Universidade Federal do Rio Grande do Norte - UFRN
Departamento de Engenharia da Computação e Automação - DCA
DCA3703 - Programação Paralela

# Tarefa 11: Impacto das cláusulas Schedule e Collapse

Discente: Quelita Míriam
Docente: Samuel Xavier de Souza

## I. INTRODUÇÃO

Neste relatório, avalia-se o impacto das cláusulas *schedule* e *collapse* sobre o desempenho de uma simulação numérica complexa, como as baseadas nas equações de Navier-Stokes. A análise visa compreender como essas diretivas influenciam o tempo de execução da aplicação.

Neste trabalho, foi desenvolvida uma simulação numérica do movimento de um fluido tridimensional ao longo do tempo utilizando a equação de Navier-Stokes simplificada, considerando apenas os efeitos da viscosidade. Para facilitar a implementação, foram desconsideradas as componentes de pressão e quaisquer forças externas. A discretização do domínio foi feita usando o método de diferenças finitas, com espaçamento uniforme no espaço e passos de tempo pequenos para garantir a estabilidade da simulação. A simulação foi feita em um cubo tridimensional de tamanho fixo, com o fluido inicialmente em repouso, exceto por uma pequena perturbação localizada no centro do domínio. Essa perturbação serviu para testar se o campo de velocidades se comportaria de forma esperada, ou seja, se a energia inicial se dissiparia suavemente com o tempo devido à viscosidade.

## II. METODOLOGIA

Foi utilizado um código[1] em C com OpenMP que simula a propagação de uma perturbação em um fluido tridimensional, aplicando a equação de difusão por diferenças finitas. A malha utilizada é de dimensões 50×50×50 e a simulação ocorre por 100 passos de tempo.

A metodologia envolveu primeiramente a implementação da simulação de forma sequencial, com a atualização iterativa do campo de velocidades utilizando a equação de difusão. Em seguida, o código foi paralelizado com OpenMP, permitindo testar diferentes estratégias de distribuição de trabalho entre os núcleos do processador. Foram utilizadas três formas de escalonamento (***schedule***) para dividir os laços de iteração entre as threads: *static*, *dynamic* e *guided*. Além disso, foi explorada a diretiva ***collapse***, que paraleliza mais de um laço aninhado de forma combinada, aumentando o grau de paralelismo. Diversos tamanhos de blocos de iteração (*chunks*) também foram testados.

Ao usar o OpenMP para paralelizar loops, a cláusula *schedule* define como as iterações do laço serão divididas entre as threads. Cada uma dessas estratégias tem um comportamento diferente:

- **Static**: divide as iterações do loop em partes iguais entre as threads, de forma fixa e antecipada, antes do loop começar.
- **Dinamic**: as iterações são divididas em blocos menores (chunks), e as threads pegam um novo bloco assim que terminam o anterior.

[1] Código: https://github.com/quelita2/programacao-paralela/tree/main/topico02/tarefa11

- **Guided**: os primeiros blocos são maiores e vão diminuindo de tamanho conforme a execução avança.

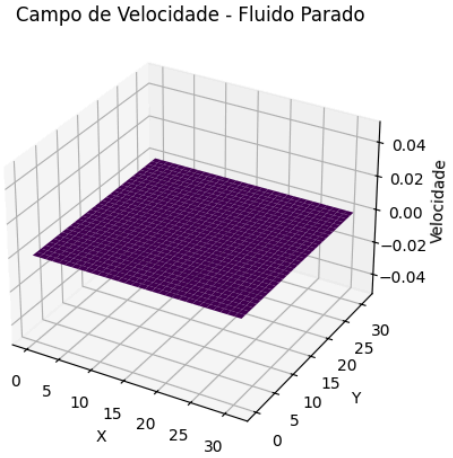
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <omp.h>
#include <string.h>

#define N 32 //Tamanho da grade (NxNxN)
#define STEPS 100 // Número de passos de tempo
#define DX 1.0 // Espaçamento entre pontos (discretização espacial)
#define DT 0.01 // Passo de tempo
#define VISC 0.1

double u[N][N][N], u_new[N][N][N];

// Medição de tempo
double get_time() {
   struct timeval tv;
   gettimeofday(&tv, NULL);
   return tv.tv_sec + tv.tv_usec * 1e-6;
}

