UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

Relatório da Tarefa 11 - Análise de Desempenho das Cláusulas `collapse` e `schedule` em OpenMP

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 21 de setembro de 2025

Lista de Figuras

Sumário

1 Introdução

A simulação computacional de fenômenos físicos constitui um pilar fundamental da ciência e engenharia contemporâneas. Diversos problemas complexos, como dinâmica de fluidos, transferência de calor e propagação de ondas, são modelados por meio de equações diferenciais parciais e resolvidos numericamente em malhas computacionais. Um dos principais desafios nesse contexto é a paralelização eficiente de laços aninhados (nested loops) que iteram sobre essas malhas, uma vez que eles concentram a maior parte do custo computacional.

Nesse cenário, a equação da difusão — um caso simplificado das equações de Navier-Stokes — destaca-se como um problema-modelo ideal. Ela incorpora a essência computacional de um estêncil de cinco pontos, um padrão de acesso a dados recorrente em simulações científicas.

O objetivo deste projeto é analisar e comparar o impacto de diferentes estratégias de paralelização de laços oferecidas pelo OpenMP para este problema. Foram desenvolvidas e avaliadas abordagens distintas para entender como a distribuição de trabalho e o escalonamento de iterações afetam o desempenho final. As versões analisadas incluem:

1. **Implementação Serial:** Versão sequencial para estabelecer a linha de base (*baseline*) de desempenho.
2. **Paralelismo Simples:** Utilização da diretiva `#pragma omp parallel for` apenas no laço mais externo.
3. **Paralelismo Otimizado com `collapse`:** Aplicação da cláusula `collapse(2)` para criar um espaço de iteração unificado.
4. **Análise de Escalonamento (`schedule`):** Estudo comparativo do impacto das políticas `static`, `dynamic` e `guided` sobre o tempo de execução da versão otimizada.

Este relatório busca, através da análise empírica dessas implementações, ilustrar a importância da escolha da estratégia de paralelização adequada, destacando as diferenças de desempenho e produtividade entre as abordagens.

2 Metodologia do Experimento

O experimento consiste na implementação e análise de desempenho de múltiplas versões de um programa que simula a difusão de um campo de velocidade em 2D.

2.1 O Modelo Físico e a Discretização

O problema é governado pela equação da difusão vetorial, uma simplificação da equação de Navier-Stokes que considera apenas os efeitos da viscosidade (ν):

$$\frac{\partial \vec{v}}{\partial t} = \nu \nabla^2 \vec{v} \tag{2.1}$$

Para a solução numérica, o domínio foi discretizado em uma grade 2D e o método das diferenças finitas foi aplicado. A derivada temporal foi aproximada por um esquema de Euler explícito de primeira ordem, e o operador Laplaciano (∇^2) foi aproximado por um estêncil de cinco pontos. Isso resulta na seguinte equação de atualização para o componente de velocidade u em cada ponto (i, j) da grade a cada passo de tempo Δt:

$$u_{i,j}^{t+1} = u_{i,j}^t + \frac{\nu \Delta t}{(\Delta x)^2} \left(u_{i+1,j}^t + u_{i-1,j}^t + u_{i,j+1}^t + u_{i,j-1}^t - 4u_{i,j}^t \right) \tag{2.2}$$

Uma equação análoga é utilizada para o componente v. O domínio utiliza condições de contorno periódicas, simulando uma superfície toroidal.

2.2 Implementações Avaliadas

Para analisar o impacto das diferentes estratégias de paralelização, foram desenvolvidas quatro versões principais do código em C com a biblioteca OpenMP.

2.2.1 Versão 1: Serial (Baseline e Validação)

Foram criadas duas variantes do código serial:

- **Código para Medição de Tempo:** Uma implementação limpa e direta do algoritmo, focada em medir o tempo de execução sequencial que serve como *baseline* para os cálculos de *speedup* (Apêndice **??**).
- **Código para Geração de Gráficos:** Uma versão mais elaborada que salva o estado da simulação em arquivos de dados. Estes arquivos foram utilizados para gerar os gráficos de validação apresentados na Seção 3.1 (Apêndice **??**).

2.2.2 Versão 2: Paralelismo Simples

Nesta abordagem (Apêndice **??**), a diretiva `#pragma omp parallel for` é aplicada apenas ao laço mais externo. Esta estratégia representa uma primeira tentativa de paralelização, onde o *runtime* do OpenMP distribui as linhas da grade entre as *threads*.

2.2.3 Versão 3: Paralelismo Otimizado

Esta versão (Apêndice **??**) aplica otimizações com base na estrutura do problema:

- **Cláusula `collapse(2)`:** É utilizada nos laços aninhados principais para tratá-los como um único laço, aumentando o espaço de iteração e melhorando o balanceamento de carga.
- **Cláusula `schedule`:** Sobre a versão com `collapse`, foram conduzidos experimentos variando a política de escalonamento (`static`, `dynamic`, `guided`) para quantificar seu impacto.

3 Análise e Discussão

3.1 Validação da Implementação Serial

A correção da simulação numérica foi verificada visualmente a partir dos dados gerados pelo código apresentado no Apêndice **??**. O processo de validação consiste em observar se a perturbação de velocidade, inicialmente concentrada no centro do domínio, se comporta de acordo com o modelo físico da difusão. O comportamento esperado é que a perturbação se espalhe suavemente pela grade, com sua amplitude máxima diminuindo ao longo do tempo.

As Figuras 2 e 3 demonstram este fenômeno de forma inequívoca para os componentes de velocidade u e v, respectivamente. Em ambas as figuras, o snapshot no **Step 0** mostra um pico de velocidade agudo e localizado. Conforme a simulação avança para os passos 2000, 4000, até 10000, observa-se visualmente que o pico diminui progressivamente em altura e se alarga em sua base. A utilização de uma escala vertical fixa nos gráficos (eixo Z e barra de cores) torna essa dissipação de energia quantificável, confirmando que a amplitude da perturbação está de fato decaindo.

