 Silva, Apolo Alves Ferreira da
Análise comparativa do desempenho: implementação serial e paralela do método das diferenças finitas para a transferência de calor em uma chapa bidimensional / Apolo Alves Ferreira da Silva - 2025.
75 f.

Orientador: Said Moraes Sirio Rocha
Trabalho de conclusão de curso apresentado como requisito parcial à finalização do curso de Engenharia Mecânica da Faculdade de Engenharia de Resende da Associação Educacional Dom Bosco.

1. Engenharia mecânica. 2. Transferência de calor. 3. Diferença finita. 4. Simulação numérica. I. Rocha, Said Moraes Sirio. II. Faculdade de Engenharia de Resende. III. Associação Educacional Dom Bosco. IV. Título.

CDU 536.24(043)

Bibliotecária Maria Ismênia de Oliveira – CRB – 7 / 7667

APOLO ALVES FERREIRA DA SILVA

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA A OBTENÇÃO DO DIPLOMA DE "**GRADUADO EM ENGENHARIA MECÂNICA**"

APROVADO EM SUA FORMA FINAL PELA BANCA EXAMINADORA

BANCA EXAMINADORA:

Prof. (a).: Said Moraes Sirio Rocha
Orientador

Prof. (a).: José Salvador da Motta Reis
Membro da Banca

Prof.(a) Rafael Raider Leoni
Membro Externo

Novembro de 2025

Dedico este trabalho de modo especial, à minha querida esposa e filhas: Letícia Ambrósio de Souza, Rebecca Ambrósio Alves e Isis Eloah Ambrósio Alves.

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus, fonte da vida e da graça. Agradeço a minha vida, minha inteligência, minha família e meus amigos;

ao meu orientador, Prof. Said Moraes Sírio Rocha que jamais deixou de me incentivar. Sem a sua orientação, dedicação e auxílio, o estudo aqui apresentado seria praticamente impossível;

aos meus pais André Luís Ferreira da Silva e Dilma Alves Ferreira da Silva, que apesar das dificuldades enfrentadas, sempre incentivaram meus estudos;

aos funcionários da empresa Gustavo Amorim, Willie Prudente, Marivaldo Félix, Adan Araújo, Claudio Nogueira e Luís Barbosa, pela dedicação, presteza e principalmente pela vontade de ajudar;

aos funcionários das Faculdades Dom Bosco pela dedicação e alegria no atendimento.

“Devemos acreditar que somos talentosos para algumas coisas, e que essa coisa, a qualquer custo, deve ser alcançada.”

Marie Curie

RESUMO

Este estudo tem como objetivo analisar o desempenho computacional de duas abordagens — serial e paralela — aplicadas à simulação da transferência de calor em uma chapa bidimensional utilizando o método das diferenças finitas. A implementação será realizada em linguagem *Python*, com a versão paralela utilizando uma biblioteca adequada para aceleração do código computacional implementado. A análise será conduzida comparando o tempo de execução e a eficiência das abordagens em diferentes tamanhos de malha (resolução espacial) e número de iterações. Os resultados mostram que a implementação paralela apresenta ganhos significativos de desempenho para grandes domínios de simulação, mantendo a precisão dos resultados térmicos. Esta comparação é relevante para aplicações em engenharia que exigem simulações eficientes em tempo real ou com alta resolução espacial.

PALAVRAS-CHAVE: Transferência de Calor. Método das Diferenças Finitas. Simulação Numérica. Processamento Paralelo. Desempenho Computacional.

ABSTRACT

This study aims to analyze the computational performance of two approaches — serial and parallel programming — applied to the simulation of heat transfer in a two-dimensional plate using the finite difference method. The implementation will be carried out in *Python*, with the parallel version using a suitable library for computational acceleration. The analysis will be conducted by comparing the execution time and efficiency of the approaches in different mesh sizes (spatial resolution) and number of iterations. The results will show that the parallel implementation presents significant performance gains for large simulation domains, while maintaining the accuracy of the thermal results. This comparison is relevant for engineering applications that require efficient simulations in real time or with high spatial resolution.

KEYWORDS: Heat Transfer. Finite Difference Method. Numerical Simulation. Parallel Processing. Computational Performance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

2D	Duas dimensões
3D	Três dimensões
EDP	Equações Diferenciais Parciais
Gb	*Gigabytes*
HPC	*High Perfomance Computing*
IDE	*Integrated Development Environment*
JIT	*Just-in-Time*
LNE	Laminação Normalizada de Estruturas
MDF	Método das Diferenças Finitas
PVC	Problema de Valor de Contorno
RAM	*Random Access Memory*
Sp	*Speedup*
TCC	Trabalho de Conclusão de Curso
TEA	Taxa de acúmulo de energia térmica interna sensível
TEG	Taxa total de energia térmica sensível gerada
TLE	Taxa líquida de entrada de calor
TLS	Taxa líquida de saída de calor

LISTA DE SÍMBOLOS

Δx	comprimento (m)
Δy	espessura (m)
Δz	profundidade (m)
T	temperatura do local (º C)
$\dot{E}_g$	Energia gerada (W)
$\dot{E}_{acu}$	Energia acumulada (W)
$\dot{E}_s$	Energia de fluxo de saída (W)
$\dot{E}_e$	Energia de fluxo de entrada (W)
ΔT	variação de temperatura (º C)
t	tempo (s)
α	difusividade térmica (m²/s)
k	condutividade térmica (W/m.K)
ρ	densidade (kg/m³)
c_p	calor específico (J/kg.K)
$\dot{q}$	taxa de calor gerado (W/m²)
T_s	tempo de execução da programação em paralelo (s)
T_p	tempo de execução da programação em paralelo (s)
N	número da dimensão da matriz
N*	número de núcleos do processador
L	comprimento (m)
W	espessura (m)
k^*	número do passo

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização

A ciência que se preocupa com a determinação das taxas de transferências de energia é a transferência de calor. O calor é definido como a forma de energia que pode ser transferida de um sistema para outro em consequência da diferença de temperatura entre eles, que sempre ocorre da maior para a menor temperatura (Bergman; Levine, 2019; Çengel; Ghajar, 2011).

O estudo dos fenômenos térmicos representa uma área de grande relevância dentro das ciências exatas, principalmente por sua ampla aplicação em processos que envolvem a transferência de calor. A importância desse tema se destaca pela presença constante do calor em trânsito em diversos contextos de simulações da engenharia e da pesquisa científica.

Segundo Bergman et al. (2019), os problemas de natureza térmica podem ser classificados, de acordo com a forma como o calor se propaga, em três tipos principais: condução, convecção e radiação. Tão importante quanto identificar o tipo de processo envolvido é compreender como o calor se distribui em determinado meio. Essa distribuição é descrita por uma equação diferencial conhecida como equação da difusão de calor.

A solução analítica da propagação de calor é obtida a partir das equações diferenciais que exercem um papel essencial na modelagem de fenômenos físicos e permitem descrever e prever o comportamento de sistemas em simulações numéricas. No entanto, para as geometrias mais complexas, as soluções analíticas tornam-se extremamente difíceis ou até inviáveis devido à complexidade dos problemas (De Brito et al., 2025; Kisabo Aliyu; Festus Olatoyinbo, 2021; Santos, 2012).

Com o avanço da tecnologia computacional e a crescente demanda por soluções precisas e rápidas, surgiram os métodos numéricos, que possibilitam otimizações com soluções aproximadas para problemas complexos. Esses métodos transformam um problema contínuo, com um número infinito de variáveis, em um problema discreto, com um número finito de variáveis, o que permite sua resolução por meio de códigos computacionais. Os métodos numéricos, se baseiam na substituição da equação diferencial pelo conjunto de equações algébricas para temperaturas desconhecidas, bem como são utilizados para a resolução de problemas com geometrias e condições de contorno complexas ou propriedades variáveis, e

podem ser obtidas por computadores (Çengel; Ghajar, 2011; De Brito et al., 2025).

Entre os diversos métodos disponíveis para resolver equações diferenciais, este trabalho adota a técnica numérica: método das diferenças finitas, com o objetivo de determinar a temperatura central de uma placa, mediante a transferência de calor do problema de condução térmica na placa bidimensional, no regime transiente para convergir até o regime estacionário, obtendo assim a temperatura de equilíbrio.

1.2 Justificativa

A rápida evolução dos computadores de alta velocidade, aliada à criação de programas cada vez mais poderosos e fáceis de usar, impulsionou o surgimento e a popularização dos métodos numéricos, tornando possível a criação de algoritmos para a resolução por meio de computadores e otimizando problemas de alta escalabilidade (De Brito et al., 2025; Kisabo Aliyu; Festus Olatoyinbo, 2021; Santos, 2012) .

Uma forma de obter uma solução particular de um Problema de Valor de Contorno (PVC) é utilizando o Método das Diferenças Finitas (MDF) para aproximações associado as Equações Diferenciais Parciais (EDP) do problema da difusão de calor. Para isto acontecer é necessário entender que o MDF consiste em reduzir um problema contínuo, com um número finito de incógnitas. Onde os valores entre os pontos discretizados deve ser representado por uma série de pontos dispostos, onde a relação entre os pontos é determinada por meio de expansões truncadas da série de Taylor que permite a substituição das derivadas parciais por fórmulas discretas de diferenças (De Brito et al., 2025; Kisabo Aliyu; Festus Olatoyinbo, 2021; Santos, 2012).

Os métodos numéricos por meio de recursos computacionais, tem exercido grande influência na educação e na otimização de problemas da engenharia nos últimos anos. Assim, compreender e desenvolver essas ferramentas é essencial para o crescimento e o aprimoramento profissional e estudos comparativos de métodos iterativos permitem particionar o sistema de tal modo que os cálculos podem ser realizados em processadores diferentes (Michels et al., 2020).

1.3 Objetivos

1.3.1 Geral

Implementar códigos de programação serial e em paralelo para análise da transferência de calor numa placa bidimensional em regime transiente em aplicações de engenharia, onde será utilizado o método das diferenças finitas e encontrar a temperatura central da placa quando atingir o equilíbrio.

1.3.2 Específicos

Dos objetivos específicos temos que:

- Investigar e realizar estudos teóricos sobre a difusão de calor em duas dimensões;
- Elaborar um procedimento metodológico para a resolução do problema;
- Obter a Solução Analítica para a Temperatura central da placa em estudo;
- Demonstrar o Método das Diferenças Finitas para obter a solução numérica através da implementação de códigos computacionais em *Python* para obter pontos de temperaturas ao longo da placa para diferentes tamanhos de matrizes (7x7, 11x11, 25x25, 51x51, 75x75, 101x101, 207x207, 301x301, 501x501, 733x733, 851x851 e 1001x1001);
- Realizar a análise de desempenho entre implementações serial e paralela, encontrando o: *Speedup* e Eficiência, conforme tamanho das dimensões das matrizes;
- Comparar as diferenças de erro entre a solução analítica e as soluções numéricas encontradas através dos códigos implementados.

1.4 Estrutura do trabalho

Para o desenvolvimento do presente trabalho foi necessário realizar um levantamento bibliográfico com fontes relacionadas a teoria de transferência de calor, métodos numéricos e computação paralela, com a finalidade de esclarecer o processo de propagação de calor, quais suas características e como quantificá-los em forma de equação para encontramos as soluções analítica e numérica do problema.

Com a formulação da equação do calor, que é a peça fundamental para a solução analítica de problemas de natureza térmica, obtém-se a um ponto de controle aleatório que será nossa temperatura de referência, para este estudo foi escolhido a temperatura central da placa. Consequentemente, consideram-se as condições de contorno às quais uma placa está sendo submetida e a utilização do método numérico: MDF para obter a solução numérica do problema. Para a implementação computacional utilizou-se o *software Pycharm*, utilizando a linguagem de programação: *Python*. O *software* foi escolhido como ferramenta de trabalho, devido a praticidade e sua aceitação no meio acadêmico.

2 FUNDAMENTAÇÃO TEÓRICA

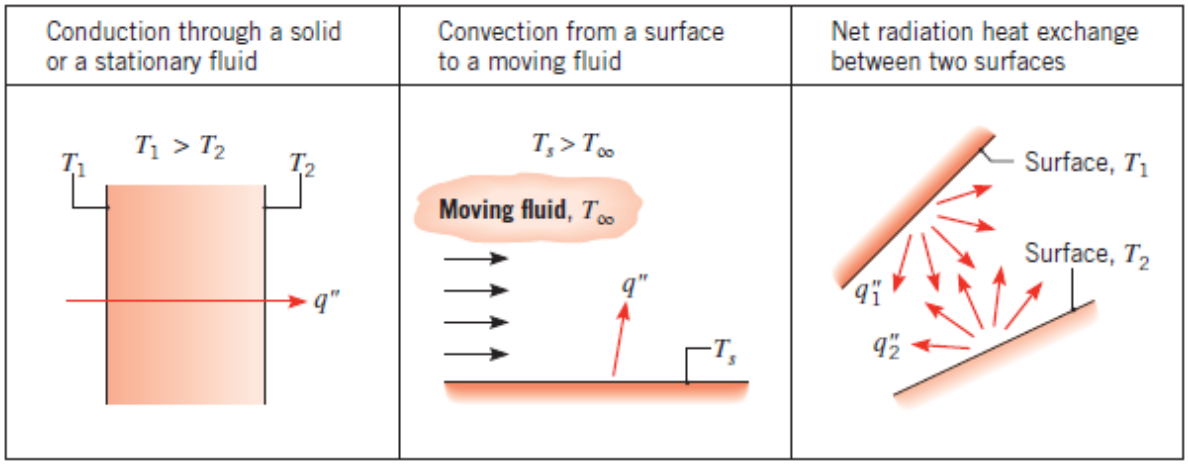
2.1 Transferência de calor

O calor é uma energia térmica em trânsito gerada do resultado da diferença de temperaturas no espaço. Logo, obedece a um dos enunciados da termodinâmica que diz que o calor sempre é transferido de um meio de alta concentração para um meio de baixa concentração de calor (Bergman; Levine, 2019; Çengel; Ghajar, 2011).

2.1.1 Propagação de Calor

A transferência de calor ocorre por três mecanismos principais: condução, convecção e radiação.

Figura 1 - Modo de transferência de calor: condução, convecção e radiação



Fonte: Bergman; Levine (2019)

As moléculas interagem entre si realizando o mecanismo de condução, quando uma

substância sólida se aquece, por exemplo, os elétrons e átomos vibram com grande intensidade por conta da alta temperatura exposta. A energia associada a estas vibrações são transferidas através de colisões entre os átomos, as partículas mais energéticas transferem energia na forma de calor para as menos energéticas (Walker; Resnick; Halliday, 2014).

2.1.2 Difusão de calor em 3D

Para equilíbrio de energia térmica, temos a seguinte formulação:

Equação 1 - Equilíbrio térmico

$$[TEA] = [TLE] + [TLS] + [TEG]$$

Sendo,

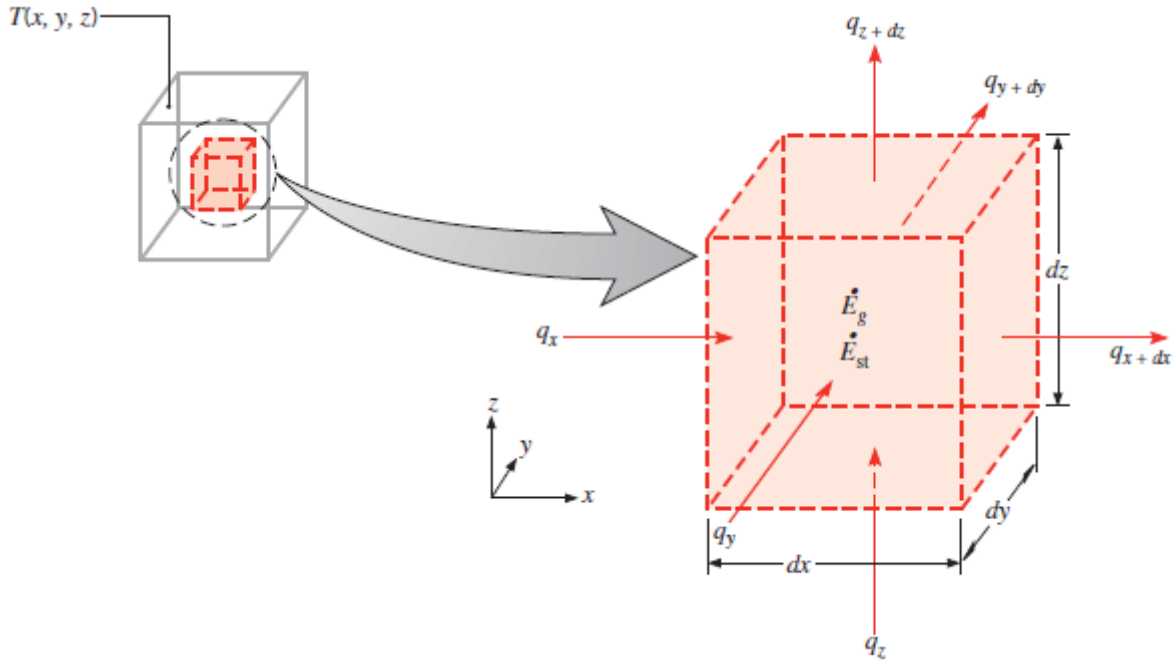
[TEA] = Taxa de acúmulo de energia térmica interna sensível no elemento infinitesimal;

[TLE] = Taxa líquida de entrada de calor no elemento infinitesimal;

[TLS] = Taxa líquida de saída de calor no elemento infinitesimal;

[TEG] = Taxa total de energia térmica sensível gerada por unidade de volume do elemento infinitesimal;

Figura 2 - Difusão de Calor no volume infinitesimal diferencial para análise de condução de calor em coordenadas cartesianas



Fonte: Bergman; Levine (2019)

Pela equação da difusão térmica de calor nas três direções do plano cartesiano tridimensional, temos os fluxos:

De Entrada:

$$q_x\,, q_y\; e\; q_z$$

De Saída:

$$q_{x+dx} \;=\; q_x \;+\; \frac{\partial q_x}{\partial x}dx$$

$$q_{y+dy} \;=\; q_y \;+\; \frac{\partial q_y}{\partial y}dy$$

$$q_{z+dz} \;=\; q_z \;+\; \frac{\partial q_z}{\partial z}dz$$

A energia gerada no ponto infinitesimal:

$$\dot{E}_g \;=\; \dot{q}dxdydz \text{ (2.1)}$$

O acúmulo de energia no ponto infinitesimal:

$$\dot{E}_{acu} = \; \rho c_p \frac{\partial T}{\partial t}dxdydz \text{ (2.2)}$$

Utilizando as equações de cada direção associadas a difusão térmica para deduzirmos as expressões da transferência de calor, segue que:

$$\dot{E}_{acu} = \; \dot{E}_e - \dot{E}_s + \; \dot{E}_g \quad \text{(2.3)}$$

Substituindo os fluxos de entrada e saída com as equações: (2.1), (2.2) e (2.3) na Equação 1, temos que:

$$\rho c_p \frac{\partial T}{\partial t}dxdydz = \; q_x + \; q_y + \; q_z - \left(q_x \; + \frac{\partial q_x}{\partial x}dx + q_y \; + \frac{\partial q_y}{\partial y}dy + \; q_z \; + \frac{\partial q_z}{\partial z}dz\right) + \dot{q}dxdydz \text{ (2.4)}$$

Logo, reduzimos a equação para:

$$\rho c_p \frac{\partial T}{\partial t}dxdydz = \; -\left(+ \frac{\partial q_x}{\partial x}dx + \frac{\partial q_y}{\partial y}dy \; + \frac{\partial q_z}{\partial z}dz\right) + \dot{q}dxdydz \text{ (2.5)}$$

Lembrando que para as três direções, temos:

$$q_x = \; -kdydz\frac{\partial T}{\partial t}\,;\, q_y = \; -kdxdz\frac{\partial T}{\partial t} \text{ e } q_z = \; -kdxdy\frac{\partial T}{\partial t}$$

Derivando, temos:

$$\rho c_p \frac{\partial T}{\partial t} = \frac{\partial}{\partial x}\left(k \frac{\partial T}{\partial x}\right) + \frac{\partial}{\partial y}\left(k \frac{\partial T}{\partial y}\right) + \frac{\partial}{\partial z}\left(k \frac{\partial T}{\partial z}\right) + \dot{q} \quad (2.6)$$

Dividindo a equação (2.6) por k, temos a equação geral da difusão de calor (Bergman; Levine (2019)):

Equação 2 - Equação Geral de Calor

$$\frac{1}{\alpha}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} + \frac{\dot{q}}{k} \quad (2.7)$$

Onde: $\alpha = \frac{k}{\rho c_p}$, sendo os termos:

$\alpha = difusividade\ térmica, k = condutividade\ térmica$ e $c_p = calor\ específico.$

2.1.3 Solução analítica para transferência por condução bidimensional

Para cálculos em geometrias complexas ou condições não homogêneas, métodos numéricos são preferíveis, porém para um simples caso de uma placa bidimensional, a equação de calor pode ser resolvida analiticamente usando o método de separação de variáveis.

