



BITS Pilani
Pilani Campus

Message Passing Interface (MPI)

K Hari Babu
Department of Computer Science & Information Systems



BITS Pilani
Pilani Campus



Programming with MPI: Collective Communication

Types of Collective Operations

- Synchronization
 - processes wait until all members of the group have reached the synchronization point
- Data Movement
 - broadcast, scatter/gather, all to all.
- Collective Computation (reductions)
 - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data

Scope

- Collective communication routines must involve all processes within the scope of a communicator
 - All processes are by default, members in the communicator `MPI_COMM_WORLD`.
- Additional communicators can be defined by the programmer
- Collective communication routines do not take message tag arguments
- Can only be used with MPI predefined datatypes

Trapezoidal Example

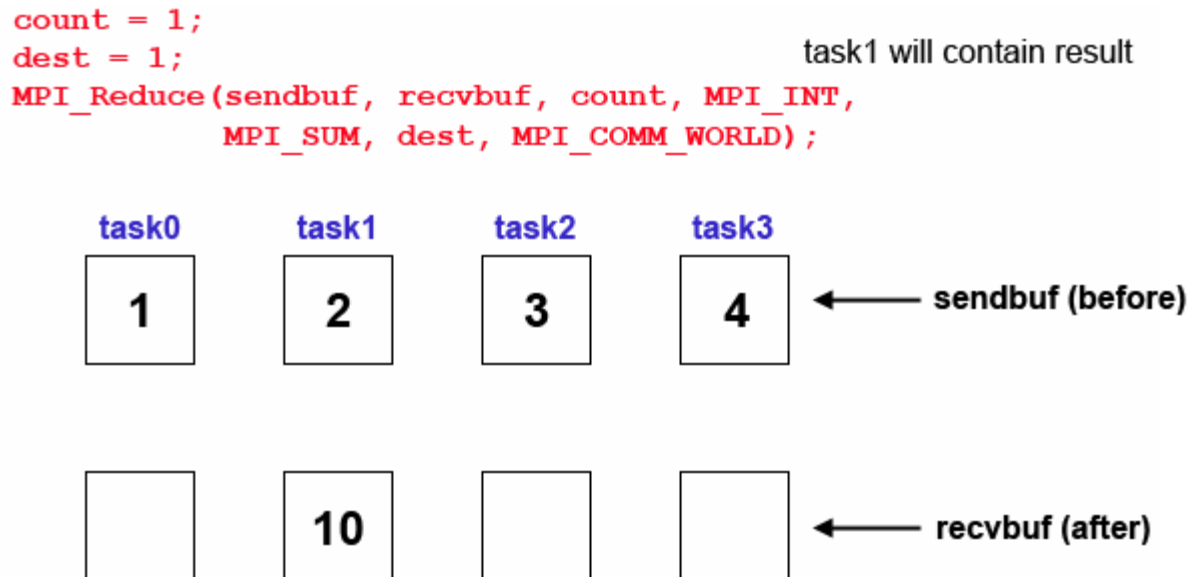
- In the code given on left, processes $P_1 \dots P_{n-1}$ are simply calculating local integral and sending
- P_0 is calculating as well as receiving and summing up all integrals
- This not equitable load distribution
 - Leads to increase in serial code thus reducing speedup

```
if (my_rank != 0)
    Send local_integral to process 0;
else{ /* my rank == 0 */
    total_integral = local_integral;
    for (proc = 1; proc < comm_sz; proc++){
        Receive local_integral from proc;
        total_integral += local_integral;
    }
}
if(my_rank == 0)
    print result;
```