A Figura 4 complementa a análise, mostrando a evolução da magnitude total da velocidade ($\sqrt{u^2 + v^2}$) em um mapa de calor 2D. A difusão é visível através do alargamento da área aquecida central e da diminuição de sua intensidade máxima (a cor passa de um amarelo brilhante para um vermelho mais escuro).

Com base nessas evidências visuais, que demonstram o decaimento suave e a dispersão da perturbação inicial, a implementação serial do simulador é considerada fisicamente correta e validada.

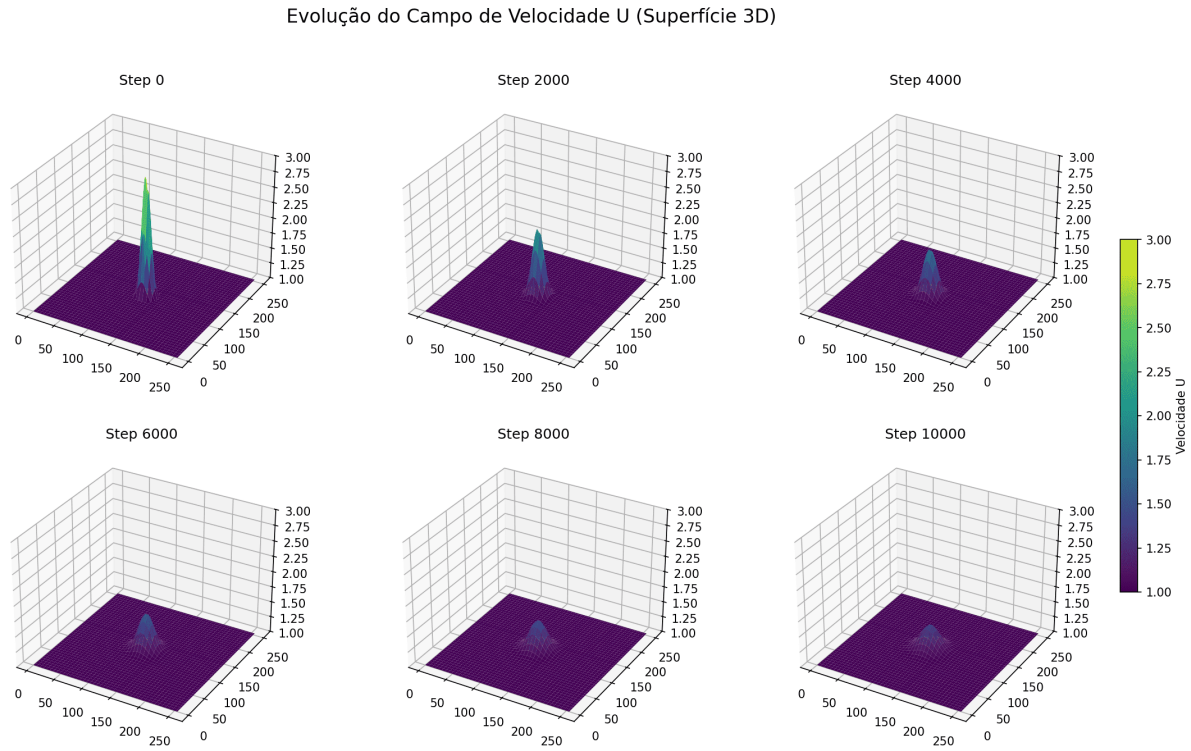

Figura 2 – Evolução temporal do componente U da velocidade em 6 snapshots. A diminuição da amplitude e o alargamento da base da perturbação são claramente visíveis, confirmando o efeito da difusão.