Iniciamos a resolução, atribuindo para nosso problema as seguintes hipóteses:

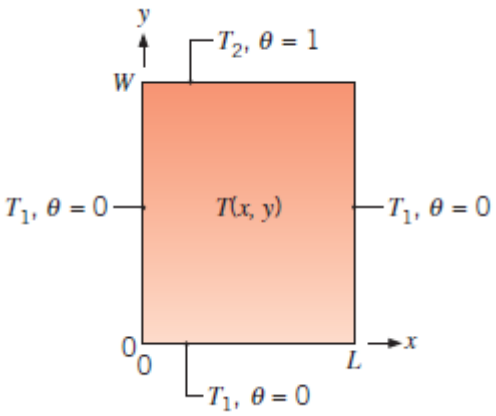
1) Para condução de Calor na Placa 2D, devemos seguir as seguintes hipóteses:
2) Sem geração interna de calor: $\frac{\dot{q}}{k} = 0$
3) Condução de Calor Bidimensional: $\frac{\partial^2 T}{\partial z^2} = 0$
4) Regime Transiente: $\frac{1}{\alpha}\frac{\partial T}{\partial t} \neq 0$

Substituindo, as condições de hipóteses, temos:

Equação 3 - Equação de Calor em Regime Transiente

$$\frac{1}{\alpha}\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \quad (2.8)$$

Figura 3 - Condução bidimensional em uma placa retangular delgada



Fonte: Bergman; Levine (2019)

Para a o ponto de interesse em função de $T(x, y)$, temos que introduzir uma transformação, sendo:

$$\theta \equiv \frac{T - T_1}{T_2 - T_1} \quad (2.9)$$

A equação diferencial deve ser transformada em uma equação da seguinte forma:

$$\frac{\partial^2 \theta}{\partial x^2} + \frac{\partial^2 \theta}{\partial y^2} = 0 \quad (2.10)$$

Como a equação é de segunda ordem em x em y, duas condições de contorno são necessárias para cada uma das coordenadas da equação para encontrarmos a Solução Final. O desenvolvimento e operações matemáticas necessárias para entendermos a propagação de calor na placa com as especificações e condições iniciais e condições de contorno será desenvolvida no capítulo 4 – Resultados e Discussões deste TCC.

2.2 Métodos numéricos

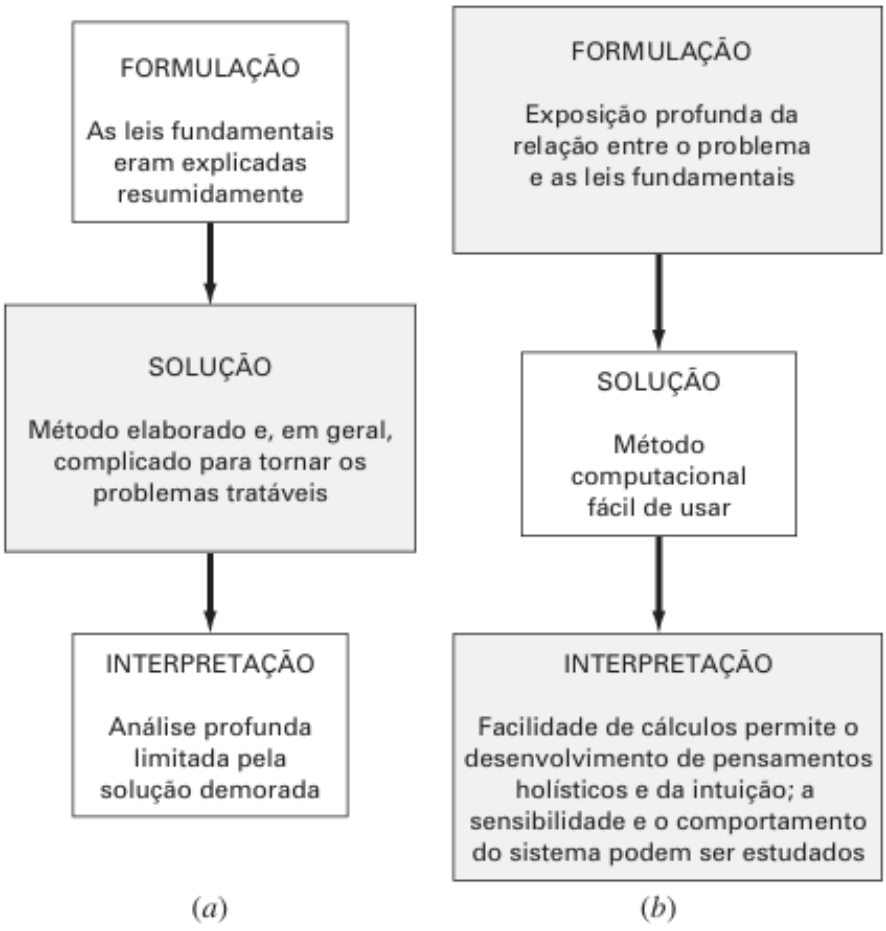
Na matemática, os problemas podem ser resolvidos de duas formas: soluções analíticas e soluções numéricas. As soluções analíticas buscam uma resposta abrangente e contínua, as soluções numéricas são obtidas discretizando o domínio e calculando a função apenas em pontos específicos, resultando em aproximações. O uso de métodos numéricos traz vantagens como automatização dos cálculos e modelos próximos da realidade, sendo amplamente

utilizados em engenharia para resolver problemas práticos (Atkinson; Han, 2004).

Segundo Chapra et al. (2006) os métodos numéricos são técnicas pelas quais os problemas matemáticos são formulados de modo que possam ser resolvidos com operações aritméticas. Embora existam muitos tipos de métodos numéricos, eles têm uma característica em comum: invariavelmente envolvem grande número de cálculos aritméticos tediosos. Não é nada surpreendente que, com o desenvolvimento de computadores digitais rápidos e eficientes, tornando relevante o papel dos métodos numéricos na resolução de problemas de engenharia.

Na era pré computadores, mostrados a seguir na Figura 4 (a) e na era do computador Figura 4 (b), percebemos que a etapas de: formulação, tratativas dos dados e interpretação das soluções são diferentes.

Figura 4 - Fases de Resolução de problemas de engenharia



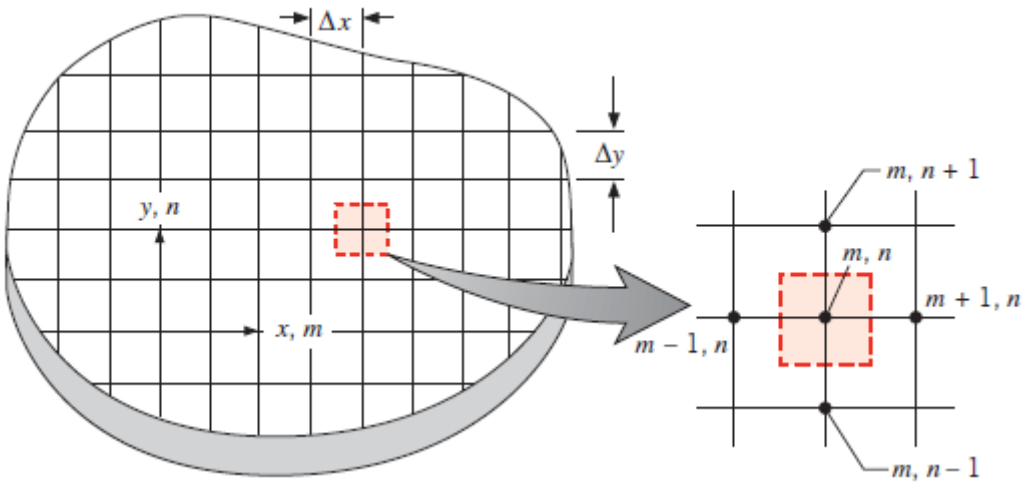
Fonte: Chapra; Canale (2006)

As etapas para resolução de problemas de engenharia utilizando métodos numéricos, é a formulação do problema, modelagem do sistema, aplicação do método numérico e interpretar os resultados. Para todas as etapas envolvidas do processo desde a formulação até a interpretação do problema é relevante a escolha de um método que satisfaça as equações com a finalidade de obter a resposta aproximada.

2.1.4 Método das diferenças finitas

É uma técnica numérica que consiste em resolver equações diferenciais, que discretiza o domínio contínuo do problema em uma malha de pontos aproximando derivadas por diferenças locais. Para Majumdar (2005) o processo de discretização finitas envolve primeiramente a divisão de uma região de solução em uma malha de linhas que se cruzam, e que são traçadas paralelamente aos eixos de coordenadas. Os pontos de intersecção discretos dessas linhas de grade são chamados de nós ou pontos nodais. A precisão da solução melhora com o aumento dos pontos de nós ou diminuição do tamanho da grade. O MDF pode ser aplicado para PVC que varia no tempo, podendo ser calculado por meio de iterações em esquema explícito que depende do passo anterior ou esquema implícito que depende do tempo posterior.

Figura 5 - Rede Nodal



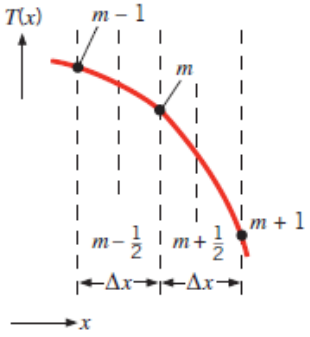
Fonte: Bergman; Levine (2019)

Para cada parte da discretização existirá um nó e em torno deste nó existirão 4 pontos que serão relevantes para utilização da técnica numérica. A aproximação numérica é obtida

através da diferença finita entre os nós. Tal que ocorre:

$$\left.\frac{\partial T}{\partial x}\right|_{m-1/2,n} = \frac{T_{m,n}-T_{m-1,n}}{\Delta x} \quad (2.11) \qquad \left.\frac{\partial T}{\partial x}\right|_{m+1/2,n} = \frac{T_{m+1,n}-T_{m,n}}{\Delta x} \quad (2.12)$$

Figura 6 - Aproximação da Diferença Finita



Fonte: Bergman; Levine (2019)

Para determinarmos a distribuição de temperatura numericamente, devemos ter em mente a equação de conservação apropriada seja escrita para cada um dos pontos nodais de temperatura desconhecida. O conjunto de equações resultante pode então ser resolvido simultaneamente para a temperatura em cada nó. Para qualquer nó interior de um sistema bidimensional sem geração e com condutividade térmica uniforme, a forma exata da exigência de conservação de energia é dada pela equação do calor, Equação de Laplace. No entanto, se o sistema for caracterizado em termos de uma rede nodal, é necessário trabalhar com uma forma aproximada, ou de diferenças finitas, desta equação. Uma equação de diferenças finitas que é adequada para os nós interiores de um sistema bidimensional pode ser inferida diretamente na equação de Laplace. Considere a segunda derivada, $\partial^2 T/\partial x^2$. A partir da Figura 4, o valor desta derivada no ponto nodal (m, n) pode ser aproximado como:

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{m,n} \approx \frac{\partial T/\partial x|_{m+1/2,n} - \partial T/\partial x|_{m-1/2,n}}{\Delta x} \quad (2.13)$$

Os gradientes de temperatura podem, por sua vez, ser expressos como uma função das temperaturas nodais. Ou seja,

$$\left.\frac{\partial T}{\partial x}\right|_{m+1/2,n} \approx \frac{T_{m+1,n} - T_{m,n}}{\Delta x} \quad (2.14)$$

$$\left.\frac{\partial T}{\partial x}\right|_{m-1/2,n} \approx \frac{T_{m,n} - T_{m-1,n}}{\Delta x} \quad (2.15)$$

Derivando as equações, teremos aproximação da diferença finita:

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{m,n} \approx \frac{T_{m+1,n}+T_{m-1,n}-2T_{m,n}}{(\Delta x)^2} \quad (2.16)$$

$$\left.\frac{\partial^2 T}{\partial y^2}\right|_{m,n} \approx \frac{T_{m,n+1}+T_{m,n-1}-2T_{m,n}}{(\Delta y)^2} \quad (2.17)$$

Se $\Delta x = \Delta y$, podemos substituir as equações 2.16 e 2.17 na equação 2.8, o que nos leva a seguinte equação:

$$T_{m,n+1} + T_{m,n-1} + T_{m+1,n} + T_{m-1,n} - 4T_{m,n} \quad (2.18)$$

Logo, temos que para o nó $T_{m.n}$ teremos a seguinte equação:

$$T_{m,n} = \frac{T_{m,n+1}+T_{m,n-1}+T_{m+1,n}+T_{m-1,n}}{4} \quad (2.19)$$

Para discretização do tempo no esquema explícito, ou seja, dependendo das condições de tempo anterior, utilizaremos a equação 3 (2.8) como referência para nossos cálculos. Tal que reorganizamos para:

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \quad (2.20)$$

Desta forma, podemos agora discretizar pelo MDF, todos os membros para obter a difusão a cada de passo k^*, portanto temos:

$$\frac{T_{m,n}^{k^*+1}-T_{m,n}^{k^*}}{\Delta t} = \alpha\left(\frac{T_{m+1,n}^{k^*+1}-2T_{m,n}^{k^*+1}+T_{m-1,n}^{k^*+1}}{(\Delta x)^2} + \frac{T_{m,n+1}^{k^*+1}-2T_{m,n}^{k^*+1}+T_{m,n-1}^{k^*+1}}{(\Delta y)^2}\right) (2.21)$$

Sendo assim, a equação temporal para as iterações do nosso algoritmo é:

$$T_{m,n}^{k^*+1} = T_{m,n}^{k^*} + \alpha \cdot \Delta t \cdot \left(\frac{T_{m+1,n}^{k^*+1}-2T_{m,n}^{k^*+1}+T_{m-1,n}^{k^*+1}}{(\Delta x)^2} + \frac{T_{m,n+1}^{k^*+1}-2T_{m,n}^{k^*+1}+T_{m,n-1}^{k^*+1}}{(\Delta y)^2}\right) (2.22)$$

Onde, k^* é o número do passo e utilizaremos a equação (2.22) para desenvolvimento da

solução numérica do PVC.

2.3 Programação paralela

2.3.1 Contextualização

Um dos principais fatores que permitiram à computação manter seu ritmo acelerado de evolução nos últimos anos está na exploração e no desenvolvimento das arquiteturas paralelas. Essas arquiteturas se baseiam no uso simultâneo de múltiplas unidades de processamento, possibilitando que várias tarefas sejam executadas ao mesmo tempo (Ignácio; Dias, 2023a, 2023b; Parhami, 2002; Tsega, 2022).

Atualmente, diferentes níveis de paralelismo são explorados — desde o processamento interno dos núcleos multicore e multiprocessadores, até o emprego de sistemas distribuídos, como clusters e grids computacionais que se estendem em escala global. Essa abordagem permite alcançar um desempenho significativamente superior ao obtido em arquiteturas tradicionais mais simples, tornando os sistemas computacionais mais eficientes e capazes de atender às crescentes demandas de processamento de dados (Jin et al., 2011; Morais; Silva; Silva, 2020; Santos, 2012).

No entanto, além do avanço nas arquiteturas paralelas (hardware), é fundamental que as aplicações de software também sejam desenvolvidas com base no paradigma do paralelismo. Essa integração entre hardware e software possibilita explorar de forma mais eficiente os benefícios da *HPC – High Performance Computing*, resultando em um desempenho computacional significativamente superior ao obtido em arquiteturas convencionais (Gebali, 2011; Michels et al., 2020; Parhami, 2002).

Dessa maneira, por meio do desenvolvimento e da execução de algoritmos paralelos, torna-se viável resolver problemas de maior complexidade e escala dentro de tempos de processamento aceitáveis, ampliando as possibilidades de aplicação da *HPC (High Performance Computing)* em diversos campos científicos e tecnológicos (Bell; Gray, 2002; Costa et al., 2020; Michels et al., 2020).

2.3.2 Conceitos do paralelismo

Na Programação Paralela ou paralelismo é realizada a divisão de uma determinada aplicação de tarefas em partes, de maneira que essas partes possam ser executadas

simultaneamente, pelos módulos do processador. Os elementos deste processamento devem cooperar entre si utilizando primitivas de comunicação e sincronização, realizando a quebra do paradigma de execução sequencial do fluxo de instruções (Gebali, 2011). Então, a programação paralela permite otimizar recursos disponibilizados pelas arquiteturas paralelas, assim, é necessário que os algoritmos das aplicações estejam aptos para operar neste tipo de arquitetura, a fim de melhorar a sua velocidade de processamento. Para a programação de ambientes de execução paralela, tem-se que utilizar bibliotecas de comunicação paralela que possibilitam a implementação de programas paralelos em ambientes com memória compartilhada e distribuída (Jin et al., 2011).

2.3.3 Bibliotecas

As bibliotecas referem-se a um conjunto de módulos e funções prontas, desenvolvidas para otimizar o desenvolvimento de algoritmos. Essas bibliotecas contêm códigos reutilizáveis que executam tarefas específicas, permitindo que o programador se concentre na lógica do problema sem precisar reescrever funções básicas.

A biblioteca *Numba* é voltada para cálculos numéricos de alto desempenho (Lam; Pitrou; Seibert, 2015). Ela atua como um compilador *JIT* (*Just-In-Time*), que traduz partes do código Python — especialmente os que utilizam a biblioteca *NumPy* — em código de máquina altamente otimizado (Harris et al., 2020).

A *Matplotlib* é uma biblioteca de visualização de dados em Python, amplamente utilizada em pesquisas científicas, análise de dados e engenharia, que permite gerar gráficos 2D e 3D, criando figuras personalizadas e gifs para exibir resultados numéricos de forma visual e interpretável. Sendo possível representar curvas, campos de temperatura, distribuições estatísticas e diversos tipos de visualização gráfica, o que auxilia na análise e interpretação dos resultados computacionais da engenharia.

2.3.4 A métrica de desempenho computacional

Atingir um alto desempenho computacional em sistemas paralelos exige uma gestão rigorosa da localidade de dados, pois o custo de mover informações entre processadores, ou da memória para o processador, frequentemente domina o custo da própria computação (PACHECO, 2011).

O tempo de execução em programação é relevante quando pensa em otimizar os processos de programação. Quando se utiliza memória e escalabilidade entre as versões serial

e paralela, nota-se um ganho no tempo de execução que pode feito uma análise de desempenho entre as programações serial e paralela.

Para uma análise de desempenho é relevante obter o *Speedup* e a eficiência com os tempos de execução obtidos. O *Speedup* ideal, igual ao número de processadores, é uma meta teórica. Na prática, ele é limitado tanto pela porção inerentemente serial do código (Lei de Amdahl) quanto pelos custos de comunicação entre os núcleos e a eficiência de um algoritmo paralelo é o verdadeiro termômetro de seu sucesso; ela nos diz quão bem estamos utilizando o poder computacional disponível, exigindo que o tempo gasto em cálculo exceda largamente o tempo gasto em gerenciamento (PACHECO, 2011).

$$Sp = \frac{\mathrm{T}_s}{T_p}$$

Onde, T_s é o tempo de execução da programação serial e T_p é o tempo de execução da programação em paralelo.

$$Eficiência = \frac{Sp}{N^*}$$

Onde, N^* é a quantidade de núcleos que o processador possui e utiliza nas programações em paralelo, para o i9 de 14ª geração a quantidade é de 32 núcleos disponíveis para a execução dos algoritmos desenvolvidos.

3 MATERIAIS E MÉTODOS

A metodologia de pesquisa adotada para o desenvolvimento do projeto é baseada numa pesquisa de natureza aplicada, buscando desenvolver uma solução para um problema de difusão de calor bidimensional, tendo como objetivo explicativo comparar e avaliar o desempenho das soluções encontradas através de algoritmos: serial e paralelo, buscando analisar sua eficiência. A abordagem de pesquisa é quantitativa porque trata dados mensuráveis e a análise se baseia em cálculos matemáticos, utilizando técnica numérica para resolução do problema. O Procedimento da pesquisa envolve simulação e modelagem com manipulação controlada de variáveis, assim como: condição inicial, condições de contorno, tamanho da placa, especificações do material da placa e tamanho das matrizes, utilizando dois métodos de programação: serial e paralelo, mantendo o problema sob as mesmas condições com a

finalidade de observar os resultados e realizar a análise de desempenho computacional para obter o resultados e assim produzir a simulação dos gráficos das matrizes de temperatura devido a difusão de calor em regime transiente. Temos o painel adaptado de metodologia de pesquisa deste TCC, a seguir:

Figura 7 - Painel de Metodologia de pesquisa.

