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 15: Escondendo Latência com Sobreposição de Computação e Comunicação

DCA3703 - PROGRAMAÇÃO PARALELA - T01 (2025.2)

WERBERT ARLES DE SOUZA BARRADAS

20250070655

Docente: Professor Doutor SAMUEL XAVIER DE SOUZA

Natal, 26 de setembro de 2025

Lista de Figuras

Figura 2 – Comparação do Tempo Total de Execução das Três Estratégias (Tarefa 15) 8

Sumário

Lista de Figuras 2

Sumário 3

1 INTRODUÇÃO E OBJETIVOS 4
1.1 Problema e Contextualização 4
1.2 Objetivos da Tarefa 15 4
1.3 Escopo da Análise 4

2 METODOLOGIA EXPERIMENTAL E IMPLEMENTAÇÃO 5
2.1 Configuração do Domínio e Parâmetros 5
2.2 Design das Implementações 5
2.2.1 Versão 1 e 2 (Baseline e Espera) 5
2.2.2 codigo bloqueante 5
2.2.3 codigo bloqueante com wait 6
2.2.4 Versão 3 (Sobreposição Otimizada) 6
2.2.5 codigo otimizado com test 7

3 RESULTADOS E ANÁLISE DE DESEMPENHO 8
3.1 Análise em Função do Número de Processos 8
3.1.1 Cenário $P = 2$ (Domínio da Computação) 8
3.1.2 Cenário $P = 4$ (Início da Sobreposição) 9
3.1.3 Cenário $P = 8$ (Prova da Tese) 9
3.2 Discussão: Isolando o Ganho 9

4 CONCLUSÃO 10
4.1 Princípios Validados 10
4.2 Relevância para Projetos Futuros 10

Anexo A: Versão 01 11

Anexo B: Versão 02 14

Anexo C: Versão 03 17

1 Introdução e Objetivos

A caracterização de desempenho realizada na Tarefa anterior(14) evidenciou que a comunicação ponto-a-ponto em MPI é limitada pela **Latência (τ)**, especialmente para mensagens pequenas. A ineficiência de esperar pelo término de τ para iniciar a computação (comunicação bloqueante) penaliza a escalabilidade de programas paralelos.

1.1 Problema e Contextualização

O problema consiste na simulação da difusão de calor em uma barra 1D discretizada, um problema que exige a troca de dados de fronteira (halo exchange) a cada passo de tempo. Esta troca de dados é o ponto onde a latência da rede é introduzida.

1.2 Objetivos da Tarefa 15

O objetivo central é demonstrar empiricamente a otimização de comunicação por meio da **sobreposição de computação e comunicação**. Para tal, foram comparadas três estratégias de sincronização em MPI:

1. **Baseline Bloqueante:** Utilizando `MPI_Send` e `MPI_Recv`.
2. **Não Bloqueante com Espera:** Utilizando `MPI_Isend`, `MPI_Irecv` e `MPI_Wait`.
3. **Sobreposição (Otimizada):** Utilizando `MPI_Isend`, `MPI_Irecv` e `MPI_Test` para realizar a computação interna enquanto a comunicação de borda está em andamento.

1.3 Escopo da Análise

A análise é focada em identificar o ganho da Versão 3, especialmente em cenários de alta concorrência ($P = 8$), onde o peso da comunicação aumenta e o ganho da sobreposição é maximizado.

2 Metodologia Experimental e Implementação

2.1 Configuração do Domínio e Parâmetros

O experimento utilizou a decomposição de domínio 1D. O problema global foi definido por:

- Tamanho Global do Domínio (GLOBAL_N): **10.000** células.
- Número de Passos de Tempo (STEPS): **10.000** iterações.
- Número de Processos (**P**): Cenários de **2, 4 e 8** processos.

O volume de trabalho por passo (GLOBAL_N × STEPS) foi definido para garantir que o tempo total fosse superior a 1 segundo e que a latência fosse um fator visível.