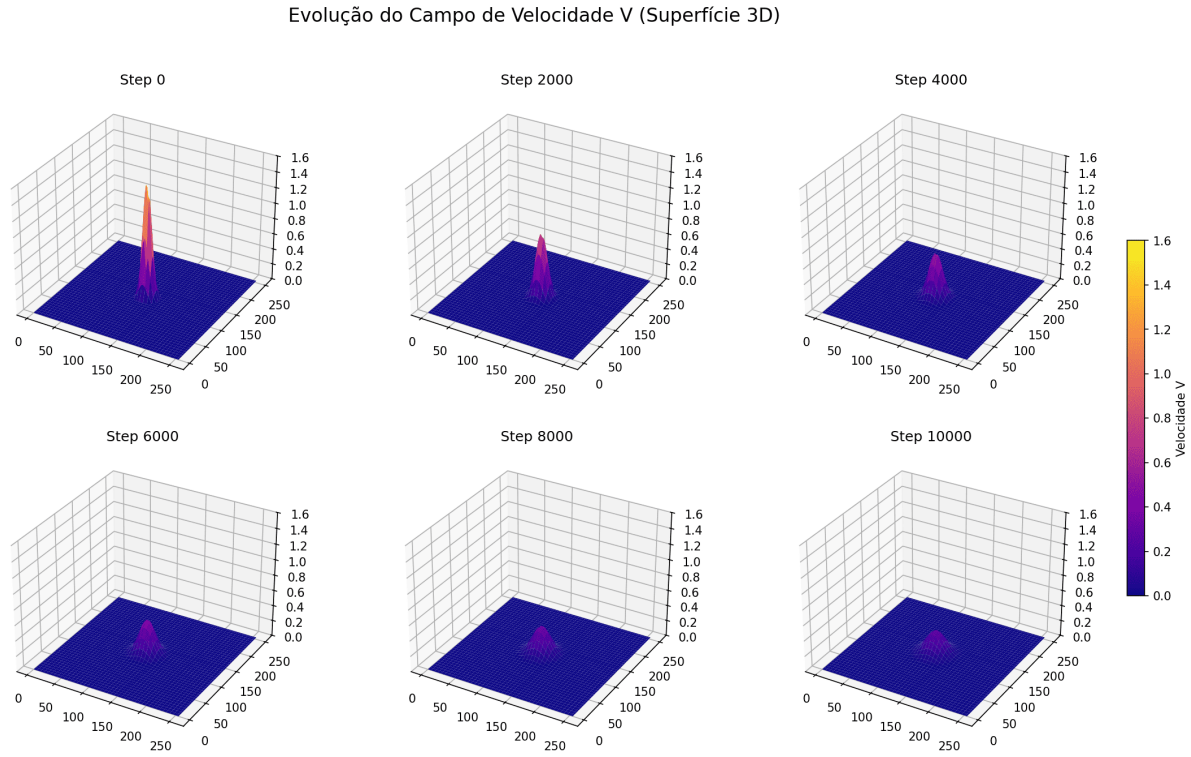

Figura 3 – Evolução temporal do componente V da velocidade. O comportamento de difusão é análogo ao observado para o componente U.

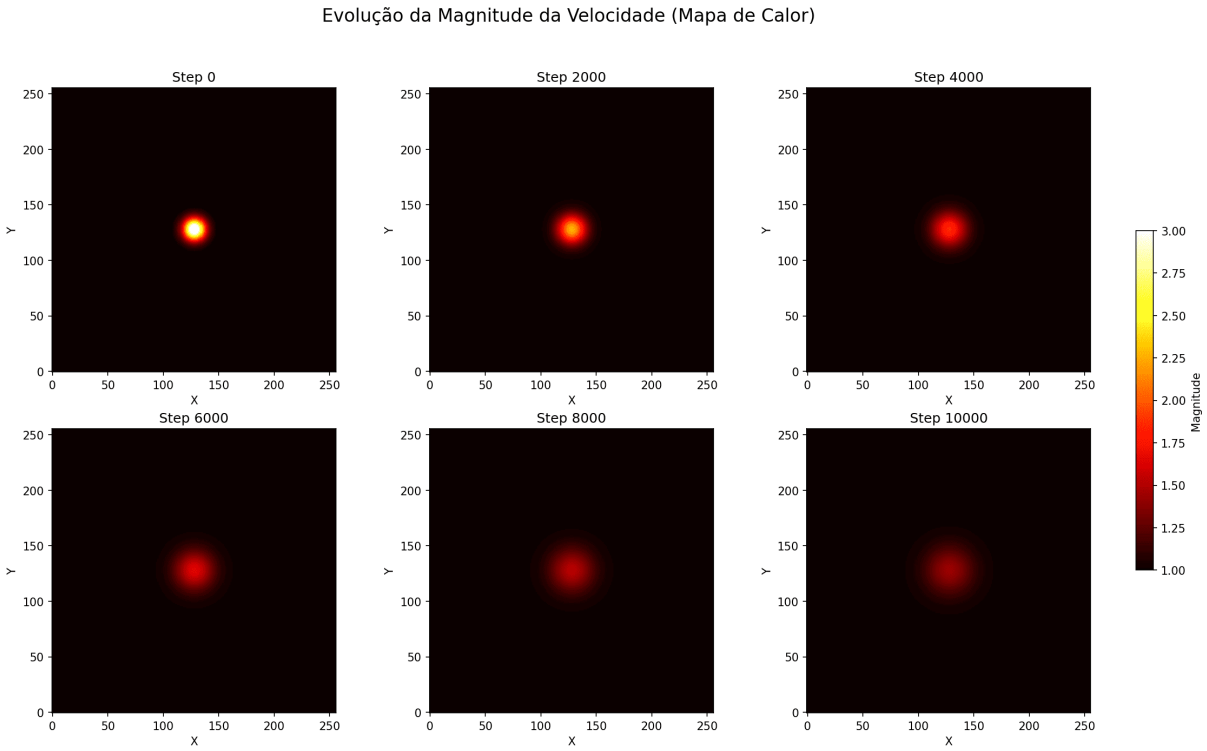

Figura 4 – Evolução da magnitude da velocidade em 6 snapshots (mapa de calor). A mancha de calor central se espalha e perde intensidade ao longo do tempo.

3.2 Análise de Desempenho e Produtividade

A análise de desempenho foi conduzida em um nó computacional com 2 processadores Intel Xeon E5-2620 v3, totalizando 12 núcleos físicos e 24 threads lógicas. O tempo da versão serial foi estabelecido como a média de 5 execuções, resultando em um *baseline* de **8.635 segundos** para um total de 100.000 passos de tempo.

3.2.1 Resultados Quantitativos

Os tempos de execução para cada estratégia de paralelização e número de threads foram coletados (média de 5 execuções) e os respectivos speedups foram calculados. Os resultados estão consolidados na Tabela 1.

3.2.2 Resultados Quantitativos

Os tempos de execução para cada estratégia de paralelização foram coletados, e os respectivos speedups foram calculados em relação ao *baseline* serial. Os resultados estão consolidados na Tabela 1.

Tabela 1 – Tabela de tempos de execução e speedup para cada implementação.

Estratégia de Paralelização	**N. Threads**	**Média (s)**	**Speedup**
1. Serial	1	8.635	1.00x
2. Paralelo Simples (sem collapse)	1	12.238	0.71x
	2	6.177	1.40x
	4	3.839	2.25x
	8	2.365	3.65x
	16	1.932	**4.47x**
3. Otimizado (collapse + static)	1	13.541	0.64x
	2	6.407	1.35x
	4	4.022	2.15x
	8	2.402	3.59x
	16	1.926	**4.48x**
4. Otimizado (collapse + dynamic)	1	65.751	**0.13x**
	2	83.962	0.10x
	4	237.363	0.04x
	8	480.546	0.02x
	16	950.265	0.01x
5. Otimizado (collapse + guided)	1	12.732	0.68x
	2	7.082	1.22x
	4	4.791	1.80x
	8	3.308	2.61x
	16	2.775	**3.11x**
5. Otimizado (collapse + guided)(section)	1	12.701	0.68x
	2	6.999	1.23x
	4	4.944	1.75x
	8	3.821	2.26x
	16	3.35	**2.58x**

3.2.3 Análise de Escalabilidade (Speedup)

A Figura 5 ilustra a escalabilidade das principais estratégias de paralelização. O gráfico compara o speedup obtido por cada abordagem em função do aumento do número de threads, tendo como referência o speedup linear ideal (linha preta tracejada). Esta visualização permite avaliar a eficiência com que cada estratégia aproveita os recursos computacionais adicionais e identificar os pontos onde a aceleração começa a diminuir.