Natureza	Objetivo	Abordagem	Procedimento de pesquisa
Básico	Exploratório	Quantitativo	Experimental
Aplicado	Descritivo	Qualitativo	Prisma
	Explicativo	Combinado	Estudo Bibliográfico
	Normativo		Modelagem
			Matriz de maturidade
			Estudo de Caso
			Simulação

Fonte: Adaptado de Cronin; George (2023); Gregório et al. (2021) e Kothari; Garg, (2020)

Utilizamos para o experimento o computador desktop particular com as seguintes especificações:

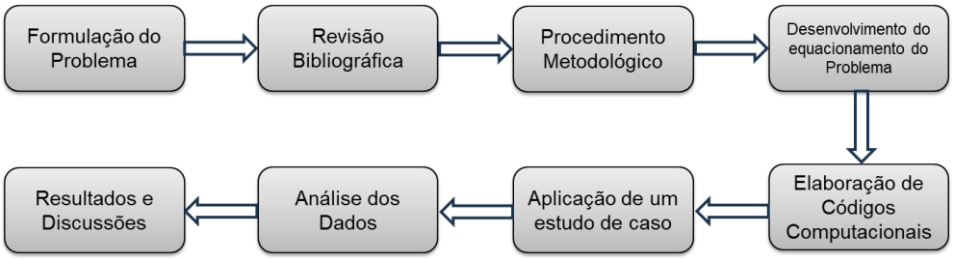
- Processador: Intel® Core™ i9 de Modelo: 14900K;
- Sistema operacional Windows 11 de 64 bits;
- 128 Gb de memória RAM; e
- Placa de vídeo GeForce 4080 de 16 Gb.

Para o presente trabalho, como *IDE* (*Integrated Development Environment*) utilizamos o *software Pycharm* utilizando a linguagem de programação: *Python* em conjunto com as bibliotecas: *Numba*, *NumPy* e *Matplotlib*, que são bibliotecas disponíveis no *Pycharm*.

A metodologia adotada neste trabalho foi organizada de forma sequencial e lógica, conforme apresentado na Figura 7, que ilustra o fluxograma das etapas desenvolvidas ao longo da pesquisa. O processo inicia-se com a formulação do problema, etapa em que se definem os objetivos e se delimita o escopo do estudo. Em seguida, é realizada a revisão bibliográfica, com o propósito de reunir conceitos teóricos e estudos anteriores que fundamentem a pesquisa. Posteriormente, define-se o procedimento metodológico, no qual são estabelecidas as técnicas e ferramentas empregadas para a resolução do problema proposto. A etapa seguinte consiste no desenvolvimento do equacionamento do problema, que serve de base para a elaboração dos códigos computacionais, responsáveis pela implementação numérica do modelo. Na sequência, ocorre a aplicação de um estudo de caso, permitindo validar a metodologia proposta e gerar os

dados necessários para a análise dos resultados. Por fim, a fase de resultados e discussões consolida as conclusões obtidas, possibilitando a interpretação crítica e a verificação do comportamento físico e numérico do sistema estudado.

Figura 8 - Fluxograma de etapas

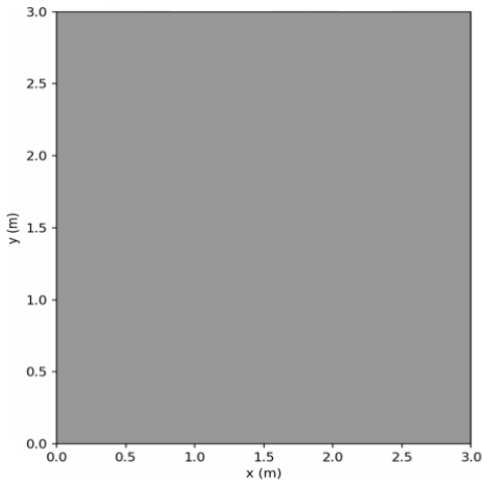


Fonte: Autor (2025)

4 RESULTADOS E DISCUSSÃO

Consideramos no problema uma placa de aço LNE230 com dimensões específicas: 3000x3000x20 mm, sendo uma chapa com a espessura muito menor que a largura e a altura da placa. A condição inicial é que toda a placa inicia com temperatura de 100º C, tendo como condições de contorno: a temperatura da borda superior de 400ºC, a temperatura da borda inferior com 100ºC, a temperatura da borda direita a 100ºC e temperatura da borda esquerda com 100ºC.

Figura 9 - Placa Bidimensional 3000 x 3000 mm



Fonte: Autor (2025)

Para encontrarmos nossa temperatura alvo, ou seja, a temperatura central da placa seguiremos a resolução analítica, admitindo as seguintes C.C. para $\theta(x,y)$:

y = H: $\theta(x,H) = \quad y = H : T(x,H) - 100º\, C = 400 - 100 = 300$

y = 0: $\theta(x,0) = \quad y = 0 : T(x,0) - 100º\, C = 100 - 100 = 0$

x = 0: $\theta(0,y) = \; x = 0 : T(0,y) - 100º\, C = 100 - 100 = 0$

x = L: $\theta(L,y) = \; x = L : T(L,y) - 100º\, C = 100 - 100 = 0$

A Solução de $\theta(x,y)$ pelo método de separação por variáveis, segue o padrão:

$$\theta(x,y) = X(x) \cdot Y(y)$$

$$\frac{X''(x)}{X(x)} + \frac{Y''(y)}{Y(y)} = 0$$

Temos, que:

$$\frac{X''(x)}{X(x)} = -\frac{Y''(y)}{Y(y)}$$

Para obter, a constante de separação λ , temos as seguintes equações:

$$\frac{\mathrm{d}^2\mathrm{X}}{\mathrm{dx}^2}(x) + \lambda^2 X \; = 0$$

$$\frac{\mathrm{d}^2\mathrm{Y}}{\mathrm{dy}^2}(x) - \lambda^2 Y \; = 0$$

As C.C. são homogêneas para X(x) são:

$$X(0) = 0 \;\text{ e }\; X(L) = 0$$

Para X(x) teremos uma solução não trivial que ocorre quando: $\lambda^2 < 0$

Resultando em senos e cossenos para X(x) ou $-\lambda^2$ hiperbólico para Y(y).

As soluções gerais serão:

$$\mathrm{X} = c_1 \cdot \cos \lambda x + c_2 \cdot \sin \lambda x$$
$$Y = c_3 \cdot e^{-\lambda y} + c_4 \cdot e^{+\lambda y}$$

Portanto, para forma geral da solução bidimensional é:

$$\theta = (c_1 \cdot \cos \lambda x + c_2 \cdot \sin \lambda x) \cdot \left(c_3 \cdot e^{-\lambda y} + c_4 \cdot e^{+\lambda y}\right)$$

Aplicando a condição $\theta(0, y) = 0$, fica evidente que $c_1 = 0$ e podemos considerar:

$$c_2 \cdot \sin \lambda x \cdot (c_3 + c_4) = 0$$

Que será satisfeita, quando $c_3 = -c_4$. Se a partir desta colocação, utilizarmos a condição: $\theta(L, y) = 0$, temos:

$$c_2 \cdot c_4 \cdot \sin \lambda x \cdot \left(e^{\lambda y} - e^{-\lambda y}\right) = 0$$

A única forma na qual esta condição pode ser satisfeita é que λ assuma valores discretos para $\sin(\lambda L) = 0$, então:

$$\lambda = \frac{n\pi}{L}$$

Onde, n = 1,2, 3,..., sendo que quando n = 0, o valor será descartado, pois implica $\theta(x, y) = 0$. Desenvolvendo o algebrismo, teremos que a solução pode ser descrita como:

$$\theta(x, y) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) \sinh\left(\frac{n\pi y}{L}\right)$$

Para determinar c_n utilizamos a condição de retorno restante que tem a forma

$$\theta(x, W) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) \sinh\left(\frac{n\pi y}{L}\right)$$

Utilizando uma expansão em série infinita em termos de funções ortogonais, teremos um conjunto de funções: $\mathrm{g}_1(x)$, $\mathrm{g}_2(x)$,..., $\mathrm{g}_n(x)$ sobre o domínio do intervalo: $a \leq x \leq b$ se:

$$\int_a^b \mathrm{g}_m(x)\, \mathrm{g}_n(x) dx = 0 \qquad , \qquad m \neq n$$

Muitas funções exibem ortogonalidade incluindo funções trigonométricas. Para resolver esta equação, devemos representar em termo de uma série infinita de funções ortogonais.

$$f(x) = \sum_{n=1}^{\infty} A_n\, \mathrm{g}_n(x)$$

As formas dos coeficientes A_n na série pode ser determinada pela multiplicação de cada lado da equação por $\mathrm{g}_m(x)$, seguida pela integração entre os limites a e b.

$$\int_a^b f(x)\, \mathrm{g}_m(x) dx = \int_a^b \mathrm{g}_m(x) \sum_{n=1}^{\infty} A_n\, \mathrm{g}_n(x)\, dx$$

Percebemos que todos os termos exceto um, no lado direito deve ser nulo, portanto, temos:

$$\int_a^b f(x)\, g_m(x)dx = A_m \int_a^b g_m^2(x)dx$$

Logo explicitando A_m e reconhecendo que o resultado vale para qualquer A_n ao mudar m por n:

$$A_n = \frac{\int_a^b f(x)\, g_n(x)dx}{\int_a^b g_n^2(x)dx}$$

As propriedades das funções podem ser usadas para resolver a equação e encontrar c_n , através da formulação de uma série infinita com a forma apropriada para $f(x)$.

Logo, teremos que:

$$A_n = \frac{\int_0^L \sin\left(\frac{n\pi x}{L}\right)dx}{\int_0^L \sin\left(\frac{n\pi x}{L}\right)dx} = \frac{2}{\pi}\cdot\frac{(-1)^{n+1}+1}{n}$$

Portanto, a partir desta equação, podemos concluir que:

$$c_n = \frac{2}{n\pi}\cdot\frac{(-1)^{n+1}+1}{\sinh\left(n\pi\frac{W}{L}\right)} \quad , \text{ n=1,2,3,...}$$

Tendo conhecimento de c_n, podemos expressar como Solução Final:

Equação 4 - Equação da Solução Final de Transferência de Calor Bidimensional

$$\theta(x,y) = \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^{n+1}+1}{n}\cdot\sin\left(\frac{n\pi x}{L}\right)\cdot\frac{\sinh\left(\frac{n\pi y}{L}\right)}{\sinh\left(\frac{n\pi W}{L}\right)}$$

Aplicando a identidade: $\sinh(2\theta) = 2\cdot\sinh(\theta)\cdot\cosh(\theta)$ e manipulando os termos da equação, temos por fim a equação:

$$T(x,y) = 100 + \sum_{n=1}^{\infty}\frac{600\cdot(-1)^{\frac{n+1}{2}}}{n\cdot\pi}\cdot\frac{1}{\cosh\left(\frac{n\cdot\pi}{2}\right)}$$

Considerando, a temperatura central em: $T(1{,}5;\,1{,}5)$ e atribuindo os valores de n = 1,3,5 e 7 obteremos assim a solução analítica. Para valores maiores que 7, o termo não apresentará contribuição significativa para os cálculos, pois serão valores muito pequenos. A tabela 1, apresenta a contribuição dos termos de n=1,3,5 e 7 calculados:

Tabela 1 - Tabela de cálculo da série de Fourier para Solução Analítica

n	$\frac{(n-1)}{2}$	$(-1)^{\frac{(n\cdot\pi)}{2}}$	$\frac{(n\cdot\pi)}{2}$	$cosh\left(\frac{n\cdot\pi}{2}\right)$	$\frac{600}{n\cdot\pi}$	*Contribuição do Termo*
1	0	1	1,5708	2,5092	190,9859	76,1149
3	1	−1	4,7124	55,6634	63,6620	−1,1437
5	2	1	7,8540	1287,9854	38,1972	−0,0297
7	3	−1	10,9956	29804,8708	27,2837	−0,0009

Fonte: Autor (2025)

Obs.: O desenvolvimento em série de Fourier utilizado neste trabalho para a representação da contribuição para a propagação de calor é truncado no harmônico de ordem n = 7. Essa escolha se justifica pela rápida convergência da série observada para o caso em estudo. Análises detalhadas do comportamento dos coeficientes de Fourier revelam que, a partir de n = 7, o módulo dos termos subsequentes n ≥ 9 decai abruptamente, tornando-se desprezível em relação aos harmônicos de baixa ordem. Especificamente, os coeficientes para n ≥ 9 contribuem com uma alteração na magnitude do erro de truncamento inferior a 0,00092 de pico da função. Portanto, a inclusão de termos de ordem maior que 7 não resulta em ganho significativo na

precisão da solução, mas, por outro lado, eleva desnecessariamente a complexidade computacional do modelo. A retenção dos harmônicos até n=7 assegura um balanço ótimo entre a fidelidade da representação analítica de propagação de calor.

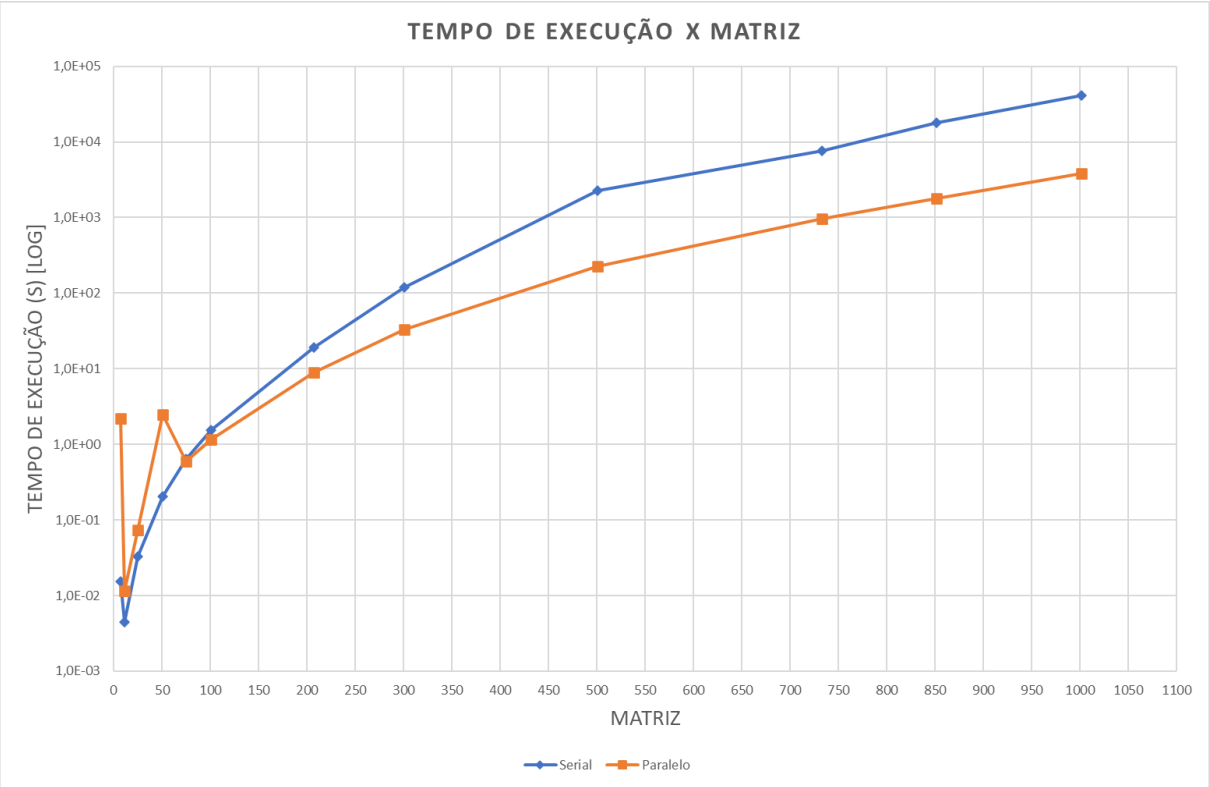
O total do somatório das contribuições é de: 74,99997028, temos então que o valor das contribuições somados com 100 graus celsius será a solução analítica. Logo temos o seguinte resultado:

$$T(1{,}5\,;1{,}5) \;=100\;+\;74{,}99997028 \cong \mathbf{174,99997}\,º\,\boldsymbol{C}$$

Para encontramos a solução numérica através de um algoritmo para estudo deste caso, utilizamos o ambiente de desenvolvimento *Pycharm* com a linguagem de programação em *Python*. Como o algoritmo representa as iterações que ocorre e as temperaturas são calculadas pelo esquema explícito, ou seja, dependendo da temperatura anterior. Determinamos o seguinte critério de parada: o valor do erro que será: $\varepsilon = T_{m,n}^{k^*+1} - T_{m,n}^{k^*}$, devendo obedecer a tolerância de: ($\varepsilon \leq 10^{-08}$). Esta tolerância do erro foi determinada exclusivamente pensando nas matrizes maiores, pois para as matrizes de maiores dimensão não ocorre a convergência para o regime estacionário se utilizarmos um $\varepsilon = 10^{-4}$, por exemplo. Quando imaginamos uma matriz 1001x1001 que terão aproximadamente 1002001 de incógnitas, a taxa de convergência é baixíssima, portanto, é necessário reduzir o valor do erro, que será o critério de parada do algoritmo para encontrarmos as temperaturas quando a placa atingir o equilíbrio.

No apêndice A, temos os dois códigos computacionais, sendo o primeiro da programação serial e o segundo da programação em paralelo para resolução numérica da propagação de calor na placa bidimensional, conforme problema sugerido neste TCC.

Figura 10 - Gráfico do Tempo de Execução x Matriz



Fonte: Autor(2025)

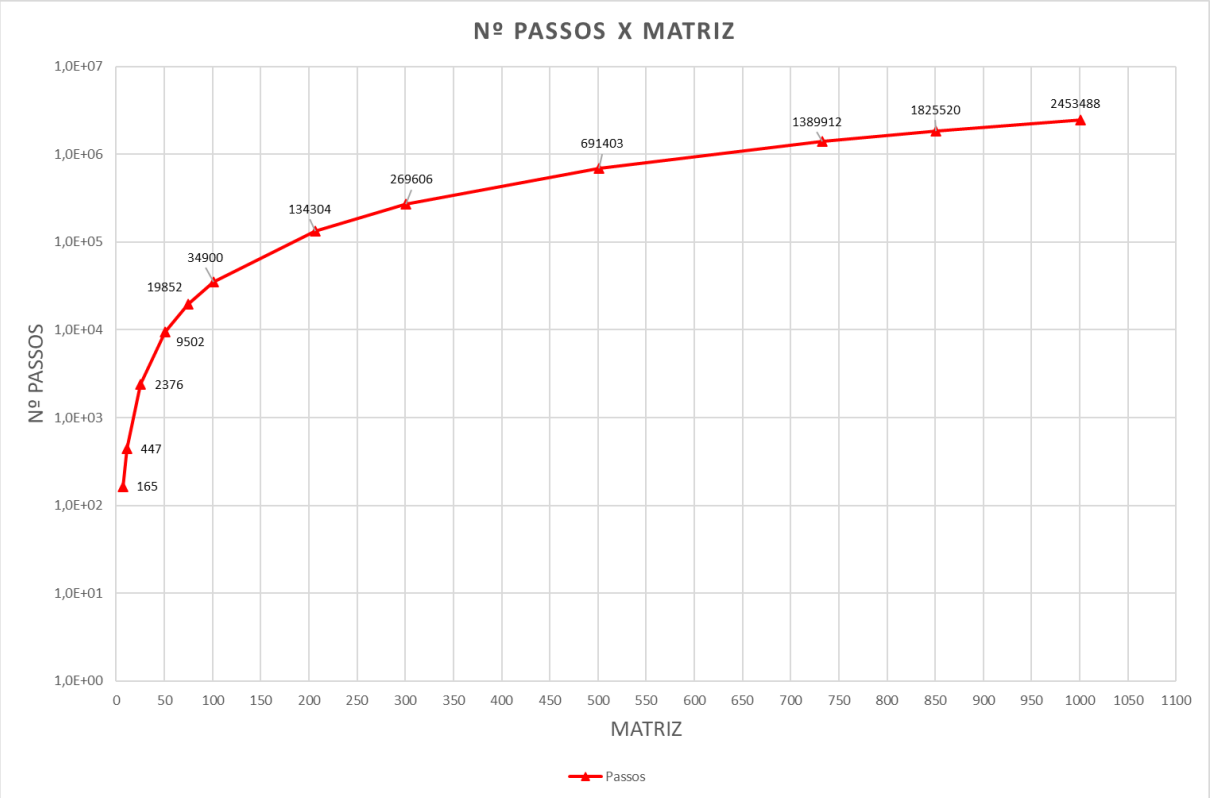
No Gráfico 01, são apresentados os tempos de execução em função do tamanho das amostras das matrizes utilizadas das simulações realizadas. Observa-se que no eixo vertical (y) utiliza escala logarítmica para apresentar os tempos de processamento das abordagens serial e paralela, em função do tamanho das dimensões das matrizes do eixo horizontal (x). A escala logarítmica é crucial, que permite uma visualização da diferença da taxa de crescimento das abordagens. Nota-se que no ponto de dimensão das matrizes: 75x75, os tempos de execução das abordagens serial e paralela são equivalentes. Este ponto demarca a transição na eficiência das metodologias das programações serial e paralela estudadas neste TCC.