// Inicializa campo
void initialize(int perturb) {
   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
         for (int k = 0; k < N; k++)
            u[i][j][k] = (perturb && i == N/2 && j == N/2 && k == N/2) ? 1.0 : 0.0;
}

// Executa um passo da simulação com diferentes opções
void step(const char* schedule, int collapse, int chunk_size) {
   if (strcmp(schedule, "static") == 0) {
      if (collapse)
         #pragma omp parallel for collapse(3) schedule(static,chunk_size)
         for (int i = 1; i < N-1; i++)
            for (int j = 1; j < N-1; j++)
               for (int k = 1; k < N-1; k++)
                  u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                     (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
      else
         #pragma omp parallel for schedule(static,chunk_size)
         for (int i = 1; i < N-1; i++)
```

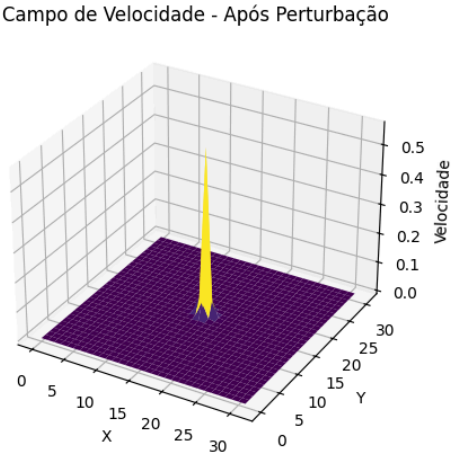
```
            for (int j = 1; j < N-1; j++)
                for (int k = 1; k < N-1; k++)
                    u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                        (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
    } else if (strcmp(schedule, "dynamic") == 0) {
        if (collapse)
            #pragma omp parallel for collapse(3) schedule(dynamic,chunk_size)
            for (int i = 1; i < N-1; i++)
                for (int j = 1; j < N-1; j++)
                    for (int k = 1; k < N-1; k++)
                        u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                            (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
        else
            #pragma omp parallel for schedule(dynamic,chunk_size)
            for (int i = 1; i < N-1; i++)
                for (int j = 1; j < N-1; j++)
                    for (int k = 1; k < N-1; k++)
                        u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                            (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
    } else if (strcmp(schedule, "guided") == 0) {
        if (collapse)
            #pragma omp parallel for collapse(3) schedule(guided,chunk_size)
            for (int i = 1; i < N-1; i++)
                for (int j = 1; j < N-1; j++)
                    for (int k = 1; k < N-1; k++)
                        u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                            (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
        else
            #pragma omp parallel for schedule(guided,chunk_size)
            for (int i = 1; i < N-1; i++)
                for (int j = 1; j < N-1; j++)
                    for (int k = 1; k < N-1; k++)
                        u_new[i][j][k] = u[i][j][k] + VISC * DT / (DX * DX) *
                            (u[i+1][j][k] + u[i-1][j][k] + u[i][j+1][k] + u[i][j-1][k] + u[i][j][k+1] + u[i][j][k-1] -
6 * u[i][j][k]);
    }

    // Copia de volta para preparar o próximo passo de tempo
    for (int i = 1; i < N-1; i++)
        for (int j = 1; j < N-1; j++)
            for (int k = 1; k < N-1; k++)
                u[i][j][k] = u_new[i][j][k];
}

// Inicializa a matriz u com ou sem pertubação
```

```
double simboraaa(const char* schedule, int collapse, int chunk_size, int perturb) {
    initialize(perturb);
    double start = get_time();
    for (int t = 0; t < STEPS; t++) step(schedule, collapse, chunk_size);
    double end = get_time();
    return end - start;
}

int main() {
    FILE *fp = fopen("resultados.csv", "w");
    fprintf(fp, "schedule,collapse,chunk_size,tempo,estado_inicial\n");

    const char* schedules[] = {"static", "dynamic", "guided"};
    int chunk_sizes[] = {1, 2, 4, 8, 16};
    int collapses[] = {0, 1};