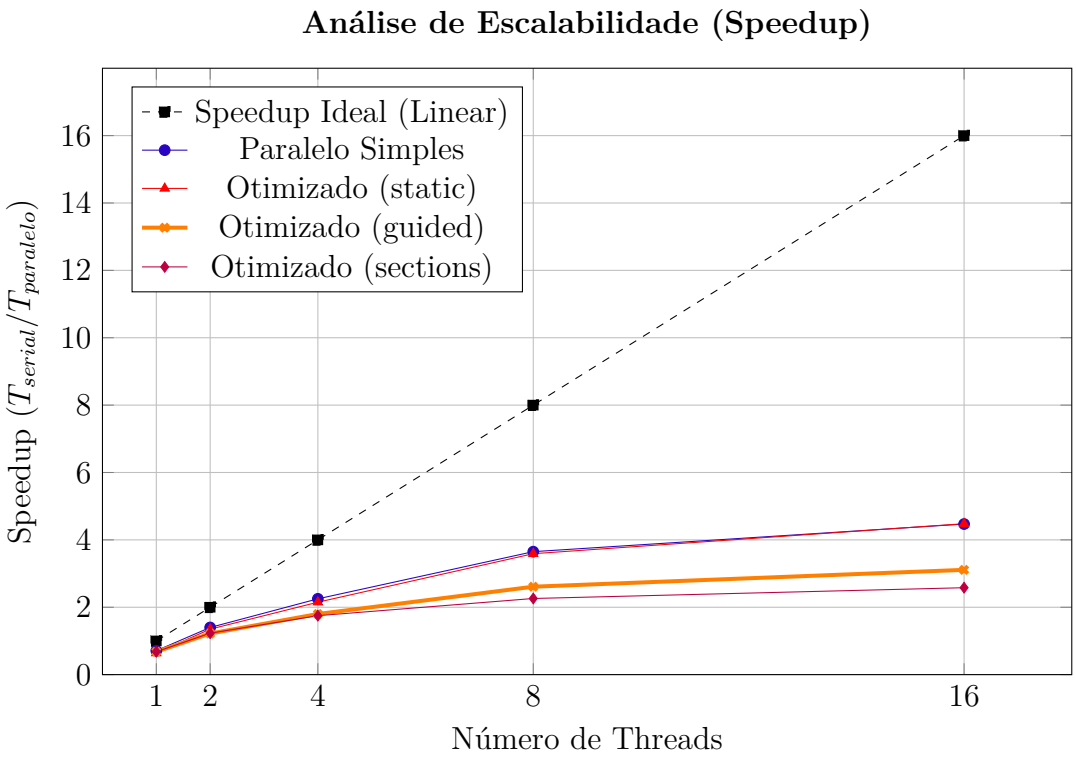

Figura 5 – Gráfico de Speedup vs. Número de Threads, gerado com `pgfplots`. A linha preta tracejada representa a escalabilidade linear ideal.

A análise do gráfico e da tabela revela os seguintes pontos:

1. **Escalabilidade Sublinear:** Nenhuma das versões atinge o speedup linear ideal. A melhor performance, obtida com 16 threads, foi de **4.48x** pela versão `collapse + static`. Essa escalabilidade sublinear é esperada e pode ser atribuída a gargalos inerentes ao sistema, como a largura de banda da memória (um fator limitante em algoritmos *memory-bound* como este) e as porções de código que permanecem seriais (Lei de Amdahl).

2. **Otimizado (static) vs. Paralelo Simples:** As duas melhores estratégias apresentaram desempenho quase idêntico. Isso indica que, para este problema de carga uniforme, a simples paralelização do laço externo já foi suficiente para distribuir bem o trabalho entre os núcleos, e a otimização com `collapse` não trouxe ganhos significativos, sugerindo que o balanceamento de carga não era o principal gargalo.

3. **Desempenho de `guided` e `sections`:** A cláusula `guided` apresentou um bom speedup (3.11x), mas inferior à `static`, devido ao seu *overhead* de agendamento, desnecessário para uma carga uniforme. A versão com `sections` para as condições de contorno também foi menos eficiente (2.58x), mostrando que, embora funcional, não foi a melhor abordagem para este problema.

3.2.4 Análise de Sobrecarga e Desaceleração Paralela

O caso mais drástico foi o da cláusula `schedule(dynamic)` sem um *chunksize*, que resultou em uma severa desaceleração paralela, como visto na Tabela 1. Com 16 threads, a execução foi aproximadamente 110 vezes mais lenta que a versão serial. Este fenômeno ocorre pelo imenso *overhead* de o *runtime* do OpenMP gerenciar e distribuir individualmente cada uma das mais de 260.000 iterações do laço interno a cada passo de tempo, confirmando que a escolha da política de agendamento é crítica para o desempenho.

3.2.5 Discussão dos Resultados

A análise dos dados completos, consolidados na Tabela 1, e a visualização da escalabilidade na Figura 5, revelam diversos pontos cruciais sobre a otimização de laços em OpenMP para este problema.

1. **O Custo Fixo do Paralelismo:** Um padrão consistente em todas as estratégias paralelas é que, ao serem executadas com apenas 1 thread, elas são consideravelmente mais lentas que o código puramente serial. Isso evidencia o *overhead* (custo computacional) introduzido pelo *runtime* do OpenMP para criar e gerenciar as regiões paralelas. Fica claro que o paralelismo só se torna vantajoso quando o ganho com a divisão do trabalho entre múltiplas threads supera esse custo inicial.

