Message Passing Interface (MPI)

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

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

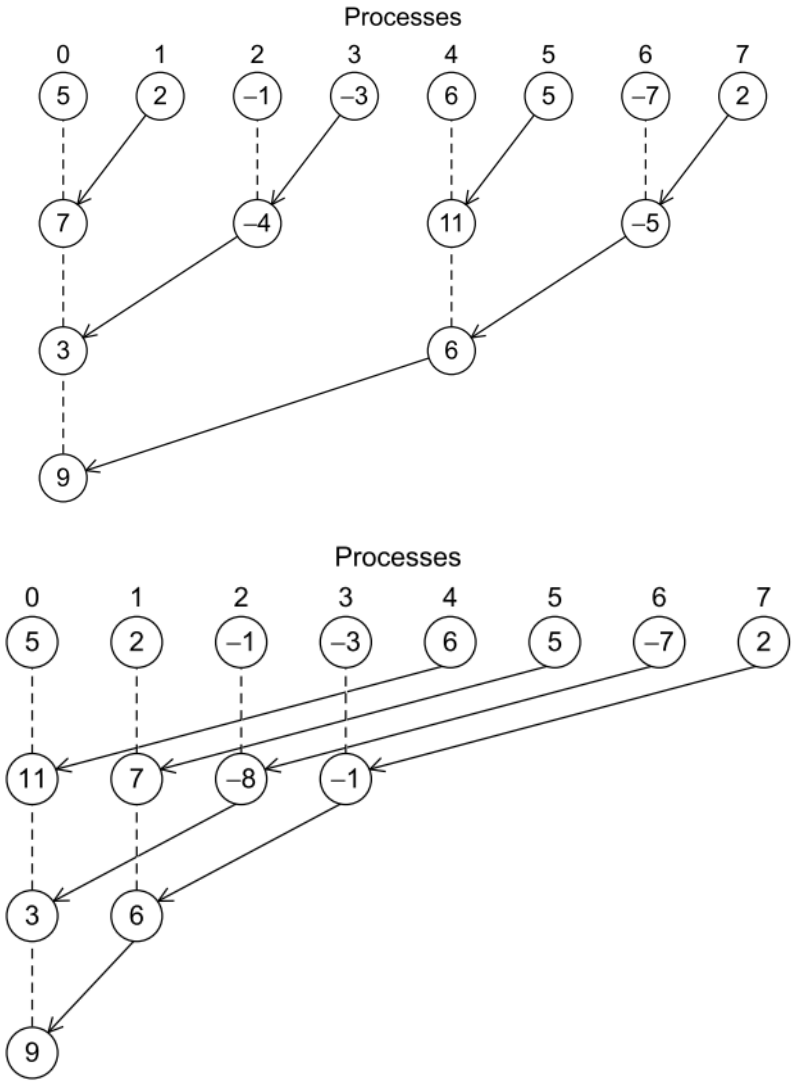
- In the code given on left, processes $P_1 \ldots P_{n-1}$ are simply calculating local integral and sending
- P_0 is calculating as well as receiving and summing up all integrals
- This not equitable load distribution
  - Leads to increase in serial code thus reducing speedup

```
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;
```

Tree-structured Communication

BITS Pilani

- Distributing load equally among the processes
  - One way: Processes 1,3,5, and 7 send their new values to processes 0,2,4, and 6 respectively. Further 2 and 6 send to 0 and 4. 4 sends to 0.
  - There can be several ways to optimize this
- MPI provides MPI_Reduce which takes care of this
  - Places the optimal implementation on the MPI API implementer not on the programmer