    for (int s = 0; s < 3; s++) {
        for (int c = 0; c < 2; c++) {
            for (int cs = 0; cs < 5; cs++) {
                for (int pert = 0; pert < 2; pert++) {
                    double tempo = simboraaa(schedules[s], collapses[c], chunk_sizes[cs], pert);
                    fprintf(fp, "%s,%d,%d,%.6f,%s\n", schedules[s], collapses[c], chunk_sizes[cs], tempo,
pert ? "perturbado" : "parado");
                    printf("%s collapse=%d chunk=%d tempo=%.6f (%s)\n", schedules[s], collapses[c],
chunk_sizes[cs], tempo, pert ? "perturbado" : "parado");
                }
            }
        }
    }

    fclose(fp);
    return 0;
}
```

## III. RESULTADOS E DISCUSSÕES

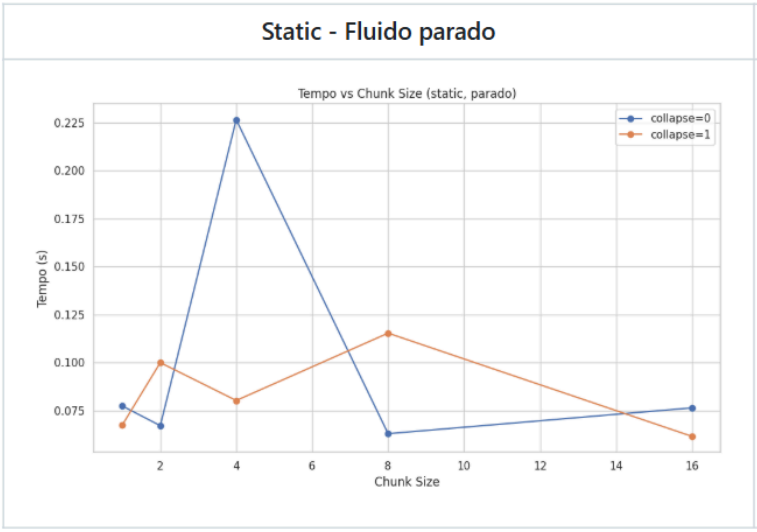
Abaixo, os gráficos representam os campos de velocidades iniciais do fluido antes e após qualquer perturbação. Cada ponto da superfície mostra a velocidade do fluido em uma posição (x, y) da malha, sendo o eixo Z a indicação da intensidade da velocidade em cada ponto da grade. Apresentando desempenho esperado para a simulação de movimento de um fluido.

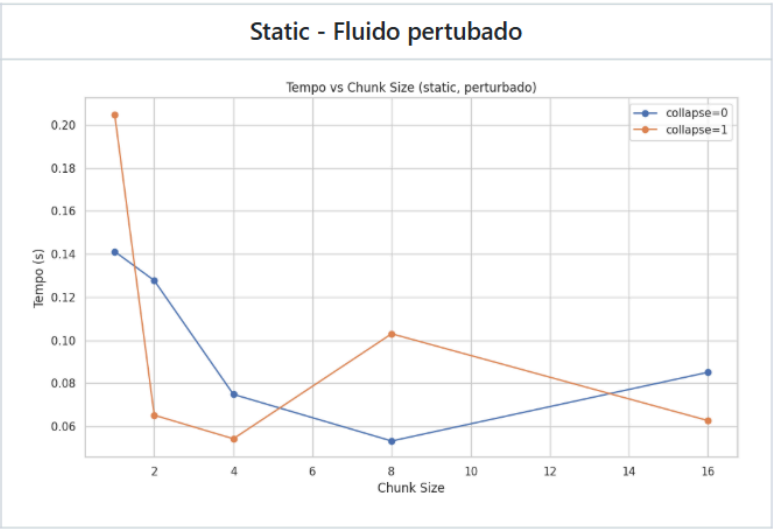



Os resultados obtidos demonstraram comportamentos distintos para cada tipo de escalonamento. O escalonamento ***static*** divide as iterações igualmente entre as threads no início da execução, sendo mais eficiente quando a carga de trabalho é equilibrada e previsível. No entanto, isso pode não ser ideal quando há variações no custo das iterações.