2. **Melhor Desempenho e Escalabilidade Sublinear:** A melhor performance foi alcançada pela versão **Otimizado (collapse + static)**, com um tempo de 1.926 segundos e um **speedup máximo de 4.48x** com 16 threads. A versão "Paralelo Simples"teve um desempenho quase idêntico (speedup de 4.47x), indicando que, para este problema de carga uniforme, a simples paralelização do laço externo já foi uma estratégia de balanceamento de carga muito eficaz. Nenhuma versão atingiu o speedup linear ideal, o que é esperado e pode ser atribuído a gargalos como a largura de banda da memória (um fator limitante em algoritmos *memory-bound*) e as porções de código que permanecem seriais (Lei de Amdahl).

3. **O Impacto do Agendamento (`schedule`):**

   - **`static`:** Provou ser a política mais eficiente, pois a carga de trabalho é perfeitamente uniforme, e seu baixo *overhead* de agendamento (dividindo o trabalho apenas uma vez no início) é ideal.
   - **`guided`:** Apresentou um bom speedup (3.11x), mas foi visivelmente inferior à `static`. Seu mecanismo de agendamento com blocos de tamanho decrescente, projetado para cargas não uniformes, introduziu um *overhead* desnecessário aqui.

- **`dynamic`:** Conforme previsto, resultou em uma desaceleração massiva. Com 16 threads, a execução foi aproximadamente **110 vezes mais lenta** que a versão serial. O custo de agendar individualmente centenas de milhares de iterações a cada passo de tempo superou completamente o trabalho computacional, servindo como um exemplo clássico de *parallel slowdown*.

4. **Análise do Paralelismo de Tarefas (`sections`):** A abordagem de usar `sections` para as condições de contorno foi funcional, mas menos eficiente que as melhores versões com paralelismo de dados, atingindo um speedup de apenas 2.58x. Isso demonstra que, para este problema, dividir os laços de dados com `omp for` foi uma estratégia mais adequada do que dividir as tarefas com `omp sections`.

Para uma comparação direta dos tempos de execução, a Figura 6 apresenta um gráfico de barras agrupado. Cada grupo no eixo X representa uma contagem de threads, e as barras coloridas mostram o tempo médio de execução das diferentes estratégias eficientes. Este gráfico facilita a identificação da abordagem de melhor desempenho para cada nível de paralelismo testado.

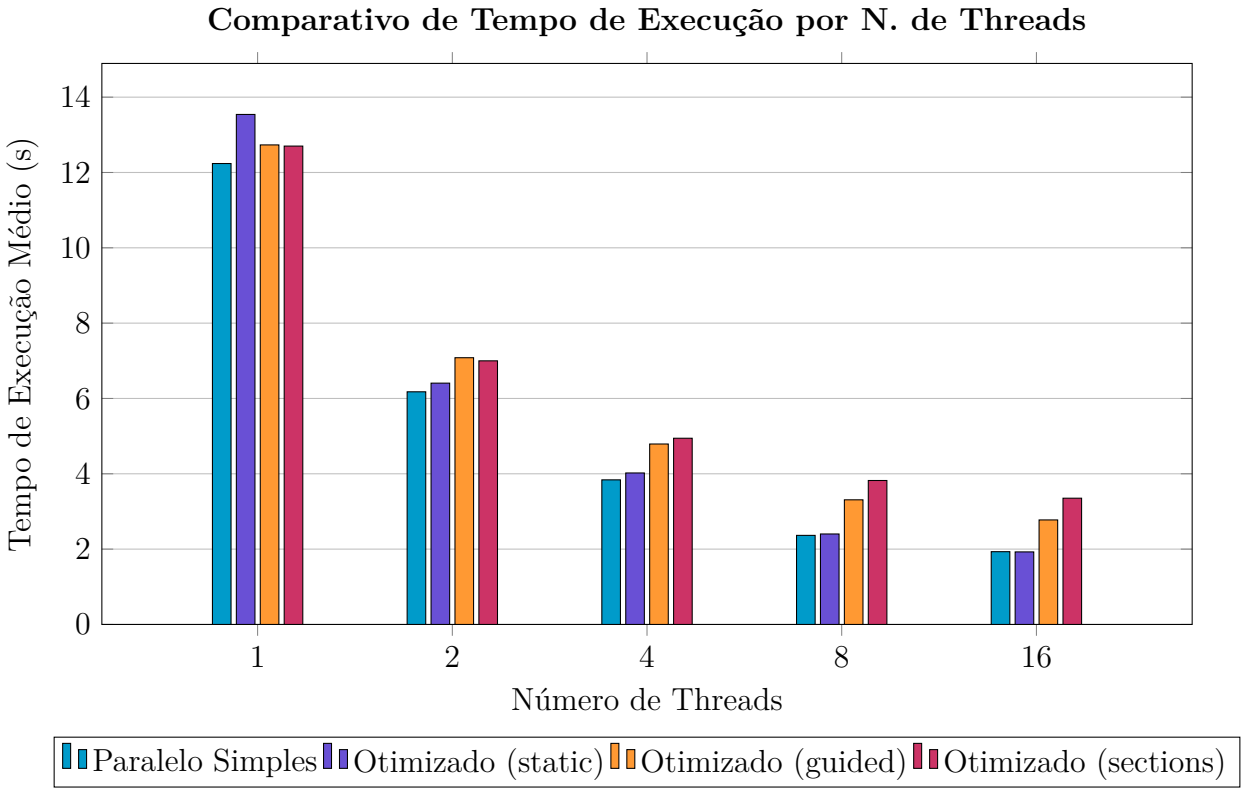

Figura 6 – Gráfico de barras agrupado comparando o tempo de execução das estratégias de paralelização eficientes. A superioridade das abordagens `static` e `simples` é visível em todas as contagens de threads.