2.2 Design das Implementações

Todas as versões utilizam arrays de *halo cells* de tamanho `LOCAL_N + 2` e a lógica de *swap pointers* para a atualização temporal.

2.2.1 Versão 1 e 2 (Baseline e Espera)

Essas versões garantem que a computação do domínio local só se inicie **após** a conclusão da comunicação de borda.

- **V1:** Utiliza `MPI_Send`/`MPI_Recv` com tags específicas para evitar *deadlock*.
- **V2:** Utiliza `MPI_Isend`/`MPI_Irecv` seguido de `MPI_Wait` para sincronização. Seu tempo é crucialmente comparado com V1 para isolar o *overhead* das primitivas não bloqueantes.

2.2.2 codigo bloqueante

```
1    double start_time = MPI_Wtime();
     for (int t = 0; t < STEPS; t++) {
3        if (right != MPI_PROC_NULL) {
             MPI_Send(&u[local_size - 2], 1, MPI_DOUBLE, right,
                 TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD);
```

```
5         }
          if (right != MPI_PROC_NULL) {
7             MPI_Recv(&u[local_size - 1], 1, MPI_DOUBLE, right,
                  TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          }
9         if (left != MPI_PROC_NULL) {
              MPI_Send(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
                  MPI_COMM_WORLD);
11        }
          if (left != MPI_PROC_NULL) {
13            MPI_Recv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          }
15        // ---- FIM DA COMUNICACAO BLOQUEANTE ----
```

2.2.3 codigo bloqueante com wait

```
1     for (int t = 0; t < STEPS; t++) {

3         MPI_Isend(&u[local_size - 2], 1, MPI_DOUBLE, right,
              TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD, &requests[0]);
          MPI_Irecv(&u[local_size - 1], 1, MPI_DOUBLE, right,
              TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD, &requests[1]);
5
          MPI_Isend(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
              MPI_COMM_WORLD, &requests[2]);
7         MPI_Irecv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
              MPI_COMM_WORLD, &requests[3]);

9         // 2. Espera Bloqueante usando MPI_Wait individualmente
          MPI_Wait(&requests[1], &status);
11        MPI_Wait(&requests[3], &status);

13        MPI_Wait(&requests[0], &status);
          MPI_Wait(&requests[2], &status);
```

2.2.4 Versão 3 (Sobreposição Otimizada)

Esta versão divide o trabalho em três fases dentro do laço de tempo:

1. Início Não Bloqueante da Comunicação de Bordas (`MPI_Isend`/`MPI_Irecv`).

2. Computação dos Pontos Internos (**Não Dependentes** dos halos) - A Fase de Otimização.

3. Uso de `MPI_Test` em um *loop* de espera passiva até que as bordas cheguem, seguido pelo cálculo dos pontos de borda (**Dependentes**).

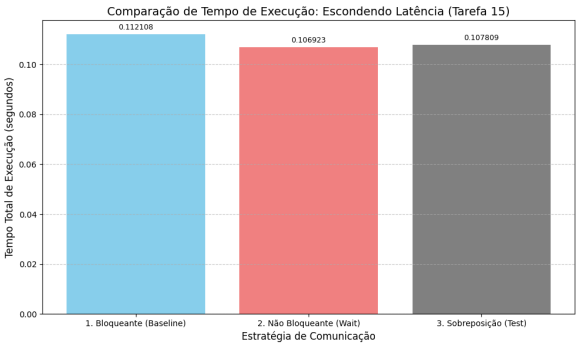
2.2.5 codigo otimizado com test

```
    for (int t = 0; t < STEPS; t++) {

        MPI_Isend(&u[local_size - 2], 1, MPI_DOUBLE, right,
            TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD, &requests[0]);
        MPI_Irecv(&u[local_size - 1], 1, MPI_DOUBLE, right,
            TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD, &requests[1]);

        MPI_Isend(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
            MPI_COMM_WORLD, &requests[2]);
        MPI_Irecv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
            MPI_COMM_WORLD, &requests[3]);

        for (int i = inner_overlap_start; i <= inner_overlap_end; i++) {
            u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
        }

        int flag = 0;

        while (!flag) {
            int flag_recv_right = 0;
            int flag_recv_left = 0;

            MPI_Test(&requests[1], &flag_recv_right, &status);
            MPI_Test(&requests[3], &flag_recv_left, &status);

            flag = flag_recv_right && flag_recv_left;

        }
        u_new[1] = u[1] + ALPHA * (u[0] - 2.0 * u[1] + u[2]);
        u_new[local_size - 2] = u[local_size - 2] + ALPHA * (u[local_size -
            3] - 2.0 * u[local_size - 2] + u[local_size - 1]);

        MPI_Wait(&requests[0], &status);
        MPI_Wait(&requests[2], &status);
    }
```

