Tarefa 15: Otimização de Comunicação em MPI

Escondendo Latência com Sobreposição de Computação

Werbert Arles de Souza Barradas

DCA3703 - Programação Paralela
UFRN - Universidade Federal do Rio Grande do Norte

Setembro de 2025

Objetivo e Estratégia do Experimento

Tese Central

Provar que a **Latência (τ)** da rede pode ser completamente mitigada realizando trabalho útil (computação) simultaneamente.

Metodologia:

- **Problema:** Difusão 1D (Troca de *Halos* a cada passo).
- **Configuração:** $N_{Global} = 1.000.000$, STEPS $= 10.000$, Coeficiênte de difusão $= 0.1$.
- **Otimização:** Comparar V1 (Bloqueante), V2(MPI_wait) com V3 (`MPI_Test`) em cenários de alta concorrência ($P = 8, 16, 32, 64$).

As Três Estratégias de Comunicação

V1 & V2: Bloqueio Total ou Parcial

1. **V1 (Baseline):** `MPI_Send` / `MPI_Recv`.
   Processo espera a rede.
2. **V2 (Wait):** `MPI_Isend/Irecv` + `MPI_Wait`.
   Overhead de requisição, mas ainda espera bloqueando.

V3: Sobreposição Otimizada

- **Primitivas:** `MPI_Isend/Irecv` + `MPI_Test`.
- **Mecanismo:** Inicia comunicação e imediatamente computa os pontos **não dependentes** dos halos, usando o tempo de latência da rede de forma produtiva.

Análise: Início da Eficiência ($P = 8$)