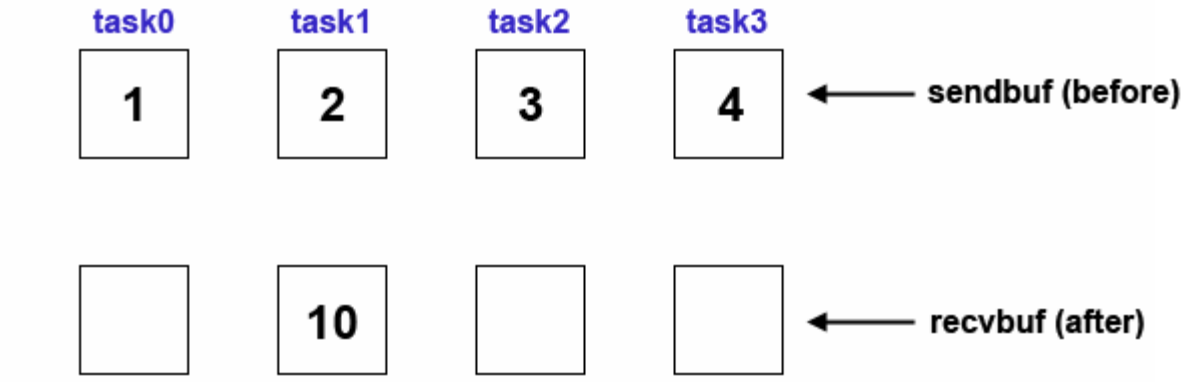


Collective Computation Operation

BITS Pilani

- 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

```
count = 1;
dest = 1;                                          task1 will contain result
MPI_Reduce(sendbuf, recvbuf, count, MPI_INT,
           MPI_SUM, dest, MPI_COMM_WORLD);
```