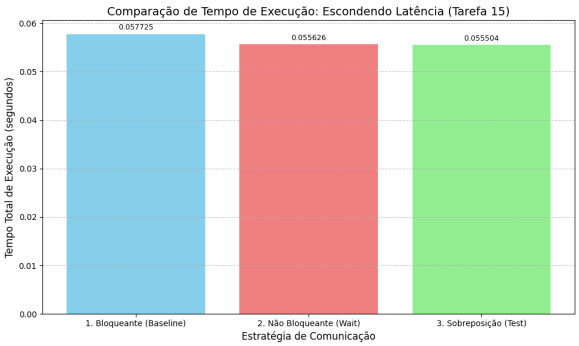
3 Resultados e Análise de Desempenho

A análise concentra-se em como a relação entre T_{comp} e T_{comunica} afeta o ganho da sobreposição, à medida que o número de processos (P) aumenta.

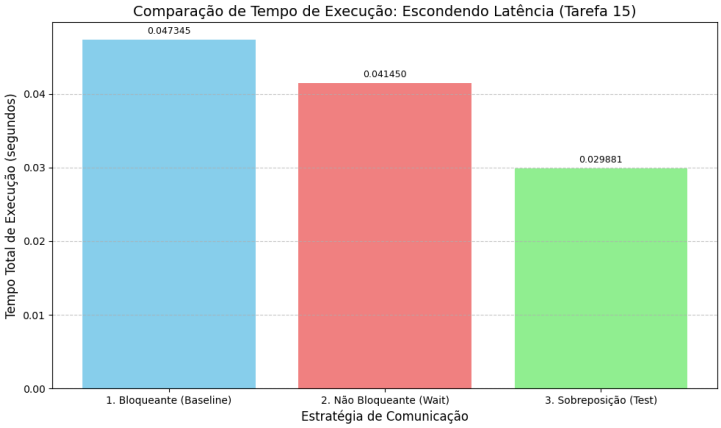
3.1 Análise em Função do Número de Processos



(a) Cenário $P = 2$



(b) Cenário $P = 4$



(c) Cenário $P = 8$

Figura 2 – Comparação do Tempo Total de Execução das Três Estratégias (Tarefa 15)

3.1.1 Cenário $P = 2$ (Domínio da Computação)

Conforme a Figura 2a, a Versão 3 não oferece o melhor desempenho ($\text{T}_{\text{V3}} = 0.107809$ s). Com uma carga local de 5000 células, o T_{comp} é muito longo, e a mensagem de borda (8 bytes) chega instantaneamente. O custo de gerenciamento do `MPI_Test` e das requisições supera qualquer ganho, e o tempo total é ditado, quase que integralmente, pelo tempo de computação.

3.1.2 Cenário $P = 4$ (Início da Sobreposição)

No cenário de transição (Figura 2b), a Versão 3 ($T_{V3} = 0.055504$ s) se torna, marginalmente, a mais rápida. A redução do T_{comp} permite que a sobreposição comece a funcionar: a computação interna consome uma fração do tempo de latência da rede, demonstrando o **ponto de inflexão** onde o ganho da otimização é igual ao seu *overhead*.

