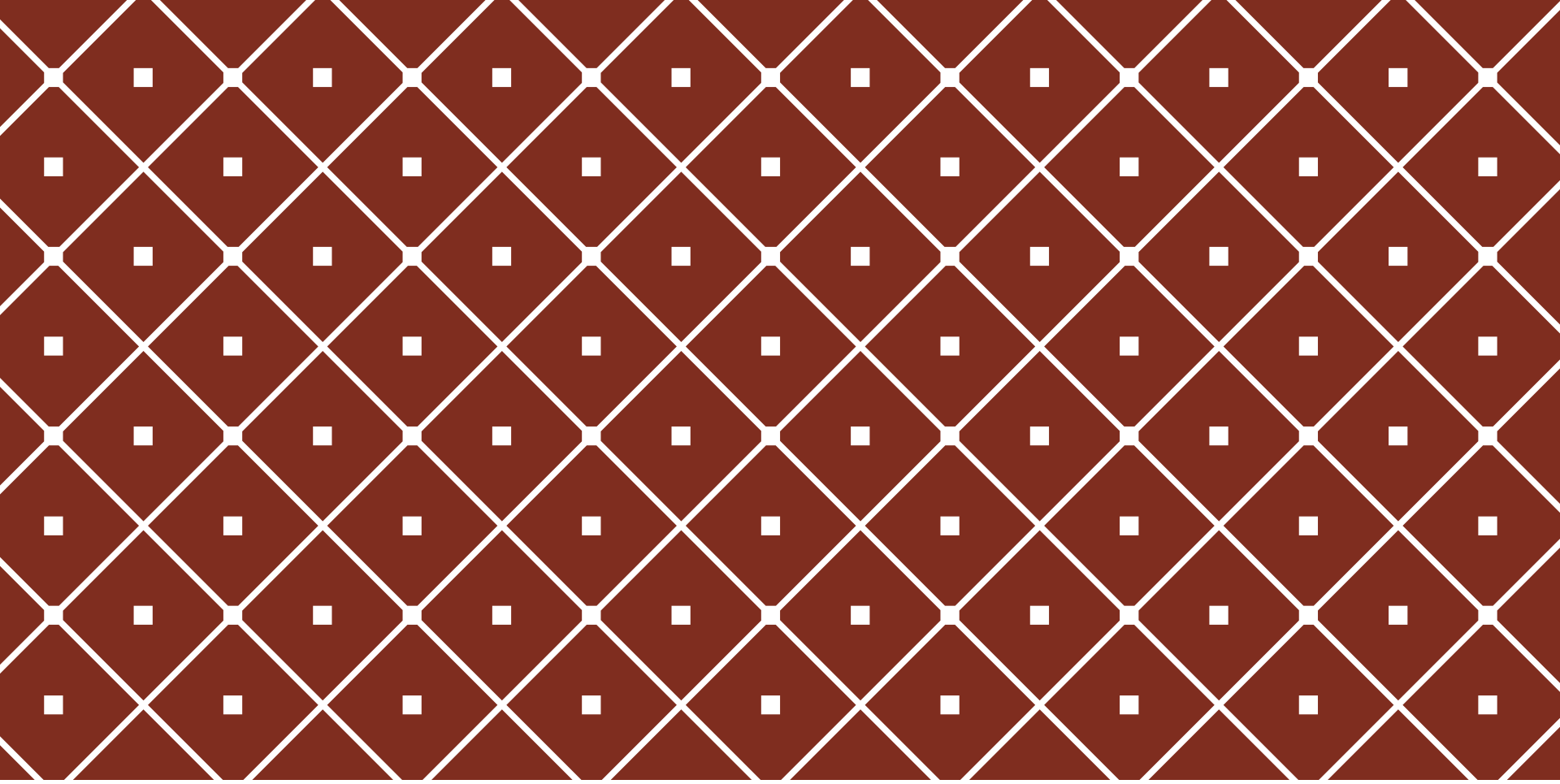


PROGRAMAÇÃO PARALELA
MPI 03 – COMUNICAÇÕES COLETIVAS

Marco A. Zanata Alves

Slides baseados nos slides de Ricardo Rocha – DCC FCUP (2009/2010)
Programação Paralela e Distribuída

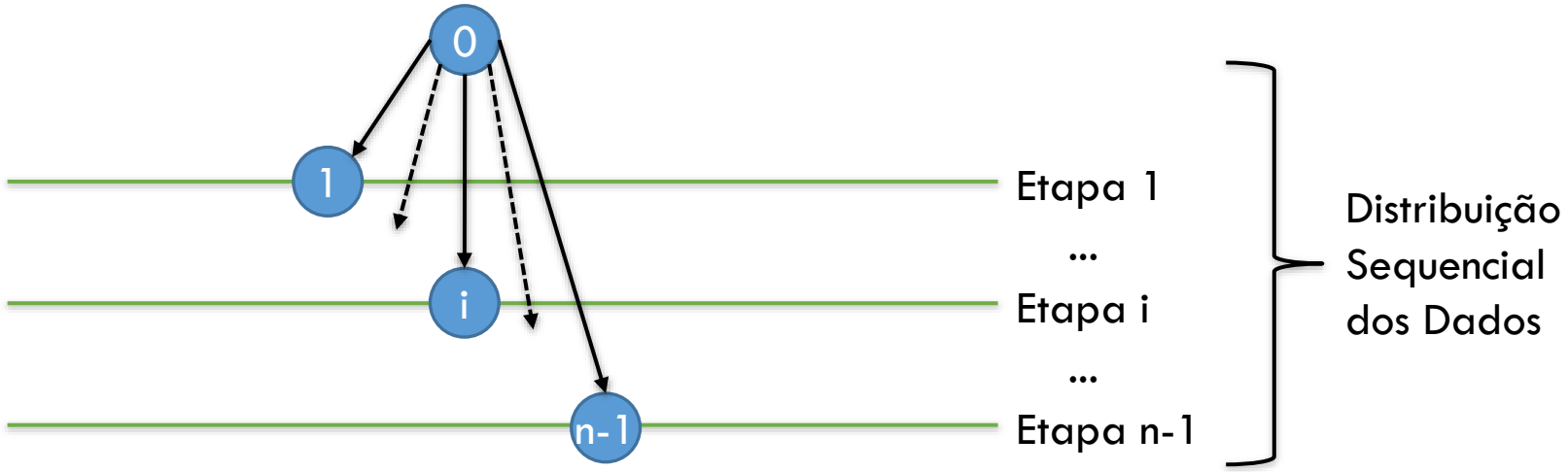
COMUNICAÇÕES COLETIVAS

Em programação paralela é habitual que, em determinadas partes do programa, um dado processo distribua o mesmo conjunto de dados para todos os processos (e.g. iniciar dados ou tarefas).

```
...
if (my_rank == 0)
  for (dest = 1; dest < n_procs; dest++)
    MPI_Send(data, count, datatype, dest, tag, MPI_COMM_WORLD);
else
  MPI_Recv(data, count, datatype, 0, tag, MPI_COMM_WORLD, &status);
...
```