Source: https://computing.llnl.gov/tutorials/mpi/

Collective Computation Operation

BITS Pilani

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

BITS Pilani

- 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

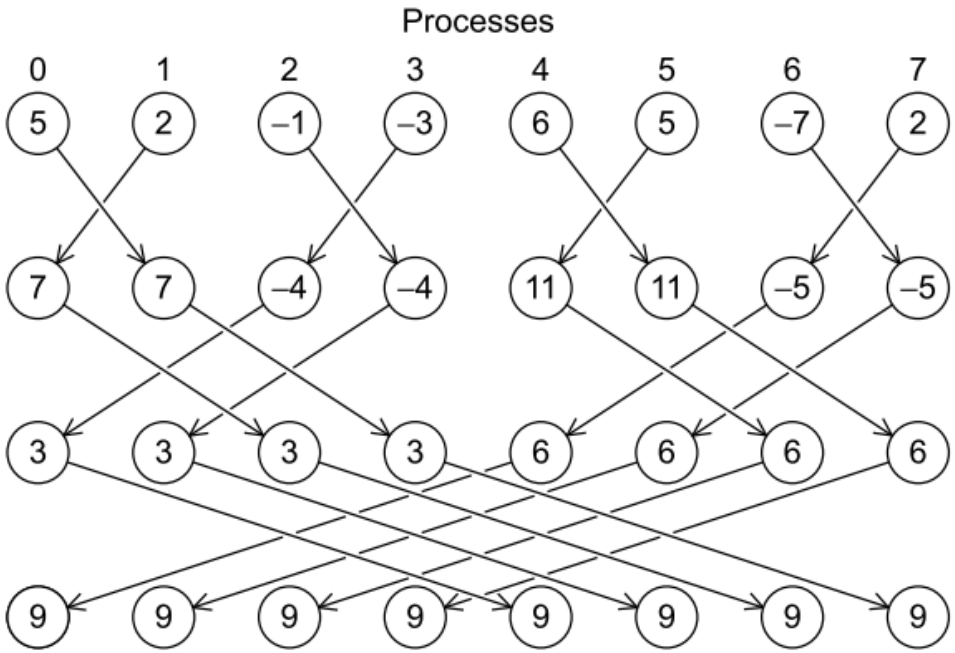
MPI_Allreduce

BITS Pilani

- 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

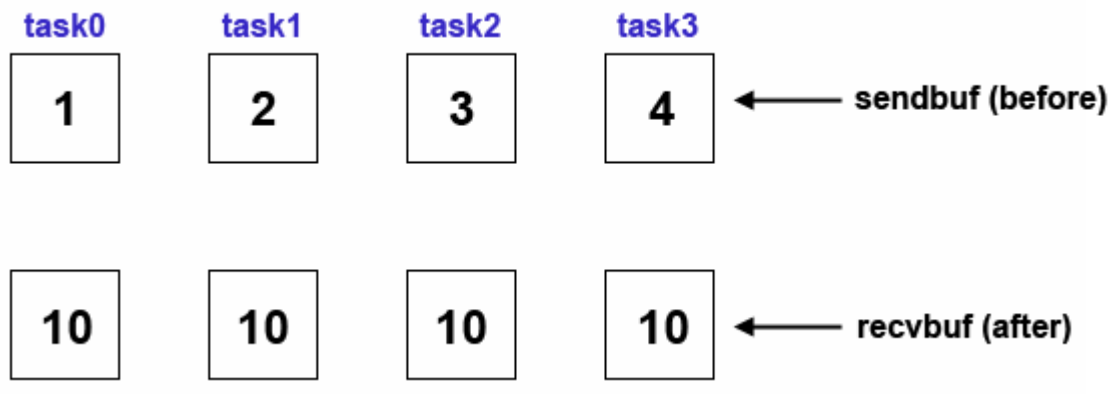
MPI provides MPI_Allreduce where the user need not worry about the communication pattern



Collective Computation Operation

BITS Pilani

- 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.



Source: https://computing.llnl.gov/tutorials/mpi/

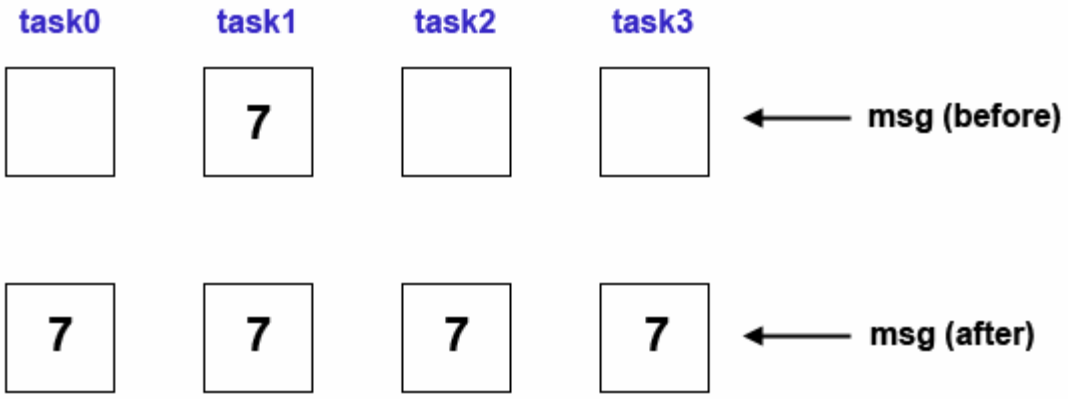
Data Movement Operation

BITS Pilani

- 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

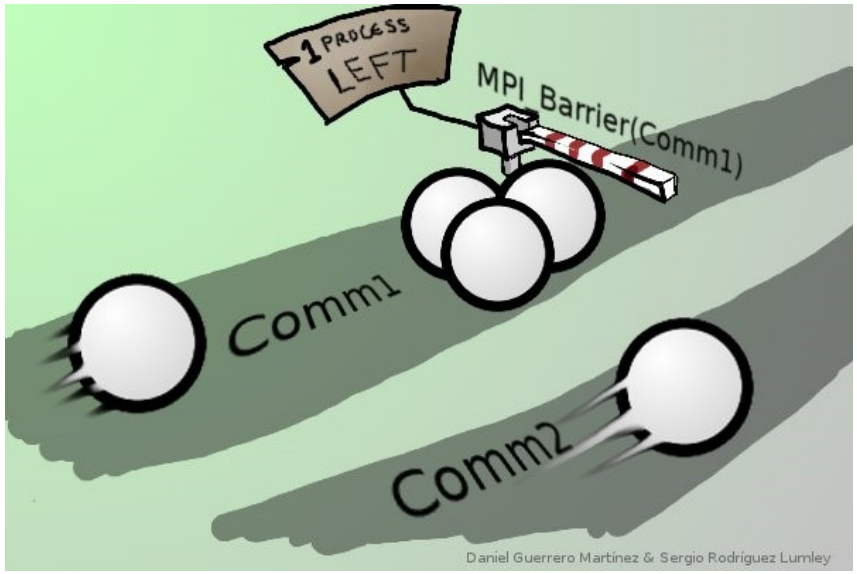


Source: https://computing.llnl.gov/tutorials/mpi/

Synchronization Operation

BITS Pilani

- *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



Source: https://lsi.ugr.es/jmantas/ppr/ayuda/mpi_ayuda.php?ayuda=MPI_Barrier&idioma=en

