Tarefa 12: Análise de Escalabilidade e Otimização Incremental

Aplicando os Modelos de Amdahl e Gustafson

Werbert Arles de Souza Barradas

Universidade Federal do Rio Grande do Norte (UFRN)
Disciplina de Programação Paralela - DCA3703

03 de outubro de 2025

Agenda

1. Teoria da Escalabilidade
2. Evolução das Implementações
3. Análise dos Heatmaps

Teoria da Escalabilidade: Lei de Amdahl

A Pergunta: Qual o speedup máximo para um problema de tamanho fixo?

A Lei de Amdahl foca em acelerar uma tarefa existente (Escalabilidade Forte).

A Ideia Central:

- O ganho de desempenho é **limitado pela porção serial** do código.
- Não importa quantos processadores você adicione, o tempo total nunca será menor que o tempo da parte sequencial.

A Fórmula:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

- $S(N)$: Speedup com N processadores
- P: Fração paralelizável do código
- N: Número de processadores

Conclusão de Amdahl

O speedup tem um limite máximo rígido, definido pela parte serial.

Teoria da Escalabilidade: Lei de Gustafson

A Pergunta: E se aumentarmos o problema junto com os processadores?

A Lei de Gustafson foca em resolver problemas **maiores** no mesmo tempo (Escalabilidade Fraca).

A Ideia Central:

- Para problemas grandes, a porção serial **se torna insignificante**.
- Com mais poder computacional, os cientistas aumentam a precisão ou o tamanho da simulação, não apenas rodam o problema antigo mais rápido.

A Fórmula:

$$S(N) = N - P \times (N - 1)$$

- $S(N)$: Speedup com N processadores
- P: Fração serial do código
- N: Número de processadores

Conclusão de Gustafson

O speedup pode escalar de forma quase linear com o número de processadores.

1. Versão Ingênua

Estratégia e Gargalo

Aplicar `#pragma omp parallel for/sections` em cada laço de trabalho de forma independente. **Gargalo:** Alto overhead de criar e destruir threads (fork/join) a cada etapa.

```
// Loop de tempo PRINCIPAL - SERIAL
for (int t = 0; t < NT; t++) {
    // 1. Cria e destr i threads aqui
    #pragma omp parallel for
    for (int i = 1; i < NX-1; i++) { /*...*/ }

    // 2. Cria e destr i threads novamente aqui
    #pragma omp parallel sections
    { /* Condicoes de contorno */ }
}
```

2. Otimização Final (com `collapse`)

Estratégia Final

Envolver todo o laço de tempo em uma região paralela única para eliminar o overhead de fork/join, e usar `collapse(2)` para otimizar o balanceamento de carga do kernel.

```
// Threads sao criadas apenas UMA VEZ aqui
#pragma omp parallel
{
    // Loop de tempo PRINCIPAL - PARALELO
    for (int t = 0; t < NT; t++) {
        // Distribuicao de trabalho com baixo custo
        #pragma omp for collapse(2) schedule(static)
        for (int i = 1; i < NX-1; i++) {
            for (int j = 1; j < NY-1; j++) { /*...*/ }
        }
        // ... demais diretivas work-sharing
    }
} // Threads sao destruidas apenas UMA VEZ aqui
```

Configuração do PaScal Analyzer

Uso do NPAD

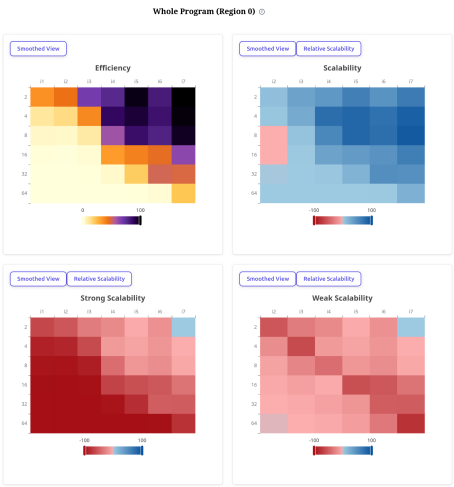
Foi configurado 1 nó (Amd 512) com 64 cores

Flags do Pascalanalyzer:

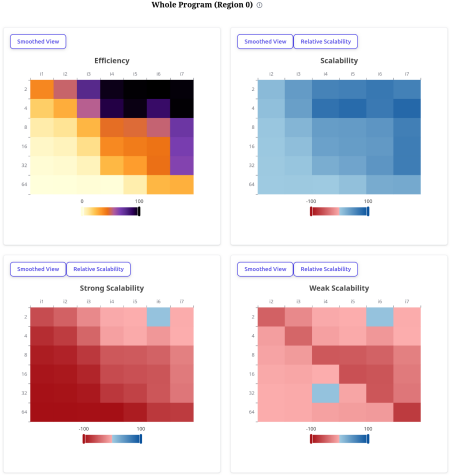
- -c 1,2,4,8,16,32,64
- -i 32,64,128,512,1024,2048,4096
- –inst aut -g -o OUTPUT.json

Análise: Comparativo Visual e Configurações

Versão Ingênua



Versão Otimizada



Análise: Diagnóstico Comparativo

Diagnóstico da Versão Ingênua

- **Causa:** O overhead massivo de *fork-join* a cada iteração do laço de tempo.
- **Efeito:** Eficiência quase nula e **escalabilidade forte negativa** (vermelho escuro), resultando em *slowdown*. O programa fica mais lento com mais processadores.