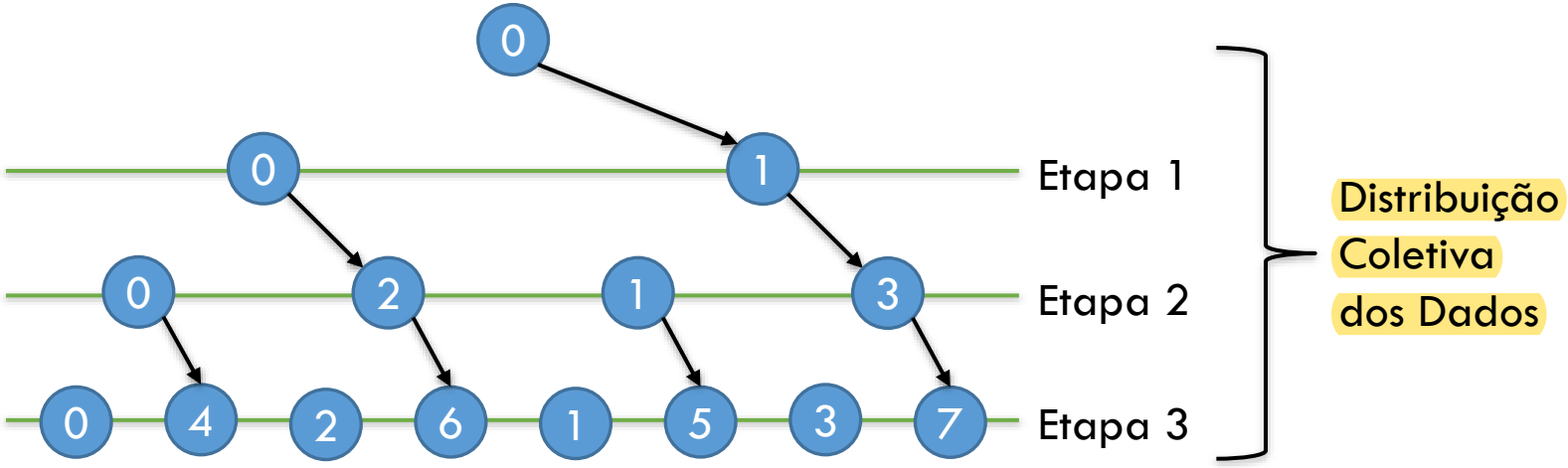
COMUNICAÇÕES COLETIVAS

Estamos basicamente fazendo uma distribuição sequencial dos dados, pois todas as comunicações são realizadas a partir do processo 0.



COMUNICAÇÕES COLETIVAS

Se mais processos colaborarem na distribuição da informação podemos reduzir significativamente o tempo total de comunicação.



Se usarmos uma topologia em árvore, tal como na figura acima, podemos distribuir os dados em $\lceil \log_2 N \rceil$ etapas em vez de $N - 1$ como na situação anterior.

COMUNICAÇÕES COLETIVAS

Para implementar a topologia em árvore, cada processo precisa de calcular em cada etapa se é um processo emissor/receptor e qual o destino/origem dos dados a enviar/receber.

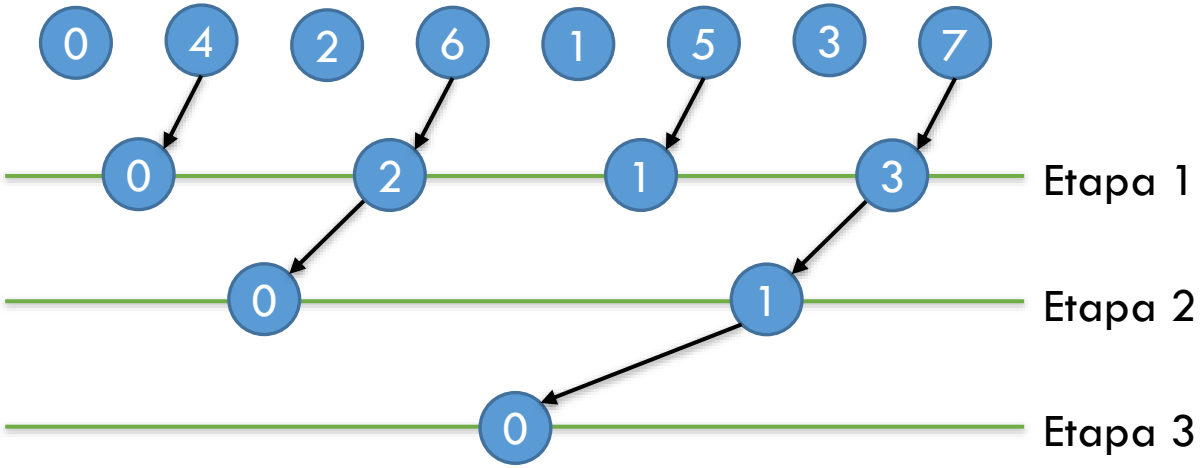
- Se $MyRank < 2^{stage-1}$, então envio para $MyRank + 2^{stage-1}$.
- Se $2^{stage-1} \leq MyRank < 2^{stage}$, então recebo de $MyRank - 2^{stage-1}$.

```
...
// Uma possível implementação:
for (stage = 1; stage <= upper_log2(n_procs); stage++)
  if (my_rank < pow(2, stage - 1))
    send_to(my_rank + pow(2, stage - 1));
  else if (my_rank >= pow(2, stage - 1) && my_rank < pow(2, stage))
    receive_from(my_rank - pow(2, stage -1));
...
```

COMUNICAÇÕES COLETIVAS

Em programação paralela é habitual que, em determinadas partes do programa, um processo (normalmente o processo 0) recolha informação dos outros processos e calcule resumos dessa informação.

Se invertermos a topologia de comunicação em árvore, podemos aplicar a mesma ideia para agrupar dados em $\lceil \log_2 N \rceil$ etapas.



MENSAGENS COLETIVAS

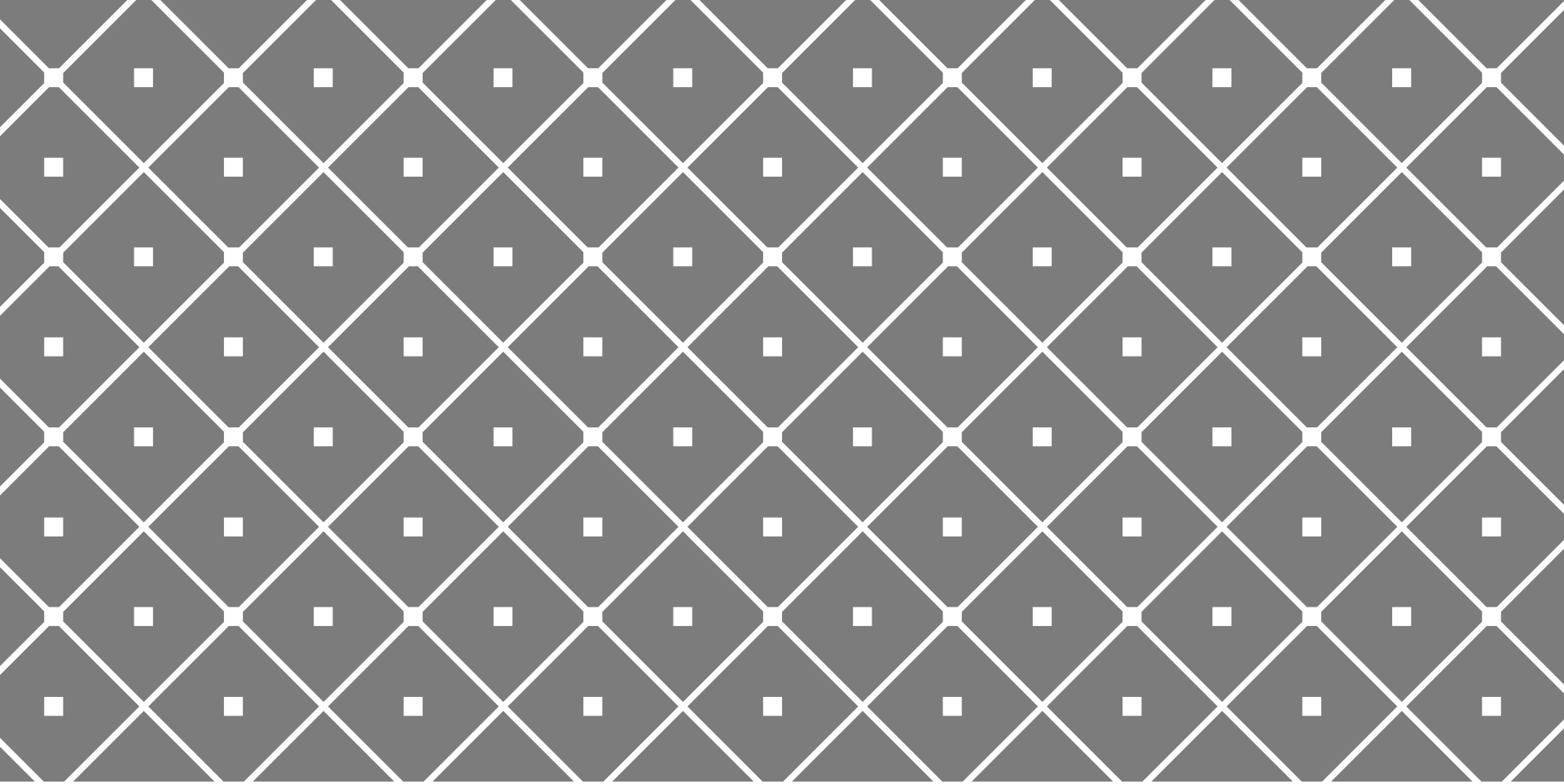
O MPI define um conjunto de rotinas para comunicações coletivas

Podemos então classificar as mensagens em:

- **Ponto-a-ponto**: a mensagem é enviada por um processo e recebida por um outro processo (e.g. todo o tipo de mensagens que vimos anteriormente).
- **Coletivas**: podem consistir de várias mensagens ponto-a-ponto concorrentes e envolvendo todos os processos de um comunicador (as mensagens coletivas têm de ser chamadas por todos os processos do comunicador).