Broadcast Example

BITS Pilani

```
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

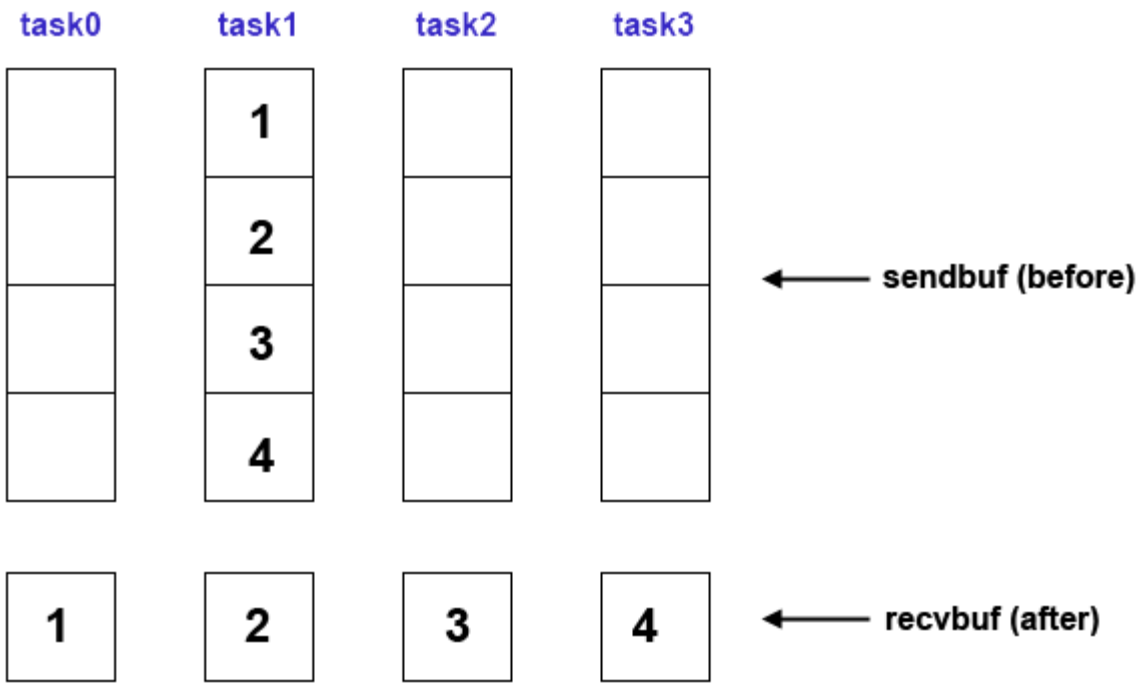
BITS Pilani

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

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