-
- The top diagram illustrates a sparse grid with 9 nodes and 6 edges. The nodes are arranged in a 3x3 grid. The top row of nodes (0, 1, 2, 3, 4, 5, 6, 7) are labeled 'Processes'. The edges are: (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (0, 3), (1, 4), (2, 5), (3, 6), (4, 7), (0, 6), (1, 7), (2, 8), (3, 9), (4, 10), (5, 11), (6, 12), (7, 13), (8, 14), (9, 15), (10, 16), (11, 17), (12, 18), (13, 19), (14, 20), (15, 21), (16, 22), (17, 23), (18, 24), (19, 25), (20, 26), (21, 27), (22, 28), (23, 29), (24, 30), (25, 31), (26, 32), (27, 33), (28, 34), (29, 35), (30, 36), (31, 37), (32, 38), (33, 39), (34, 40), (35, 41), (36, 42), (37, 43), (38, 44), (39, 45), (40, 46), (41, 47), (42, 48), (43, 49), (44, 50), (45, 51), (46, 52), (47, 53), (48, 54), (49, 55), (50, 56), (51, 57), (52, 58), (53, 59), (54, 60), (55, 61), (56, 62), (57, 63), (58, 64), (59, 65), (60, 66), (61, 67), (62, 68), (63, 69), (64, 70), (65, 71), (66, 72), (67, 73), (68, 74), (69, 75), (70, 76), (71, 77), (72, 78), (73, 79), (74, 80), (75, 81), (76, 82), (77, 83), (78, 84), (79, 85), (80, 86), (81, 87), (82, 88), (83, 89), (84, 90), (85, 91), (86, 92), (87, 93), (88, 94), (89, 95), (90, 96), (91, 97), (92, 98), (93, 99), (94, 100), (95, 101), (96, 102), (97, 103), (98, 104), (99, 105), (100, 106), (101, 107), (102, 108), (103, 109), (104, 110), (105, 111), (106, 112), (107, 113), (108, 114), (109, 115), (110, 116), (111, 117), (112, 118), (113, 119), (114, 120), (115, 121), (116, 122), (117, 123), (118, 124), (119, 125), (120, 126), (121, 127), (122, 128), (123, 129), (124, 130), (125, 131), (126, 132), (127, 133), (128, 134), (129, 135), (130, 136), (131, 137), (132, 138), (133, 139), (134, 140), (135, 141), (136, 142), (137, 143), (138, 144), (139, 145), (140, 146), (141, 147), (142, 148), (143, 149), (144, 150), (145, 151), (146, 152), (147, 153), (148, 154), (149, 155), (150, 156), (151, 157), (152, 158), (153, 159), (154, 160), (155, 161), (156, 162), (157, 163), (158, 164), (159, 165), (160, 166), (161, 167), (162, 168), (163, 169), (164, 170), (165, 171), (166, 172), (167, 173), (168, 174), (169, 175), (170, 176), (171, 177), (172, 178), (173, 179), (174, 180), (175, 181), (176, 182), (177, 183), (178, 184), (179, 185), (180, 186), (181, 187), (182, 188), (183, 189), (184, 190), (185, 191), (186, 192), (187, 193), (188, 194), (189, 195), (190, 196), (191, 197), (192, 198), (193, 199), (194, 200), (195, 201), (196, 202), (197, 203), (198, 204), (199, 205), (200, 206), (201, 207), (202, 208), (203, 209), (204, 210), (205, 211), (206, 212), (207, 213), (208, 214), (209, 215), (210, 216), (211, 217), (212, 218), (213, 219), (214, 220), (215, 221), (216, 222), (217, 223), (218, 224), (219, 225), (220, 226), (221, 227), (222, 228), (223, 229), (224, 230), (225, 231), (226, 232), (227, 233), (228, 234), (229, 235), (230, 236), (231, 237), (232, 238), (233, 239), (234, 240), (235, 241), (236, 242), (237, 243), (238, 244), (239, 245), (240, 246), (241, 247), (242, 248), (243, 249), (244, 250), (245, 251), (246, 252), (247, 253), (248, 254), (249, 255), (250, 256), (251, 257), (252, 258), (253, 259), (254, 260), (255, 261), (256, 262), (257, 263), (258, 264), (259, 265), (260, 266), (261, 267), (262, 268), (263, 269), (264, 270), (265, 271), (266, 272), (267, 273), (268, 274), (269, 275), (270, 276), (271, 277), (272, 278), (273, 279), (274, 280), (275, 281), (276, 282), (277, 283), (278, 284), (279, 285), (280, 286), (281, 287), (282, 288), (283, 289), (284, 290), (285, 291), (286, 292), (287, 293), (288, 294), (289, 295), (290, 296), (291, 297), (292, 298), (293, 299), (294, 300), (295, 301), (296, 302), (297, 303), (298, 304), (299, 305), (300, 306), (301, 307), (302, 308), (303, 309), (304, 310), (305, 311), (306, 312), (307, 313), (308, 314), (309, 315), (310, 316), (311, 317), (312, 318), (313, 319), (314, 320), (315, 321), (316, 322), (317, 323), (318, 324), (319, 325), (320, 326), (321, 327), (322, 328), (323, 329), (324, 330), (325, 331), (326, 332), (327, 333), (328, 334), (329, 335), (330, 336), (331, 337), (332, 338), (333, 339), (334, 340), (335, 341), (336, 342), (337, 343), (338, 344), (339, 345), (340, 346), (341, 347), (342, 348), (343, 349), (344, 350), (345, 351), (346, 352), (347, 353), (348, 354), (349, 355), (350, 356), (351, 357), (352, 358), (353, 359), (354, 360), (355, 361), (356, 362), (357, 363), (358, 364), (359, 365), (360, 366), (361, 367), (362, 368), (363, 369), (364, 370), (365, 371), (366, 372), (367, 373), (368, 374), (369, 375), (370, 376), (371, 377), (372, 378), (373, 379), (374, 380), (375, 381), (376, 382), (377, 383), (378, 384), (379, 385), (380, 386), (381, 387), (382, 388), (383, 389), (384, 390), (385, 391), (386, 392), (387, 393), (388, 394), (389, 395), (390, 396), (391, 397), (392, 398), (393, 399), (394, 400), (395, 401), (396, 402), (397, 403), (398, 404), (399, 405), (400, 406), (401, 407), (402, 408), (403, 409), (404, 410), (405, 411), (406, 412), (407, 413), (408, 414), (409, 415), (

Collective Computation Operation

- `MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)`
- Applies a reduction operation on all tasks in the group and places the result in one task



Collective Computation Operation

MPI Reduction Operations

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical AND
MPI_BAND	bit-wise AND
MPI_LOR	logical OR
MPI_BOR	bit-wise OR
MPI_LXOR	logical XOR
MPI_BXOR	bit-wise XOR
MPI_MAXLOC	max value and location
MPI_MINLOC	min value and location

The operation is always assumed to be associative. All predefined operations are also assumed to be commutative. Users may define operations that are assumed to be associative, but not commutative

