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 }
```