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;
```

```c
    }

    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;
}
```