Users can also define their own reduction functions by using the MPI_Op_create routine

MPI_Reduce Example

- Summing up local values

```
#include <mpi.h>
#include <stdio.h>
#define SIZE 100
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    int rank,i;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    srand(rank);
    //local
    double local_double=(rand() % (SIZE - 0 + 1)) + 0;
    printf("rank =%d. my no is %lf\n", rank, local_double);
    double global_sum;
    double global_max;
    MPI_Reduce(&local_double,&global_sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    MPI_Reduce(&local_double,&global_max,1,MPI_DOUBLE,MPI_MAX,0,MPI_COMM_WORLD);
    printf("Process %d received number %lf \n",rank,global_sum);
    printf("Process %d received number %lf \n",rank,global_max);
    MPI_Finalize();}
```

MPI_Reduce Trapezoid Example

- MPI_Send and MPI_Recv are replaced by MPI_Reduce

```
if (my_rank != 0)
    Send local_integral to process 0;
else{ /* my rank == 0 */
    total_integral = local_integral;
    for (proc = 1; proc < comm_sz; proc++){
        Receive local_integral from proc;
        total_integral += local_integral;
    }
}
if(my_rank == 0)
    print result;
```

```
/*Note: h and local_n are the same for all processes*/
h = (b-a)/n;          /* length of each trapezoid */
local_n = n/comm_sz; /* number of trapezoids per process */
/* Length of each process' interval of integration = local_n*h. */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_int = Trap(local_a, local_b, local_n, h);
/* Add up the integrals calculated by each process */
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

Collective vs Point-to-Point Communication

- Collective communications differ in several ways from point-to-point communications
 - All the processes in the communicator must call the same collective function
 - `MPI_Reduce` must be called by all processes
 - The arguments passed by each process to an MPI collective communication must be “compatible”
 - if one process passes in 0 as the dest process and another passes in 1, then the outcome of a call to MPI Reduce is erroneous
 - The *recvbuf* argument is only used on *dest* process. However, all of the processes still need to pass it
 - Point-to-point communications are matched on the basis of tags and communicators
 - Collective communications don't use tags, so they're matched solely on the basis of the communicator and the order in which they're called

