

## Programa 1: Mais simples e menos otimizado

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE 100000 //Tamanho do array
#define N_ARRAYS 1000 // Quantidade de arrays
#define MASTER 0 // id do mestre
#define POISON_PILL -2
#define FIRST_TASK -1
#define ARRAY_MSG 2 // tipo de mensagem que transmite um array
#define INDEX_MSG 1 // tipo de mensagem que transmite um índice

/* Função de que é usado pelo qsort */
int cmpfunc (const void * a, const void * b){
    return ( *(int*)a - *(int*)b );
}

/* Método onde é o escravo que requisita a tarefa */
int main(int argc, char **argv){

    int n_tasks, task_id, exit_code, i, j, index, worker, array_to_send = 0, recv_arrays = 0, task_executed = 0;
    double t1, t2;
    MPI_Status mpi_status;
    exit_code = MPI_Init(&argc,&argv);
    exit_code|= MPI_Comm_size(MPI_COMM_WORLD,&n_tasks);
    exit_code|= MPI_Comm_rank(MPI_COMM_WORLD,&task_id);

    if (exit_code != MPI_SUCCESS) {
        printf ("Error initializing MPI and obtaining task ID information\n");
        return 1;
    }

    if (task_id == MASTER){
        // ===== MESTRE =====
        t1 = MPI_Wtime();
        // Aloca as matrizes
        int (*bag_of_tasks)[ARRAY_SIZE] = malloc (N_ARRAYS * sizeof *bag_of_tasks);
        int (*results)[ARRAY_SIZE] = malloc (N_ARRAYS * sizeof *results);

        // populando nros invertidos
        for (i = 0; i < N_ARRAYS; i++){
            for(j=0; j < ARRAY_SIZE; j++){
                bag_of_tasks [i][j] = (ARRAY_SIZE-j-1);
            }
        }

        // Delega enquanto tem arrays para receber
        while(recv_arrays < N_ARRAYS) {
            for (worker = 1; worker < n_tasks; worker++){ // Recebe as requisições de trabalho...
                if (recv_arrays < N_ARRAYS) {
                    MPI_Recv(&index, 1, MPI_INT, worker, INDEX_MSG, MPI_COMM_WORLD, &mpi_status);
                    if (index != FIRST_TASK){
                        MPI_Recv(&results[index], ARRAY_SIZE, MPI_INT, worker, ARRAY_MSG, MPI_COMM_WORLD, &mpi_status);
                        ;
                        recv_arrays++;
                    }
                }
            }
        }
        // ... e envia mais vetores
        for(worker = 1; worker < n_tasks; worker++){
            if (array_to_send < N_ARRAYS){
                MPI_Send(&array_to_send, 1, MPI_INT, worker, INDEX_MSG, MPI_COMM_WORLD);
                MPI_Send(&bag_of_tasks[array_to_send], ARRAY_SIZE, MPI_INT, worker, ARRAY_MSG, MPI_COMM_WORLD);
                array_to_send++;
            }
        }

        index = POISON_PILL; // enviando POISON PILL para matar os escravos
        for (worker = 1; worker < n_tasks; worker++){
            MPI_Send(&index, 1, MPI_INT, worker, INDEX_MSG, MPI_COMM_WORLD);
        }

        free(bag_of_tasks);
        free(results);
        t2 = MPI_Wtime();
        printf("[Master] Duration [%f]\n", t2-t1);
    } else {
```

```

t1 = MPI_Wtime();
// ===== SLAVE =====
int alive = 1;
int array[ARRAY_SIZE];
int index = FIRST_TASK;

MPI_Send(&index, 1, MPI_INT, MASTER, INDEX_MSG, MPI_COMM_WORLD); // Ja pede uma tarefa de inicio
do {
    MPI_Recv(&index, 1, MPI_INT, MASTER, 1, MPI_COMM_WORLD, &mpi_status);
    if (index == POISON_PILL) {
        //printf("[WORKER %d] Received POISON_PILL, ARGH!\n", task_id);
        alive = 0;
    } else {
        //printf("[WORKER %d] Received vector %d!\n", task_id, index);
        MPI_Recv(&array, ARRAY_SIZE, MPI_INT, MASTER, ARRAY_MSG, MPI_COMM_WORLD, &mpi_status); // 0 escravo
            recebe seu vetor, ...
        qsort(array, ARRAY_SIZE, sizeof(int), cmpfunc); // ... trabalha...
        task_executed++;
        MPI_Send(&index, 1, MPI_INT, MASTER, INDEX_MSG, MPI_COMM_WORLD); // ... e envia para o mestre
        MPI_Send(&array, ARRAY_SIZE, MPI_INT, MASTER, ARRAY_MSG, MPI_COMM_WORLD);
    }
} while (alive != 0);

t2 = MPI_Wtime();
printf("[Worker %d] Duration [%f] - Tasks [%d]\n", task_id, t2-t1, task_executed);

}

MPI_Finalize();