A fim de ilustrar o impacto de uma escolha inadequada de agendamento, a Figura 7 foca exclusivamente na estratégia `Otimizado (collapse + dynamic)`. O gráfico de barras mostra como o tempo de execução não apenas falha em diminuir, mas aumenta drasticamente com a adição de mais threads, um claro exemplo de desaceleração paralela causado pela alta sobrecarga (*overhead*) do agendador.

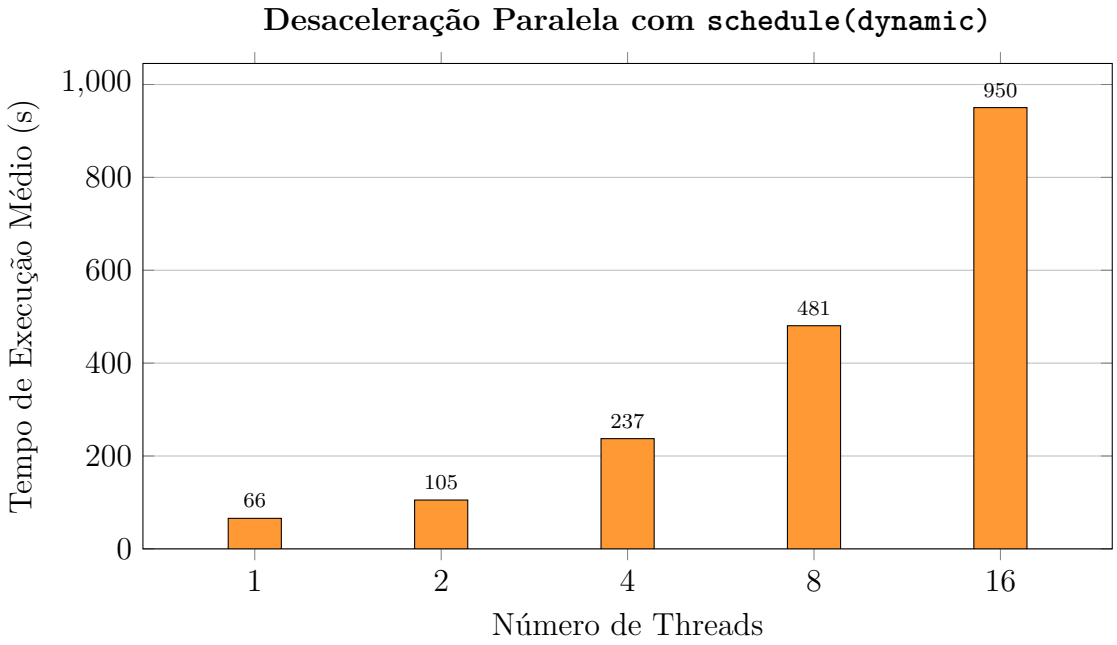

Figura 7 – Gráfico de barras mostrando o aumento do tempo de execução para a estratégia `Otimizado (collapse + dynamic)`. A adição de mais threads piora drasticamente o desempenho, um claro exemplo de desaceleração paralela devido ao alto overhead de agendamento.

4 Roteiro

Escolha da Estratégia de Paralelização de Laços

A análise das implementações permite a elaboração de um roteiro prático para a escolha da estratégia de paralelização, uma decisão fundamental para o desempenho.

4.0.1 Passo 1: Identificar Padrões de Laços Aninhados

O primeiro passo é verificar se a carga computacional principal do programa reside em laços aninhados. Se a resposta for sim, e os laços forem perfeitamente aninhados (sem código entre eles) e com limites de iteração independentes, a cláusula `collapse` deve ser a primeira ferramenta a ser considerada.

4.0.2 Passo 2: Analisar a Natureza da Carga de Trabalho

Após decidir pela paralelização do ninho de laços, a escolha da política de `schedule` depende do custo computacional de cada iteração.

4.0.2.1 Cenário 1: Carga de Trabalho Uniforme

- **Descrição:** Cada iteração do laço leva aproximadamente o mesmo tempo para ser executada. Este é o caso de estênceis simples, multiplicação de matrizes densas, etc.
- **Solução:** Utilize `schedule(static)`.
- **Justificativa:** Minimiza o *overhead* de escalonamento, pois a distribuição do trabalho pode ser feita uma única vez no início do laço de forma eficiente.

4.0.2.2 Cenário 2: Carga de Trabalho Não-Uniforme

- **Descrição:** O tempo de execução por iteração varia significativamente, por exemplo, devido a laços com número variável de iterações internas ou condicionais complexos.
- **Solução:** Utilize `schedule(dynamic)` ou `schedule(guided)`.
- **Justificativa:** Embora possuam maior *overhead*, estas políticas permitem o balanceamento dinâmico da carga, garantindo que nenhuma *thread* fique ociosa. A `guided` é frequentemente um bom ponto de partida.

5 Conclusão

A análise detalhada das cinco estratégias de paralelização para a simulação da equação de difusão permitiu extrair conclusões robustas e, em alguns casos, surpreendentes sobre a otimização de laços aninhados com OpenMP. Os resultados experimentais não apenas validaram conceitos teóricos, mas também destacaram a importância de uma análise criteriosa na escolha das ferramentas de paralelização.

As principais conclusões deste trabalho são:

1. **A estratégia de paralelização é mais importante que a micro-otimização:** A maior salto de desempenho (de 8.6 segundos para aproximadamente 1.9 segundos, um speedup de **4.48x**) não veio de uma cláusula específica, mas da decisão correta de manter o laço de tempo serial e paralelizar os laços espaciais, que continham o trabalho de dados independentes. Isso reforça que a identificação correta do paralelismo disponível em um algoritmo é o passo mais crucial para a obtenção de aceleração.