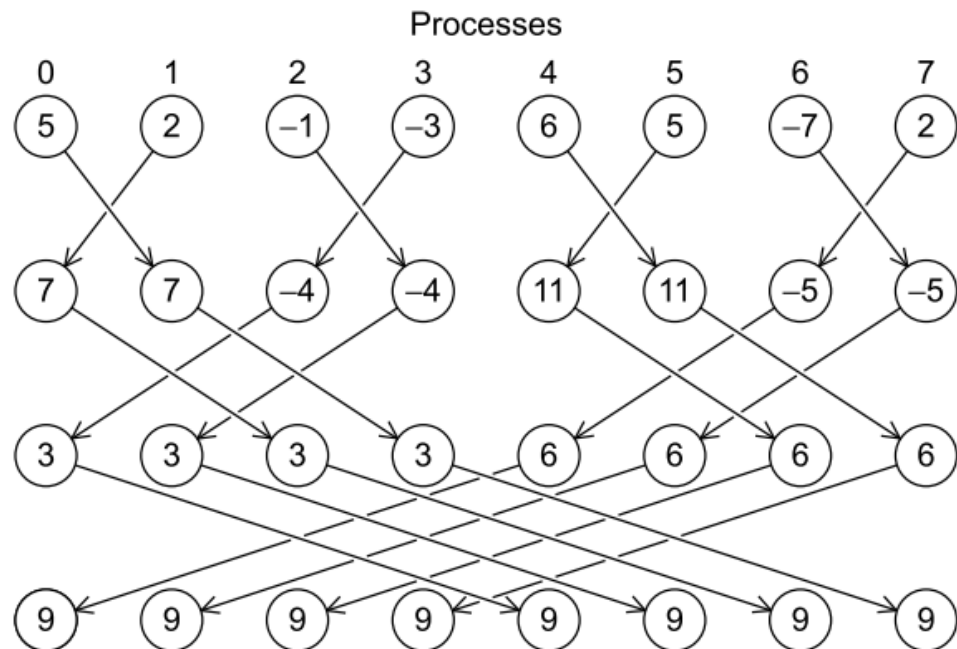
MPI_Allreduce

- In our trapezoidal rule program, we just print the result, so it's perfectly natural for only one process to get the result of the global sum
 - However, it's not difficult to imagine a situation in which all of the processes need the result of a global sum in order to complete some larger computation

What is the best way to do it?

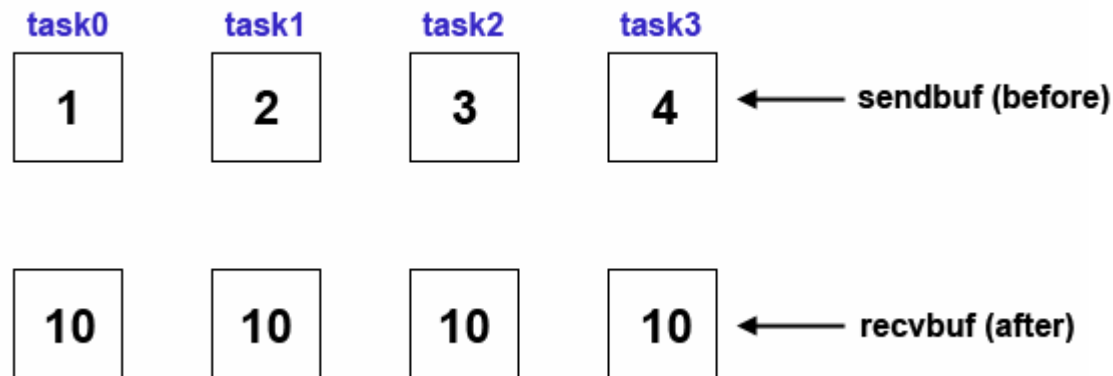
One way is to reverse the tree
Other is to butterfly communication pattern where result is not computed at one place and then distributed

MPI provides MPI_Allreduce where the user need not worry about the communication pattern



Collective Computation Operation

- `MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)`
- Collective computation operation + data movement. Applies a reduction operation and places the result in all tasks in the group. This is equivalent to an `MPI_Reduce` followed by an `MPI_Bcast`.

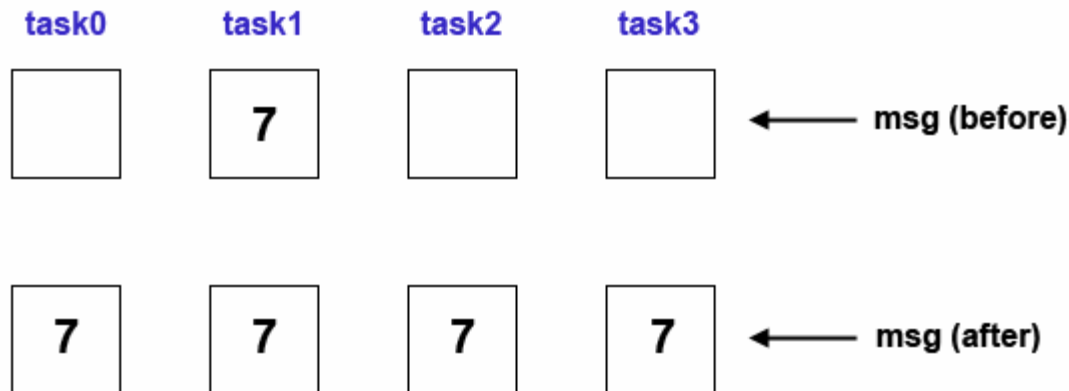


Data Movement Operation

- MPI_Bcast (&buffer,count,datatype,root,comm)
- Broadcasts (sends) a message from the process with rank "root" to all other processes in the group

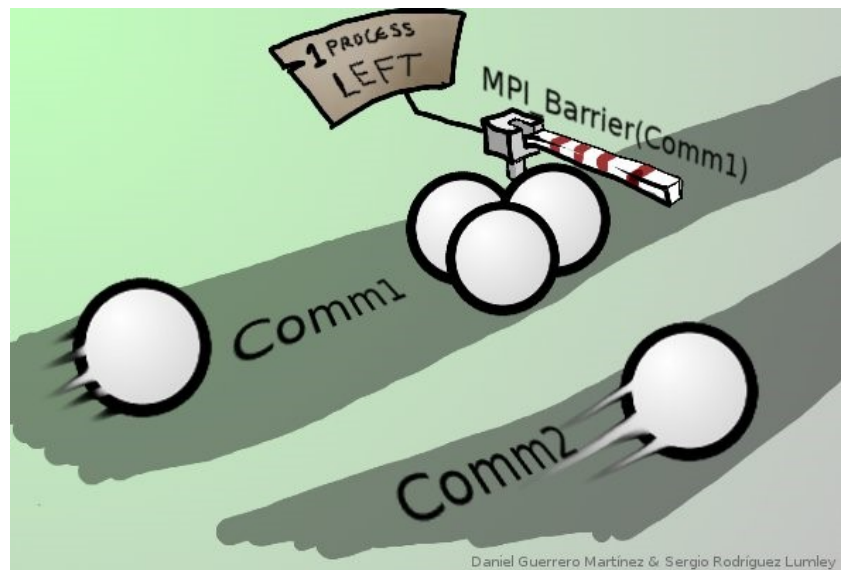
```
count = 1;  
source = 1;  
MPI_Bcast(&msg, count, MPI_INT, source, MPI_COMM_WORLD);
```

task1 contains the message to be broadcast



Synchronization Operation

- `MPI_Barrier(MPI_comm)` is a synchronization operation
 - Creates a barrier synchronization in a group
 - Each task, when reaching the `MPI_Barrier` call, blocks until all tasks in the group reach the same `MPI_Barrier` call. Then all tasks are free to proceed
- `MPI_Send`, `MPI_Recv` can be used for sync between two processes.
`MPI_Barrier` is used for collective synchronization



Broadcast Example

```
09 if (my_rank == 0) {
10     printf("Enter a, b, and n\n");
11     scanf("%lf %lf %d", a,b,n);
12 for (dest = 1; dest < comm_sz; dest++) {
13     MPI_Send(a, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
14     MPI_Send(b, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
15     MPI_Send(n, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
16 }
17 } else { /* my rank != 0 */
18     MPI_Recv(a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
20     MPI_Recv(b, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
22     MPI_Recv(n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
24 }
```

