COSC 407
Intro to Parallel Computing

MPI: Collective Communication

Outline

Previously:

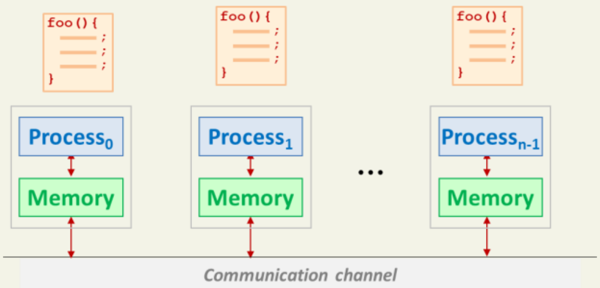
- Distributed Memory Programming
- Intro to MPI
- MPI: Point to Point communication
  - Example: Area under the curve in MPI
- Dealing with I/O

Today:

- MPI: Collective communication

Previously…

- A ***communicator*** has ***n nodes***, each with ***unique rank***.
- The nodes are interconnected by ***channels.***
- Each node can **`send`** and **`receive`**.
- Two communication scopes: ***point-to-point***, or ***collective***
- ***SPMD*** = Single-Program Multiple-Data.
- ***Message Matching*** means both sender and receiver must execute ***matching functions*** for sending and receiving.

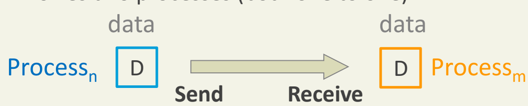


Two types of communications

Point-to-point communications:

Those function which involves two processes (both one to one)

- MPI_Send
- MPI_Recv

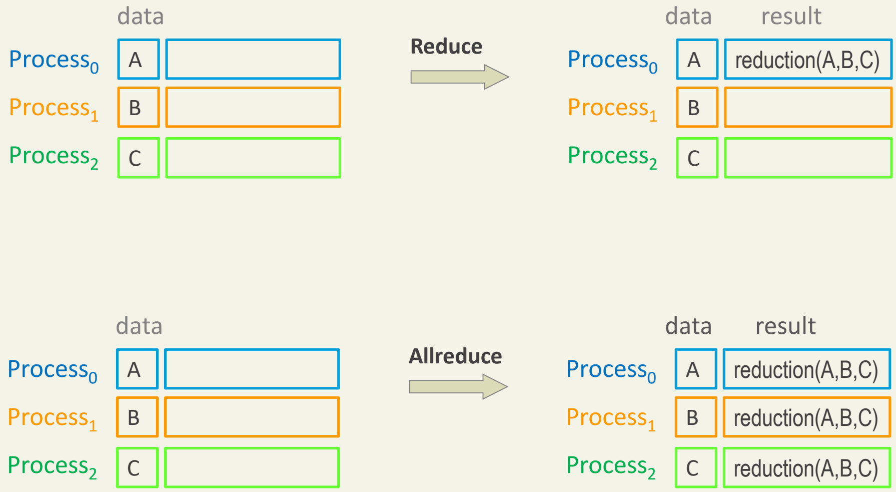


Collective Communication:

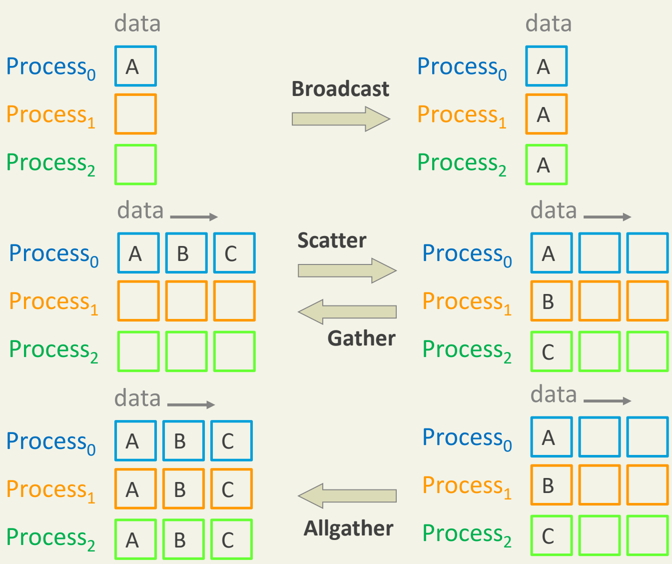
Those functions that involve all the processes in a communicator

- MPI_Reduce Reduction (all to one)
- MPI_Allreduce Reduction (all to all)
- MPI_Bcast Broadcast (one to all)
- MPI_Scatter Data distribution (one to all)
- MPI_Gather Data concatenation (all to one)
- MPI_Allgather Data concatenation (all to all)