Seu desempenho foi estável e bom com *collapse*=0, enquanto que com *collapse*=1 o tempo oscilou bastante, parecendo ineficiente, provavelmente porque a sobrecarga de colapsar dois *loops* paralelos não compensa com essa configuração.

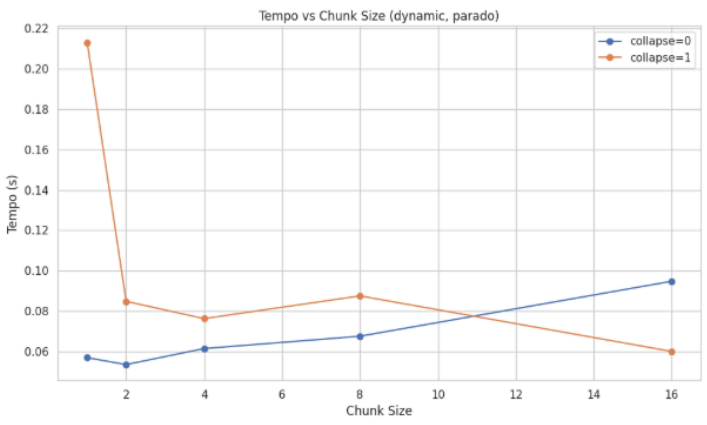


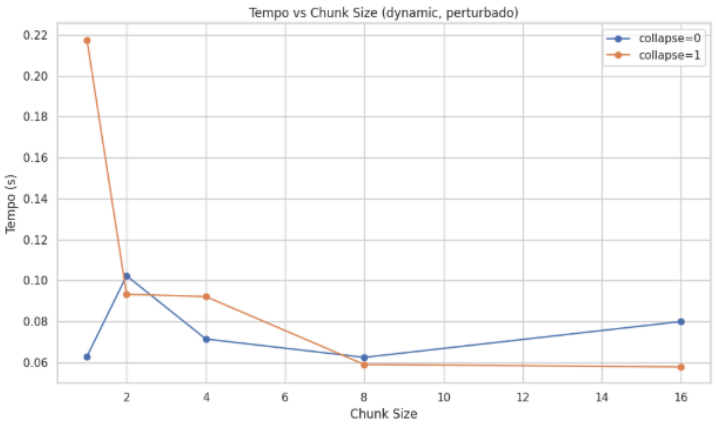

Já o ***dynamic*** atribui os blocos de iteração às *threads* conforme elas ficam disponíveis, o que é vantajoso quando a carga de trabalho é irregular, pois ajuda a manter todas as *threads* ocupadas. Os tempos com dynamic foram consistentemente baixos com ou sem collapse, mostrando boa eficiência.

O *dynamic* teve bom desempenho para *chunks* pequenos e sem *collapse*. O *collapse*=1 parece piorar o desempenho, possivelmente por conflito com a alocação dinâmica e maior sobrecarga de balanceamento.

**Dynamic - Fluido parado**



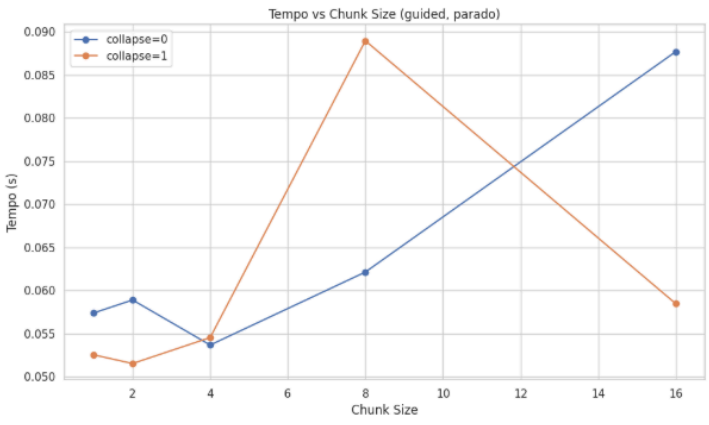

**Dynamic - Fluido pertubado**



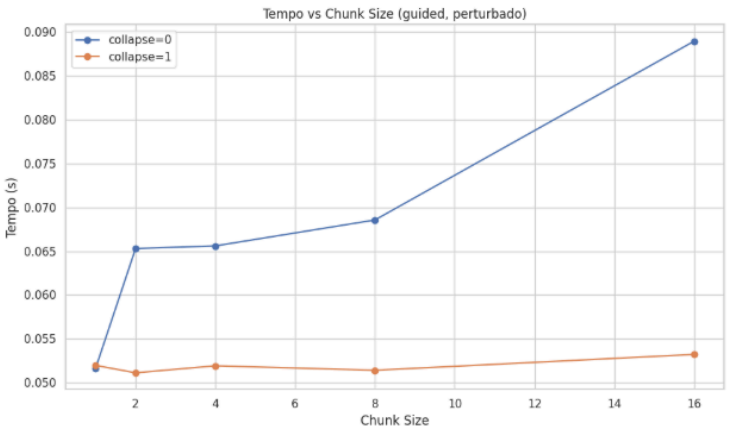

Por fim, o ***guided*** é uma abordagem intermediária, onde blocos maiores são atribuídos inicialmente e vão diminuindo com o tempo. Isso combina o balanceamento de carga com a redução de *overhead*, e apresentou os melhores tempos de execução entre todas as estratégias.

O *guided* é eficiente quando bem configurado, mas muito sensível a *chunk size*, especialmente com *collapse*=1. Foi ideal para a simulação de perturbação.

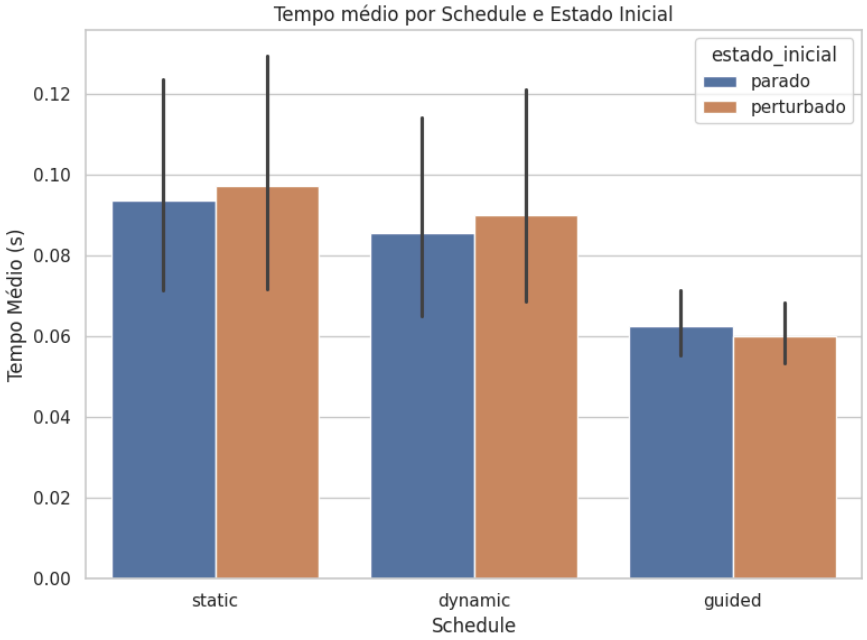