```
void Get_input(int argc, char* argv[], int my_rank, double* n_p){
    if (my_rank == 0) //stdin is accessible only to rank 0
        scanf("%lf", n_p);
    // Broadcasts value of n to each process
    MPI_Bcast(n_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // negative n ends the program
} /* Get_input */
```


Data Distribution

- Suppose there are 10000 elements in a vector and to be distributed across `comm_sz` processes
- If we use `MPI_Bcast`, all processes need to allocate memory for 10000 elements
- If there is a way, these elements can be distributed to equally to all processes, that is `MPI_Scatter`
 - `MPI_Scatter`
(`&sendbuf`,`sendcnt`,`sendtype`,`&recvbuf`,`recvcnt`,`recvtype`,`root`,`comm`)
 - divides the data referenced by `sendbuf` into `comm sz` pieces—the first piece goes to process 0, the second to process 1, the third to process 2, and so on ...
 - `sendcnt` is the amount of data going to each process; it's not the amount of data in the memory referred to by `sendbuf`
 - If the data is not evenly divisible by `comm_sz`, use `MPI_Scatterv`

Data Movement Operation

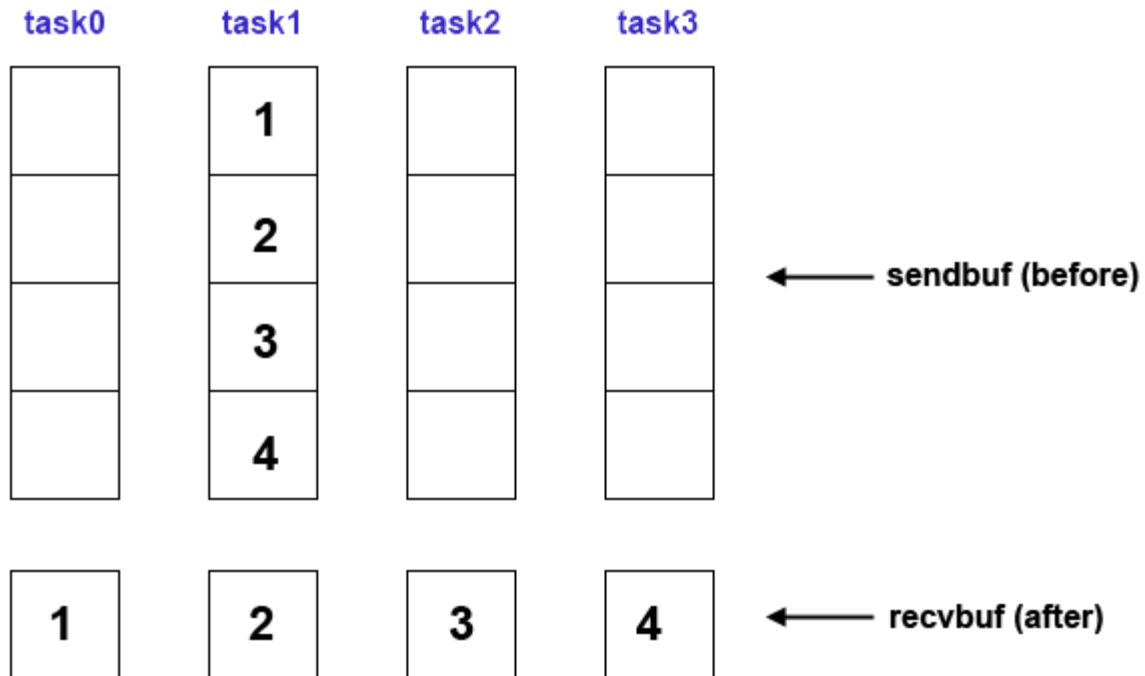
- MPI_Scatter
(&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)

```
sendcnt = 1;  
recvcnt = 1;  
src = 1;
```

```
MPI_Scatter(sendbuf, sendcnt, MPI_INT  
            recvbuf, recvcnt, MPI_INT  
            src, MPI_COMM_WORLD);
```

task1 contains the data to be scattered

- Distributes distinct messages from a single source task to each task in the group



Scatter Example

```
1. #define SIZE 4
2. int main (int argc, char *argv[])
3. {
4.     int numtasks, rank, sendcount, recvcount, source;
5.     float sendbuf[SIZE][SIZE] = {
6.         {1.0, 2.0, 3.0, 4.0},
7.         {5.0, 6.0, 7.0, 8.0},
8.         {9.0, 10.0, 11.0, 12.0},
9.         {13.0, 14.0, 15.0, 16.0}
10.    };
11.    float recvbuf[SIZE];
12.    MPI_Init (&argc, &argv);
13.    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
14.    MPI_Comm_size (MPI_COMM_WORLD,
        &numtasks);
```