Para matrizes de menor dimensão — especificamente 7x7, 11x11, 25x25 e 51x51 — a execução serial demonstrou desempenho superior, com tempos de processamento mais reduzidos, isto é justificado pelo overhead inicial (tempo gasto na criação, gerenciamento e sincronização de threads ou processos paralelos) é maior que o tempo poupado pela divisão do trabalho, tornando a programação serial mais rápida para problemas de baixa escala. Por outro lado, para matrizes de maior dimensão - especificamente 101x101, 207x207, 301x301, 501x501, 733x733, 851x851 e 1001x1001, a abordagem paralela, apesar de envolver maior escalabilidade e número de etapas, apresentou tempos de execução menores aos da abordagem

serial, evidenciando sua eficiência em cenários de maior escalabilidade computacional.

Para matrizes de maior dimensão (a partir de 101x101 até 1001x1001), a abordagem paralela demonstra sua superioridade em termos de escalabilidade, com uma taxa de crescimento de tempo significativamente menor que a serial. Temos dois conceitos importantes notados: 1) Matrizes de Maior Dimensão: Onde o volume de dados e a complexidade computacional são altos, o tempo poupado pelo paralelismo supera amplamente o overhead inicial. 2) Ganho de Ordem de Magnitude: Para a maior matriz simulada (1001x1001), o tempo de execução serial é drasticamente maior. O tempo serial atinge cerca de 3×10^4 segundos, enquanto o tempo paralelo se mantém em cerca de 3×10^3 segundos. Esta diferença representa um ganho de aproximadamente uma ordem de magnitude (cerca de 10 vezes) no desempenho do algoritmo paralelo, confirmando sua eficiência em cenários de alta complexidade.

Figura 11 - Gráfico do Número de Passos x Matriz



Fonte: Autor (2025)

Lembrando que cada interação da matriz é contabilizada por número de passos até encontrarmos uma temperatura central de equilíbrio. A Temperatura de equilíbrio é atingida

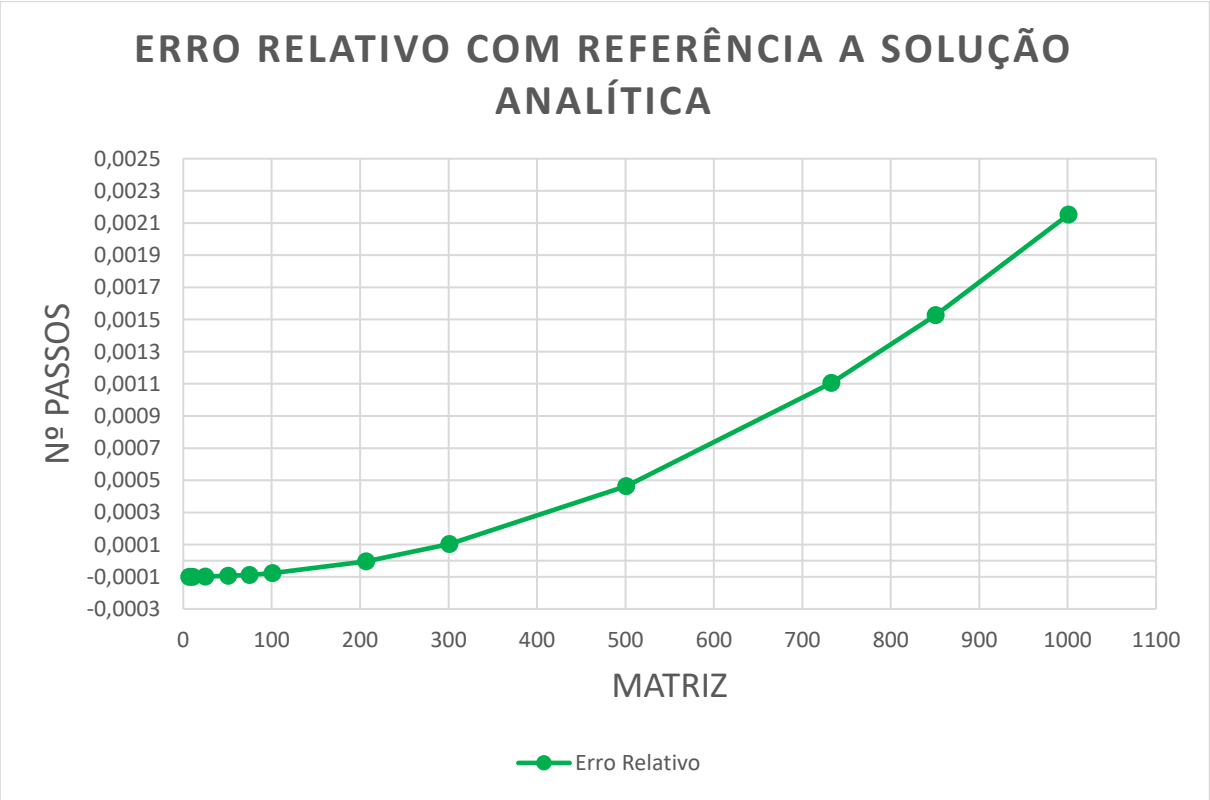
quando existe uma diferença entre a Temperatura do passo anterior e a temperatura do novo passo sendo igual ou menor que o resultado da diferença do erro de: 1,0E-08. Quanto maior a dimensão da matriz, maior serão os números de passos até que a placa bidimensional atinja a estabilidade ou pontos de equilíbrio.

O gráfico apresentado ilustra a relação entre o Número de Passos de um algoritmo interativo e o Tamanho da Matriz (NxN) processada, com o eixo vertical (Número de Passos) em escala logarítmica com a finalidade de entender a complexidade computacional e a escalabilidade do método à medida que a dimensão do problema aumenta.

A curva vermelha, que representa o número de passos, demonstra um crescimento contínuo e progressivo em função do tamanho da matriz, devido à escala logarítmica, a inclinação da curva sugere um comportamento sub-exponencial ou polinomial de baixa ordem. Isso indica que, embora o número de operações cresça com o tamanho da matriz, a taxa de crescimento se torna menos acentuada à medida que o tamanho da dimensão da matriz aumenta.

Ao analisar os pontos de dados, podemos citar três pontos importantes: 1) Baixa Complexidade Inicial: Para matrizes pequenas, o crescimento é rápido em termos de magnitude absoluta, mas ainda gerencialmente baixo. Por exemplo, uma matriz de 50x50 exige 34.900 passos, 2) Crescimento Moderado: O aumento do tamanho da matriz de 50x50 para 200x200 (aumento de 4 vezes em N) resulta em um crescimento de cerca de quatro vezes no número de passos, atingindo 134.304 e 3) Tendência à Estabilização (na escala logarítmica): Ao observar os pontos de dados para grandes matrizes, a taxa de crescimento aparente diminui. Passar de uma matriz de 700x700 (1.389.912 passos) para 1000x1000 (2.453.488 passos) representa um aumento de apenas 76% no número de passos para um aumento de cerca de 43% em N.

Figura 12 - Gráfico do Erro relativo com referência a Solução Analítica



Fonte: Do Autor

O gráfico acima apresenta o erro relativo (em relação a uma solução analítica de referência) em função do tamanho da matriz NxN utilizada na simulação numérica. O eixo vertical (Erro Relativo) está em escala linear, variando de -0,0003 a 0,0025, o que já indica que o erro é extremamente pequeno na ordem de 10^{-3} a 10^{-4}.

O padrão de crescimento do erro com base na curva revela um comportamento estável. Nota-se que para matrizes pequenas (até $N \approx 200$) temos que o erro relativo é praticamente nulo ou, em alguns pontos, levemente negativo, permanecendo abaixo de 0,0001. Isso sugere uma concordância muito alta entre a solução numérica e a solução analítica para problemas de menor dimensão.

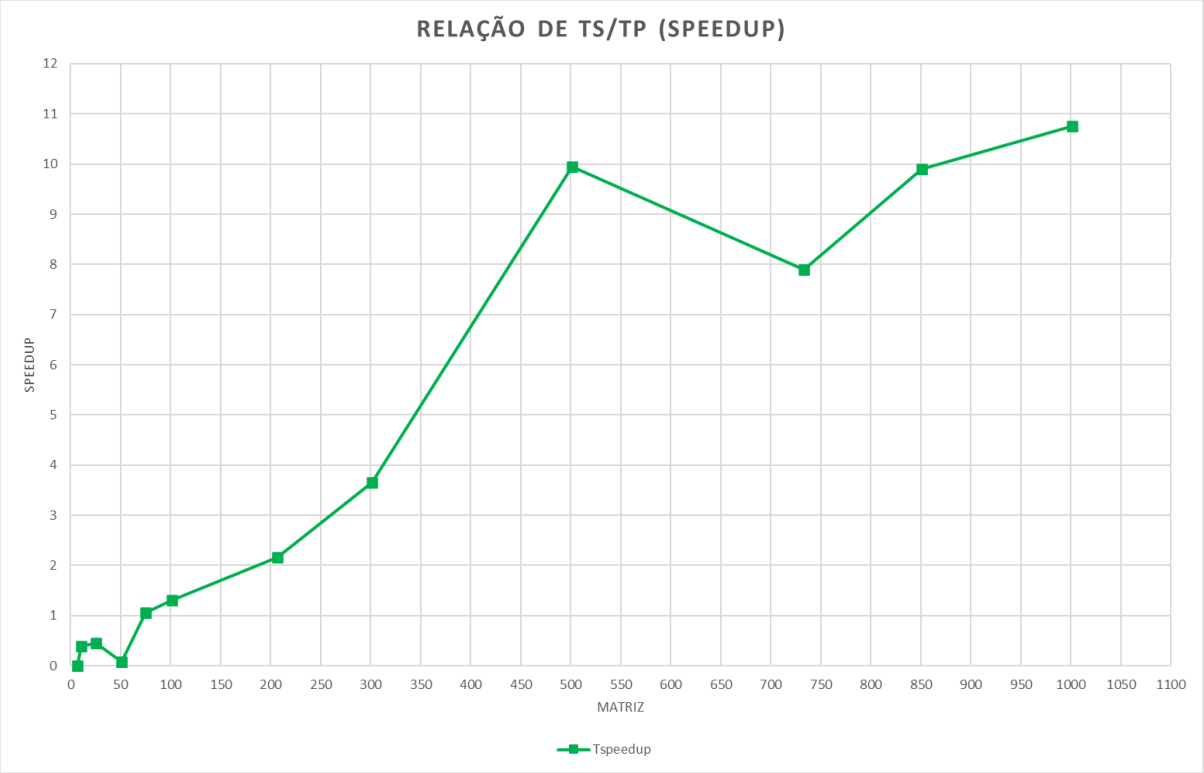
Para matrizes grandes (de $N \approx 200$ a $N = 1001$), observa-se a que partir de $N = 200$, o erro relativo começa a crescer de forma monotônica e acelerada. O erro atinge seu valor máximo de aproximadamente 0,0022 para a maior matriz testada $N = 1001$. O crescimento da curva é quase quadrático ou exponencialmente lento em N, indicando que o erro aumenta

substancialmente à medida que a dimensão do problema cresce.

Notamos que o programa durante sua execução inclui, armazena e soma diversos resultados com tolerância de 10^{-8}. Quanto maior a dimensão das matrizes, o erro aumenta, devido ao truncamento de valores e acúmulos dos erros. O aumento do erro com o tamanho da matriz é um comportamento típico em métodos interativos. Em matrizes maiores, o número de passos e, consequentemente, o número de operações de ponto flutuante, é muito maior (conforme visto no gráfico anterior de "Nº Passos X Matriz"). Cada operação introduz um pequeno erro de arredondamento devido à precisão finita dos computadores. Em grandes simulações, esses pequenos erros se acumulam ao longo dos milhões de passos, resultando no erro relativo final observado.

Embora haja uma perda de precisão nas casas decimais, a diferença final é muito pequena e aceitável para a maioria das aplicações de engenharia, indicando que o algoritmo numérico mantém uma alta fidelidade em relação à solução analítica.

Figura 13 - Relação entre Tempo de Execução Serial sobre Tempo de Execução em Paralelo



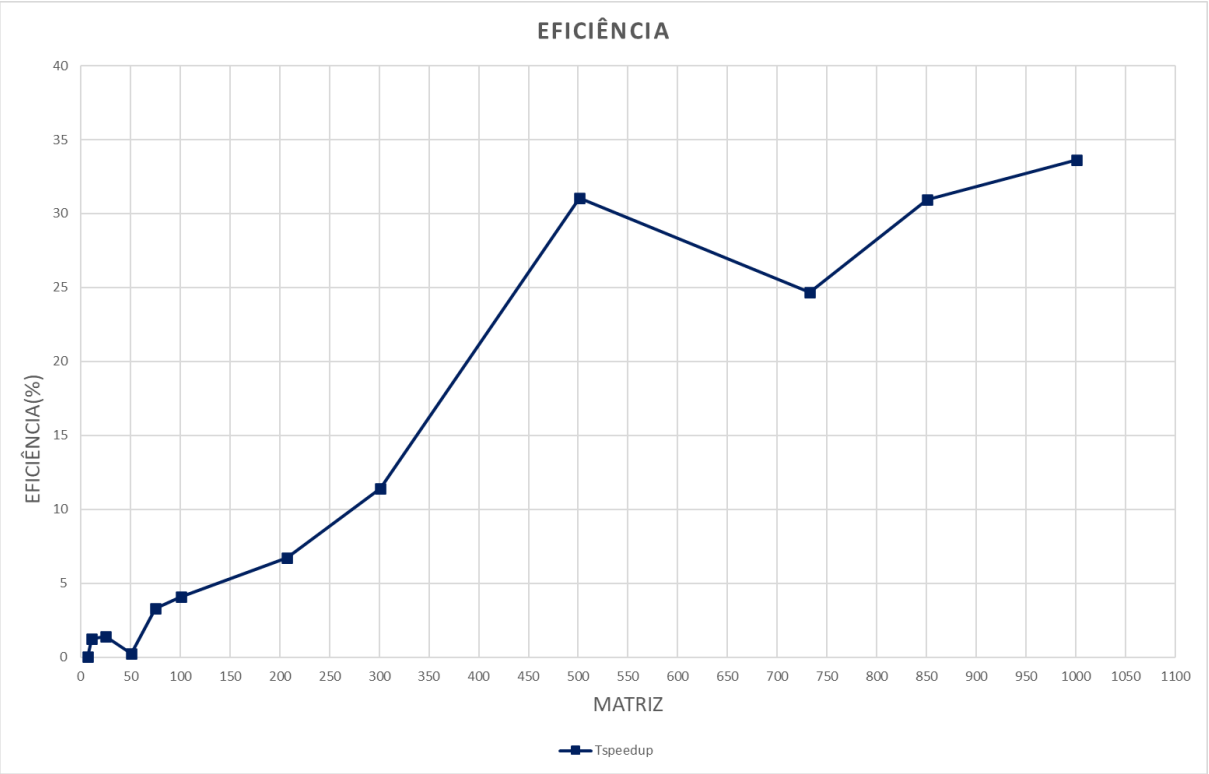
Fonte: Do Autor

O *Speedup* é definido como a razão entre o tempo de execução Serial (T_s) e o tempo de execução Paralelo (T_p), em função do Tamanho da Matriz (NxN). O *Speedup* é um meio de avaliar a eficiência da paralelização: um valor de *Speedup* maior que 1 indica que a abordagem paralela é mais rápida que a serial, enquanto um valor igual a 1 indica equivalência.

Para matrizes de dimensão muito pequena, o desempenho da paralelização é modesto e ineficiente, pois o *Speedup* é baixo, oscilando em torno de 0,5 a 1,5. Em alguns pontos, o *Speedup* é inferior a 1, confirmando que o *overhead* (custo de inicialização e comunicação) da abordagem paralela supera o ganho de velocidade para problemas com pouca carga computacional. Isso está em total acordo com o ponto de crossover observado no gráfico de Tempo de Execução.

A medida que o tamanho da matriz aumenta, o *Speedup* cresce de forma acentuada e quase linear, demonstrando a eficácia da paralelização em problemas mais complexos. Para $N \approx 300$, o *Speedup* atinge aproximadamente 3,6, o que significa que o tempo paralelo é cerca de 3,6 vezes mais rápido que o serial. O pico de eficiência neste intervalo é atingido em $N \approx 500$, onde o *Speedup* alcança impressionantes 10. Isso representa uma redução de tempo de 90% em comparação com a execução serial. Há uma queda notável no *Speedup* para $N \approx 700$, caindo para cerca de 8. Essa queda pode ser atribuída a fatores como contenção de recursos, thrashing de memória ou ineficiências de sincronização que se manifestam em cargas de trabalho específicas. No entanto, o *Speedup* se recupera nas maiores matrizes. Para $N \approx 850$, ele volta a 10, e o pico máximo é atingido na matriz de $N \approx 1001$, com um *Speedup* de aproximadamente de 10,8 vezes, sendo a relação entre os tempos de execução entre a programação Serial e a programação em paralelo.

Figura 14 - Gráfico da Eficiência do Processador utilizando 32 núcleos



Fonte: Do Autor

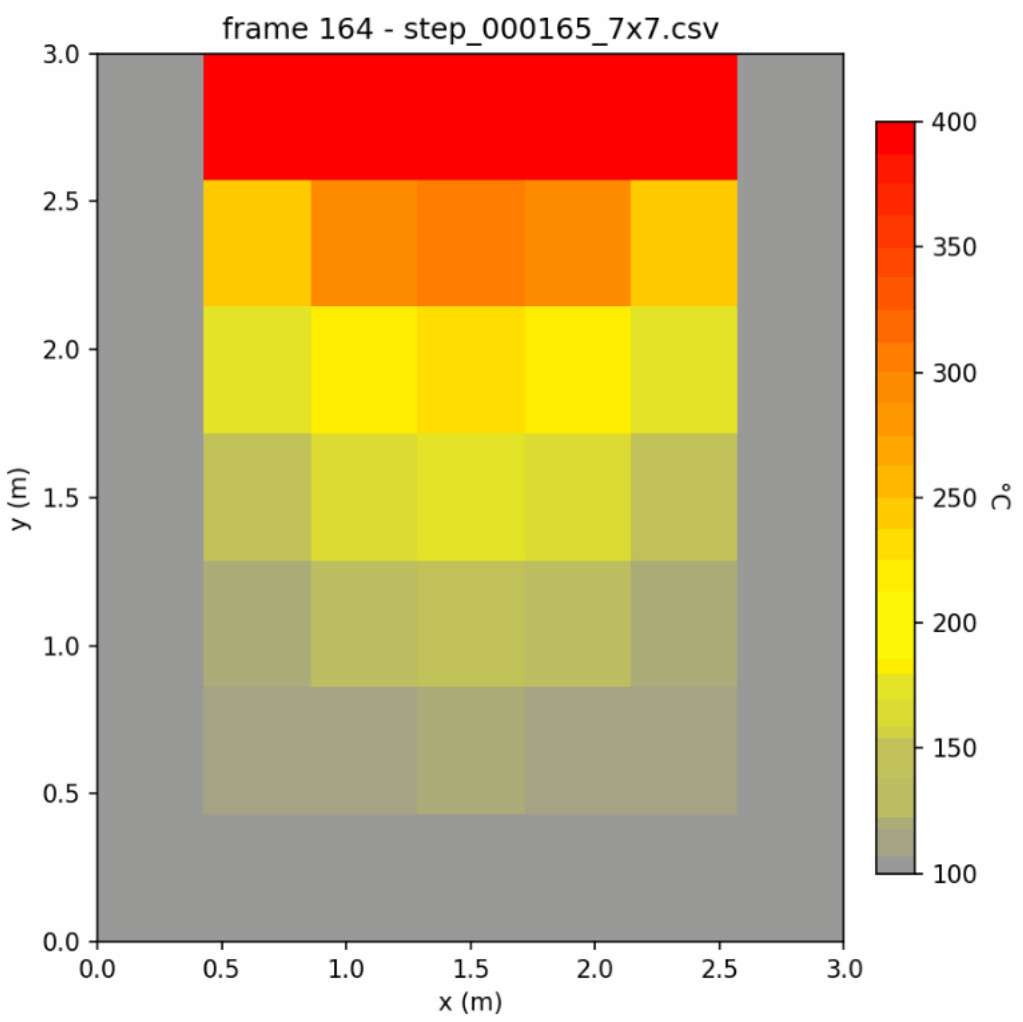
Para matrizes pequenas (até $N < 75$), a eficiência é muito baixa, indicando um valor entre de 1% a 3%. Este valor confirma que o custo de overhead (tempo gasto na comunicação, sincronização e inicialização dos processos paralelos) é muito alto em comparação com o tempo de computação útil. O algoritmo paralelo está utilizando a maior parte do tempo para gerenciar a paralelização, e não para resolver o problema.