As mensagens coletivas são variações ou combinações das seguintes 4 operações primitivas: **Broadcast, Reduce, Scatter, Gather.**

Todos os processos deverão chamar a mesma rotina (o processo raiz e os demais)



BROADCAST (DIFUNDIR)

BROADCAST

```
MPI_Bcast(void *buf, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```

MPI_Bcast() faz chegar uma mensagem de um processo a todos os outros processos no comunicador.

buf é o endereço inicial dos dados a enviar/receber.

count é o número de elementos do tipo datatype a enviar/receber.

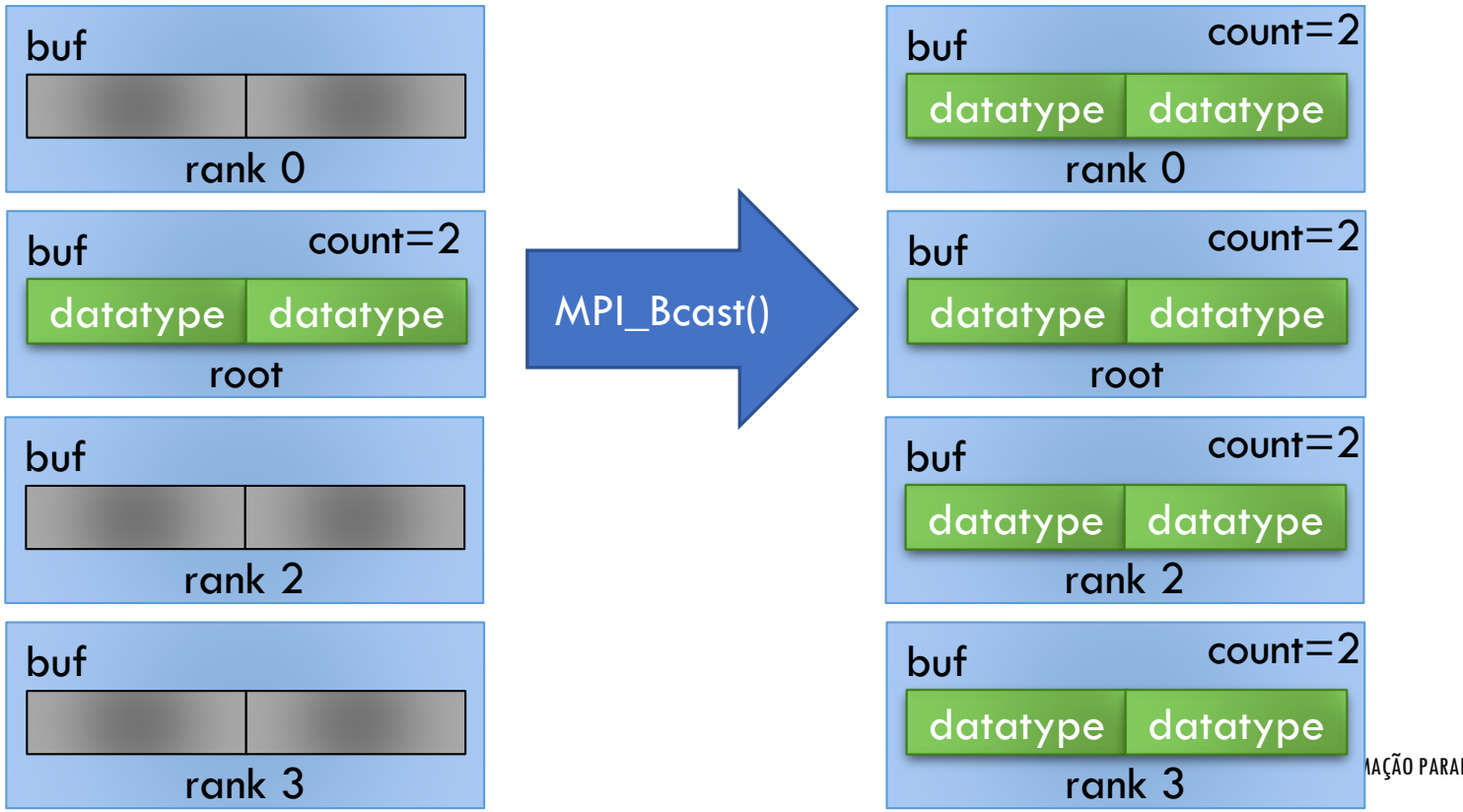
datatype é o tipo de dados a enviar/receber.

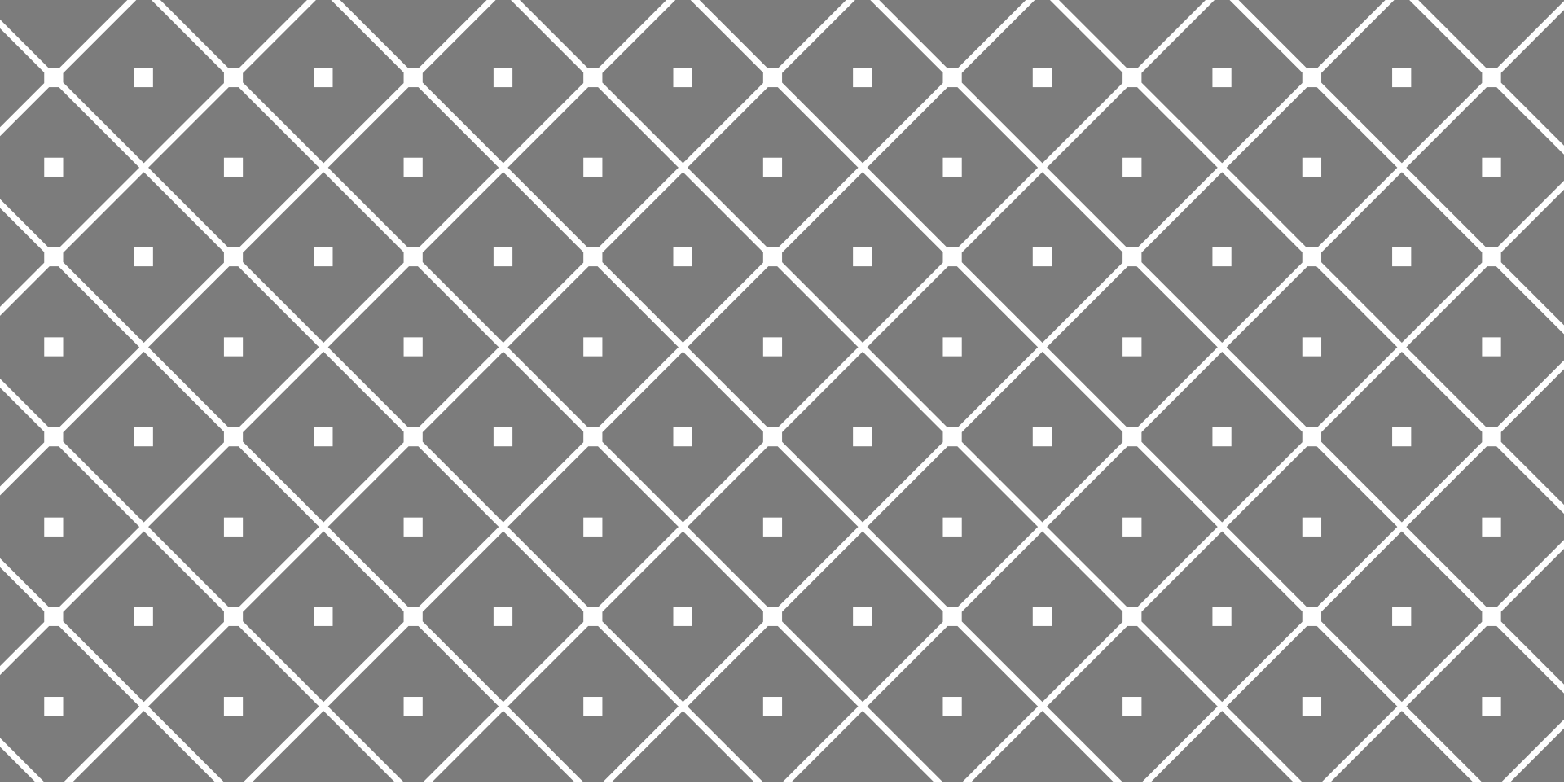
root é a posição do processo, no comunicador comm, que possui à partida a mensagem a enviar.

comm é o comunicador dos processos envolvidos na comunicação.

BROADCAST