```
1.     if (numtasks == SIZE)
2.     {
3.         // define source task and elements to
           // send/receive, then perform collective scatter
4.         source = 1;
5.         sendcount = SIZE;
6.         recvcount = SIZE;
7.         MPI_Scatter (sendbuf, sendcount,
           MPI_FLOAT, recvbuf, recvcount,
8.         MPI_FLOAT, source, MPI_COMM_WORLD);
9.         printf ("rank= %d Results: %f %f %f %f\n",
           rank, recvbuf[0],
10.        recvbuf[1], recvbuf[2], recvbuf[3]);
11.    }
12.    else
13.        printf ("Must specify %d processors.
           Terminating.\n", SIZE);
14.
15.    MPI_Finalize ();
16. }
```

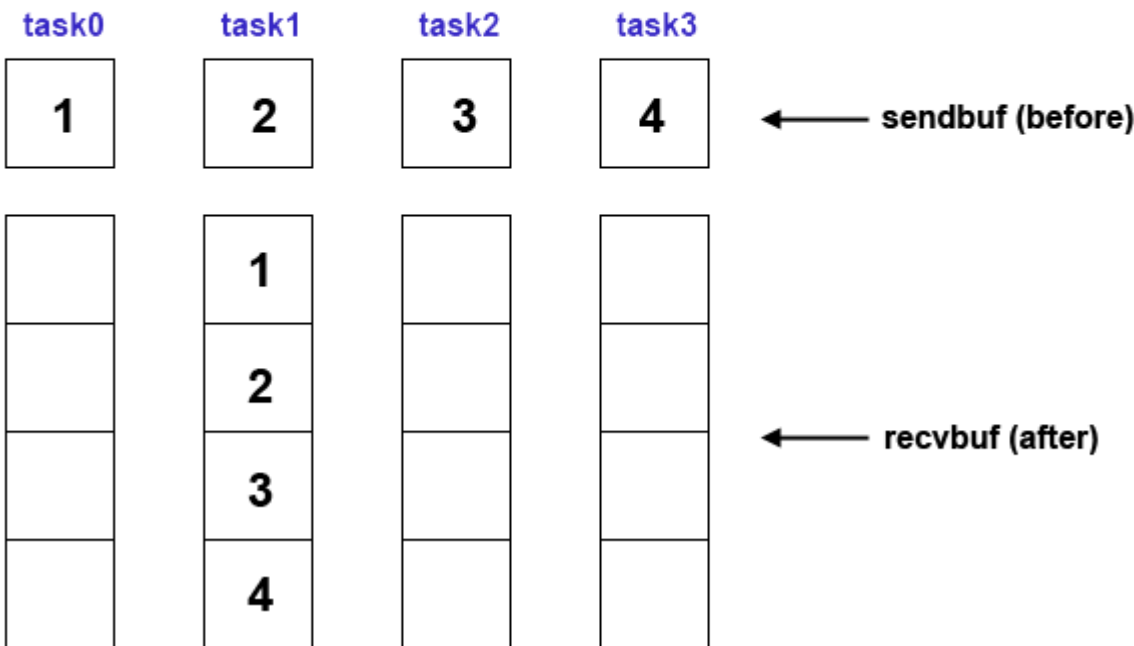
Data Movement Operation

- **MPI_Gather**
(&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,root,comm)
- Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

```
sendcnt = 1;
recvcnt = 1;
src = 1;
```

```
MPI_Gather(sendbuf, sendcnt, MPI_INT
recvbuf, recvcnt, MPI_INT
src, MPI_COMM_WORLD);
```

message will be gathered into task1

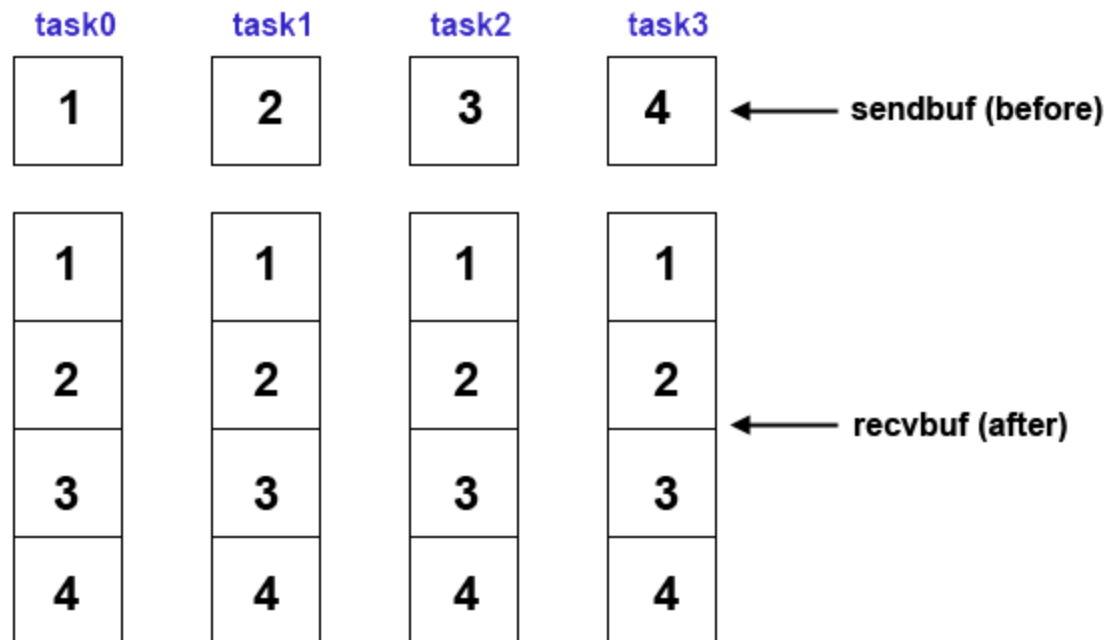


Data Movement Operation

- MPI_Allgather
(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)