2. **A política de agendamento (`schedule`) deve ser adequada à carga de trabalho:** A comparação das políticas de `schedule` foi o diferencial mais dramático. A `schedule(static)` provou ser a ideal para a carga de trabalho uniforme do problema, oferecendo o melhor desempenho devido ao seu baixo *overhead*. Em contrapartida, a `schedule(dynamic)` resultou em uma desaceleração catastrófica (tempo de execução $\sim$110x maior que o serial), demonstrando que o custo do agendamento pode superar massivamente o trabalho computacional se a política for inadequada para o problema.

3. **O impacto de otimizações avançadas como `collapse` pode ser sutil:** Embora a teoria aponte a cláusula `collapse` como uma otimização poderosa, seu impacto neste experimento foi marginal em comparação com uma paralelização simples do laço externo. O desempenho quase idêntico entre as versões "Paralelo Simples"e "Otimizado (static)"sugere que o principal gargalo, após a paralelização, não era o balanceamento de carga, mas provavelmente fatores de hardware como a largura de banda da memória.

4. **Abstrações de alto nível devem ser priorizadas, mas compreendidas:** O experimento confirmou que as abstrações do OpenMP são poderosas. No entanto, a escolha da abstração correta é fundamental. Para este problema, a combinação de `omp for` com `collapse` e `schedule(static)` representa a solução mais idiomática

e de melhor desempenho, enquanto o uso inadequado de `schedule(dynamic)` ilustra os perigos de uma aplicação incorreta dessas abstrações.

Em suma, a tarefa demonstrou que o desenvolvimento de código paralelo eficiente exige mais do que a simples aplicação de diretivas. Requer uma análise cuidadosa dos padrões de dependência de dados, a seleção de uma estratégia de paralelização compatível e, finalmente, o ajuste fino das políticas de agendamento para minimizar a sobrecarga e maximizar a utilização dos recursos computacionais.

serial/navier_stokes_simul_serial_tempo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NX 512
#define NY 512
#define NT 2000
#define DT 0.001
#define NU 0.01

int main() {
    // Alocar arrays 2D
    double **u = (double**)malloc(NX * sizeof(double*));
    double **v = (double**)malloc(NX * sizeof(double*));
    double **u_new = (double**)malloc(NX * sizeof(double*));
    double **v_new = (double**)malloc(NX * sizeof(double*));

    for (int i = 0; i < NX; i++) {
        u[i] = (double*)malloc(NY * sizeof(double));
        v[i] = (double*)malloc(NY * sizeof(double));
        u_new[i] = (double*)malloc(NY * sizeof(double));
        v_new[i] = (double*)malloc(NY * sizeof(double));
    }

    // Inicialização
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            u[i][j] = 1.0;
            v[i][j] = 0.0;
            double dx = i - NX/2, dy = j - NY/2;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < 20.0) {
                u[i][j] += 2.0 * exp(-dist*dist/100.0);
                v[i][j] += 1.5 * exp(-dist*dist/100.0);
            }
        }
    }

    double start_time = omp_get_wtime();

    // Simulação principal
    for (int step = 0; step < NT; step++) {
        for (int i = 1; i < NX-1; i++) {
            for (int j = 1; j < NY-1; j++) {
                double d2u_dx2 = (u[i+1][j] - 2.0*u[i][j] + u[i-1][j]);
                double d2u_dy2 = (u[i][j+1] - 2.0*u[i][j] + u[i][j-1]);
                double d2v_dx2 = (v[i+1][j] - 2.0*v[i][j] + v[i-1][j]);
                double d2v_dy2 = (v[i][j+1] - 2.0*v[i][j] + v[i][j-1]);

```

```c
                u_new[i][j] = u[i][j] + DT * NU * (d2u_dx2 + d2u_dy2);
                v_new[i][j] = v[i][j] + DT * NU * (d2v_dx2 + d2v_dy2);
            }
        }

        // Condições de contorno periódicas
        for (int i = 0; i < NX; i++) {
            u_new[i][0] = u_new[i][NY-2];
            u_new[i][NY-1] = u_new[i][1];
            v_new[i][0] = v_new[i][NY-2];
            v_new[i][NY-1] = v_new[i][1];
        }
        for (int j = 0; j < NY; j++) {
            u_new[0][j] = u_new[NX-2][j];
            u_new[NX-1][j] = u_new[1][j];
            v_new[0][j] = v_new[NX-2][j];
            v_new[NX-1][j] = v_new[1][j];
        }

        // Trocar arrays
        double **temp_u = u;
        double **temp_v = v;
        u = u_new;
        v = v_new;
        u_new = temp_u;
        v_new = temp_v;
    }

    double end_time = omp_get_wtime();

    // Saída apenas do tempo
    printf("%.6f\n", end_time - start_time);

    // Liberar memória
    for (int i = 0; i < NX; i++) {
        free(u[i]);
        free(v[i]);
        free(u_new[i]);
        free(v_new[i]);
    }
    free(u);
    free(v);
    free(u_new);
    free(v_new);

