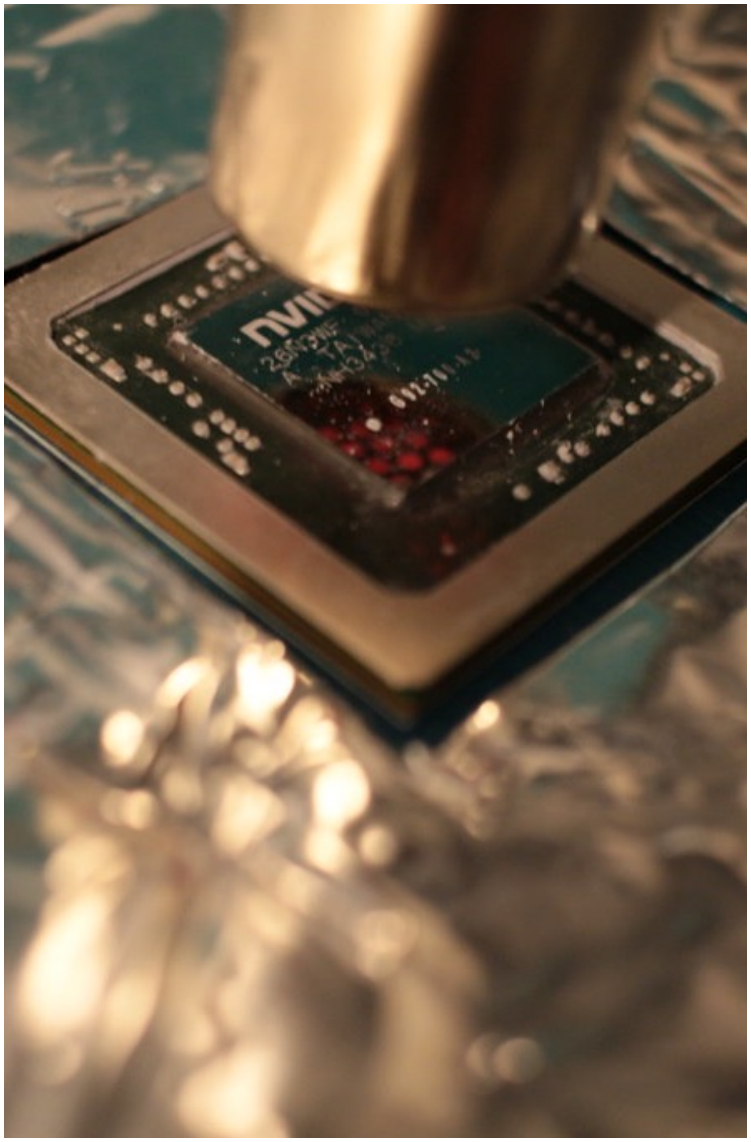
Insper

Supercomputação



MPI

- Scatter e Gather

Revisitando

- Broadcast

```
MPI_Bcast(buffer, count, datatype, root, comm)
```

buffer
data to be distributed

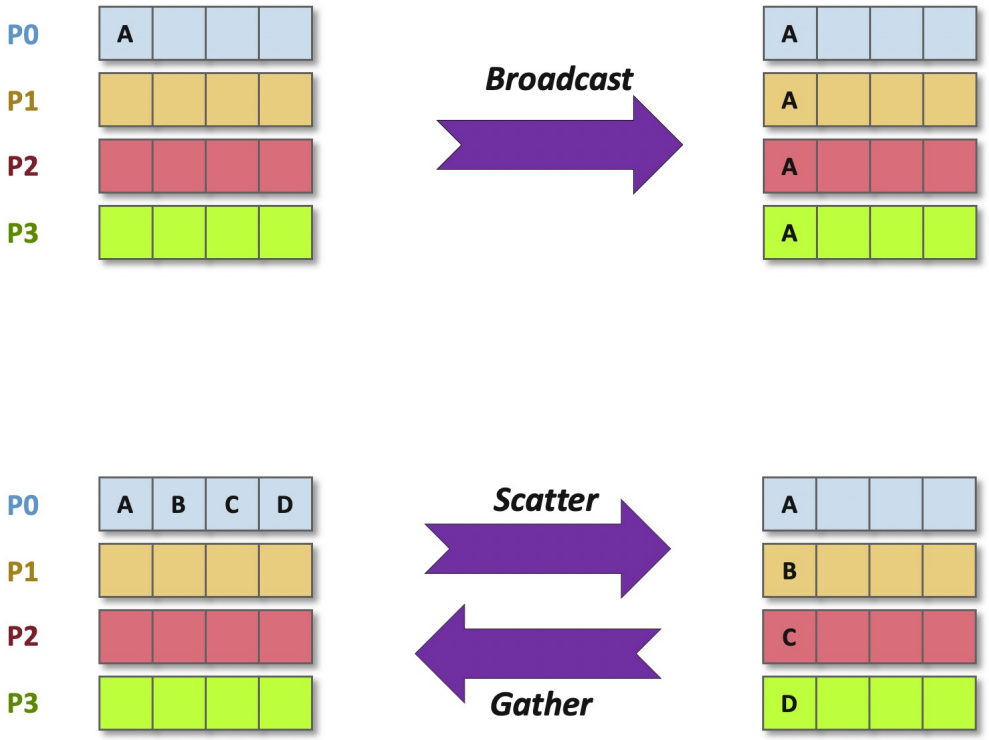
count
number of entries in buffer

datatype
data type of buffer

root
rank of broadcast root

comm
communicator

Broadcast vs Scatter / Gather



Scatter

- Task root sends an equal share of data to all other processes

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount,
recvtype, root, comm)
```

sendbuf
send buffer (data to be scattered)

sendcount
number of elements sent to each process

sendtype
data type of send buffer elements

recvbuf
receive buffer

recvcount
number of elements to receive at each process

recvtype
data type of receive buffer elements

root
rank of sending process

comm
communicator

Exemplo de Código SEND / RECV

- Enviando um vetor

```cpp
#include <iostream>
#include <mpi.h>

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);

    const int ARRAY_SIZE = 100;
    int array[ARRAY_SIZE];

    if (rank == 0) {
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i;
        }
    }

    int chunk_size = ARRAY_SIZE / size;
    int recv_array[chunk_size];

    if (rank == 0) {
        for (int i = 1; i < size; i++) {
            MPI_Send(&array[i * chunk_size], chunk_size, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
        MPI_Recv(recv_array, chunk_size, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Imprime o vetor recebido
        std::cout << "Nó " << rank << " recebeu: ";
        for (int i = 0; i < chunk_size; i++) {
            std::cout << recv_array[i] << " ";
        }
        std::cout << std::endl;
    }

    MPI_Finalize();
    return 0;
}
```

Exemplo de Código SCATTER

- Enviando um vetor

```cpp
#include <iostream>
#include <mpi.h>

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);

    const int ARRAY_SIZE = 100;
    int array[ARRAY_SIZE];