```
sendcnt = 1;
recvcnt = 1;
MPI_Allgather(sendbuf, sendcnt, MPI_INT,
recvbuf, recvcnt, MPI_INT,
MPI_COMM_WORLD);
```

- Concatenation of data to all tasks in a group. Each task in the group, in effect, performs a one-to-all broadcasting operation within the group.



Matrix Multiplication

```
#include <mpi.h>
#include <stdio.h>
#define SIZE 4
int
main (int argc, char *argv[])
{
    int numtasks, rank, sendcount, recvcount,
    source, i;
    float sendbuf[SIZE][SIZE] = {
        {1.0, 2.0, 3.0, 4.0},
        {5.0, 6.0, 7.0, 8.0},
        {9.0, 10.0, 11.0, 12.0},
        {13.0, 14.0, 15.0, 16.0} };
    float m2[] = { 1.0, 2.0, 3.0, 4.0 };
    float y[SIZE] = { 0.0, 0.0, 0.0, 0.0 };
    float result[SIZE];
    float recvbuf[SIZE];

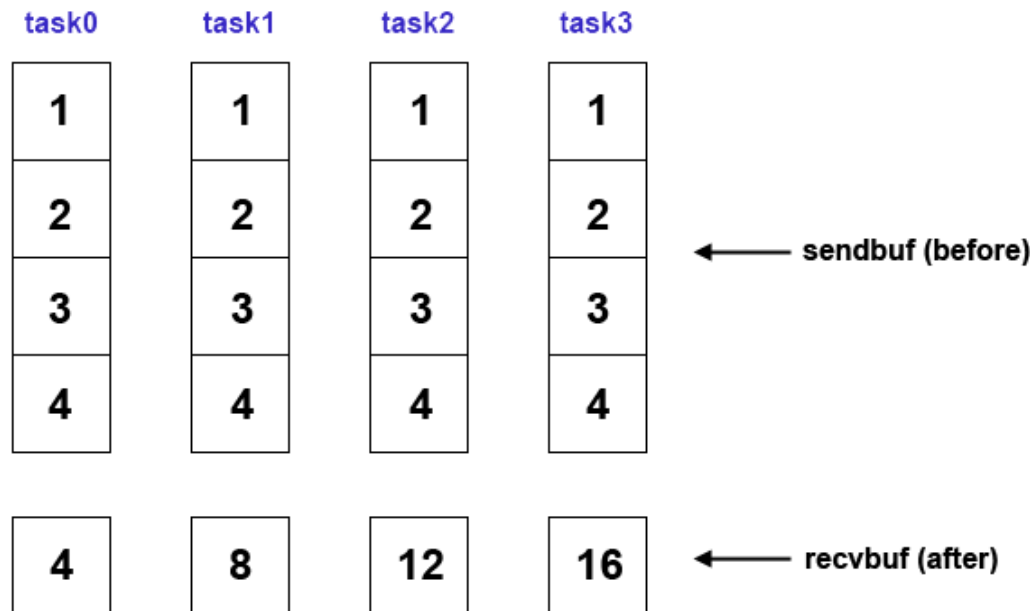
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD,
    &rank);
    MPI_Comm_size (MPI_COMM_WORLD,
    &numtasks);
```