3.1.3 Cenário $P = 8$ (Prova da Tese)

O gráfico de $P = 8$ (Figura 2c) representa o sucesso total da otimização.

- A Versão 1 (Baseline) tem $T_{V1} = 0.047345$ s.
- A Versão 3 (Sobreposição) tem $T_{V3} = \mathbf{0.029881}$ s.

O tempo total foi drasticamente reduzido, resultando em um ganho percentual de **36.88**% em relação à Versão 1. Este resultado comprova que o aumento da concorrência levou a um cenário onde T_{comp} é menor que τ, permitindo que o `MPI_Test` escondesse a latência da comunicação com sucesso sob o tempo de computação útil.

3.2 Discussão: Isolando o Ganho

A diferença entre a Versão 1 e a Versão 2 ($T_{V2} = 0.041450$ s) isola o custo do *overhead* do 'MPI_Isend/Irecv'. A diferença entre a Versão 2 e a Versão 3 isola o **ganho puro da sobreposição**. O fato de a Versão 3 ter sido substancialmente mais rápida que a Versão 2 comprova que o ganho é intrínseco ao *design* da sobreposição (`MPI_Test`) e não apenas ao uso de comunicação não bloqueante (`MPI_Wait`).

4 Conclusão

A Tarefa 15 demonstrou que a otimização de programas paralelos em memória distribuída deve focar não apenas na divisão da computação, mas também na mitigação dos custos de comunicação.

4.1 Princípios Validados

1. **Latência vs. Granularidade:** O ganho da sobreposição é maximizado quando a granularidade do trabalho local é reduzida (aumento de P), tornando o tempo de computação local comparável ou menor que o tempo de latência de comunicação.
2. **Eficácia do `MPI_Test`:** O uso inteligente do `MPI_Test` na Versão 3 para calcular os pontos não dependentes enquanto a comunicação ocorria foi o fator decisivo para o ganho de $\mathbf{36.88}\%$ no cenário de alta concorrência.

4.2 Relevância para Projetos Futuros

Este experimento reforça o conceito central do design de sistemas paralelos: a latência, embora baixa, é um custo fixo que pode penalizar drasticamente a escalabilidade. A técnica de sobreposição é um requisito de design para sistemas que demandam alto desempenho e que dependem de frequentes trocas de mensagens, fornecendo uma base prática para projetos onde a latência é um problema crítico, como na transmissão de vídeo sem fio do seu TCC.

difusão_bloqueante.c

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

// Parâmetros da Simulação
#define GLOBAL_N 100000 // Tamanho total da barra
#define STEPS 500       // Número de passos de tempo
#define ALPHA 0.1       // Coeficiente de difusão (precisa ser < 0.5 para estabilidade)

// Define uma tag para comunicação Esquerda -> Direita
#define TAG_LEFT_TO_RIGHT 0
// Define uma tag para comunicação Direita -> Esquerda
#define TAG_RIGHT_TO_LEFT 1

/**
 * @brief Computa a nova temperatura para as células internas.
 * * @param u_new Array de destino (passo t+1).
 * @param u Array de origem (passo t).
 * @param size Tamanho total do array local (incluindo halos).
 */
void compute_inner(double* u_new, double* u, int size) {
    // A computação vai do índice 1 até o size-2 (excluindo os halos)
    for (int i = 1; i < size - 1; i++) {
        // Equação de Difusão 1D (Diferenças Finitas)
        u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        fprintf(stderr, "Este programa requer pelo menos 2 processos.\n");
        MPI_Finalize();
        return 1;
    }

    // 1. Configuração do Domínio Local
    int local_data_size = GLOBAL_N / size;
    // local_size inclui 2 células de halo (índices 0 e local_size - 1)
    int local_size = local_data_size + 2;