}

```

Programa 2: Mestre recebendo sem precisar saber a origem (mpi any source)

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE 100000 //Tamanho do array
#define N_ARRAYS 1000 // Quantidade de arrays
#define MASTER 0 // id do mestre
#define POISON_PILL -2
#define FIRST_TASK -1
#define ARRAY_MSG 2 // tipo de mensagem que transmite um array

/* Função de que é usado pelo qsort */
int cmpfunc (const void * a, const void * b){
    return ( *(int*)a - *(int*)b );
}

/* Método onde é o escravo que requisita a tarefa */
int main(int argc, char **argv){

    int n_tasks, task_id, exit_code, i, j, index, worker, array_to_send = 0, received_arrays = 0,
        task_executed = 0, sending = 1;
    int msg_size = (ARRAY_SIZE+1); // tamanho da mensagem trafegada entre os processos
    int index_pos = msg_size -1; // posição onde estará o índice do vetor
    double t1, t2; // tempos para medição de duração de execuções
    MPI_Status mpi_status;

    exit_code = MPI_Init(&argc,&argv);
    exit_code|= MPI_Comm_size(MPI_COMM_WORLD,&n_tasks);
    exit_code|= MPI_Comm_rank(MPI_COMM_WORLD,&task_id);

    if (exit_code != MPI_SUCCESS) {
        printf ("Error initializing MPI and obtaining task ID information\n");
        return 1;
    }

    if (task_id == MASTER){
        // ===== MESTRE =====
        t1 = MPI_Wtime();
        // Aloca as matrizes, com a última posição reservada para o índice
        int (*bag_of_tasks)[msg_size] = malloc (N_ARRAYS * sizeof *bag_of_tasks);
        int (*results)[msg_size] = malloc (N_ARRAYS * sizeof *results);
        //buffer usado para transmissão de mensagem entre mestre e escravos
        int buffer[msg_size];
        // Define a última posição como identificador do array
        for (i=0; i < N_ARRAYS; i++){
            bag_of_tasks[i][index_pos] = i;
        }
        // populando nros invertidos
        for (i = 0; i < N_ARRAYS; i++){
            for(j = 0; j < (msg_size-1); j++){ //última posição reservada
                bag_of_tasks [i][j] = (ARRAY_SIZE-j)*(i+1);
            }
        }

        // Delega enquanto tem arrays para receber
        while(sending != 0) {
            MPI_Recv(&buffer, msg_size, MPI_INT, MPI_ANY_SOURCE, ARRAY_MSG, MPI_COMM_WORLD, &mpi_status);
            index = buffer[index_pos];
            if (index != FIRST_TASK){
                // Copia o resultado retornado do escravo pro saco de tarefas
                memcpy(results[index], buffer, msg_size * sizeof(int));
                received_arrays++; // registra o recebimento de um array
            }
            // Verifica se já enviou todos
            if (array_to_send < N_ARRAYS){
                MPI_Send(&bag_of_tasks[array_to_send], msg_size, MPI_INT, mpi_status.MPI_SOURCE, ARRAY_MSG,
                    MPI_COMM_WORLD);
                array_to_send++;
            }
            // Se o mestre já recebeu todos os vetores, então para o processo de envio
            if (received_arrays == N_ARRAYS) sending = 0;
        }

        buffer[index_pos] = POISON_PILL; // enviando POISON PILL para matar os escravos
        for (worker = 1; worker < n_tasks; worker++){
            MPI_Send(&buffer, msg_size, MPI_INT, worker, ARRAY_MSG, MPI_COMM_WORLD);
        }

        t2 = MPI_Wtime();
    }
}
```

```

    printf("[Master] Duration [%f]\n", t2-t1);
    free(bag_of_tasks);
    free(results);
} else {

    t1 = MPI_Wtime();
    // ===== SLAVE =====
    int alive = 1;
    int worker_buffer[msg_size];
    worker_buffer[index_pos] = FIRST_TASK;
    MPI_Send(&worker_buffer, msg_size, MPI_INT, MASTER, ARRAY_MSG, MPI_COMM_WORLD); // Já pede uma tarefa de
        início
    do {
        MPI_Recv(&worker_buffer, msg_size, MPI_INT, MASTER, ARRAY_MSG, MPI_COMM_WORLD, &mpi_status);
        index = worker_buffer[index_pos];
        if (index == POISON_PILL) {
            alive = 0;
        } else {
            // lembrando que a última posição é o índice, ou seja, não deve ser usado no algoritmo de ordenação
            qsort(worker_buffer, (msg_size-1), sizeof(int), cmpfunc); // ... trabalha...
            task_executed++;
            MPI_Send(&worker_buffer, msg_size, MPI_INT, MASTER, ARRAY_MSG, MPI_COMM_WORLD);
        }
    } while (alive != 0);

    t2 = MPI_Wtime();
    printf("[WORKER %d] Duration [%f] - Tasks [%d]\n", task_id, t2-t1, task_executed);
}
MPI_Finalize();
}

```