```
    if (numtasks == SIZE){
        // define source task and elements to send/receive,
        then perform collective scatter
        source = 1;
        sendcount = SIZE;
        recvcount = SIZE;
        MPI_Scatter (sendbuf, sendcount, MPI_FLOAT,
        recvbuf, recvcount, MPI_FLOAT, source,
        MPI_COMM_WORLD);
        //MPI_Bcast(m2,SIZE,MPI_FLOAT,0,MPI_COMM_W
        ORLD);
        for (i = 0; i < SIZE; i++)
            y[rank] += recvbuf[i] * m2[i];
        MPI_Allgather (&y[rank], 1, MPI_FLOAT, result, 1,
        MPI_FLOAT, MPI_COMM_WORLD);
        printf ("rank= %d Results: %f %f %f %f\n", rank,
        result[0],
        result[1], result[2], result[3]);
    }
    else
        printf ("Must specify %d processors. Terminating.\n",
        SIZE);
    MPI_Finalize ();
}
```

Collective Computation Operation

- `MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype,op,comm)`
 - First does an element-wise reduction on a vector across all tasks in the group. Next, the result vector is split into disjoint segments and distributed across the tasks

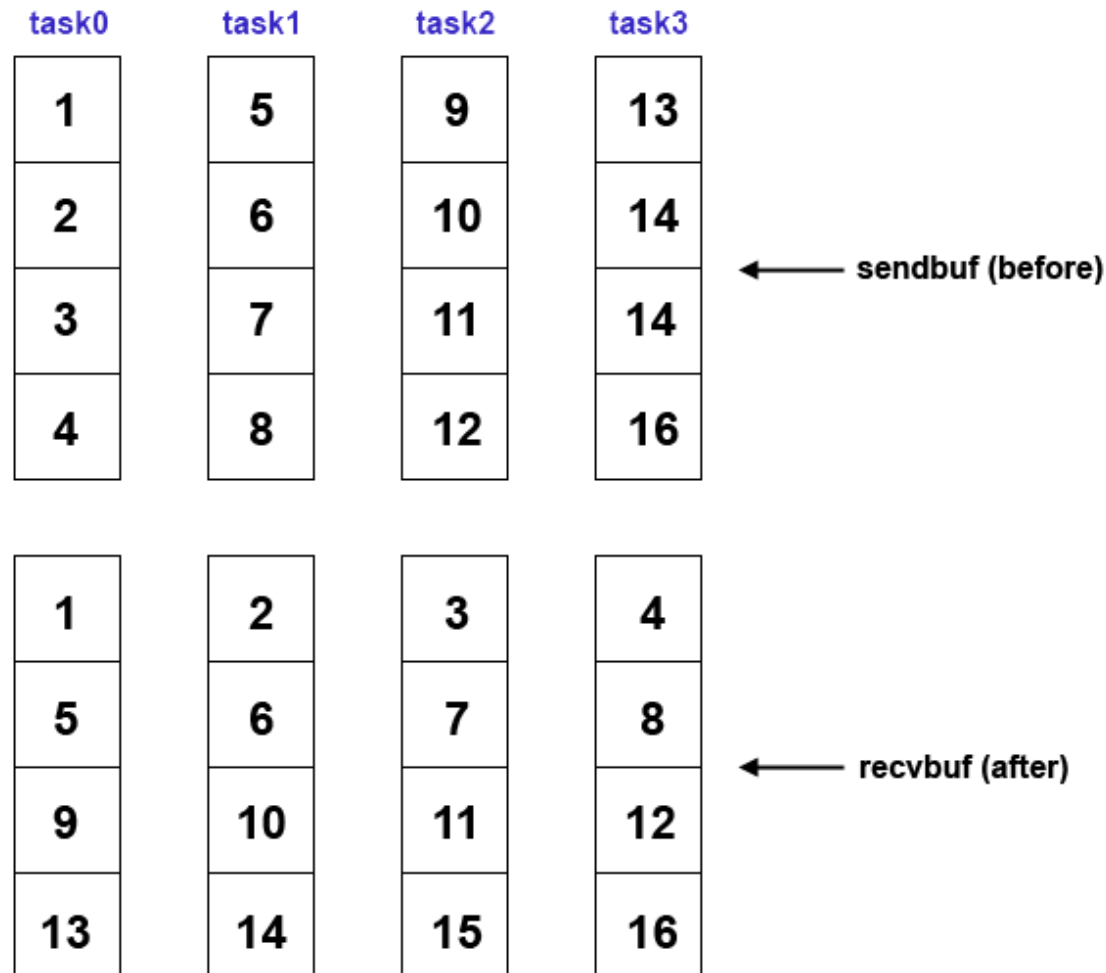
```
recvcnt = 1;  
MPI_Reduce_scatter(sendbuf, recvbuf, recvcount,  
                   MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



Data Movement

- MPI_Alltoall
(&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)
- Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index

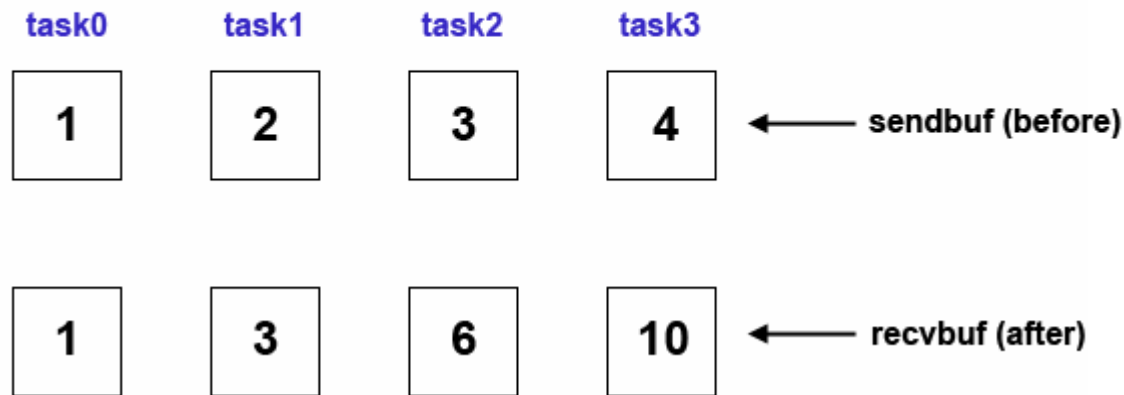
```
sendcnt = 1;
recvcnt = 1;
MPI_Alltoall(sendbuf, sendcnt, MPI_INT,
             recvbuf, recvcnt, MPI_INT,
             MPI_COMM_WORLD);
```



MPI_Scan

- `MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)`
- Performs a scan (partial reduction) operation with respect to a reduction operation across a task group.

```
count = 1;  
MPI_Scan(sendbuf, recvbuf, count, MPI_INT,  
         MPI_SUM, MPI_COMM_WORLD);
```



References

- <https://computing.llnl.gov/tutorials/mpi/>
- “An introduction to Parallel Programming”, Peter S. Pacheco (Chapter 3)



BITS Pilani
Pilani Campus



Thank You