```
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



Source: https://computing.llnl.gov/tutorials/mpi/

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. }
```

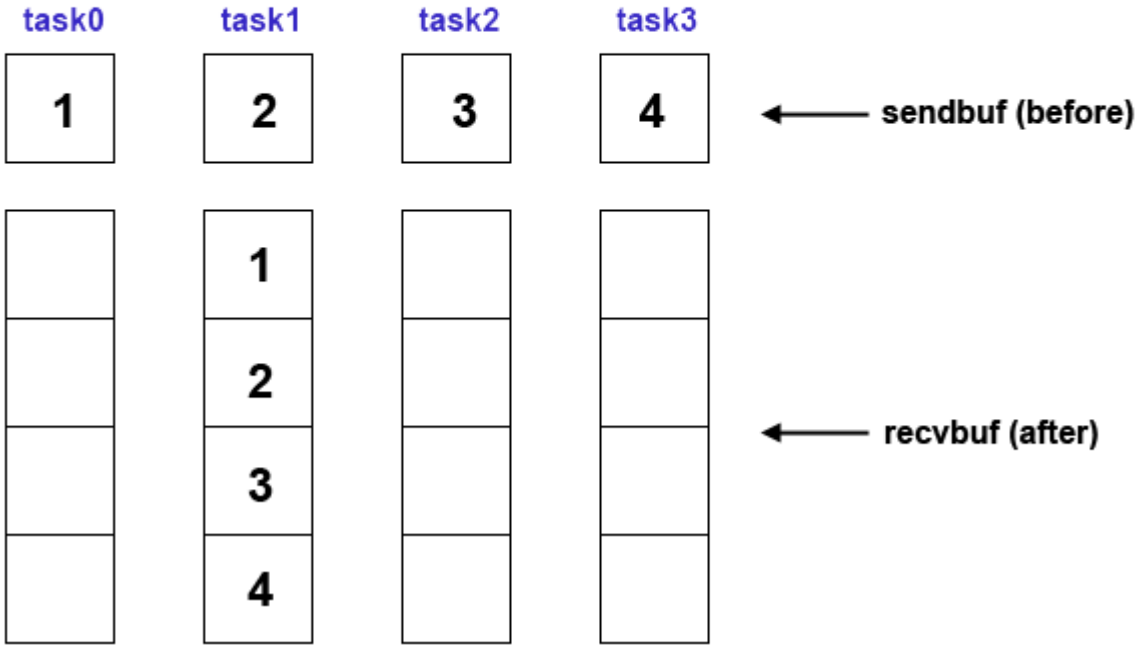
Data Movement Operation

BITS Pilani

- 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



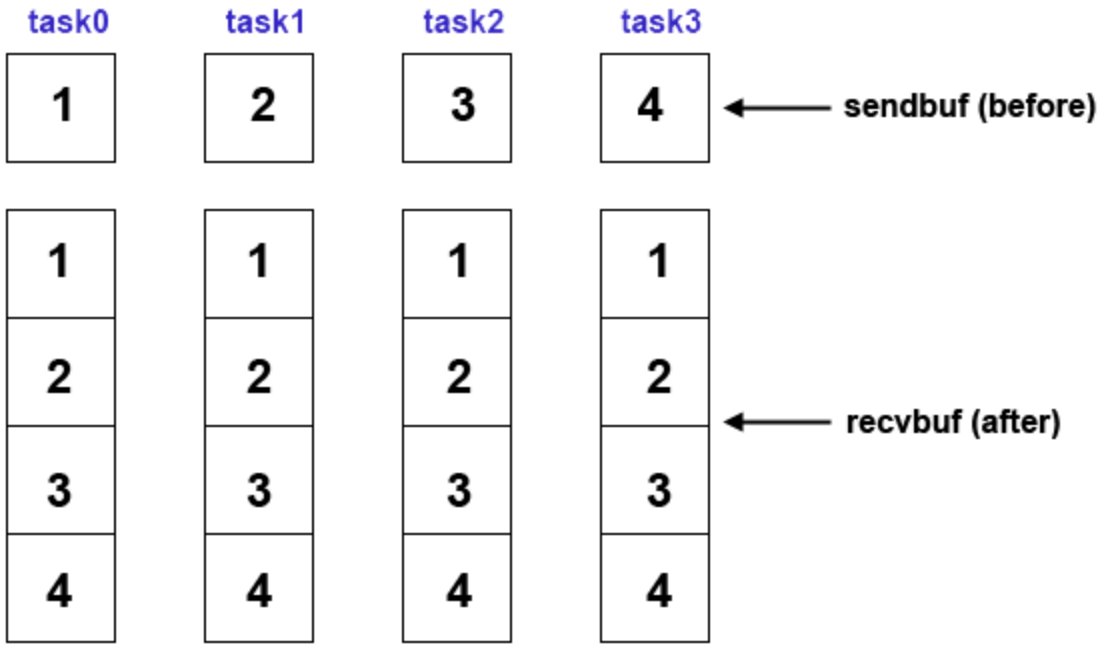
Source: https://computing.llnl.gov/tutorials/mpi/

Data Movement Operation

BITS Pilani

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

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



Source: https://computing.llnl.gov/tutorials/mpi/

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);
```