O ganho acelerado da eficiência ocorre quando a dimensão da matriz é de $N \approx 75$, a eficiência cresce rapidamente à medida que o tamanho da matriz e o aumento de passos devido as iterações aumentam. A carga de trabalho faz com que o tempo de computação útil (que pode ser paralelizado) cresça mais rapidamente do que o tempo de overhead. O pico de eficiência é atingido em $N \approx 500$, onde a curva alcança aproximadamente 31%. Isso sugere que, neste ponto, a divisão do trabalho entre os processadores está sendo otimizada em relação ao overhead inerente ao sistema. Nota-se que o comportamento para grandes matrizes e instabilidade para matrizes maiores, o comportamento da eficiência mostra instabilidade: Há uma queda notável em $N \approx 750$, onde a eficiência cai para cerca de 25%. Essa redução pode

ser causada por novos fatores limitantes que surgem com o aumento da escala, como conflitos de acesso à memória (contenção) ou o aumento da necessidade de sincronização entre os núcleos, que limitam o desempenho. Apesar da queda, a eficiência se recupera e atinge seu valor máximo final de cerca de 34% para a matriz de N=1001.

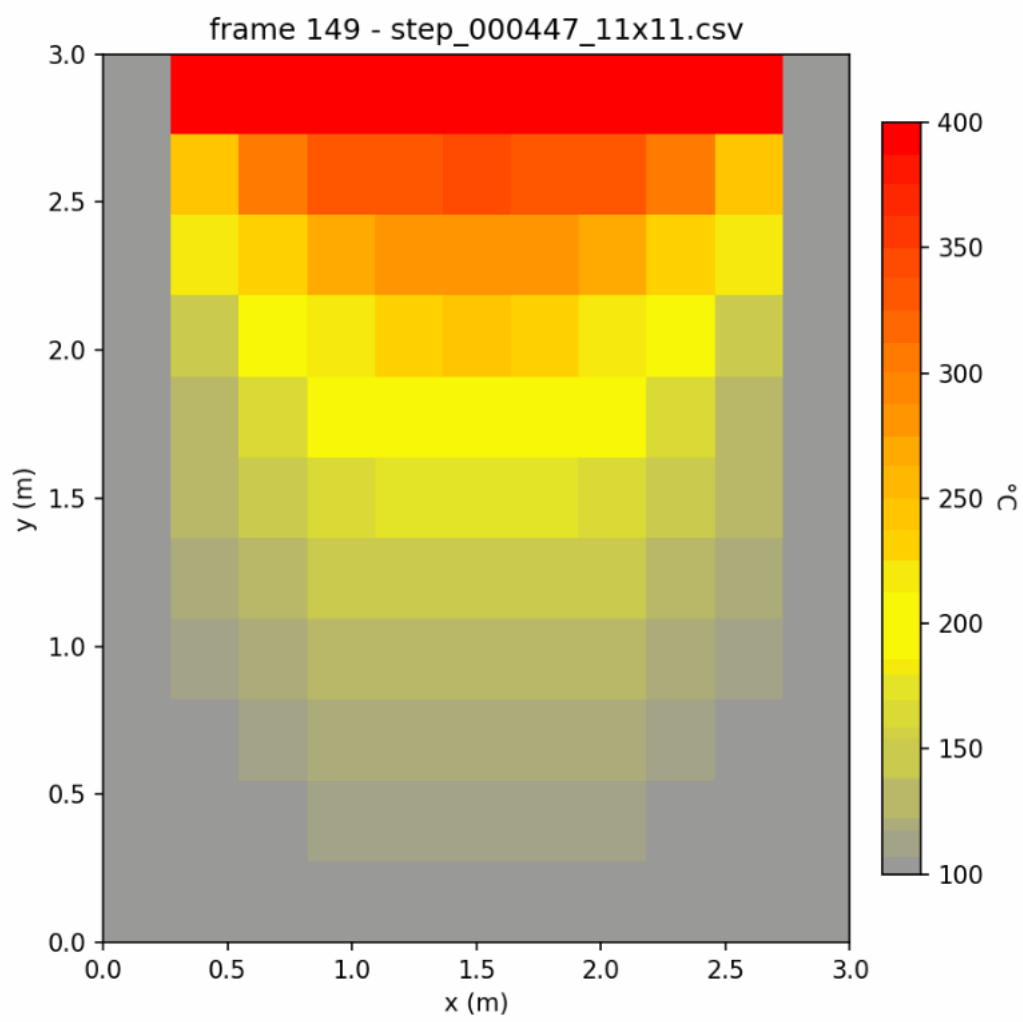
Para simularmos a propagação de calor, utilizamos um terceiro algoritmo que importa os resultados das temperaturas conforme os passos em planilhas com extensão .csv em pastas direcionadas com o nome do tamanho das matrizes. Os arquivos .csv serão utilizados no quarto algoritmo que busca as temperaturas conforme as matrizes e plota o gradiente de temperaturas, utilizando a biblioteca *Matplotlib*, e transforma as imagens em gifs para cada tamanho das matrizes selecionada. As matrizes de temperatura de acordo com as dimensões estabelecidas, no seu passo final, ou seja, ao atingir o equilíbrio são demonstradas a seguir:

Figura 15 - Matriz Temperatura de dimensão: 7x7



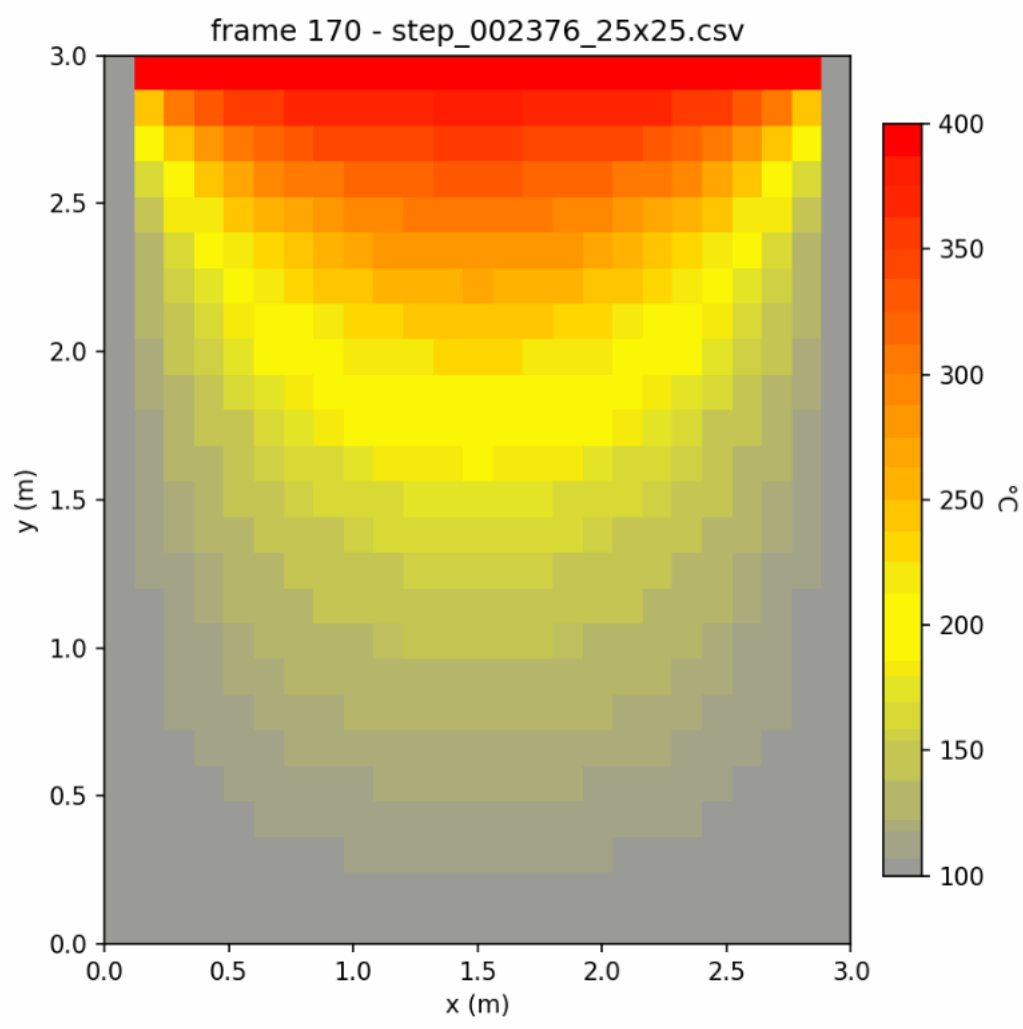
Fonte: Autor (2025)

Figura 16 - Matriz Temperatura de dimensão: 11x11



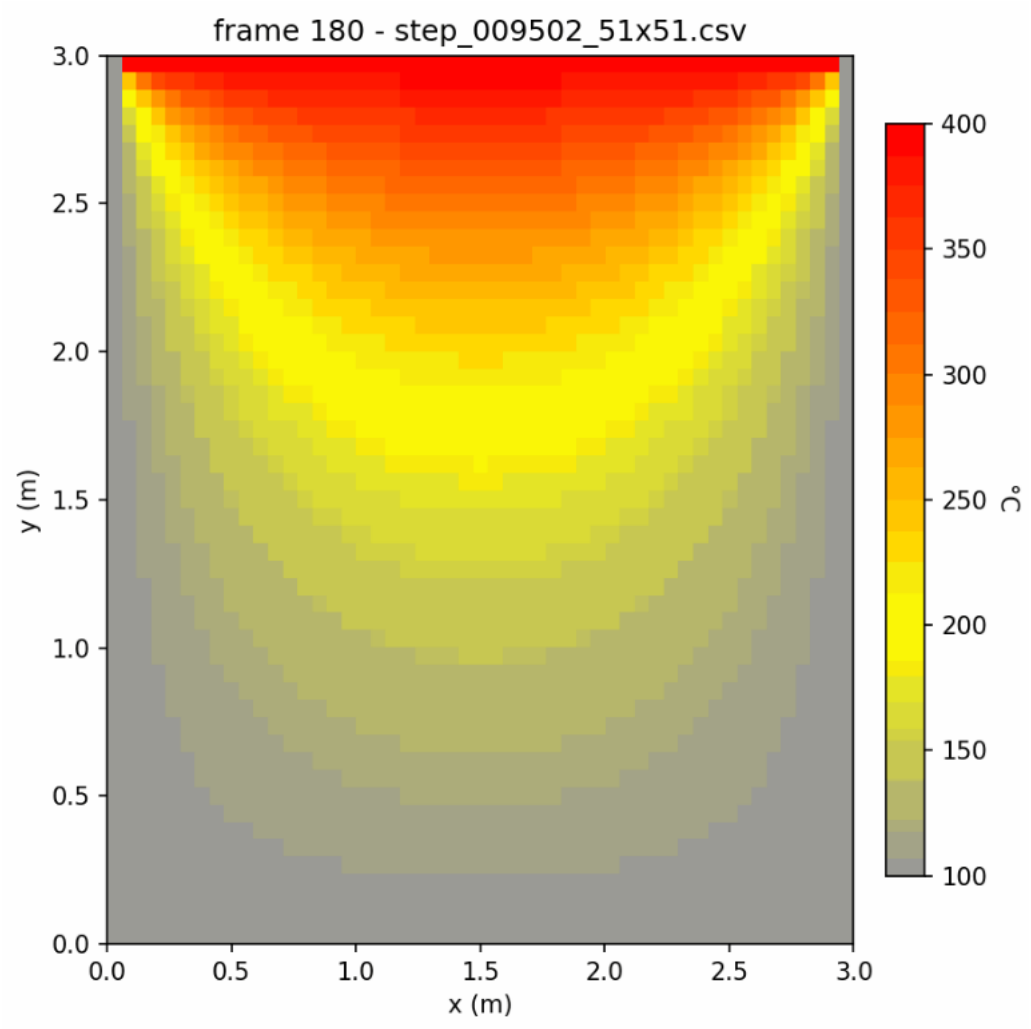
Fonte: Autor (2025)

Figura 17 - Matriz Temperatura de dimensão: 25x25



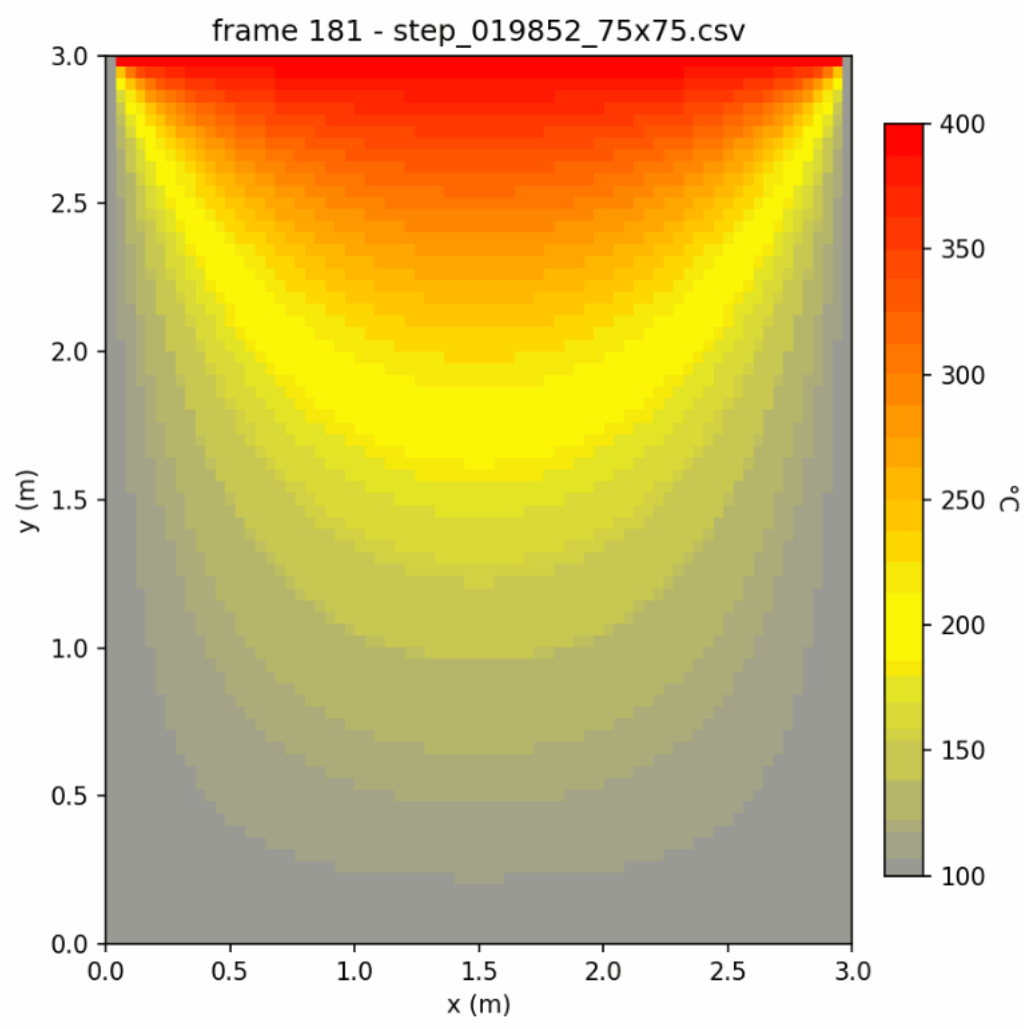
Fonte: Autor (2025)

Figura 18 - Matriz Temperatura de dimensão: 51x51



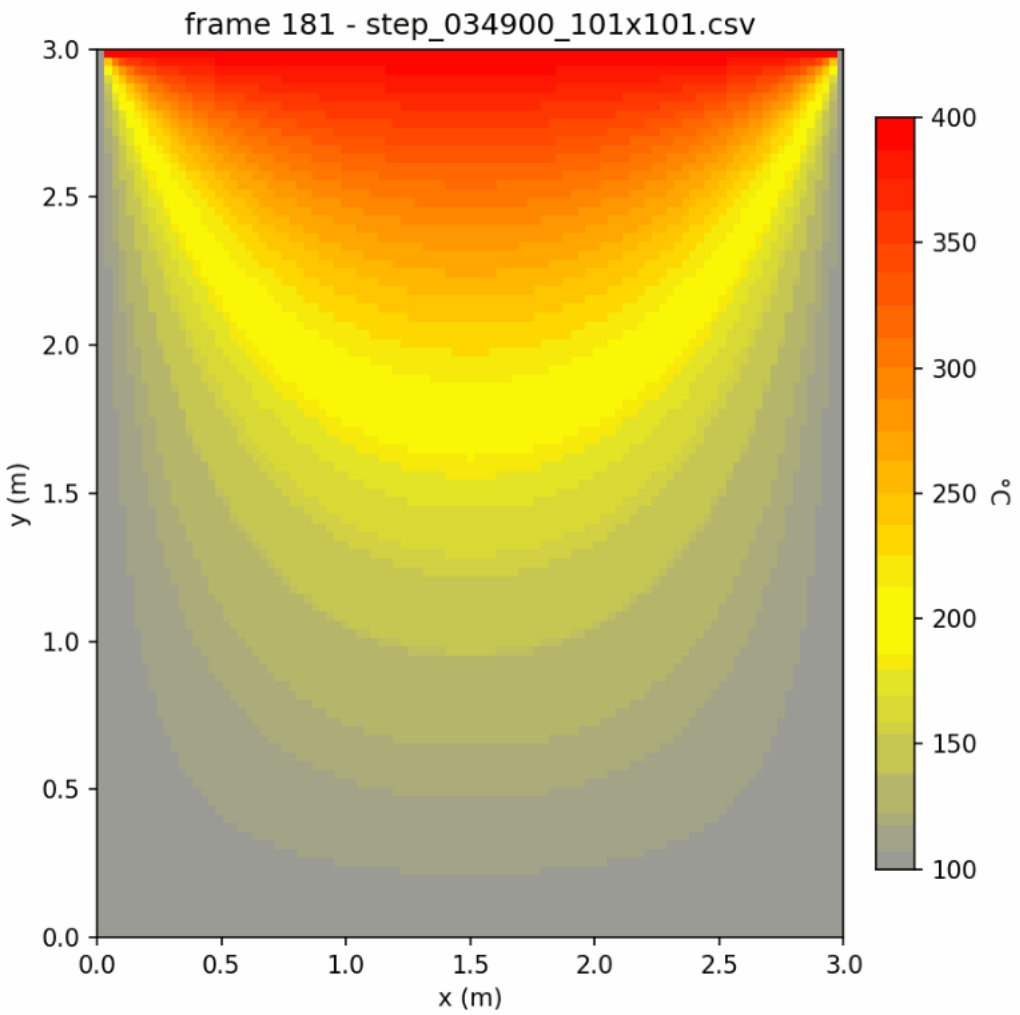
Fonte: Autor (2025)