**Guided - Fluido parado**




**Guided - Fluido pertubado**




O gráfico abaixo representa os *schedules* usados e seu tempo médio de execução. As linhas pretas são linhas de erro (*error bars*), indicando a incerteza ou variação dos dados. As linhas pequenas representam medições consistentes, enquanto linhas grandes medições instáveis ou com muita flutuação.




Quando o fluido está parado, geralmente há menos trabalho computacional a ser feito, e isso pode ser percebido nos tempos mais baixos na maioria das execuções com esse estado. Já no estado perturbado, é comum observar tempos um pouco maiores, pois há mais variações e cálculos associados à propagação do movimento no fluido.

A variação dos tempos com diferentes tamanhos de *chunk* mostra que nem sempre um *chunk* maior resulta em melhor desempenho. Isso é esperado, pois *chunks* muito grandes podem sobrecarregar algumas *threads* e causar desequilíbrio na carga, enquanto *chunks* muito pequenos geram sobrecarga na criação e gerenciamento de tarefas. O comportamento ideal depende do tipo de schedule usado.

## IV. CONCLUSÃO

Conclui-se que a paralelização do código trouxe benefícios significativos de desempenho, especialmente com o uso do escalonamento guided e da cláusula collapse. O trabalho demonstrou, de forma prática, como diferentes estratégias de distribuição de tarefas impactam no desempenho de simulações científicas, e reforça a importância de uma escolha adequada de parâmetros de paralelismo para explorar ao máximo os recursos computacionais disponíveis.