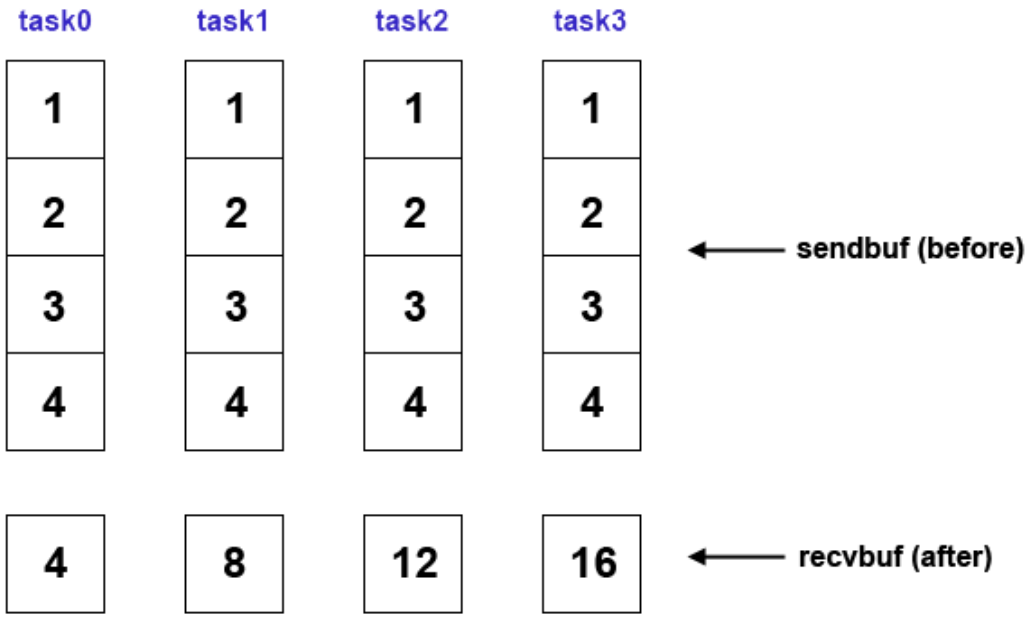
```
if (numtasks == SIZE){
// define source task and elements to send/receive,
then perform collective scatter
source = 1;
sendcount = SIZE;
recvcount = SIZE;
MPI_Scatter (sendbuf, sendcount, MPI_FLOAT,
recvbuf, recvcount, MPI_FLOAT, source,
MPI_COMM_WORLD);
//MPI_Bcast(m2,SIZE,MPi_FLOAT,0,MPI_COMM_W
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

BITS Pilani

- 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);
```