Figura 19 - Matriz Temperatura de dimensão: 75x75



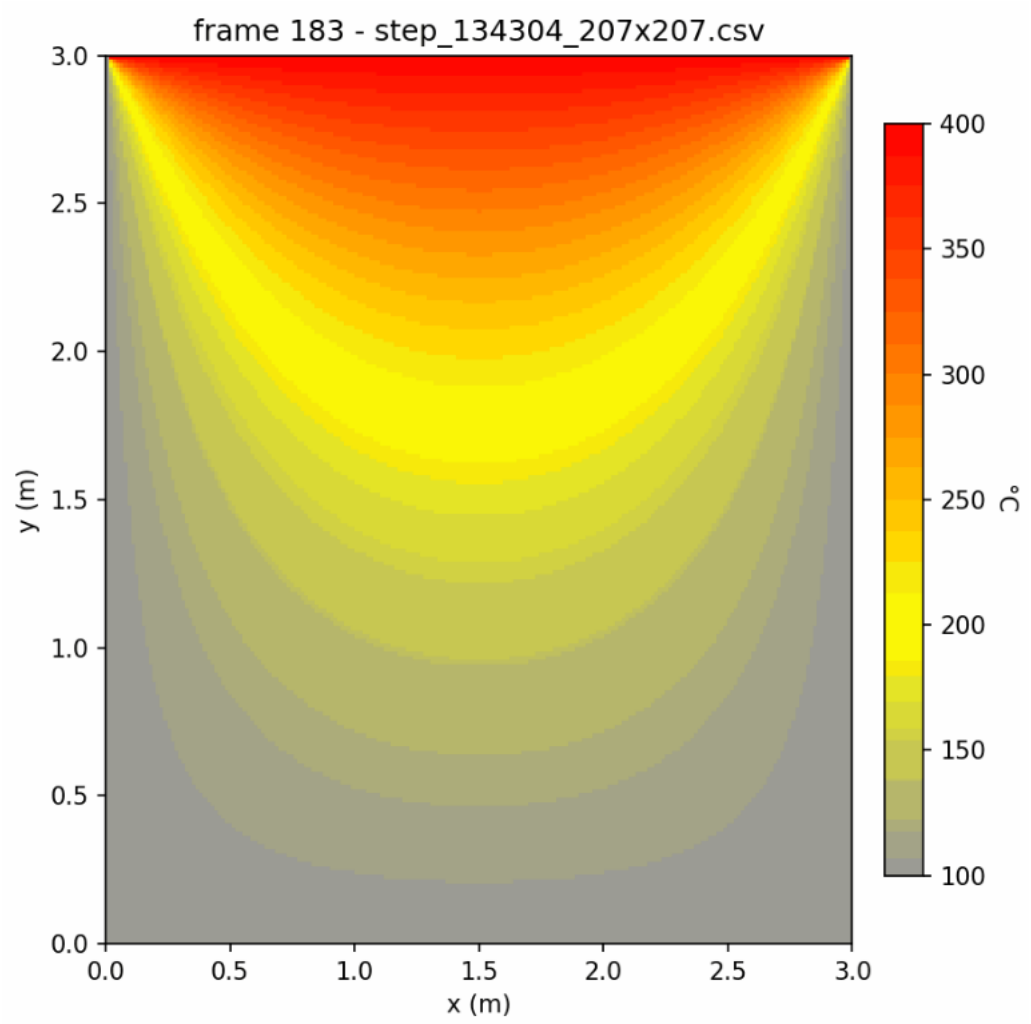
Fonte: Autor (2025)

Figura 20 - Matriz Temperatura de dimensão: 101x101



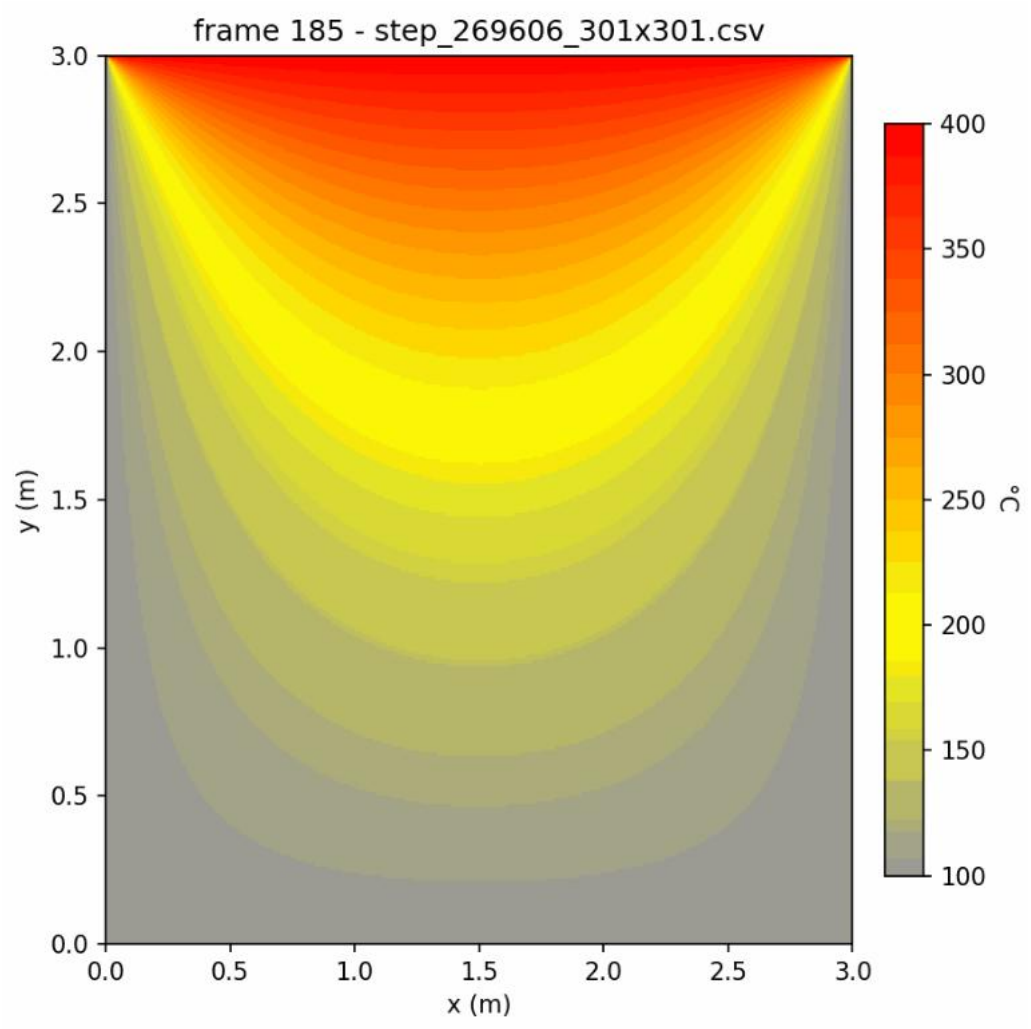
Fonte: Autor (2025)

Figura 21 - Matriz Temperatura de dimensão: 207x207



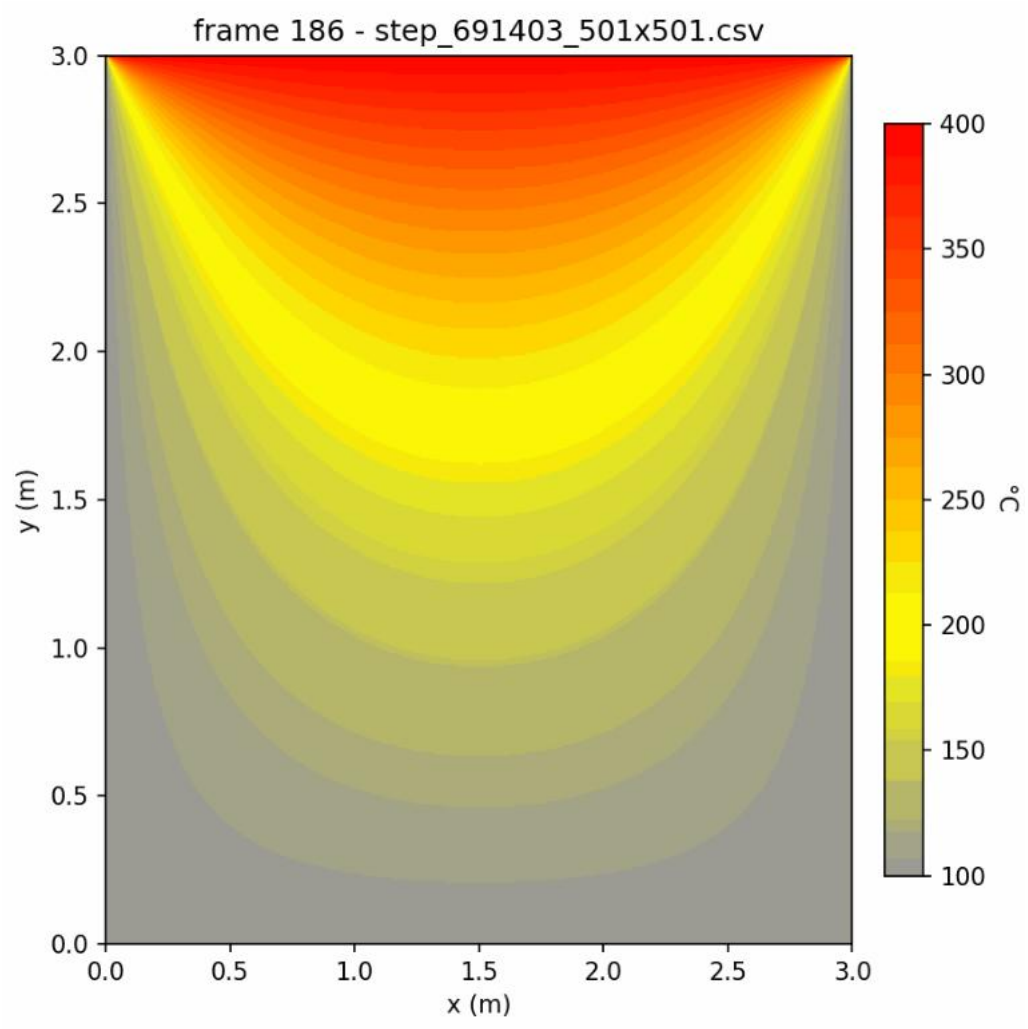
Fonte: Autor (2025)

Figura 22 - Matriz Temperatura de dimensão: 301x301



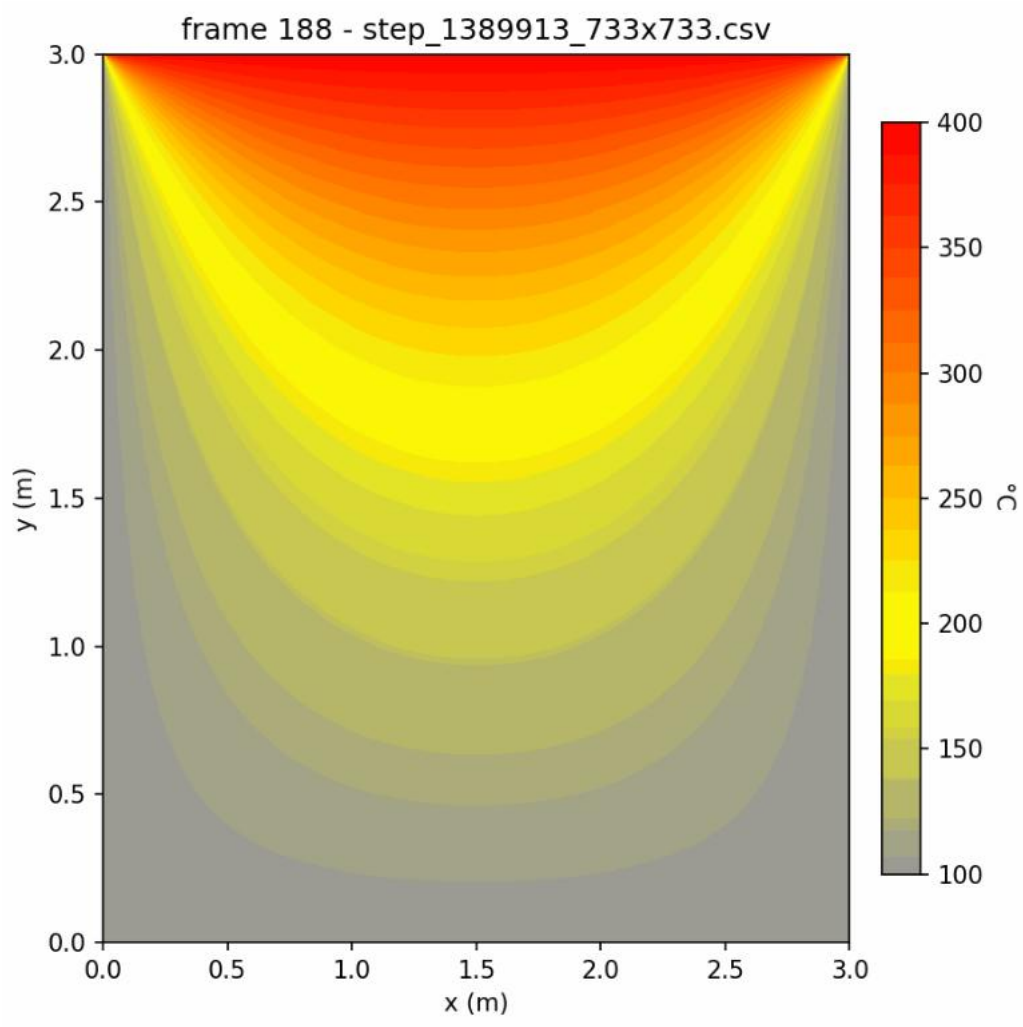
Fonte: Autor (2025)

Figura 23 - Matriz Temperatura de dimensão: 501x501



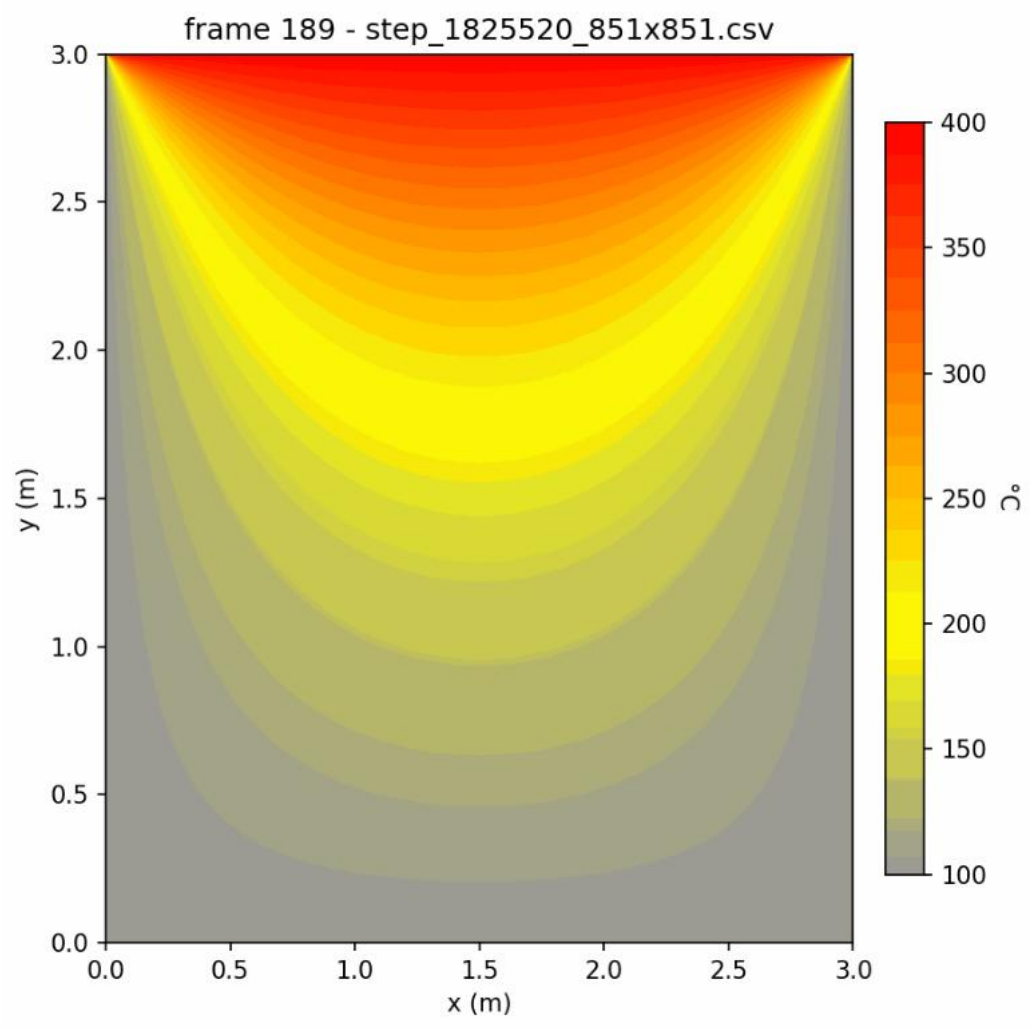
Fonte: Autor (2025)

Figura 24 - Matriz Temperatura de dimensão: 733x733



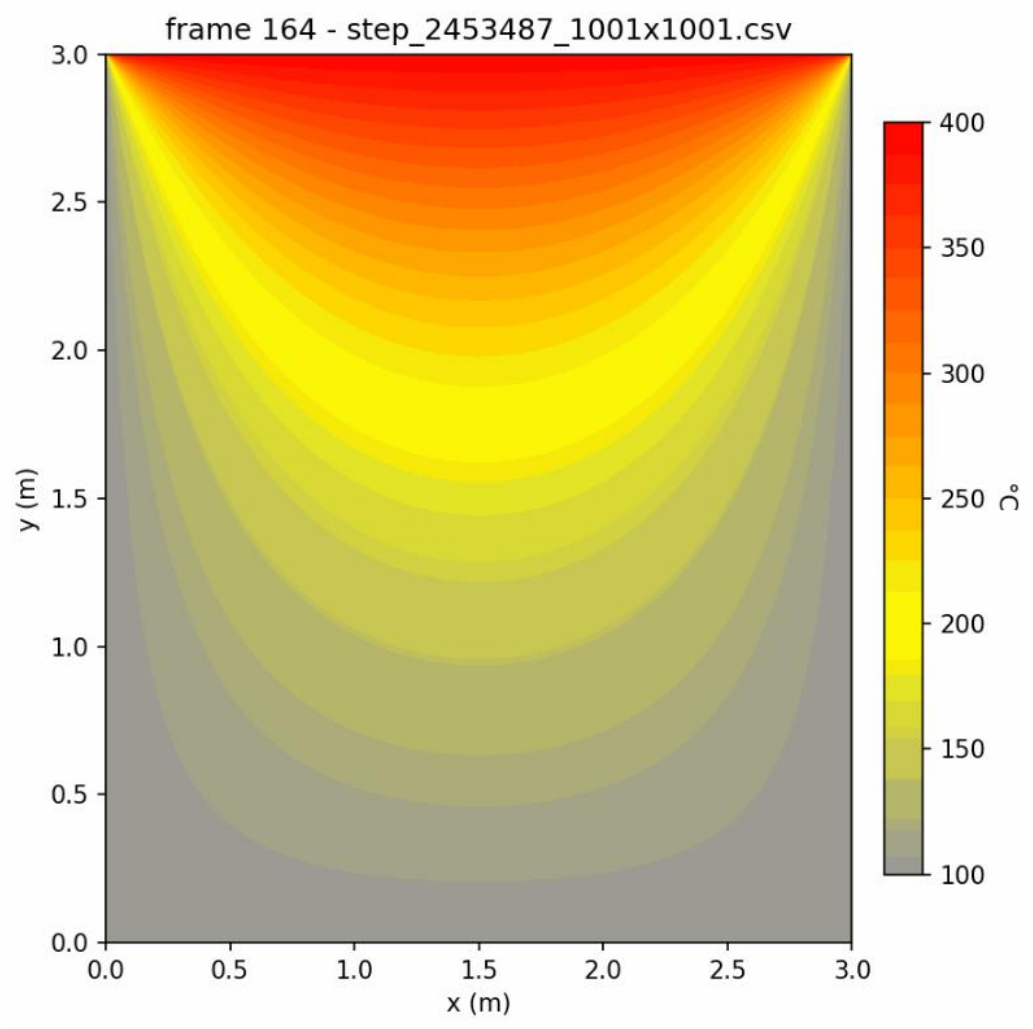
Fonte: Autor (2025)

Figura 25 - Matriz Temperatura de dimensão: 851x851



Fonte: Autor (2025)

Figura 26 - Matriz Temperatura de dimensão: 1001x1001



Fonte: Autor (2025)

5 CONCLUSÃO

O estudo demonstrou relevância sobre um estudo analítico e prático sobre a transferência de calor na placa bidimensional. Percebemos que a proposta deste trabalho de investigar com fundamentação teórica, realizar a modelagem de uma placa com condições iniciais e de contorno definidas, analisando e comparando os resultados obtidos das matrizes.

