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 }
```