Diagnóstico da Versão Otimizada

- **Causa:** O overhead foi eliminado com uma região paralela única.
- **Efeito:** O *slowdown* desaparece. A escalabilidade forte, embora não seja perfeita (vermelho claro), mostra um comportamento saudável de **retornos decrescentes**, como previsto pela Lei de Amdahl.

Conclusões Finais

- A estratégia de maior impacto foi a **redução de overhead**, consolidando o trabalho dentro de uma **região paralela única**. Isso corrigiu a escalabilidade negativa.
- A otimização é um processo incremental: a remoção do gargalo do **modelo de programação** (fork-join) revelou os gargalos inerentes ao **algoritmo e hardware** (comunicação, acesso à memória).
- A cláusula `collapse(2)` foi importante para maximizar a eficiência do kernel, melhorando o balanceamento de carga para a computação na grade 2D.
- A **Lei de Amdahl** foi demonstrada na prática: após a otimização, o desempenho geral continuou limitado pela porção do código que não escalava perfeitamente.

~/github/projeto_PP_12/navier_stokes_paralelo_ingenua.c

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

#define NT 2000

// As funções auxiliares de alocação/liberação são as mesmas
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX;

    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização (pode ser paralela, não impacta muito a análise)
    #pragma omp parallel for
    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/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) {
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
            }
        }
    }

```

```c
    double start_time = omp_get_wtime();

    for (int step = 0; step < NT; step++) {

        // --- INÍCIO DA ABORDAGEM INGÊNUA ---

        // GARGALO 1: Cria e destrói um time de threads apenas para este laço.
        #pragma omp parallel for
        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]);

                u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 + d2u_dy2);
                v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 + d2v_dy2);
            }
        }

        // GARGALO 2: Cria e destrói OUTRO time de threads apenas para as seções.
        #pragma omp parallel sections
        {
            #pragma omp section
            { // Contorno Horizontal
                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];
                }
            }
            #pragma omp section
            { // Contorno Vertical
                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];
                }
            }
        }

        // A troca de ponteiros ocorre serialmente, pela thread mestre, após os joins.
        double **temp_u = u;
        double **temp_v = v;
        u = u_new;
        v = v_new;
        u_new = temp_u;
        v_new = temp_v;

```

```
102             // --- FIM DA ABORDAGEM INGÊNUA ---
103         }
104
105         double end_time = omp_get_wtime();
106         printf("%.6f\n", end_time - start_time);
107
108         free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);
109
110         return 0;
111     }
112
```

~/github/projeto_PP_12/navier_stokes_paralelo_otm2.c

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

// NT agora é uma constante interna.
#define NT 2000

// Funções auxiliares agora recebem as dimensões como parâmetros
double** allocate_grid(int nx, int ny) {
    double *data = (double*)malloc(nx * ny * sizeof(double));
    double **array = (double**)malloc(nx * sizeof(double*));
    for (int i = 0; i < nx; i++) {
        array[i] = &(data[i * ny]);
    }
    return array;
}

void free_grid(double** array) {
    free(array[0]);
    free(array);
}

int main(int argc, char *argv[]) {
    // Agora esperamos o tamanho da grade (NX) como argumento
    if (argc != 2) {
        fprintf(stderr, "Uso: %s <TAMANHO_DA_GRADE>\n", argv[0]);
        fprintf(stderr, "Exemplo: %s 512\n", argv[0]);
        return 1;
    }
    int NX = atoi(argv[1]);
    int NY = NX; // NY será sempre igual a NX

    // Alocação de memória usa as variáveis NX e NY
    double **u = allocate_grid(NX, NY);
    double **v = allocate_grid(NX, NY);
    double **u_new = allocate_grid(NX, NY);
    double **v_new = allocate_grid(NX, NY);

    // Inicialização usa as variáveis NX e NY
    #pragma omp parallel for
    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/2.0, dy = j - NY/2.0;
            double dist = sqrt(dx*dx + dy*dy);
            if (dist < (NX / 25.0)) { // Condição inicial relativa ao tamanho
da grade
                u[i][j] += 2.0 * exp(-dist*dist/(NX/5.0));
                v[i][j] += 1.5 * exp(-dist*dist/(NX/5.0));
```

```
            }
        }
    }

    double start_time = omp_get_wtime();

    #pragma omp parallel
    {
        for (int step = 0; step < NT; step++) {

            #pragma omp for collapse(2) schedule(static)
            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-1][j]);

                    u_new[i][j] = u[i][j] + (0.001 * 0.01) * (d2u_dx2 +
d2u_dy2); // DT e NU podem ser fixados
                    v_new[i][j] = v[i][j] + (0.001 * 0.01) * (d2v_dx2 +
d2v_dy2);
                }
            }

            #pragma omp for
            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];
            }

            #pragma omp for
            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];
            }

            #pragma omp single
            {
                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();
    printf("%.6f\n", end_time - start_time);

    free_grid(u); free_grid(v); free_grid(u_new); free_grid(v_new);

    return 0;
}

```