Percebemos que a solução obtida através do método analítico é relevante como referência, o seguinte passo foi estruturar códigos para obter o tempo de execução em serial e em paralelo para obter gráficos e realizar comparações ao longo deste estudo. O comportamento térmico sob condução bidimensional a partir do método das diferenças finitas mostrou-se eficaz nas simulações do problema, obtendo um erro de 0,002 º C em matrizes de dimensão 1001x1001, sendo aceitável em engenharia e a implementação paralela proporcionou ganhos significativos de desempenho a partir de matrizes superiores as dimensões 75x75.

Nota-se que o trabalho contribui para o entendimento da transferência de calor e pode ser expandido para simulações de outras geometrias e materiais e outros fatores como diferentes temperaturas e matrizes de maiores dimensões. A programação paralela indicou que tivemos em alguns pontos a otimização do tempo e em questão a eficiência os resultados apresentados foram: para matriz de dimensão: 75x75 encontramos o resultado da eficiência de 3,32%, para matriz 301x301 de 11,43%, para matriz 501x501 de 31,07% e para matriz 1001x1001 de 33,62%.

Concluímos com este trabalho que para programas que simulam muitas iterações tem-se vantagem em utilizar a programação em paralelo devido a duas vantagens principais: 1) Resultados Válidos: com os resultados da solução numérica percebemos que existe coerência e pequena diferença entre os valores do resultados da solução analítica em comparação com a solução numérica, ou seja, conseguimos demonstrar e validar o método aplicado e 2) Otimização de tempo: o tempo de execução ser bem menor do que a programação serial, demonstrou que a validade dos procedimentos aplicados.

A partir do trabalho realizado, é possível fazer recomendações para trabalhos futuros, como estudo em novas geometrias como um cubo tridimensional, com aplicação de outro método numérico como elementos finitos ou volumes finitos.

6 REFERÊNCIAS

ATKINSON, Kendall E. ..; HAN, Weimin. **Elementary numerical analysis**. *[S.l.]*: John Wiley & Sons, 2004.

BELL, Gordon; GRAY, Jim. What's next in high-performance computing? **Communications of the ACM**, v. 45, n. 2, p. 91–95, fev. 2002.

BERGMAN, T. L. ..; LEVINE, Adrienne S. .. **Fundamentals of heat and mass transfer**. *[S.l.]*: John Wiley & Sons, Inc., 2019.

ÇENGEL, Yunus A. ..; GHAJAR, Afshin J. .. **Heat and mass transfer : fundamentals & applications**. *[S.l.]*: McGraw-Hill, 2011.

CHAPRA, Steven C. ..; CANALE, Raymond P. .. **Numerical methods for engineers**. *[S.l.]*: McGraw-Hill Higher Education, 2006.

CRONIN, Matthew A.; GEORGE, Elizabeth. The Why and How of the Integrative Review. **Organizational Research Methods**, v. 26, n. 1, p. 168–192, 6 jan. 2023.

DE BRITO, Leandro W. *et al.* **Proceeding Series of the Brazilian Society of Computational and Applied Mathematics Aplicação do Método das Diferenças Finitas Implícito na Obtenção da Distribuição de Temperatura em Placa Plana**. Toledo, PR: *[S.n.]*. Disponível em: <https://github.com/LeandroWrzecionek/2dHeatEquation/blob/main/MDF%20impl>.

GEBALI, Fayez. **Algorithms and parallel computing**. *[S.l.]*: Wiley, 2011.

GREGÓRIO, João *et al.* The role of Design Science Research Methodology in developing pharmacy eHealth services. **Research in Social and Administrative Pharmacy**, v. 17, n. 12, p. 2089–2096, dez. 2021.

HARRIS, Charles R. *et al.* Array programming with NumPy. **Nature**, v. 585, n. 7825, p. 357–362, 17 set. 2020.

JIN, Haoqiang *et al.* High performance computing using MPI and OpenMP on multi-core parallel systems. **Parallel Computing**, v. 37, n. 9, p. 562–575, set. 2011.

KISABO ALIYU, Bhar; FESTUS OLATOYINBO, Seyi. Explicit and Implicit Solutions to 2-D Heat Equation. 16 fev. 2021.

KOTHARI, C. R.; GARG, G. (2019). **Research methodology : methods and techniques**. *[S.l.]*: New Age International (P) Limited, Publishers, 2020.

LAM, Siu Kwan; PITROU, Antoine; SEIBERT, Stanley. Numba. *In*: New York, NY, USA: ACM, 15 nov. 2015.

MAJUMDAR, Pradip. **Computational Methods for Heat and Mass Transfer**. *[S.l.]*: CRC Press, 2005.

MICHELS, Félix *et al.* Otimização de Aplicações Paralelas em Aceleradores Vetoriais NEC SX-Aurora. *In*: Sociedade Brasileira de Computação, 21 out. 2020.

MORAIS, Erikosn; SILVA, Iara; SILVA, Felipe. Estudo Comparativo entre a Implementação Sequencial e Paralela dos Métodos Gauss-Jacobi e Gauss-Seidel. **Revista Eletrônica de Iniciação Científica em Computação**, v. 18, n. 2, 21 jul. 2020.

PACHECO, Peter S. ;. MALENSEK, Matthew. **An Introduction to Parallel Programming**. 1st. ed. Burlington, MA: Morgan Kaufmann, 2011.

PARHAMI, Behrooz. **Introduction to parallel processing : algorithms and architectures**. *[S.l.]*: Kluwer Academic, 2002.

SANTOS, Ezequiel. REAPROVEITAMENTO DE COMPUTADORES ATRAVÉS DE MÁQUINAS VIRTUAIS PARALELAS (PVM). 1 jun. 2012.

WALKER, Jearl.; RESNICK, Robert.; HALLIDAY, David. **Halliday & Resnick Fundamentals of physics**. *[S.l.]*: Wiley, 2014.

7 APÊNDICE A – MODELOS

Código de Programação Serial

```
import numpy as np
import time


def transferencia_calor_2d_serial(L, C, alpha, nx, ny, erro_max,
passos_monitoramento, salvar_matriz_final=False):
    dx = L / (nx - 1)
    dy = C / (ny - 1)

    # Condição de estabilidade mais exata para esquema explícito 2D:
    dt_stable = 1.0 / (2.0 * alpha * (1.0/dx**2 + 1.0/dy**2))


    # usar fator de segurança
    dt = 0.9 * dt_stable

    # Inicialização
    T = np.full((ny, nx), 100.0, dtype=np.float64)

    # Condições de contorno (Dirichlet)

    T[0, :] = 400.0   # superior
    T[-1, :] = 100.0  # inferior
    T[:, 0] = 400.0   # esquerda
    T[:, -1] = 100.0  # direita

    Tnew = T.copy()
    passo = 0
    erro = 1.0

    inicio = time.time()

    nome_arquivo = f'saida_transferencia_{nx}x{ny}.txt'
    with open(nome_arquivo, 'w') as f:
        f.write(f"Simulação: {nx}x{ny}, dx={dx:.6e}, dy={dy:.6e},
dt={dt:.6e}\n")
        f.write("Formato de monitoramento: passo, erro_max, Tmax, Tmin,
Tcentro\n")

        # precomputar coeficientes para eficiência

        cx = alpha * dt / dx**2
        cy = alpha * dt / dy**2

        # loop temporal (explícito)
        while erro > erro_max:
            # Atualização vetorizada (somente pontos internos)
            Tnew[1:-1, 1:-1] = (
                T[1:-1, 1:-1]
                + cx * (T[2:, 1:-1] - 2.0 * T[1:-1, 1:-1] + T[:-2, 1:-1])
```

```
                + cy * (T[1:-1, 2:] - 2.0 * T[1:-1, 1:-1] + T[1:-1, :-2])
            )

            # Re-aplicar condições de contorno (garante fixidez)

            Tnew[0, :] = 400.0
            Tnew[-1, :] = 100.0
            Tnew[:, 0] = 400.0
            Tnew[:, -1] = 100.0

            # erro máximo (inf-norm) entre passos

            erro = np.max(np.abs(Tnew - T))

            # swap (evita cópia completa)

            T, Tnew = Tnew, T
            passo += 1

            # monitoramento a cada 'passos_monitoramento'

            if passo % passos_monitoramento == 0:
                temp_max = np.max(T)
                temp_min = np.min(T)
                temp_centro = T[ny // 2, nx // 2]
                f.write(f"{passo}, {erro:.6e}, {temp_max:.6f}, {temp_min:.6f},
{temp_centro:.6f}\n")

                # imprimir também no console ocasionalmente

                print(f"Passo {passo}: erro={erro:.3e},
Tcentro={temp_centro:.4f}")

            # safety: evitar loop infinito (limite superior de iterações)
            if passo % 10000 == 0 and passo > 0:
                # estimativa de tempo decorrido e alerta
                tempo_dec = time.time() - inicio
                print(f"[ALERTA] já {passo} passos, tempo decorrido
{tempo_dec:.1f}s, erro={erro:.3e}")
                # opcionalmente, poderia inserir break se demorar demais

    fim = time.time()
    duracao = fim - inicio

    # gravar resultados finais (append)
    with open(nome_arquivo, 'a') as f:
        f.write("\n*** Convergência Atingida ***\n")
        f.write(f"Total de Passos: {passo}\n")
        f.write(f"Tempo de execução: {duracao:.4f} segundos\n")
        f.write(f"Temperatura Central: {T[ny // 2, nx // 2]:.6f} °C\n")
        f.write(f"Temperatura Máxima Final: {np.max(T):.6f} °C\n")
        f.write(f"Temperatura Mínima Final: {np.min(T):.6f} °C\n")

    if salvar_matriz_final:
        np.save(f"matriz_final_{nx}x{ny}.npy", T)
```

```
        print(f"Matriz final salva em matriz_final_{nx}x{ny}.npy")

    return T[ny // 2, nx // 2], passo, duracao

if __name__ == "__main__":
    L = 3.0
    C = 3.0
    alpha = 1.172e-5  # difusividade térmica (m^2/s)
    erro_max = 1e-8
    PASSOS_MONITORAMENTO = 1000

    # recomendação: comece com malhas pequenas para testar
    for nx, ny in [(1001,1001)]:  # altere para (1001,1001) com cuidado
        print(f"Iniciando simulação para malha {nx}x{ny}. Monitoramento a cada
{PASSOS_MONITORAMENTO} passos.")
        tcentro, passos, tempo = transferencia_calor_2d_serial(
            L, C, alpha, nx, ny, erro_max, PASSOS_MONITORAMENTO,
salvar_matriz_final=False
        )
        print(f"Simulação para {nx}x{ny} concluída: passos={passos},
tempo={tempo:.2f}s, Tcentro={tcentro:.4f}")
```

Código de Programação em Paralelo

```
import numpy as np
import time
from numba import njit, prange  # Importa Numba para Just-In-Time compilation
e paralelismo


# ----------------------------------------------------------------------
# Função principal que contém a lógica do loop temporal
# ----------------------------------------------------------------------
# Usamos njit(parallel=True) para compilação JIT e habilitação de paralelismo.
@njit(parallel=True, fastmath=True)
def atualizacao_paralela_numba(T, Tnew, cx, cy, ny, nx):
    """
    Função JIT-compilada e paralela para o passo de atualização de
temperatura.
    Usa prange para paralelizar o loop sobre as linhas.
    """
    # Paralelizando o loop principal sobre as linhas internas (y)
    for i in prange(1, ny - 1):  # i percorre de 1 até ny-2 (linhas internas)
        for j in range(1, nx - 1):  # j percorre de 1 até nx-2 (colunas
internas)
            # Equação explícita de diferenças finitas 2D:
            Tnew[i, j] = (
                    T[i, j]
                    + cx * (T[i + 1, j] - 2.0 * T[i, j] + T[i - 1, j])
                    + cy * (T[i, j + 1] - 2.0 * T[i, j] + T[i, j - 1])
            )


# ----------------------------------------------------------------------
```

```
# Função que gerará e exibirá as matrizes de gradiente (Gx e Gy)
# ------------------------------------------------------------------------
def calcular_gradiente_e_imprimir(T, nx, ny, dx, dy):
    """
    Calcula o gradiente de temperatura (dT/dx e dT/dy) usando diferenças
centrais.
    """
    # Usa np.gradient para um cálculo eficiente do gradiente
    # Retorna uma tupla: (d/dy (linhas), d/dx (colunas))
    Gy, Gx = np.gradient(T, dy, dx)

    print("\n" + "=" * 50)
    print(f"Resultado Final para malha {nx}x{ny}:")
    print(f"Temperatura Central: {T[ny // 2, nx // 2]:.6f} °C")
    print("=" * 50 + "\n")

    # Imprime Gx (gradiente na direção X)
    print("Gradiente de Temperatura (dT/dx) na grade central:")
    # Imprimindo uma fatia do meio para visualização
    slice_x = Gx[ny // 4:3 * ny // 4, nx // 4:3 * nx // 4]
    print(slice_x)
    print(f"\nValor Máximo de Gradiente X (dT/dx): {np.max(np.abs(Gx)):.4e}
°C/m")

    # Imprime Gy (gradiente na direção Y)
    print("\nGradiente de Temperatura (dT/dy) na grade central:")
    slice_y = Gy[ny // 4:3 * ny // 4, nx // 4:3 * nx // 4]
    print(slice_y)
    print(f"\nValor Máximo de Gradiente Y (dT/dy): {np.max(np.abs(Gy)):.4e}
°C/m")

    return Gx, Gy


# ------------------------------------------------------------------------
# Função principal (mantém a lógica de inicialização, monitoramento, I/O)
# ------------------------------------------------------------------------
def transferencia_calor_2d_paralela(L, C, alpha, nx, ny, erro_max,
passos_monitoramento, salvar_matriz_final=False):
    dx = L / (nx - 1)
    dy = C / (ny - 1)
    dt_stable = 1.0 / (2.0 * alpha * (1.0 / dx ** 2 + 1.0 / dy ** 2))
    dt = 0.9 * dt_stable  # Fator de segurança

    # Inicialização
    T = np.full((ny, nx), 100.0, dtype=np.float64)
    # Condições de contorno (Dirichlet)
    T[0, :] = 400.0  # superior
    T[-1, :] = 100.0  # inferior
    T[:, 0] = 100.0  # esquerda
    T[:, -1] = 100.0  # direita

    Tnew = T.copy()
    passo = 0  # Variável de contagem (singular)
    erro = 1.0
    inicio = time.time()
```

```
    nome_arquivo = f'saida_transferencia_PARALELA_{nx}x{ny}.txt'

    # Precomputar coeficientes para eficiência
    cx = alpha * dt / dx ** 2
    cy = alpha * dt / dy ** 2

    with open(nome_arquivo, 'w') as f:
        f.write(f"Simulação (Paralela Numba): {nx}x{ny}, dx={dx:.6e},
dy={dy:.6e}, dt={dt:.6e}\n")
        f.write("Formato de monitoramento: passo, erro_max, Tmax, Tmin,
Tcentro\n")

        # Loop temporal (explícito)
        while erro > erro_max:
            # 1. Atualização Paralelizada (pontos internos)
            atualizacao_paralela_numba(T, Tnew, cx, cy, ny, nx)

            # 2. Re-aplicar condições de contorno (garante fixidez)
            Tnew[0, :] = 400.0
            Tnew[-1, :] = 100.0
            Tnew[:, 0] = 100.0
            Tnew[:, -1] = 100.0

            # 3. Cálculo do erro máximo
            erro = np.max(np.abs(Tnew - T))

            # 4. Swap (evita cópia completa)
            T, Tnew = Tnew, T

            passo += 1

            # 5. Monitoramento
            if passo % passos_monitoramento == 0:
                temp_max = np.max(T)
                temp_min = np.min(T)
                temp_centro = T[ny // 2, nx // 2]
                f.write(f"{passo}, {erro:.6e}, {temp_max:.6f}, {temp_min:.6f},
{temp_centro:.6f}\n")
                print(f"Passo {passo}: erro={erro:.3e},
Tcentro={temp_centro:.4f}")

            # 6. Safety check
            if passo % 10000 == 0 and passo > 0:
                tempo_dec = time.time() - inicio
                print(f"[ALERTA] já {passo} passos, tempo decorrido
{tempo_dec:.1f}s, erro={erro:.3e}")

    fim = time.time()
    duracao = fim - inicio

    # Gravar resultados finais (append)
    with open(nome_arquivo, 'a') as f:
        f.write("\n*** Convergência Atingida ***\n")
        f.write(f"Total de Passos: {passo}\n")
```

```
            f.write(f"Tempo de execução: {duracao:.4f} segundos\n")
            f.write(f"Temperatura Central: {T[ny // 2, nx // 2]:.6f} °C\n")

    if salvar_matriz_final:
        np.save(f"matriz_final_PARALELA_{nx}x{ny}.npy", T)
        print(f"Matriz final salva em matriz_final_PARALELA_{nx}x{ny}.npy")

    # Chamada para calcular e imprimir o gradiente de temperatura
    calcular_gradiente_e_imprimir(T, nx, ny, dx, dy)

    # LINHA CORRIGIDA: Usa a variável 'passo' (singular)
    return T[ny // 2, nx // 2], passo, duracao, T  # Retorna a matriz final T


# ------------------------------------------------------------------------
# Bloco de execução principal
# ------------------------------------------------------------------------
if __name__ == "__main__":
    L = 3.0
    C = 3.0
    alpha = 1.172e-5  # difusividade térmica (m^2/s)
    erro_max = 1e-8
    PASSOS_MONITORAMENTO = 1000

    # Malhas selecionadas para teste de paralelismo
    malhas_teste = [(1001,1001)]

    # Use esta lista para testes mais extensivos, se desejar:
    # malhas_teste =
[(7,7),(11,11),(25,25),(51,51),(75,75),(101,101),(207,207),(301,301),(501,501)
,(733,733),(851,851),(1001,1001)]

    for nx, ny in malhas_teste:
        print("\n" + "#" * 70)
        print(f"Iniciando simulação PARALELA para malha {nx}x{ny}.
Monitoramento a cada {PASSOS_MONITORAMENTO} passos.")
        print("#" * 70)

        tcentro, passos_finais, tempo, T_final =
transferencia_calor_2d_paralela(
            L, C, alpha, nx, ny, erro_max, PASSOS_MONITORAMENTO,
salvar_matriz_final=False
        )
        print(
            f"\nSimulação PARALELA para {nx}x{ny} CONCLUÍDA:
passos={passos_finais}, tempo={tempo:.2f}s, Tcentro={tcentro:.4f}")
        print("---")
```