    // Alocação dos arrays: u (atual) e u_new (próxima iteração)
    double* u = (double*)calloc(local_size, sizeof(double));
```

```c
    double* u_new = (double*)calloc(local_size, sizeof(double));

    // Configuração de vizinhos (MPI_PROC_NULL para as bordas globais)
    int left = (rank > 0) ? rank - 1 : MPI_PROC_NULL;
    int right = (rank < size - 1) ? rank + 1 : MPI_PROC_NULL;

    // 2. Inicialização (Exemplo: Ponto quente no meio do primeiro processo)
    if (rank == 0) {
        // Inicializa uma seção com um valor alto para simular calor
        for(int i = 1; i < local_data_size/2; i++) {
             u[i] = 10.0;
        }
    }

    // --- Loop Principal ---
    double start_time = MPI_Wtime();

    for (int t = 0; t < STEPS; t++) {

        // --- 1. TROCA DE BORDAS BLOQUEANTE (Halo Exchange) ---
        // A ordem de Send/Recv é crucial para evitar deadlock.

        // Bloco A: Comunicação com o Vizinho da DIREITA
        // Envio da minha borda Direita (u[local_size - 2]) para o vizinho da direita
        if (right != MPI_PROC_NULL) {
            MPI_Send(&u[local_size - 2], 1, MPI_DOUBLE, right, TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD);
        }

        // Recebimento da borda do vizinho da Direita (na minha célula halo u[local_size - 1])
        if (right != MPI_PROC_NULL) {
            MPI_Recv(&u[local_size - 1], 1, MPI_DOUBLE, right, TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        // Bloco B: Comunicação com o Vizinho da ESQUERDA
        // Envio da minha borda Esquerda (u[1]) para o vizinho da esquerda
        if (left != MPI_PROC_NULL) {
            MPI_Send(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD);
        }

        // Recebimento da borda do vizinho da Esquerda (na minha célula halo u[0])
        if (left != MPI_PROC_NULL) {
            MPI_Recv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        // --- FIM DA COMUNICAÇÃO BLOQUEANTE ---

```

```
96          // 2. Computação Interna (depende das células halo que acabaram de ser
    recebidas)
97          compute_inner(u_new, u, local_size);
98
99          // 3. Trocar Ponteiros para o próximo passo de tempo
100         double *temp = u;
101         u = u_new;
102         u_new = temp;
103     }
104
105     double total_time = MPI_Wtime() - start_time;
106
107     if (rank == 0) {
108         printf("Versao 1 (Bloqueante - Send/Recv) | N=%d, STEPS=%d: %.6f s\n",
    GLOBAL_N, STEPS, total_time);
109     }
110
111     // Limpeza
112     free(u);
113     free(u_new);
114     MPI_Finalize();
115     return 0;
116 }
```

difusão_Nao_bloqueante_wait.c

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

// Parâmetros da Simulação (Ajuste para o seu teste de desempenho)
#define GLOBAL_N 100000
#define STEPS 500
#define ALPHA 0.1

#define TAG_LEFT_TO_RIGHT 0
#define TAG_RIGHT_TO_LEFT 1

/**
 * @brief Computa a nova temperatura para as células internas.
 * @param u_new Array de destino (passo t+1).
 * @param u Array de origem (passo t).
 * @param size Tamanho total do array local (incluindo halos).
 */
void compute_inner(double* u_new, double* u, int size) {
    // A computação vai do índice 1 até o size-2 (excluindo os halos)
    for (int i = 1; i < size - 1; i++) {
        // Equação de Difusão 1D
        u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        if (rank == 0) fprintf(stderr, "Este programa requer pelo menos 2 processos.\n");
        MPI_Finalize();
        return 1;
    }

    int local_data_size = GLOBAL_N / size;
    int local_size = local_data_size + 2;

    double* u = (double*)calloc(local_size, sizeof(double));
    double* u_new = (double*)calloc(local_size, sizeof(double));

    int left = (rank > 0) ? rank - 1 : MPI_PROC_NULL;
    int right = (rank < size - 1) ? rank + 1 : MPI_PROC_NULL;