```
MPI_Bcast(void *buf, int count,
MPI_Datatype datatype, int root, MPI_Comm comm)
```





REDUCE (REDUZIR)

REDUCE

```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

MPI_Reduce() permite realizar operações globais de resumo fazendo chegar mensagens de todos os processos a um único processo no comunicador.

sendbuf é o endereço inicial dos dados a enviar.

recvbuf é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo **root**).

count é o número de elementos do tipo **datatype** a enviar.

datatype é o tipo de dados a enviar.

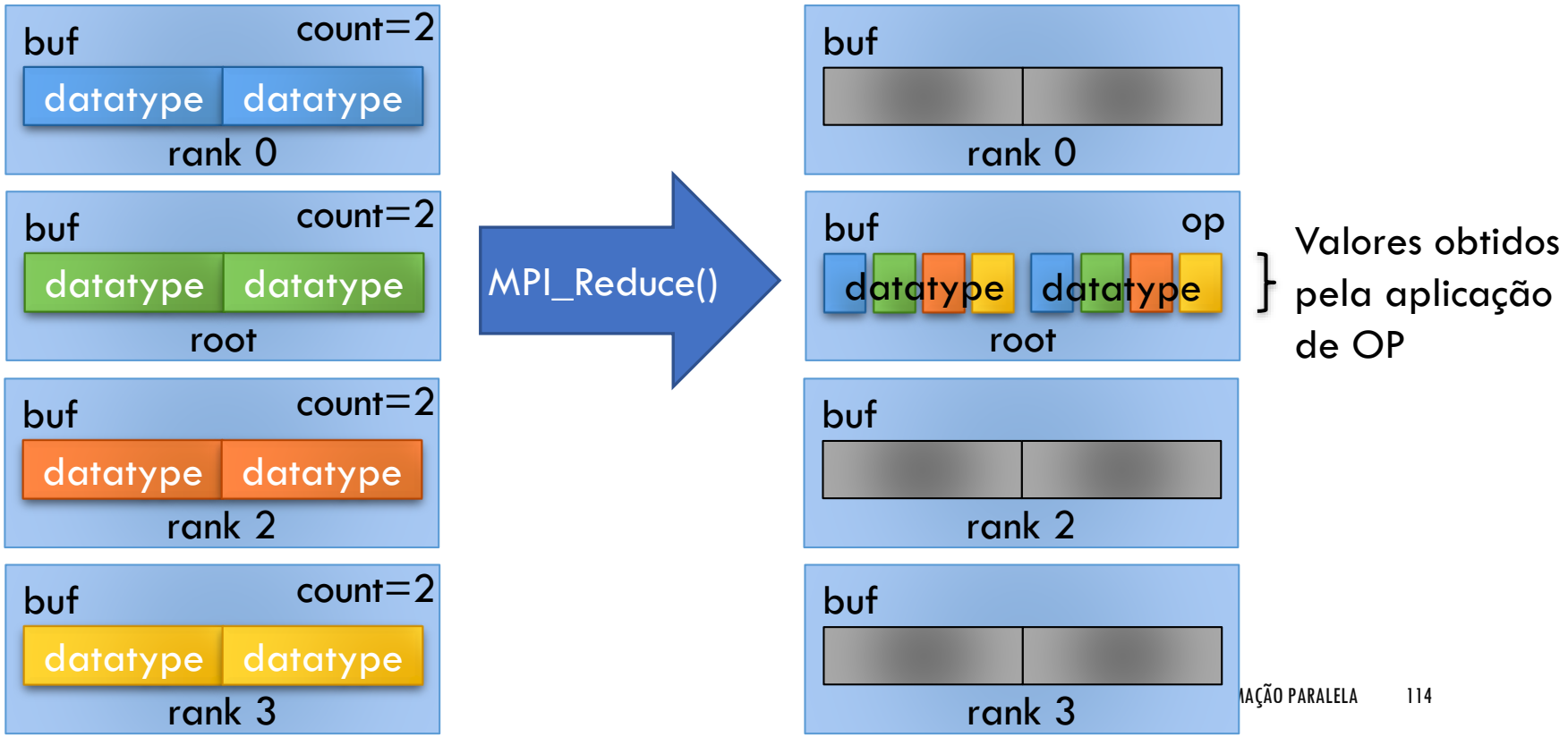
op é a operação de redução a aplicar aos dados recebidos.

root é a posição do processo, no comunicador **comm**, que recebe e resume os dados.

comm é o comunicador dos processos envolvidos na comunicação.