Código para salvar arquivos em csv de passos relevantes das medições de temperatura:

```
import os
import time
import numpy as np
from numba import njit, prange, set_num_threads, get_num_threads
```

```
# (Opcional) limitar threads Numba aqui, se desejar:
# set_num_threads(16)

@njit(parallel=True, fastmath=True)
def atualizacao_paralela_numba(T, Tnew, cx, cy, ny, nx):
    for i in prange(1, ny - 1):
        for j in range(1, nx - 1):
            Tnew[i, j] = (
                T[i, j]
                + cx * (T[i + 1, j] - 2.0 * T[i, j] + T[i - 1, j])
                + cy * (T[i, j + 1] - 2.0 * T[i, j] + T[i, j - 1])
            )

def calcular_gradiente_e_imprimir(T, nx, ny, dx, dy):
    Gy, Gx = np.gradient(T, dy, dx)
    print("\n" + "=" * 50)
    print(f"Resultado Final para malha {nx}x{ny}:")
    print(f"Temperatura Central: {T[ny // 2, nx // 2]:.6f} °C")
    print("=" * 50 + "\n")
    slice_x = Gx[ny // 4:3 * ny // 4, nx // 4:3 * nx // 4]
    print(slice_x)
    print(f"\nValor Máximo de Gradiente X
(dT/dx): {np.max(np.abs(Gx)):.4e} °C/m")
    print("\nGradiente de Temperatura (dT/dy) na grade central:")
    slice_y = Gy[ny // 4:3 * ny // 4, nx // 4:3 * nx // 4]
    print(slice_y)
    print(f"\nValor Máximo de Gradiente Y
(dT/dy): {np.max(np.abs(Gy)):.4e} °C/m")
    return Gx, Gy

def transferencia_calor_2d_paralela(
    L, C, alpha, nx, ny, erro_max, passos_monitoramento,
    salvar_matriz_final=False,
    save_csv=True,
    save_every_steps=1,
    save_interval=None,            # <-- Novo: se definido, salva em 1,
interval, 2*interval, ...
    max_saves=None,
    save_final=True,               # <-- Novo: garantir salvar o último passo
    output_root="resultados"
):
    """
    save_csv: bool -> salvar CSVs
    save_every_steps: int -> salvar apenas a cada N passos (p.ex. 1 para cada
passo) [usado se save_interval is None]
    save_interval: int ou None -> quando definido, salva nos passos 1,
interval, 2*interval, ...
    max_saves: None ou int -> limite máximo de arquivos a salvar (None = sem
limite)
    save_final: bool -> garante que o último passo seja salvo (se save_csv
True)
    output_root: pasta base onde os CSVs serão salvos
    """
    dx = L / (nx - 1)
    dy = C / (ny - 1)
```

```
    dt_stable = 1.0 / (2.0 * alpha * (1.0 / dx ** 2 + 1.0 / dy ** 2))
    dt = 0.9 * dt_stable

    T = np.full((ny, nx), 100.0, dtype=np.float64)
    T[0, :] = 400.0
    T[-1, :] = 100.0
    T[:, 0] = 100.0
    T[:, -1] = 100.0
    Tnew = T.copy()

    passo = 0
    erro = 1.0
    inicio = time.time()

    nome_arquivo = f'saida_transferencia_PARALELA_{nx}x{ny}.txt'
    cx = alpha * dt / dx ** 2
    cy = alpha * dt / dy ** 2

    # Preparar pasta para CSVs
    output_dir = os.path.join(output_root, "csv", f"{nx}x{ny}")
    if save_csv:
        os.makedirs(output_dir, exist_ok=True)
        print(f"[INFO] CSVs serão salvos em: {output_dir}")

    saved_count = 0
    last_saved_step = -1  # para rastrear se o último passo já foi salvo

    with open(nome_arquivo, 'w') as f:
        f.write(f"Simulação (Paralela Numba): {nx}x{ny}, dx={dx:.6e},
dy={dy:.6e}, dt={dt:.6e}\n")
        f.write("Formato de monitoramento: passo, erro_max, Tmax, Tmin,
Tcentro\n")

        # Loop temporal
        while erro > erro_max:
            atualizacao_paralela_numba(T, Tnew, cx, cy, ny, nx)

            Tnew[0, :] = 400.0
            Tnew[-1, :] = 100.0
            Tnew[:, 0] = 100.0
            Tnew[:, -1] = 100.0

            erro = np.max(np.abs(Tnew - T))
            T, Tnew = Tnew, T
            passo += 1

            # Monitoramento e logging
            if passo % passos_monitoramento == 0:
                temp_max = np.max(T)
                temp_min = np.min(T)
                temp_centro = T[ny // 2, nx // 2]
                f.write(f"{passo}, {erro:.6e}, {temp_max:.6f}, {temp_min:.6f},
 {temp_centro:.6f}\n")
                print(f"Passo {passo}: erro={erro:.3e},
Tcentro={temp_centro:.4f}")
```

```
            # Decidir se salva esse passo (nova lógica)
            if save_csv:
                should_save = False
                # Prioridade para save_interval (comportamento solicitado: 1,
15000, 30000, ...)
                if save_interval is not None:
                    if passo == 1 or (passo % save_interval == 0):
                        should_save = True
                else:
                    # comportamento legado: salva a cada save_every_steps
                    if save_every_steps is not None and (passo % save_every_st
eps == 0):
                        should_save = True

                # Se deve salvar, verificar limite max_saves
                if should_save:
                    if (max_saves is None) or (saved_count < max_saves):
                        nome_csv = os.path.join(output_dir, f"step_{passo:06d}
_{nx}x{ny}.csv")
                        np.savetxt(nome_csv, T, delimiter=",", fmt="%.6f")
                        saved_count += 1
                        last_saved_step = passo
                        # feedback leve
                        if saved_count % 10 == 0:
                            print(f"[INFO] {saved_count} CSVs salvos até o
passo {passo}...")
                    else:
                        # já atingiu max_saves
                        pass

            # Safety check
            if passo % 10000 == 0 and passo > 0:
                tempo_dec = time.time() - inicio
                print(f"[ALERTA] já {passo} passos, tempo
decorrido {tempo_dec:.1f}s, erro={erro:.3e}")

    fim = time.time()
    duracao = fim - inicio

    # Garantir que o último passo seja salvo, se solicitado
    if save_csv and save_final:
        if last_saved_step != passo:  # se o último passo ainda não foi salvo
            if (max_saves is None) or (saved_count < max_saves):
                nome_csv = os.path.join(output_dir, f"step_{passo:06d}_{nx}x{n
y}.csv")
                np.savetxt(nome_csv, T, delimiter=",", fmt="%.6f")
                saved_count += 1
                print(f"[INFO] Último passo ({passo}) salvo como CSV.")
            else:
                print(f"[WARN] Não salvou o último passo ({passo}) porque já
atingiu max_saves ({max_saves}).")

    with open(nome_arquivo, 'a') as f:
        f.write("\n*** Convergência Atingida ***\n")
        f.write(f"Total de Passos: {passo}\n")
```

```
            f.write(f"Tempo de execução: {duracao:.4f} segundos\n")
            f.write(f"Temperatura Central: {T[ny // 2, nx // 2]:.6f} °C\n")
            f.write(f"Total CSVs salvos: {saved_count}\n")

    if salvar_matriz_final:
        np.save(os.path.join(output_dir, f"matriz_final_PARALELA_{nx}x{ny}.npy
"), T)
        print(f"Matriz final salva em matriz_final_PARALELA_{nx}x{ny}.npy")

    calcular_gradiente_e_imprimir(T, nx, ny, dx, dy)
    return T[ny // 2, nx // 2], passo, duracao, T

# ----------------------
# Execução principal
# ----------------------
if __name__ == "__main__":
    L = 3.0
    C = 3.0
    alpha = 1.172e-5
    erro_max = 1e-8
    PASSOS_MONITORAMENTO = 1000

    # Escolha pequenas malhas para testar salvamento completo (7x7, 11x11
etc).
    malhas_teste = [(101,101)]

    for nx, ny in malhas_teste:
        print("\n" + "#" * 70)
        print(f"Iniciando simulação PARALELA para malha {nx}x{ny}.
Monitoramento a cada {PASSOS_MONITORAMENTO} passos.")
        print("#" * 70)

        # Exemplos de parâmetros de salvamento:
        # Usar save_interval=15000 para salvar em 1,15000,30000,...
        save_interval = 193
        max_saves = None  # ou informe um inteiro se quiser limitar
        tcentro, passos_finais, tempo, T_final = transferencia_calor_2d_parale
la(
            L, C, alpha, nx, ny, erro_max, PASSOS_MONITORAMENTO,
            salvar_matriz_final=False,
            save_csv=True,
            save_every_steps=1,   # ignorado se save_interval não for None
            save_interval=save_interval,
            max_saves=max_saves,
            save_final=True,
            output_root="resultados"
        )
        print(f"\nSimulação PARALELA para {nx}x{ny} CONCLUÍDA:
passos={passos_finais}, tempo={tempo:.2f}s, Tcentro={tcentro:.4f}")
        print("---")
```

Código para coletar arquivos csv e plotar os gráficos de temperatura com a extensão *Matplotlab*, conforme os passos relevantes pré selecionados:

```
# gerar_gifs_todas_subpastas_personalizado.py
import os
import glob
import re
import io
from pathlib import Path
from typing import List, Tuple

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
import imageio


# ---------------------- CONFIGURAÇÃO ----------------------
# Ajuste estas variáveis conforme seu caso
root_csv_dir = Path(r"E:\Elaboracao_Proj\resultados\csv")  # pasta que contém
subpastas (7x7, 11x11, ...)
L_domain = 3.0   # comprimento em x (m)
C_domain = 3.0   # comprimento em y (m)
out_gif_template = "transferencia_{subfolder}.gif"

colormap_name = None  # se None usaremos colormap personalizado
fps = 6
dpi = 150
sample_every_for_scale = 1  # 1 = usa todos os CSVs para calcular vmin/vmax;
>1 pula alguns arquivos para acelerar
max_frames = None           # limite de frames por subpasta (None = todos)
save_pngs = False           # True para também salvar PNGs em cada
subpasta/pngs/
pngs_folder_name = "pngs"   # nome da subpasta para PNGs, se save_pngs=True
# ----------------------------------------------------------


def criar_colormap_personalizado():
    """Cria colormap cinza -> amarelo -> laranja -> vermelho."""
    colors = [
        (0.6, 0.6, 0.6),   # cinza (frio)
        (1.0, 1.0, 0.0),   # amarelo (médio)
        (1.0, 0.5, 0.0),   # laranja
        (1.0, 0.0, 0.0)    # vermelho (quente)
    ]
    return LinearSegmentedColormap.from_list("cinza_yel_org_red", colors, N=25
6)


def find_and_sort_csvs_in_folder(csv_folder: Path) -> List[Path]:
    """Encontra CSVs e ordena por número em 'step_000123_...csv'. Se não
achar, lista todos alfabeticamente."""
    files = list(csv_folder.glob("*.csv"))
    rx = re.compile(r"step[_\-]?0*([0-9]+)_?.*\.csv$", re.IGNORECASE)
    matched = []
    for f in files:
        m = rx.search(f.name)
        if m:
            step = int(m.group(1))
```

```
            matched.append((step, f))
    if matched:
        matched.sort(key=lambda x: x[0])
        return [p for (_, p) in matched]
    return sorted(files)


def compute_vmin_vmax(csv_paths: List[Path], sample_every: int = 1) -
> Tuple[float, float]:
    """Calcula vmin/vmax global para um conjunto de CSVs (pode pular arquivos
para acelerar)."""
    vmin = float("inf")
    vmax = float("-inf")
    for idx, p in enumerate(csv_paths):
        if idx % sample_every != 0:
            continue
        try:
            arr = np.loadtxt(p, delimiter=",")
        except Exception as e:
            print(f"[WARN] falha ao ler {p}: {e} - pulando")
            continue
        vmin = min(vmin, np.nanmin(arr))
        vmax = max(vmax, np.nanmax(arr))
    if vmin == float("inf") or vmax == float("-inf"):
        raise RuntimeError("Não foi possível determinar vmin/vmax. Verifique
os CSVs nesta pasta.")
    return vmin, vmax


def plot_temperatura_to_imagearray(arr: np.ndarray,
                                   L: float, C: float,
                                   nx: int, ny: int,
                                   vmin: float, vmax: float,
                                   cmap,
                                   title: str = None,
                                   dpi: int = 150,
                                   figsize=(6, 6)):
    """
    Gera um array de imagem (RGB) a partir de uma matriz T.
    Usa origin='upper' para colocar a primeira linha do array no topo.
    """
    extent = [0.0, L, 0.0, C]  # xmin, xmax, ymin, ymax (metros)
    if title is None:
        title = f"Temperatura (°C) - malha {nx}x{ny}"

    fig, ax = plt.subplots(figsize=figsize, dpi=dpi)
    im = ax.imshow(arr, origin="upper", extent=extent, cmap=cmap, vmin=vmin, v
max=vmax, aspect='auto')
    ax.set_title(title, fontsize=12)
    ax.set_xlabel("x (m)", fontsize=10)
    ax.set_ylabel("y (m)", fontsize=10)

    # ticks "amigáveis" (aprox 5 ticks)
    def nice_ticks(length_m):
        max_ticks = 6
        raw_step = max(length_m / (max_ticks - 1), 1e-12)
```

```
        candidates = [1.0, 0.5, 0.25, 0.2, 0.1, 0.05, 0.02, 0.01]
        step = min(candidates, key=lambda c: abs(c - raw_step))
        ticks = np.arange(0, length_m + 1e-12, step)
        return ticks

    ax.set_xticks(nice_ticks(L))
    ax.set_yticks(nice_ticks(C))

    cbar = fig.colorbar(im, ax=ax, fraction=0.046, pad=0.04)
    cbar.set_label("°C", rotation=270, labelpad=12)

    plt.tight_layout()

    buf = io.BytesIO()
    fig.savefig(buf, format='png', bbox_inches='tight')
    plt.close(fig)
    buf.seek(0)
    img = imageio.v2.imread(buf)  # retorna ndarray RGB
    return img


def process_folder(csv_folder: Path, L: float, C: float, out_gif_name: str,
                   cmap, fps: int, dpi: int,
                   sample_every_for_scale: int, max_frames: int, save_pngs: bo
ol):
    print(f"\n=== Processando pasta: {csv_folder} ===")
    csvs = find_and_sort_csvs_in_folder(csv_folder)
    if len(csvs) == 0:
        print("[INFO] Nenhum CSV encontrado nesta pasta. Pulando.")
        return

    if max_frames is not None:
        csvs = csvs[:max_frames]

    try:
        vmin, vmax = compute_vmin_vmax(csvs, sample_every=sample_every_for_sca
le)
    except Exception as e:
        print(f"[ERRO] Não foi possível calcular vmin/vmax
em {csv_folder}: {e}")
        return

    print(f"[INFO] {len(csvs)} CSVs a processar. vmin={vmin:.6f},
vmax={vmax:.6f}")

    frames = []
    png_dir = csv_folder / pngs_folder_name if save_pngs else None
    if save_pngs:
        png_dir.mkdir(parents=True, exist_ok=True)

    for idx, p in enumerate(csvs):
        try:
            arr = np.loadtxt(p, delimiter=",")
        except Exception as e:
            print(f"[WARN] Erro lendo {p}: {e} - pulando")
```

```
            continue

        ny, nx = arr.shape
        title = f"frame {idx} - {p.name}"
        img = plot_temperatura_to_imagearray(arr, L, C, nx, ny, vmin, vmax, cm
ap, title=title, dpi=dpi)
        frames.append(img)

        if save_pngs:
            png_path = png_dir / f"frame_{idx:04d}.png"
            imageio.v2.imwrite(str(png_path), img)

        if (idx + 1) % 20 == 0 or (idx + 1) == len(csvs):
            print(f"[INFO] Gerados {idx+1}/{len(csvs)} frames
(último: {p.name})")

    if len(frames) == 0:
        print("[ERRO] Nenhum frame válido gerado nesta pasta.")
        return

    gif_path = csv_folder / out_gif_name
    try:
        imageio.mimsave(str(gif_path), frames, fps=fps)
        print(f"[OK] GIF salvo em: {gif_path}")
        if save_pngs:
            print(f"[OK] PNGs individuais salvos em: {png_dir}")
    except Exception as e:
        print(f"[ERRO] Falha ao salvar GIF em {gif_path}: {e}")


def process_all_subfolders(root_csv_dir: Path, L: float, C: float, out_gif_tem
plate: str,
                           cmap, fps: int, dpi: int, sample_every_for_scale: i
nt, max_frames: int, save_pngs: bool):
    if not root_csv_dir.exists():
        print(f"[ERRO] Diretório raiz não existe: {root_csv_dir}")
        return
    subfolders = [p for p in root_csv_dir.iterdir() if p.is_dir()]
    if len(subfolders) == 0:
        print(f"[INFO] Nenhuma subpasta encontrada em: {root_csv_dir}")
        return

    print(f"[INFO] Encontradas {len(subfolders)} subpastas. Iniciando
processamento...")
    for sub in subfolders:
        out_gif_name = out_gif_template.format(subfolder=sub.name)
        process_folder(sub, L, C, out_gif_name, cmap, fps, dpi, sample_every_f
or_scale, max_frames, save_pngs)


if __name__ == "__main__":
    # Prepara colormap
    cmap = criar_colormap_personalizado() if colormap_name is None else plt.ge
t_cmap(colormap_name)
```

```
    process_all_subfolders(root_csv_dir, L_domain, C_domain, out_gif_template,
                           cmap, fps, dpi, sample_every_for_scale, max_frames,
save_pngs)
```

ANEXO A

Tabela para encontrar a solução analítica:

Calculo_Série_Fourier .XLSX

Arquivo Editar Ver Inserir Formatar Dados Ferramentas Ajuda

Menus

F33

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1						Solução Analítica para Transmissão de Calor 2D											
2		Condições Inicial (° C):		100												
3		Condições de Contorno (° C):	Borda Superior:	400	Preencher: (Tmáx, Tmín):	Tmáx	Tmín	(Tmáx-Tmín)	2*(Tmáx-Tmín)							
4			Borda Inferior:	100		400	100	300	600							
5			Borda Esquerda:	100								Tsup:	400			
6			Borda Direita:	100												
7																
8		n	(n-1)/2	(-1)^(n-1)/2	(n*pi)/2	cosh((n*pi)/2)	Fator 2*(Tmáx-Tmín)/(n*pi)	Contribuição do Termo			Tesquerda:					Tesquerda:
9		1	0	1	1,570796327	2,509178479	190,9859317	76,114925			100					100
10		3	1	-1	4,712388988	55,66338089	63,66197724	-1,143696								
11		5	2	1	7,853981634	1287,965442	38,19718634	0,02966								
12		7	3	-1	10,99557429	29804,87075	27,28370453	-0,00092								
13																
14						Total do somatório da Série Fourier:		74,99997028	°C							
15							Tmenor da placa:	100,0000000	°C							
16						Tcentral Solução Analítica:		174,9999702	°C							
17																
18																
19																
20																
21																

Tabela feita no Excel para análise da programação serial e paralela:

Data: 06/10/2025	*Tcentral Analítico =*	174,9999	Erro	1,00E-08	*Condições de Contorno:*	*Temperatura Borda Superior:*	400º C		TCC - Focar este
	Condição Inicial =	100,0000				*Temperatura Demais Bordas:*	100º C		

Critério Parada	Notepad escreverá:	Matriz	Tempo de Execução(s)		Tcentral(º C)		Passos		Tamanho dos arquivos (KB)		Erro Relativo		Tspeedup	Eficiencia	Tempo de Execução(min)		Tempo de Execução(hora)	
Erro:	Imprimi a cada: ___ passos		Serial	Paralelo	Serial	Paralelo	Serial	Paralelo	Serial	Paralelo	Serial	Paralelo			Serial	Paralelo	Serial	Paralelo
1,00E-08	1000	7	0,0155	2,1949	175,0000	175,0000	165	165	1	1	-0,0001	-0,0001	0,0071	0,0221	0,0003	0,03658166667	0,0000	0,0006096944444
1,00E-08	1000	11	0,0045	0,0115	175,0000	175,0000	447	447	1	1	-0,0001	-0,0001	0,3913	1,2228	0,0001	0,0001916666667	0,0000	0,0000031944444444
1,00E-08	1000	25	0,0330	0,0730	175,0000	175,0000	2376	2376	1	1	-0,0001	-0,0001	0,4521	1,4127	0,0006	0,001216666667	0,0000	0,0000202777778
1,00E-08	1000	51	0,2043	2,4781	175,0000	175,0000	9502	9502	1	1	-0,0001	-0,0001	0,0824	0,2576	0,0034	0,04130166667	0,0001	0,0006883611111
1,00E-08	1000	75	0,6373	0,6001	175,0000	175,0000	19852	19852	2	2	-0,0001	-0,0001	1,0620	3,3187	0,0106	0,01000166667	0,0002	0,0001666944444
1,00E-08	1000	101	1,5295	1,1671	175,0000	175,0000	34900	34900	3	3	-0,0001	-0,0001	1,3105	4,0954	0,0255	0,01945166667	0,0004	0,0003241944444
1,00E-08	1000	207	19,2584	8,9306	174,9999	174,9999	134304	134304	8	8	0,0000	0,0000	2,1565	6,7389	0,3209733333	0,1488433333	0,0053	0,002480722222
1,00E-08	1000	301	119,8686	32,7621	174,9998	174,9998	269606	269606	16	16	0,0001	0,0001	3,6588	11,4336	1,99781	0,546035	0,0333	0,009100583333
1,00E-08	1000	501	2244,8390	225,7990	174,9994	174,9994	691403	691403	40	40	0,0005	0,0005	9,9418	31,0680	37,41398333	3,763316667	0,6236	0,06272194444
1,00E-08	1000	733	7597,4841	961,7471	174,9988	174,9988	1389912	1389912	80	80	0,0011	0,0011	7,8997	24,6865	126,624735	16,02911833	2,1104	0,2671519722
1,00E-08	1000	851	17884,3728	1806,3739	174,9984	174,9984	1825520	1825520	105	105	0,0015	0,0015	9,9007	30,9397	298,07288	30,10623167	4,9679	0,5017705278
1,00E-08	1000	1001	40966,8107	3808,3084	174,9977	174,9977	2453488	2453488	141	141	0,0022	0,0022	10,7572	33,6163	682,7801783	63,47180667	11,3797	1,057863444