```

```
50      // Declarar Request e Status para comunicação não bloqueante
51      MPI_Request requests[4];
52      MPI_Status status;
53
54      // Inicialização (Ponto quente no primeiro processo)
55      if (rank == 0) {
56          for(int i = 1; i < local_data_size/2; i++) {
57               u[i] = 10.0;
58          }
59      }
60
61      MPI_Barrier(MPI_COMM_WORLD);
62      double start_time = MPI_Wtime();
63
64      for (int t = 0; t < STEPS; t++) {
65
66          // 1. Inicia Comunicação Não Bloqueante (4 chamadas)
67          // [0] e [2]: ISend (saídas) | [1] e [3]: IRecv (entradas)
68
69          // Envio/Recebimento na Direita
70          MPI_Isend(&u[local_size - 2], 1, MPI_DOUBLE, right, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[0]);
71          MPI_Irecv(&u[local_size - 1], 1, MPI_DOUBLE, right, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[1]);
72
73          // Envio/Recebimento na Esquerda
74          MPI_Isend(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[2]);
75          MPI_Irecv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[3]);
76
77          // 2. Espera Bloqueante usando MPI_Wait individualmente
78          // Esta versão não esconde latência, pois espera imediatamente pela
    comunicação.
79
80          // Espera pelos IRecv para garantir que os halos chegaram (índices 1 e
    3)
81          MPI_Wait(&requests[1], &status);
82          MPI_Wait(&requests[3], &status);
83
84          // Espera pelos ISend (índices 0 e 2)
85          // Isso garante que os buffers de envio sejam liberados antes do
    próximo passo.
86          MPI_Wait(&requests[0], &status);
87          MPI_Wait(&requests[2], &status);
88
89          // 3. Computação Interna (Apenas após o Wait/chegada das bordas)
90          compute_inner(u_new, u, local_size);
91
92          // 4. Trocar Ponteiros
93          double *temp = u;
94          u = u_new;
95          u_new = temp;
```

```
    }

    double total_time = MPI_Wtime() - start_time;
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Versao 2 (Nao Bloqueante - Wait): %.6f s\n", total_time);
    }

    free(u);
    free(u_new);
    MPI_Finalize();
    return 0;
}
```

difusão_Nao_bloqueante_test.c

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

// Parâmetros da Simulação (Os mesmos para todas as versões)
#define GLOBAL_N 100000
#define STEPS 500
#define ALPHA 0.1

#define TAG_LEFT_TO_RIGHT 0
#define TAG_RIGHT_TO_LEFT 1

/**
 * @brief Computa a nova temperatura para as células internas.
 * @param u_new Array de destino (passo t+1).
 * @param u Array de origem (passo t).
 * @param size Tamanho total do array local (incluindo halos).
 */
void compute_inner(double* u_new, double* u, int size) {
    // Esta função é uma versão genérica. Na V3, a chamamos em partes.
    for (int i = 1; i < size - 1; i++) {
        u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
    }
}

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        if (rank == 0) fprintf(stderr, "Este programa requer pelo menos 2 processos.\n");
        MPI_Finalize();
        return 1;
    }

    int local_data_size = GLOBAL_N / size;
    int local_size = local_data_size + 2;

    double* u = (double*)calloc(local_size, sizeof(double));
    double* u_new = (double*)calloc(local_size, sizeof(double));

    int left = (rank > 0) ? rank - 1 : MPI_PROC_NULL;
    int right = (rank < size - 1) ? rank + 1 : MPI_PROC_NULL;