REDUCE

```
MPI_Reduce(void *sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

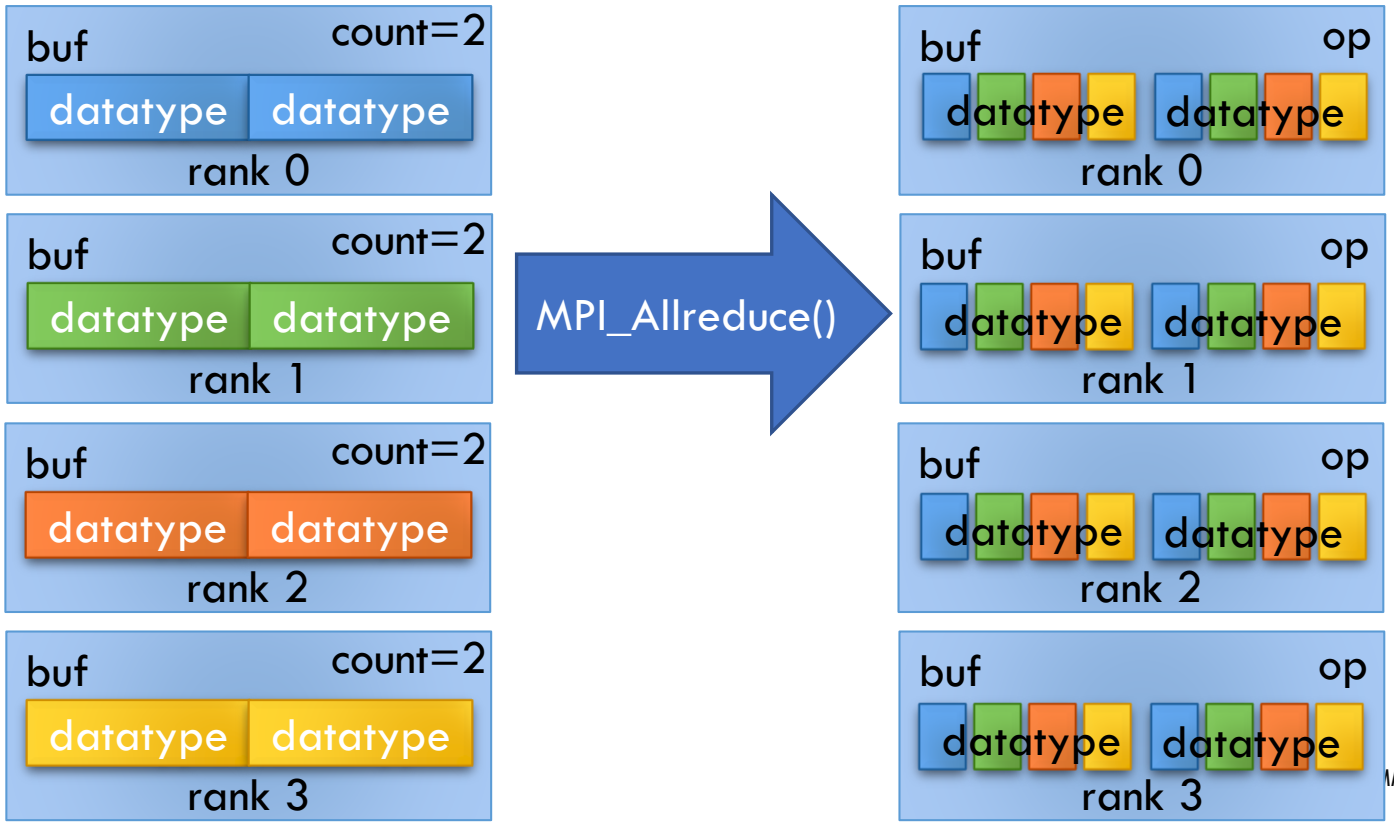


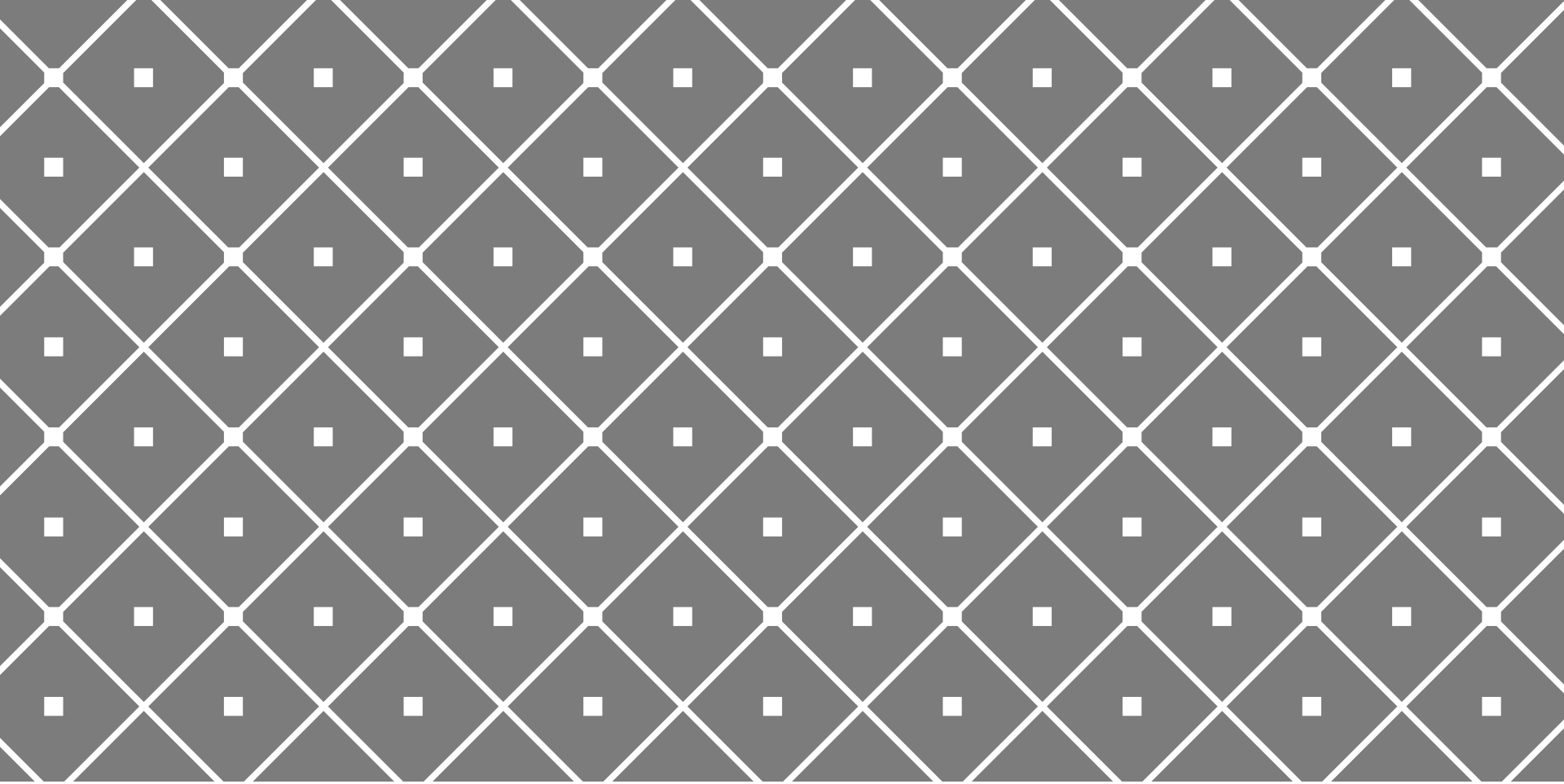
OPERAÇÕES DE REDUÇÃO

Operação	Significado
MPI_MAX	Máximo
MPI_MIN	Mínimo
MPI_SUM	Soma
MPI_PROD	Produto
MPI_LAND	E lógico
MPI_BAND	E dos bits
MPI_LOR	OU lógico
MPI_BOR	OU dos bits
MPI_LXOR	OU exclusivo lógico
MPI_BXOR	OU exclusivo dos bits

ALL REDUCE

```
MPI_Allreduce(void *sendbuf, void* recvbuf, int count,
  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```





SCATTER (ESPALHAR)

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Scatter() divide em partes iguais os dados de uma mensagem e distribui ordenadamente cada uma das partes por cada um dos processos no comunicador.

sendbuf é o endereço inicial dos dados a enviar (só é importante para o processo **root**).

sendcount é o número de elementos do tipo **sendtype** a enviar para cada processo (só é importante para o processo **root**).

sendtype é o tipo de dados a enviar (só é importante para o processo **root**).

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

recvbuf é o endereço onde devem ser colocados os dados recebidos.

recvcount é o número de elementos do tipo **recvtype** a receber por processo (normalmente o mesmo que **sendcount**).

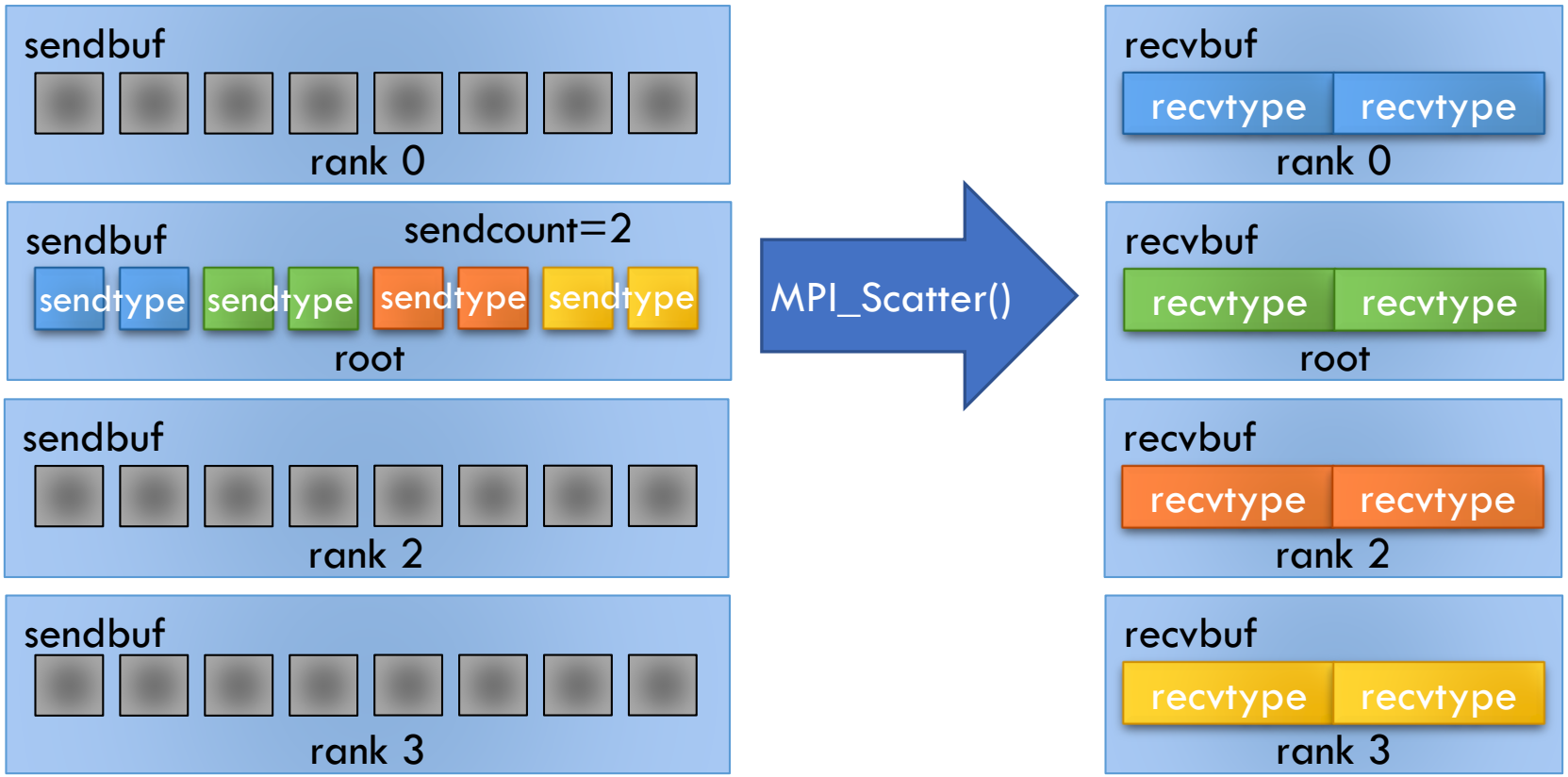
recvtype é o tipo de dados a receber (normalmente o mesmo que **sendtype**).