    return 0;
}
```

paralelo/navier_stokes_simul_paralela.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NX 512
#define NY 512
#define NT 10000
#define DT 0.001
#define NU 0.01

int main() {
    // Alocar memória
    double **u = malloc(NX * sizeof(double*));
    double **v = malloc(NX * sizeof(double*));
    double **un = malloc(NX * sizeof(double*));
    double **vn = malloc(NX * sizeof(double*));

    for (int i = 0; i < NX; i++) {
        u[i] = malloc(NY * sizeof(double));
        v[i] = malloc(NY * sizeof(double));
        un[i] = malloc(NY * sizeof(double));
        vn[i] = malloc(NY * sizeof(double));
    }

    #pragma omp parallel for
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            double dx = i - NX/2, dy = j - NY/2;
            double dist_sq = dx*dx + dy*dy;

            u[i][j] = 1.0;
            v[i][j] = 0.0;

            if (dist_sq < 400) {
                double perturbation = exp(-dist_sq/100.0);
                u[i][j] += 2.0 * perturbation;
                v[i][j] += 1.5 * perturbation;
            }
        }
    }

    double start = omp_get_wtime();

    // Loop de tempo PRINCIPAL
    for (int t = 0; t < NT; t++) {
        // 1. Atualização dos valores (um laço paralelo)
        #pragma omp parallel for
        for (int i = 1; i < NX-1; i++) {
            for (int j = 1; j < NY-1; j++) {
```

```c
                un[i][j] = u[i][j] + DT*NU*(u[i+1][j] + u[i-1][j] + u[i][j+1] + u[i][j-1] - 4*u[i][j]);
                vn[i][j] = v[i][j] + DT*NU*(v[i+1][j] + v[i-1][j] + v[i][j+1] + v[i][j-1] - 4*v[i][j]);
            }
        }
        // 2. Aplicar condições de contorno (dois laços paralelos)
        #pragma omp parallel for
        for (int i = 0; i < NX; i++) {
            un[i][0] = un[i][NY-2];
            un[i][NY-1] = un[i][1];
            vn[i][0] = vn[i][NY-2];
            vn[i][NY-1] = vn[i][1];
        }

        #pragma omp parallel for
        for (int j = 0; j < NY; j++) {
            un[0][j] = un[NX-2][j];
            un[NX-1][j] = un[1][j];
            vn[0][j] = vn[NX-2][j];
            vn[NX-1][j] = vn[1][j];
        }

        // Swap pointers (feito pelo thread mestre, serialmente)
        double **ut = u, **vt = v;
        u = un; v = vn;
        un = ut; vn = vt;
    }

    double end = omp_get_wtime();
    printf("%.6f\n", end - start);

    // Cleanup
    for (int i = 0; i < NX; i++) {
        free(u[i]); free(v[i]); free(un[i]); free(vn[i]);
    }
    free(u); free(v); free(un); free(vn);

    return 0;
}
```

paralelo/navier_stokes_simul_paralela_otm_st.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

#define NX 512
#define NY 512
#define NT 10000 // Mantendo o NT alto para testes de desempenho
#define DT 0.001
#define NU 0.01

int main() {
    // Alocar memória (sem alterações)
    double **u = malloc(NX * sizeof(double*));
    double **v = malloc(NX * sizeof(double*));
    double **un = malloc(NX * sizeof(double*));
    double **vn = malloc(NX * sizeof(double*));

    for (int i = 0; i < NX; i++) {
        u[i] = malloc(NY * sizeof(double));
        v[i] = malloc(NY * sizeof(double));
        un[i] = malloc(NY * sizeof(double));
        vn[i] = malloc(NY * sizeof(double));
    }

    // Inicialização (pode ser otimizada com collapse também)
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < NX; i++) {
        for (int j = 0; j < NY; j++) {
            double dx = i - NX/2, dy = j - NY/2;
            double dist_sq = dx*dx + dy*dy;

            u[i][j] = 1.0;
            v[i][j] = 0.0;

            if (dist_sq < 400) {
                double perturbation = exp(-dist_sq/100.0);
                u[i][j] += 2.0 * perturbation;
                v[i][j] += 1.5 * perturbation;
            }
        }
    }

    double start = omp_get_wtime();

    // Loop de tempo PRINCIPAL - continua serial
    for (int t = 0; t < NT; t++) {

        // OTIMIZAÇÃO 1: UMA ÚNICA REGIÃO PARALELA
        // As threads são criadas apenas uma vez por passo de tempo.
```

```
51          #pragma omp parallel
52          {
53              // OTIMIZAÇÃO 2: COLLAPSE E SCHEDULE
54              // A cláusula collapse(2) trata os laços 'i' e 'j' como um único
    laço,
55              // melhorando muito o balanceamento de carga.
56              // A cláusula schedule controla como as iterações são distribuídas.
57              #pragma omp for collapse(2) schedule(static) /* Mude aqui para seus
    testes: (dynamic), (guided), (static, 64), etc. */
58              for (int i = 1; i < NX-1; i++) {
59                  for (int j = 1; j < NY-1; j++) {
60                      un[i][j] = u[i][j] + DT*NU*(u[i+1][j] + u[i-1][j] + u[i]
    [j+1] + u[i][j-1] - 4*u[i][j]);
61                      vn[i][j] = v[i][j] + DT*NU*(v[i+1][j] + v[i-1][j] + v[i]
    [j+1] + v[i][j-1] - 4*v[i][j]);
62                  }
63              }
64
65              // Condições de contorno dentro da mesma região paralela
66              // O omp for possui uma barreira implícita no final, garantindo que
    o cálculo
67              // principal termine antes de aplicar as condições de contorno.
68              #pragma omp for
69              for (int i = 0; i < NX; i++) {
70                  un[i][0] = un[i][NY-2];
71                  un[i][NY-1] = un[i][1];
72                  vn[i][0] = vn[i][NY-2];
73                  vn[i][NY-1] = vn[i][1];
74              }
75
76              #pragma omp for
77              for (int j = 0; j < NY; j++) {
78                  un[0][j] = un[NX-2][j];
79                  un[NX-1][j] = un[1][j];
80                  vn[0][j] = vn[NX-2][j];
81                  vn[NX-1][j] = vn[1][j];
82              }
83          } // Fim da região paralela. As threads são sincronizadas aqui.
84
85          // Swap pointers (feito pelo thread mestre, serialmente)
86          double **ut = u, **vt = v;
87          u = un; v = vn;
88          un = ut; vn = vt;
89      }
90
91      double end = omp_get_wtime();
92      printf("%.6f\n", end - start);
93
94      // Cleanup
95      for (int i = 0; i < NX; i++) {
96          free(u[i]); free(v[i]); free(un[i]); free(vn[i]);
97      }
98      free(u); free(v); free(un); free(vn);
```

```
 99
100     return 0;
101 }
```