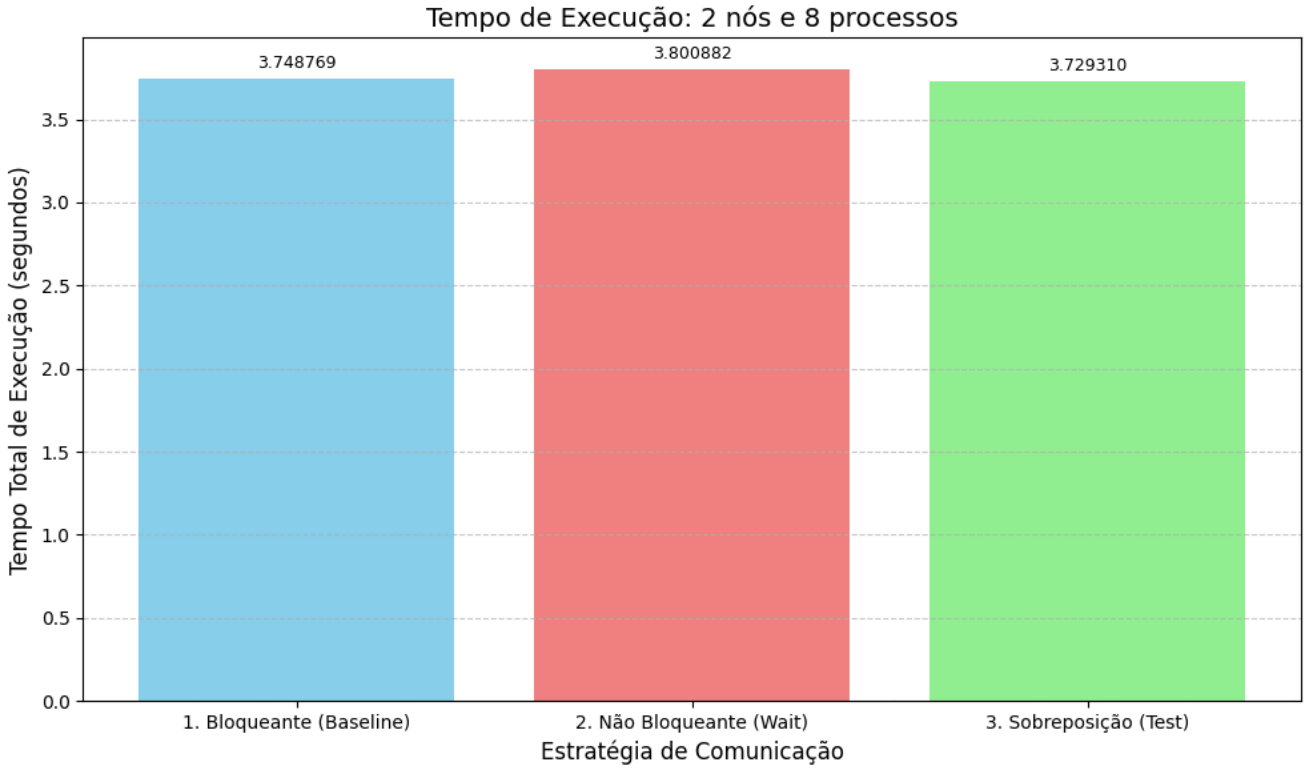

Figura: Grafico $P = 8$

- $P = 8$: A V3 não é a mais rápida. O T_{comp} é muito longo, e a Latência é insignificante, resultando em *overhead*.

Análise: Início da Eficiência ($P = 8$ vs. $P = 16$)

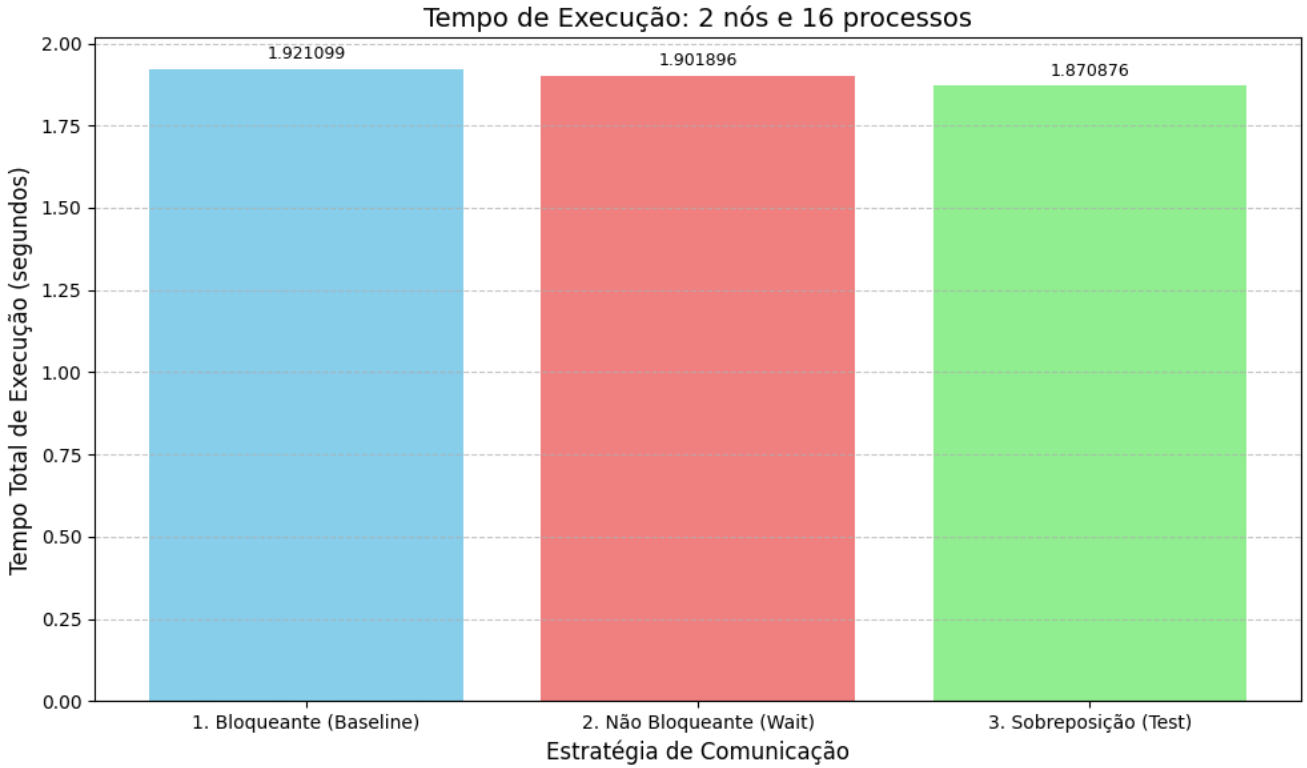

Figura: Grafico $P = 16$

- $P = 16$**:** A V3 (**1.070876** s) se torna, marginalmente, a melhor. A redução da carga local permite o início da **sobreposição efetiva**.