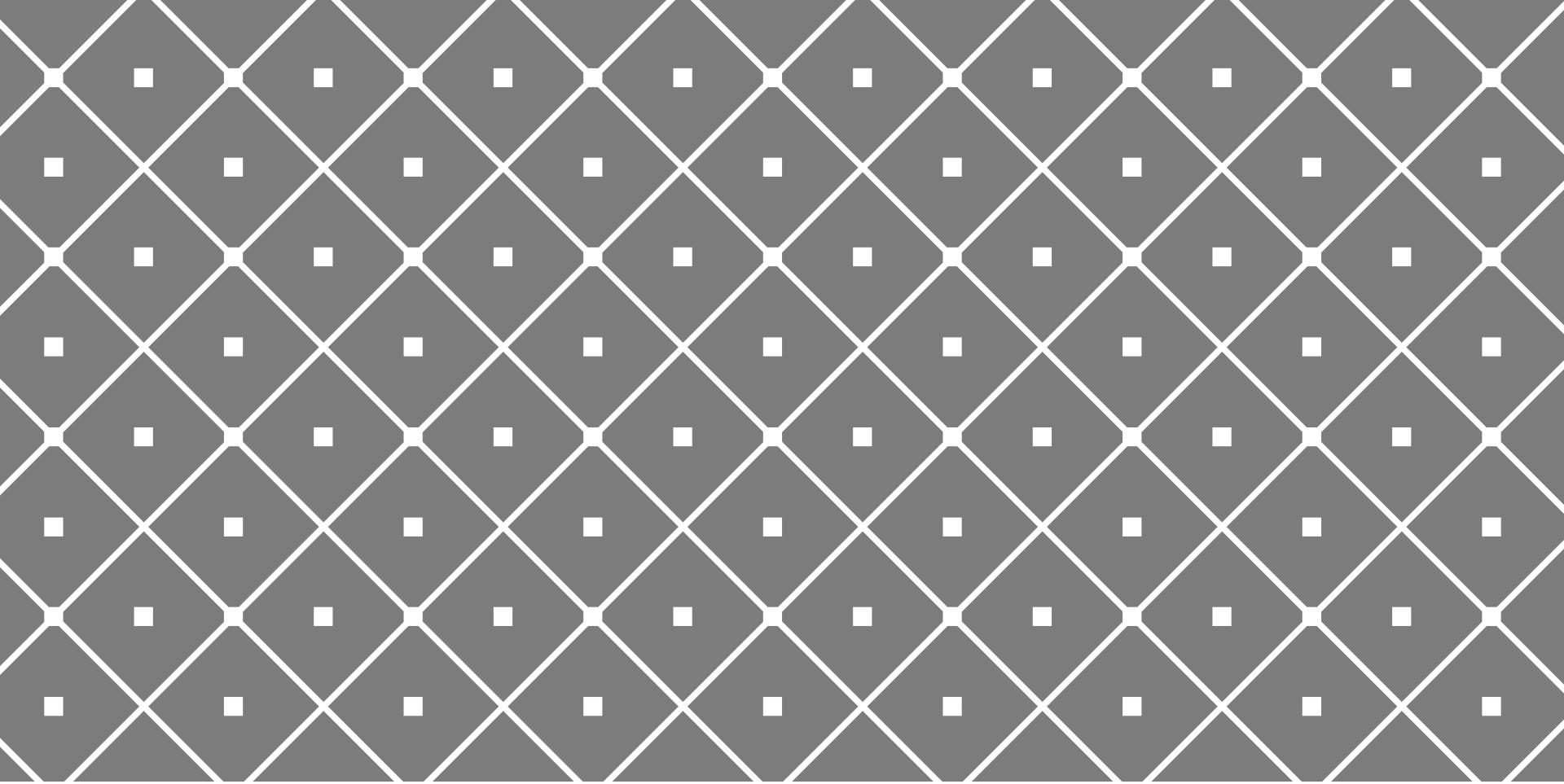
root é a posição do processo, no comunicador **comm**, que possui à partida a mensagem a enviar.

comm é o comunicador dos processos envolvidos na comunicação.