    if (rank == 0) {
        for (int i = 0; i < ARRAY_SIZE; i++) {
            array[i] = i;
        }
    }

    int chunk_size = ARRAY_SIZE / size;
    int recv_array[chunk_size];

    MPI_Scatter(array, chunk_size, MPI_INT, recv_array, chunk_size, MPI_INT, 0, MPI_COMM_WORLD);

    // Imprime o vetor recebido
    std::cout << "Nó " << rank << " recebeu: ";
    for (int i = 0; i < chunk_size; i++) {
        std::cout << recv_array[i] << " ";
    }
    std::cout << std::endl;

    MPI_Finalize();
    return 0;
}
```

MPI e OpenMP

- **Tarefa:** Calcular o quadrado de cada elemento em um array bidimensional.
- **MPI:** Divide o array entre diferentes processos.
- **OpenMP:** Paraleliza o cálculo dentro de cada processo.

Exemplo de código

```cpp
#include <iostream>
#include <mpi.h>
#include <omp.h>

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);

    const int N = 10;  // Dimensões do array bidimensional
    int data[N][N];

    // Inicialização do array pelo processo 0
    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                data[i][j] = i + j;
            }
        }
    }
```

```cpp
    // Dividir o array entre os processos
    int chunk_size = N / size;
    int local_data[chunk_size][N];
    MPI_Scatter(data, chunk_size * N, MPI_INT, local_data, chunk_size * N, MPI_INT, 0, MPI_COMM_WORLD);

    // Paralelização com OpenMP
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < chunk_size; i++) {
        for (int j = 0; j < N; j++) {
            local_data[i][j] *= local_data[i][j];  // Calcula o quadrado do elemento
        }
    }

    // Reunir os resultados no processo 0
    MPI_Gather(local_data, chunk_size * N, MPI_INT, data, chunk_size * N, MPI_INT, 0, MPI_COMM_WORLD);

    // Processo 0 imprime os resultados
    if (rank == 0) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                std::cout << data[i][j] << " ";
            }
            std::cout << std::endl;
        }
    }

    MPI_Finalize();
    return 0;
}
```