RESPOSTA CHECAGEM_01 - MPI EM LABORATÓRIO

- **Professor:** Luis Vinicius Costa Silva
- **Aluno:** João Vitor de Souza Gonçalves
- **Data de avaliação:** 05/08/2025

1.1

```
Processo 0 enviando dado: 42 para processo 1
Processo 1 recebeu dado: 42 do processo 0
```

2.1

O `MPI_Scatter`, apesar de não estar sendo utilizado no código base enviado, trata-se de um MPI responsável por distribuir partes de um array de dados do processo para todos os processos de comunicação. Isso significa que, caso o root apresente um array com 20 partes, cada processo recebe uma parte específica desse array. Basicamente ele recebe o vetor e o divide entre todos os processos. No código base da atividade é feito o uso de `MPI_Send` e `MPI_Recv` para enviar um dado do processo 0 para o processo 1.

2.2

Ao contrário do MPI_Scatter, o `MPI_Gather` é o comando de mpi responsável por receber os dados de vários processos em um único processo.

2.3

Isso ocorre pois somente o processo 0 tem todos os dados necessários para que seja realizado a ordenação.

3.1

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;
    int dado;
    int resultado[4]; // usado apenas no root

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    dado = rank * 10;

    MPI_Gather(&dado, 1, MPI_INT, resultado, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
```

```
        printf("Processo 0 reuniu os dados: ");
        for (int i = 0; i < 4; i++) {
            printf("%d ", resultado[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

3.2

```
Processo 0 reuniu os dados: 0 10 20 30
```

4.1

Para mensurar o tempo de execução de cada processo, foram utilizados os seguintes comandos:

```
# Comandos separados individualmente
time mpirun --oversubscribe -np 2 ./media_mpi_mod
time mpirun --oversubscribe -np 4 ./media_mpi_mod
time mpirun --oversubscribe -np 6 ./media_mpi_mod

# Comandos unidos
time mpirun --oversubscribe -np 2 ./media_mpi_mod && time mpirun --oversubscribe -np 4
./media_mpi_mod && time mpirun --oversubscribe -np 6 ./media_mpi_mod
```

> Resposta Adquirida:

```
Processo 0 reuniu os dados: 0 10 0 0

real    0m1.087s
user    0m0.288s
sys     0m0.575s
Processo 0 reuniu os dados: 0 10 20 30

real    0m0.407s
user    0m0.192s
sys     0m0.225s
Processo 0 reuniu os dados: 0 10 20 30

real    0m0.432s
user    0m0.293s
sys     0m0.337s
```

Aqui está a tabela com os tempos registrados:

Processos	Tempo (real)
2	1.087s

4	0.407s
6	0.432s

4.2

Sim, o uso de CPU foi balanceado entre os processos. Durante a execução do programa com `mpirun -np 4`, os quatro processos apresentaram uso semelhante de CPU verificado no `htop/top`, indicando que a carga foi distribuída de forma equilibrada.

Isso é esperado nesse caso, pois cada processo apenas realiza uma operação simples e participa igualmente da chamada `MPI_Gather`. Não há nenhuma computação pesada ou espera ativa significativa em nenhum processo.

4.3

- Com **MPI_Wtime** Utilizei `MPI_Wtime` para medir o tempo gasto na chamada `MPI_Gather`. Notei que, mesmo sendo uma operação coletiva, o tempo de execução foi muito pequeno e bem semelhante entre os processos. Isso mostra que, para um volume pequeno de dados, a sobrecarga de comunicação é mínima.
- Com **taskset** Com o `taskset`, notou-se que o uso de CPU foi bem distribuído entre os núcleos, sem migração de processos entre CPUs, o que pode melhorar desempenho em execuções maiores.
- Com **strace** Com o uso de `strace`, foi possível perceber que o comando faz chamadas de sistema relacionadas à comunicação, principalmente durante a chamada a `MPI_Gather`. Isso confirma que a biblioteca MPI está utilizando chamadas de baixo nível para sincronizar os processos e mover os dados.