SCATTER

```
MPI_Scatter(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```





GATHER (REUNIR)

GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

MPI_Gather() recolhe ordenadamente num único processo um conjunto de mensagens oriundo de todos os processos no comunicador.

sendbuf é o endereço inicial dos dados a enviar.

sendcount é o número de elementos do tipo **sendtype** a enviar por cada processo.

sendtype é o tipo de dados a enviar.

GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

recvbuf é o endereço onde devem ser colocados os dados recebidos (só é importante para o processo **root**).

recvcount é o número de elementos do tipo **recvtype** a receber de cada processo (normalmente o mesmo que **sendcount**; só é importante para o processo **root**).

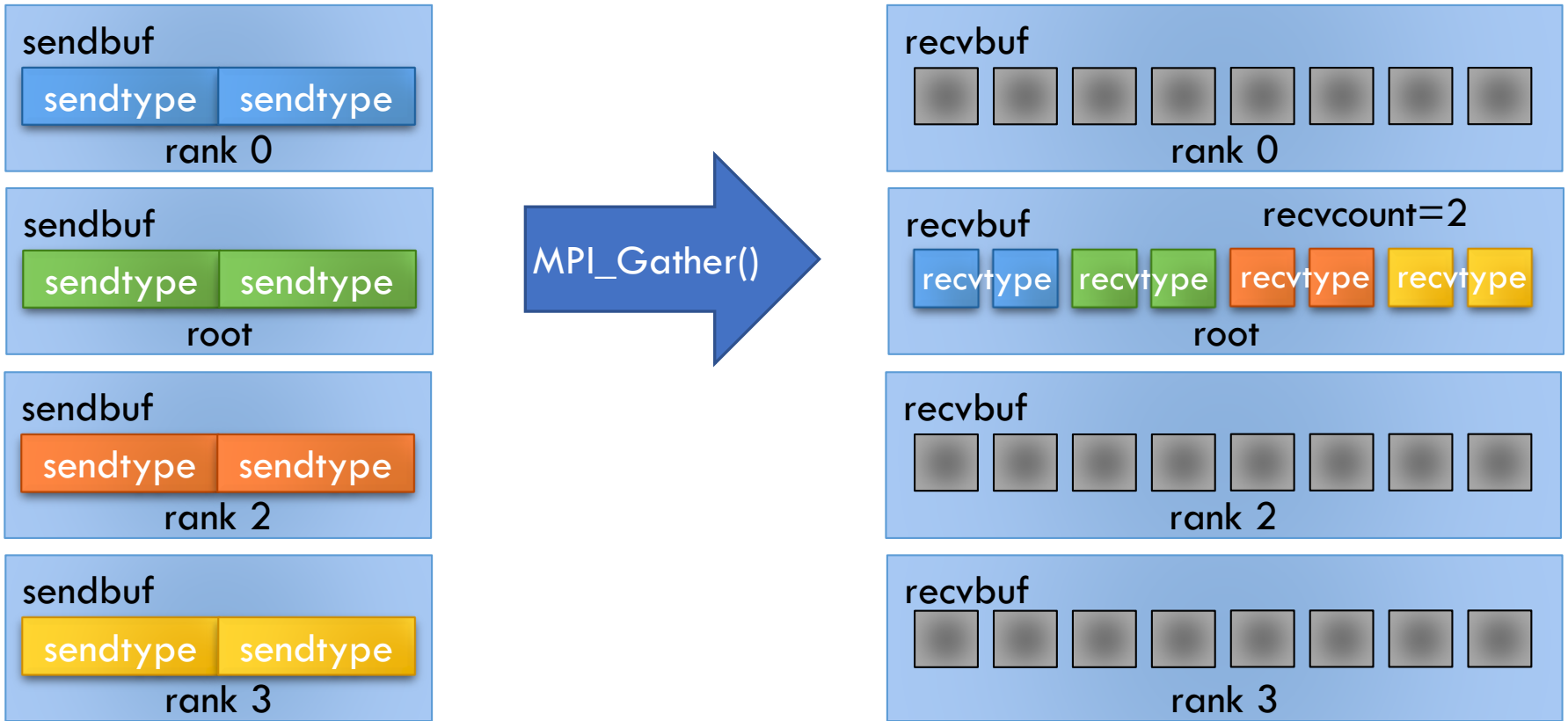
recvtype é o tipo de dados a receber (normalmente o mesmo que **sendtype**; só é importante para o processo **root**).

root é a posição do processo, no comunicador **comm**, que recebe os dados.

comm é o comunicador dos processos envolvidos na comunicação.

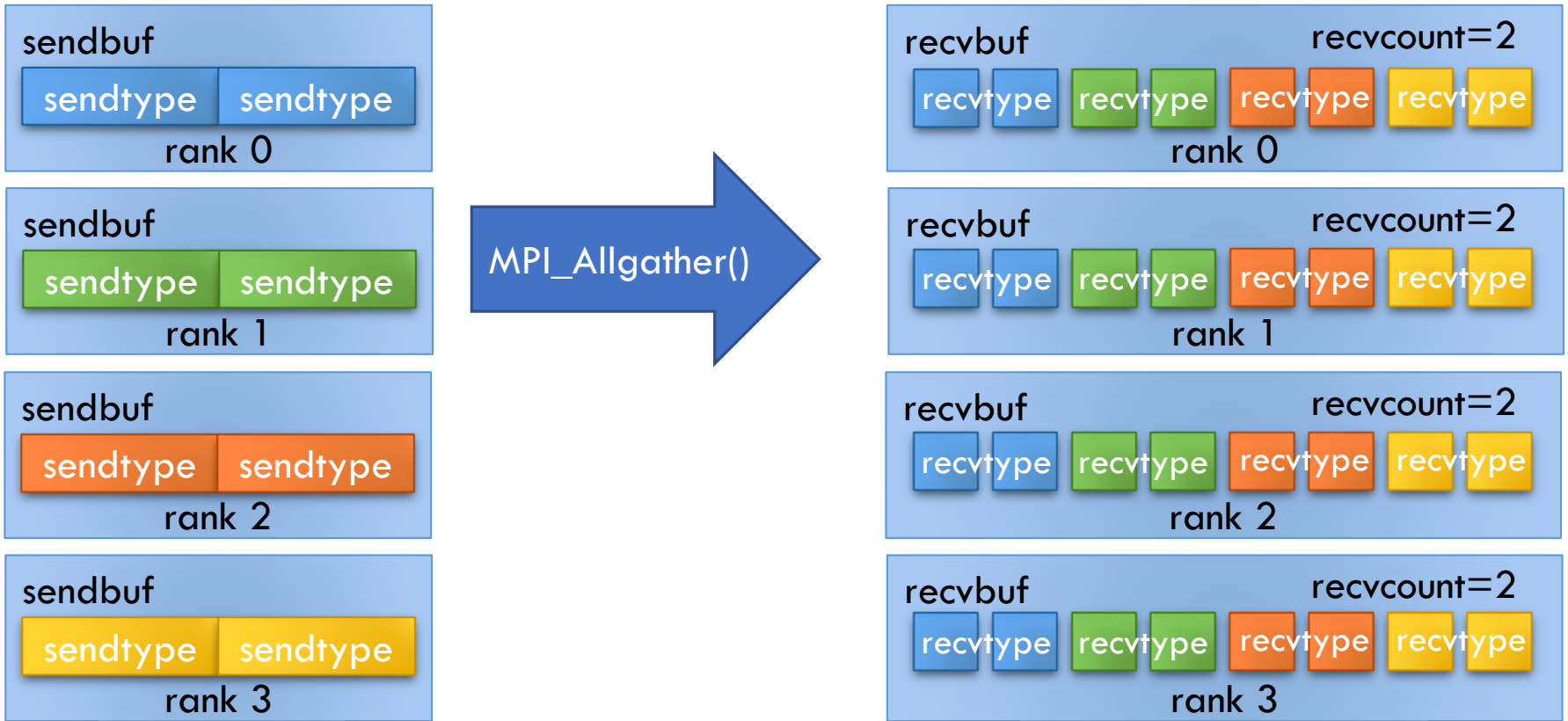
GATHER

```
MPI_Gather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



ALL GATHER

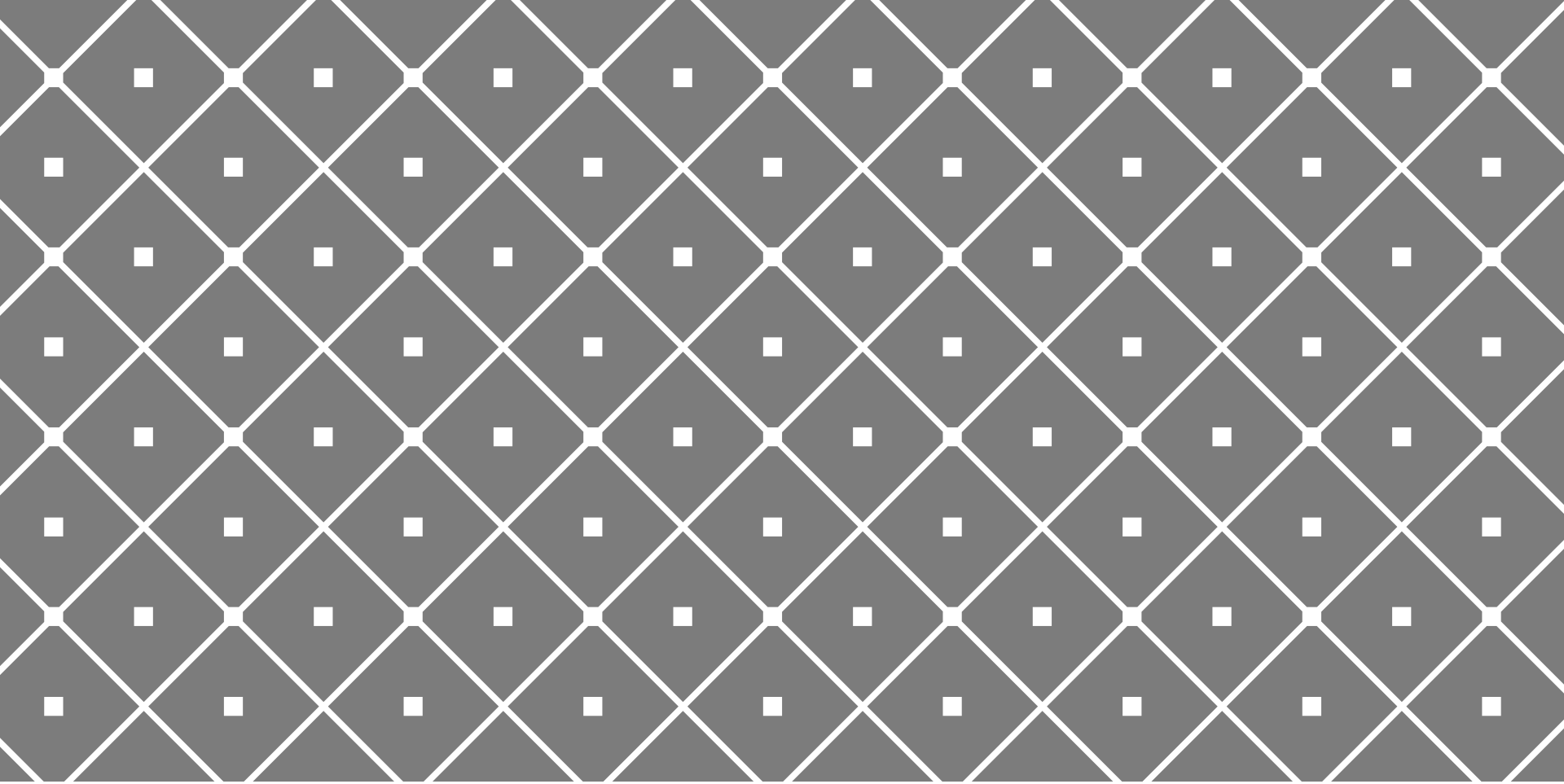
```
MPI_Allgather(void *sendbuf, int sendcount,
MPI_Datatype sendtype, void *recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)
```