Prova Final: Domínio da Latência ($P = 32$)

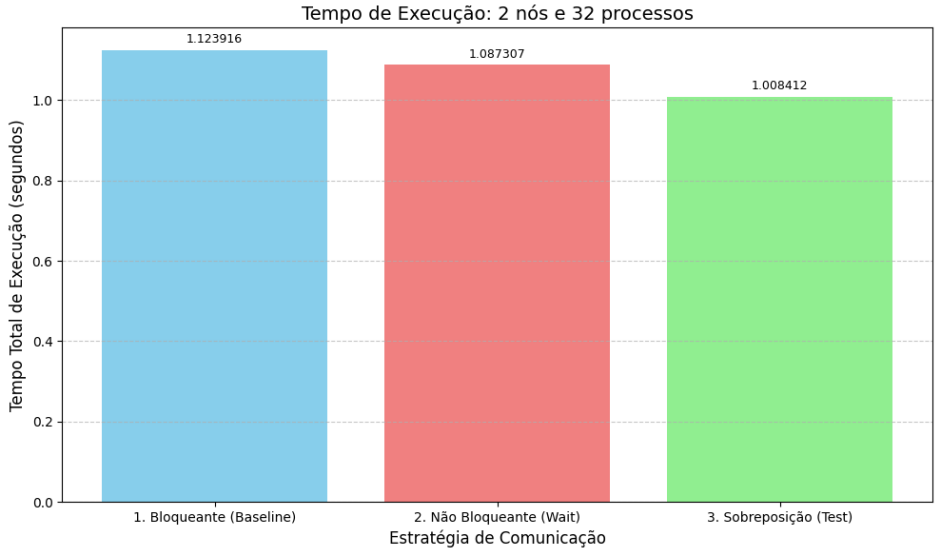

Figura: Desempenho no Cenário de Máxima Concorrência ($P = 32$)

Ganho da Sobreposição

- **V1 (Baseline):** 1.123916 s
- **V3 (Otimizada):** $\mathbf{1.008412}$ s

Ganho Total: $\approx \mathbf{10.28\%}$

Conclusão Final

- **Princípio Validado:** O ganho de $\mathbf{10.28\%}$ demonstra que, para otimizar a escalabilidade, é preciso fazer a latência da rede coincidir com o tempo de computação útil.
- **Condição Crítica:** A otimização máxima (**V3**) só se manifestou no cenário de **Latência Dominante** ($P = 32$), onde o trabalho de computação local é baixo e o custo da comunicação é alto.

difusao_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 5000       // Número de passos de tempo
#define ALPHA 0.1       // Coeficiente de difusão (precisa ser < 0.5 para estabilidade)


#define TAG_LEFT_TO_RIGHT 0

#define TAG_RIGHT_TO_LEFT 1

void compute_inner(double* u_new, double* u, int size) {

    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) {
        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;

    // Alocação dos arrays: u (atual) e u_new (próxima iteração)
    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;

```

```c

    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++) {

        if (right != MPI_PROC_NULL) {
            MPI_Send(&u[local_size - 2], 1, MPI_DOUBLE, right, TAG_RIGHT_TO_LEFT, MPI_COMM_WORLD);
        }


        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);
        }


        if (left != MPI_PROC_NULL) {
            MPI_Send(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT, MPI_COMM_WORLD);
        }


        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 ---


        compute_inner(u_new, u, local_size);

        // 3. Trocar Ponteiros para o próximo passo de tempo
        double *temp = u;
        u = u_new;
        u_new = temp;
    }

    double total_time = MPI_Wtime() - start_time;

    if (rank == 0) {
        printf("Versao 1 (Bloqueante - Send/Recv) | N=%d, STEPS=%d: %.6f s\n", GLOBAL_N, STEPS, total_time);
    }
```