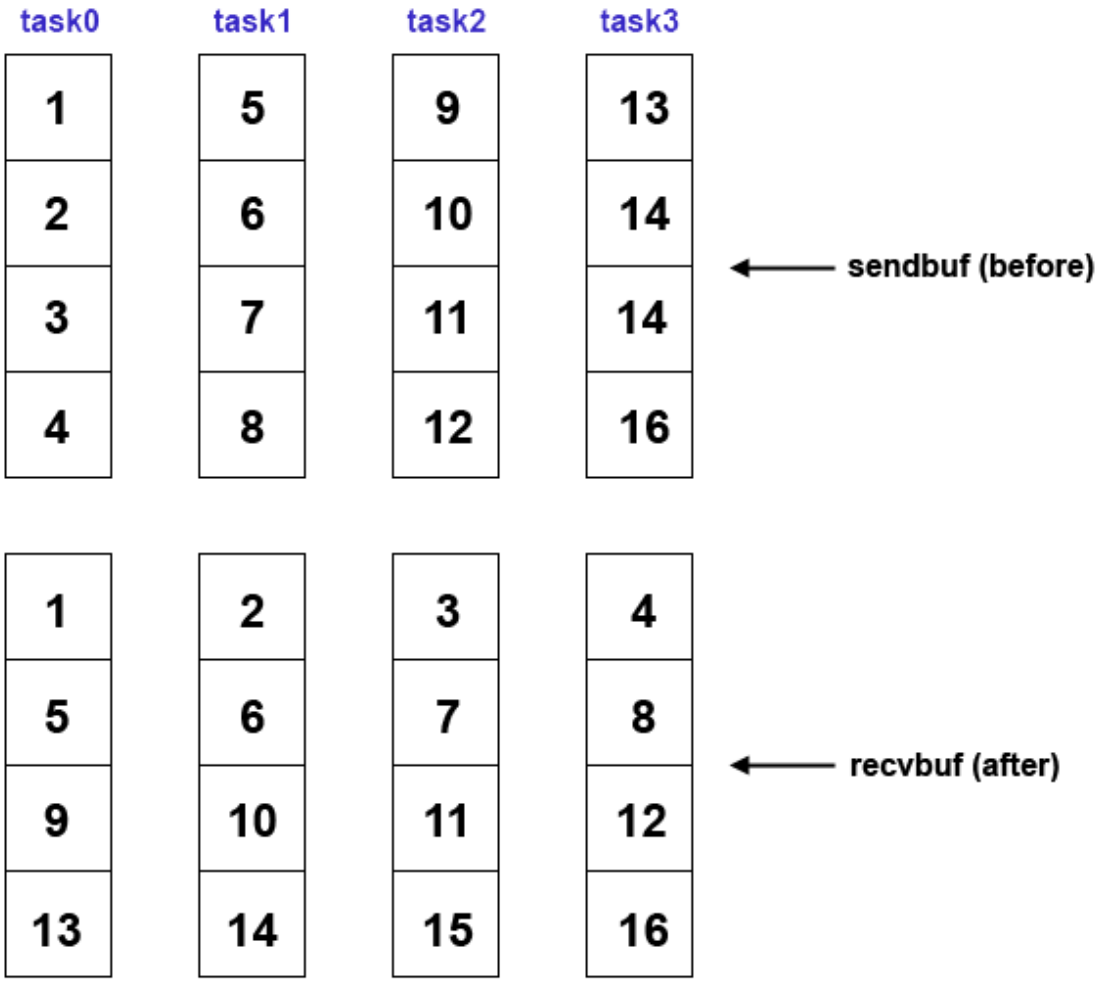


Source: https://computing.llnl.gov/tutorials/mpi/

Data Movement

- MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,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);
```

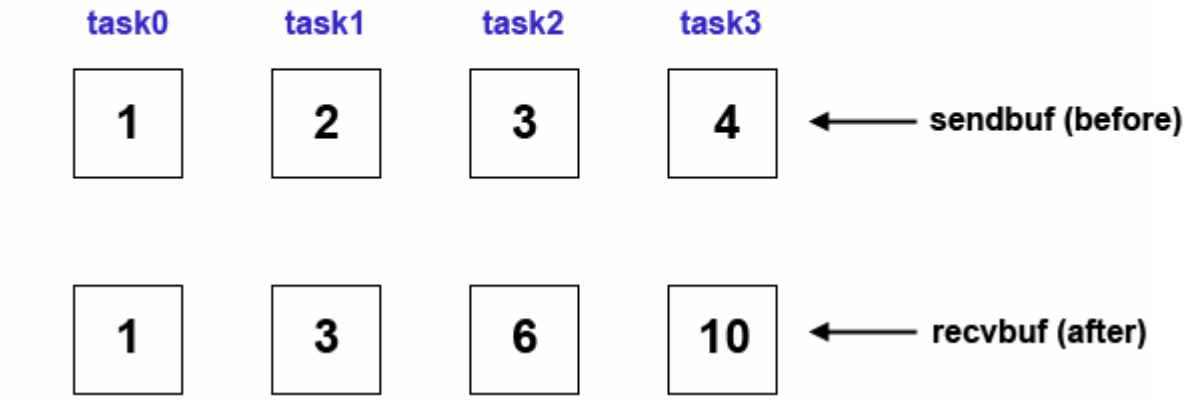


Source: https://computing.llnl.gov/tutorials/mpi/

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);
```



Source: https://computing.llnl.gov/tutorials/mpi/

References

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

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You