OUTRAS VARIAÇÕES

MPI_SCATTERV e MPI_GATHERV estendem funcionalidades, permitindo que um número variável de dados a serem enviados para cada processo, uma vez que ele suporta um vetor de tamanhos.

Também prove flexibilidade para escolher de onde os dados serão tomados do root (acesso stride/esparso), através do argumento displs.



3 EXEMPLOS

1) MÉDIA DE N NÚMEROS

```
if (world_rank == 0) {
    // Cria números aleatórios para usar de exemplo
    rand_nums = create_rand_nums(elements_per_proc * world_size);
}

// Cria um buffer para armazenar o subconjunto de números aleatórios
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);

// Espalha os números entre todos os processos
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums,
            elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Computa a média em seu subconjunto
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

// Reúne todas as médias parciais para o processo raíz
float *sub_avgs = NULL;
if (world_rank == 0) {
    sub_avgs = malloc(sizeof(float) * world_size);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Computa a média geral dos valores
if (world_rank == 0) {
    float avg = compute_avg(sub_avgs, world_size);
```

2) PRODUTO ESCALAR

O **produto escalar** de 2 vetores de dimensão N é definido por:

$$x \cdot y = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}$$

Se tivermos P processos, cada um deles pode calcular $K = \left(\frac{N}{P}\right)$ componentes do produto escalar:

Processo	Componentes
0	$x_0 y_0 + x_1 y_1 + \cdots + x_{k-1} y_{k-1}$
1	$x_k y_k + x_{k+1} y_{k+1} + \cdots + x_{2k-1} y_{2k-1}$
…	…
P-1	$x_{(p-1)k} y_{(p-1)k} + x_{(p-1)k+1} y_{(p-1)k+1} + \cdots + x_{n-1} y_{n-1}$

2) PRODUTO ESCALAR

Processo	Componentes
0	$x_0y_0 + x_1y_1 + \cdots + x_{k-1}y_{k-1}$
1	$x_ky_k + x_{k+1}y_{k+1} + \cdots + x_{2k-1}y_{2k-1}$
...	...
P-1	$x_{(p-1)k}y_{(p-1)k} + x_{(p-1)k+1}y_{(p-1)k+1} + \cdots + x_{n-1}y_{n-1}$

```
int produto_escalar(int x[], int y[], int n) {
  int i, pe = 0;
  for (i = 0; i < n; i++)
    pe = pe + x[i] * y[i];
  return pe;
}
```

2) PRODUTO ESCALAR

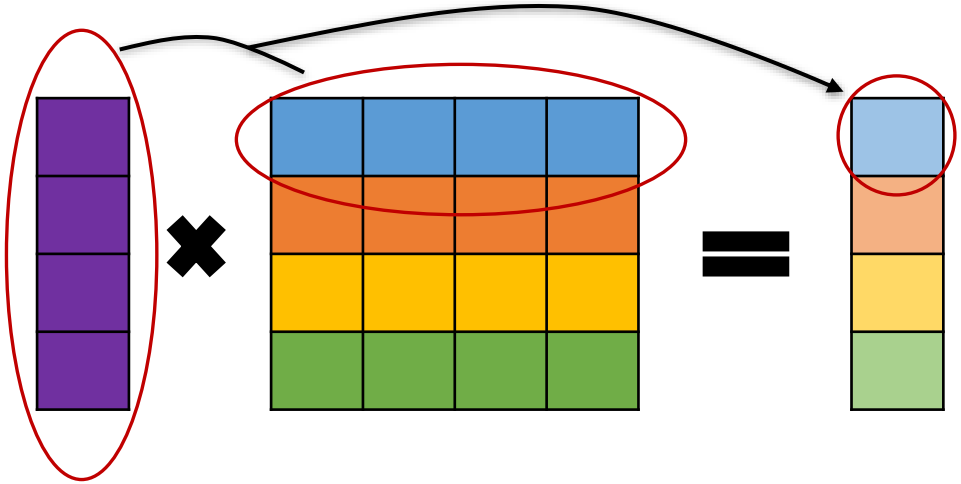
```
int *vector_x, *vector_y;
int K, pe, loc_pe, *loc_x, *loc_y;
...
if (my_rank == ROOT) {
  ... // calcular K e iniciar os vectores X e Y
}
// envia K a todos os processos
MPI_Bcast(&K, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
// aloca espaço para os vectores locais
loc_x = (int *) malloc(K * sizeof(int));
loc_y = (int *) malloc(K * sizeof(int));
// distribui as componentes dos vectores X e Y, um pedaço para cada proc.
MPI_Scatter(vector_x, K, MPI_INT, loc_x, K, MPI_INT, ROOT, MPI_COMM_WORLD);
MPI_Scatter(vector_y, K, MPI_INT, loc_y, K, MPI_INT, ROOT, MPI_COMM_WORLD);
// calcula o produto escalar e reagrupar os subtotais
loc_pe = produto_escalar(loc_x, loc_y, K);
MPI_Reduce(&loc_pe, &pe, 1, MPI_INT, MPI_SUM, ROOT,MPI_COMM_WORLD);
// apresenta o resultado
if (my_rank == ROOT)
  printf("Produto Escalar = %d \ n", pe);
...
```

3) PRODUTO MATRIZ-VECTOR

Sejam **matrix[ROWS,COLS]** e **vector[COLS]** respectivamente uma matriz e um vetor coluna.

O **produto matriz-vetor** é um vetor linha **result[ROWS]** em que cada result[i] é o produto escalar da linha i da matriz pelo vetor.

Se tivermos ROWS processos, cada um deles pode calcular um elemento do vetor resultado.



3) PRODUTO MATRIZ-VECTOR

```
int ROWS, COLS, *matrix, *vector, *result;
int pe, *linha;
... // inicia ROWS, COLS e o vetor
if (my_rank == ROOT) { ... // iniciar a matriz }
// distribui a matriz, uma linha para cada processo
MPI_Scatter(matrix, COLS, MPI_INT, linha, COLS, MPI_INT, ROOT,
MPI_COMM_WORLD);
// calcula o produto matriz-vetor e reúne os resultados
pe = produto_escalar(linha, vector, COLS);
MPI_Gather(&pe, 1, MPI_INT, result, 1, MPI_INT, ROOT, MPI_COMM_WORLD);
if (my_rank == ROOT) {
  printf("Produto Matriz-Vector: ");
  for (i = 0; i < ROWS; i++)
    printf("%d ", result[i]);
}
...
```