```
 98
 99      // Limpeza
100      free(u);
101      free(u_new);
102      MPI_Finalize();
103      return 0;
104  }
105
```

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 1000000
#define STEPS 10000
#define ALPHA 0.1

#define TAG_LEFT_TO_RIGHT 0
#define TAG_RIGHT_TO_LEFT 1


void compute_inner(double* u_new, double* u, int size) {

    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 para comunicação não bloqueante
    MPI_Request requests[4];
    MPI_Status status;

    // Inicialização (Ponto quente no primeiro processo)
```

```c
    if (rank == 0) {
        for(int i = 1; i < local_data_size/2; i++) {
             u[i] = 10.0;
        }
    }

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

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

        // 1. Inicia Comunicação Não Bloqueante (4 chamadas)
        // [0] e [2]: ISend (saídas) | [1] e [3]: IRecv (entradas)

        // Envio/Recebimento na Direita
        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]);

        // Envio/Recebimento na Esquerda
        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]);

         // Espera pelos ISend (índices 1 e 3)
        MPI_Wait(&requests[1], &status);
        MPI_Wait(&requests[3], &status);

        // Espera pelos ISend (índices 0 e 2)
        MPI_Wait(&requests[0], &status);
        MPI_Wait(&requests[2], &status);

        // 3. Computação Interna (Apenas após o Wait/chegada das bordas)
        compute_inner(u_new, u, local_size);

        // 4. Trocar Ponteiros
        double *temp = u;
        u = u_new;
        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 1000000
#define STEPS 10000
#define ALPHA 0.1

#define TAG_LEFT_TO_RIGHT 0
#define TAG_RIGHT_TO_LEFT 1


void compute_inner(double* u_new, double* u, int size) {

    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;


    MPI_Request requests[4];
    MPI_Status status;


    int inner_overlap_start = 2;
```

```
50      int inner_overlap_end = local_size - 3;
51
52
53      if (rank == 0) {
54          for(int i = 1; i < local_data_size/2; i++) {
55               u[i] = 10.0;
56          }
57      }
58
59      MPI_Barrier(MPI_COMM_WORLD);
60      double start_time = MPI_Wtime();
61
62      for (int t = 0; t < STEPS; t++) {
63
64
65
66          // Envio/Recebimento na Direita
67          MPI_Isend(&u[local_size - 2], 1, MPI_DOUBLE, right, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[0]);
68          MPI_Irecv(&u[local_size - 1], 1, MPI_DOUBLE, right, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[1]);
69
70          // Envio/Recebimento na Esquerda
71          MPI_Isend(&u[1], 1, MPI_DOUBLE, left, TAG_LEFT_TO_RIGHT,
    MPI_COMM_WORLD, &requests[2]);
72          MPI_Irecv(&u[0], 1, MPI_DOUBLE, left, TAG_RIGHT_TO_LEFT,
    MPI_COMM_WORLD, &requests[3]);
73
74          // 2. Computação da Zona Interna (SOBREPOSIÇÃO)
75          // O processador agora calcula os pontos internos que não dependem da
    comunicação.
76          for (int i = inner_overlap_start; i <= inner_overlap_end; i++) {
77              u_new[i] = u[i] + ALPHA * (u[i-1] - 2.0 * u[i] + u[i+1]);
78          }
79
80          int flag = 0;
81
82          while (!flag) {
83
84              int flag_recv_right = 0;
85              int flag_recv_left = 0;
86
87              MPI_Test(&requests[1], &flag_recv_right, &status);
88              MPI_Test(&requests[3], &flag_recv_left, &status);
89
90              flag = flag_recv_right && flag_recv_left;
91
92          }
93
94          u_new[1] = u[1] + ALPHA * (u[0] - 2.0 * u[1] + u[2]);
95          u_new[local_size - 2] = u[local_size - 2] + ALPHA * (u[local_size - 3]
    - 2.0 * u[local_size - 2] + u[local_size - 1]);
96
```

```
97          // Certifica-se que os envios também terminaram antes do próximo passo
    (Importante para o buffer)
98          MPI_Wait(&requests[0], &status);
99          MPI_Wait(&requests[2], &status);
100
101         // 5. Trocar Ponteiros
102         double *temp = u;
103         u = u_new;
104         u_new = temp;
105     }
106
107     double total_time = MPI_Wtime() - start_time;
108     MPI_Barrier(MPI_COMM_WORLD);
109
110     if (rank == 0) {
111         printf("Versao 3 (Sobreposicao - Test): %.6f s\n", total_time);
112     }
113
114     free(u);
115     free(u_new);
116     MPI_Finalize();
117     return 0;
118 }
119
```