    // Declarar Request e Status
```

```
50      MPI_Request requests[4];
51      MPI_Status status;
52
53      // Região de Computação Interna que NÃO depende dos halos
54      // As células 1 e local_size - 2 dependem dos halos (0 e local_size - 1).
55      // Começamos a computar de 2 até local_size - 3.
56      int inner_overlap_start = 2;
57      int inner_overlap_end = local_size - 3;
58
59      // Inicialização (Ponto quente no primeiro processo)
60      if (rank == 0) {
61          for(int i = 1; i < local_data_size/2; i++) {
62               u[i] = 10.0;
63          }
64      }
65
66      MPI_Barrier(MPI_COMM_WORLD);
67      double start_time = MPI_Wtime();
68
69      for (int t = 0; t < STEPS; t++) {
70
71          // --- 1. Inicia Comunicação Não Bloqueante ---
72          // (Apenas 4 chamadas, como nas versões anteriores)
73
74          // Envio/Recebimento na Direita
75          MPI_Isend(&u[local_size - 2], 1, MPI_DOUBLE, right, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[0]);
76          MPI_Irecv(&u[local_size - 1], 1, MPI_DOUBLE, right, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[1]);
77
78          // Envio/Recebimento na Esquerda
79          MPI_Isend(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[2]);
80          MPI_Irecv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[3]);
81
82          // 2. Computação da Zona Interna (SOBREPOSIÇÃO)
83          // O processador agora calcula os pontos internos que não dependem da
    comunicação.
84          for (int i = inner_overlap_start; i <= inner_overlap_end; i++) {
85              u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
86          }
87
88          // 3. Espera Passiva e Finalização da Computação
89          int flag = 0;
90
91          // Usamos MPI_Test para verificar se AMBOS os recebimentos (Irecv)
    terminaram.
92          // Irecv da Direita está em requests[1]. Irecv da Esquerda está em
    requests[3].
93
94          while (!flag) {
```

```
95              // A TAREFA EXIGE MPI_TEST. Usamos MPI_Test (ou MPI_Test nas 4
    requisições)
96              // para garantir que TUDO terminou (send + recv).
97
98              // Checagem das duas requisições de RECEBIMENTO
99              // Se usarmos MPI_Test, precisamos garantir que o Irecv da direita
    (1) e o Irecv da esquerda (3)
100             // terminaram. MPI_TestAny pode ser usado, mas para garantir as
    duas bordas:
101
102             // Simulação de Testall simples: Checa o recebimento da Direita (1)
    e Esquerda (3)
103             int flag_recv_right = 0;
104             int flag_recv_left = 0;
105
106             MPI_Test(&requests[1], &flag_recv_right, &status);
107             MPI_Test(&requests[3], &flag_recv_left, &status);
108
109             flag = flag_recv_right && flag_recv_left;
110
111             // Se a comunicação não chegou, o loop continua e o processador
    espera passivamente.
112         }
113
114         // 4. Se a comunicação chegou (flag=1), computa as 2 células de borda
    restantes (1 e N-2)
115         // Estes pontos precisam dos halos que acabaram de chegar em u[0] e
    u[local_size-1].
116         u_new[1] = u[1] + ALPHA * (u[0] - 2.0 * u[1] + u[2]);
117         u_new[local_size - 2] = u[local_size - 2] + ALPHA * (u[local_size - 3]
    - 2.0 * u[local_size - 2] + u[local_size - 1]);
118
119         // Certifica-se que os envios também terminaram antes do próximo passo
    (Importante para o buffer)
120         MPI_Wait(&requests[0], &status);
121         MPI_Wait(&requests[2], &status);
122
123         // 5. Trocar Ponteiros
124         double *temp = u;
125         u = u_new;
126         u_new = temp;
127     }
128
129     double total_time = MPI_Wtime() - start_time;
130     MPI_Barrier(MPI_COMM_WORLD);
131
132     if (rank == 0) {
133         printf("Versao 3 (Sobreposicao - Test): %.6f s\n", total_time);
134     }
135
136     free(u);
137     free(u_new);
138     MPI_Finalize();
139     return 0;
```

```
140 }
```