Summary of Collective Comm

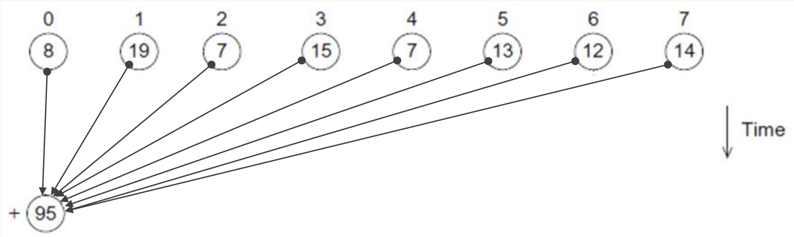


Summary of Collective Comm



Reduction: Discussing Trap Rule

- In the previous lecture, we wrote the trapezoid program using point-to-point communication as shown below.
  - Work load is not evenly distributed (process 0 has to do more work)
    - This is ok in case of reading the input
    - But this could be improved for computing the final result.
    - This is point-to-point communication



Remember: Trap Rule: V. 2

```
int main() {
   int my_rank, comm_sz, n, my_n, source;
   double a, b, h, my_a, my_b, my_sum, total_sum;
   //initialize, get rank and comm_sz
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   read_input(my_rank, comm_sz, &a, &b, &n);
   //break down the problem to subproblems
   h = (b-a) / n;                   // the same for all processes
   my_n = n / comm_sz;              // the same for all processes
   my_a = a + my_rank * my_n * h;   //unique to each process
   my_b = my_a + my_n * h;          //unique to each process
   my_sum = Trap(my_a, my_b, my_n, h); //find partial result from one process
   if (my_rank != 0) {              //send my partial result to process 0
      MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {                         //PROCESS 0 COMBINES THE PARTIAL RESULTS
      total_sum = my_sum;
      for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&my_sum,1,MPI_DOUBLE, source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        total_sum += my_sum;
      }
   }
   if (my_rank == 0)  printf("%.15e\n", total_sum);
   MPI_Finalize();
   return 0;
}
```

Tree-structured Communication

a) Process 1 sends to 0, 3 sends to 2, …etc. Processes 0, 2, 4, and 6 add in the received values.

b) Processes 2 and 6 send to 0 and 4. Processes 0 and 4 add the received values into their new values.

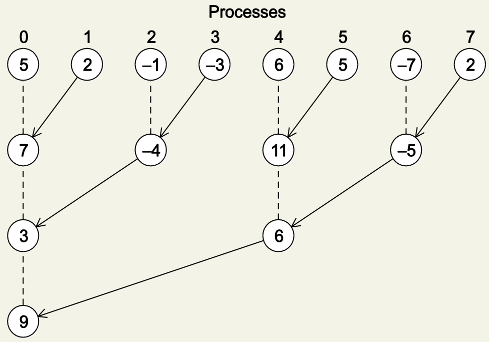
c) Process 4 sends its newest value to process 0. Process 0 adds the received value to its newest value.

The good:

Better workload distribution, Process 0 is doing less work (only 3 receives & 3 additions)

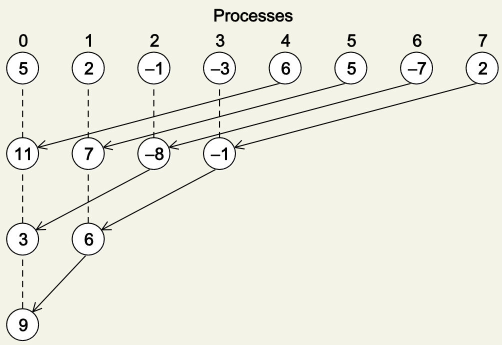
The bad:

- Processes 1,3,5,7 are still not contributing enough
- Requires Extra coding (not easy)!



An Alternative Tree-Structured Global Sum

- Tree structured communication can be planned in different ways. For example, here is another way, and there are many other possibilities.
- Note that this structure suffers from the *same problems as the previous process.*



Is there a way that doesn't involve much coding and testing to see which structure is best?

MPI Reduction

- With the endless possibilities, it's *not reasonable* to ask each MPI programmer to write an optimal global-sum function (much more code and a lot of testing to find the optimal way).
- Therefore, MPI includes implementations of reduction operators.
- This means the burden of the extra code is put on the developer of the MPI implementation, rather than the application developer.

- Two functions that can be used for global reduction:
  - MPI_Reduce Reduction (all to one)
  - MPI_Allreduce Reduction (all to all)

Obviously, reduction is a form of *collective communication*

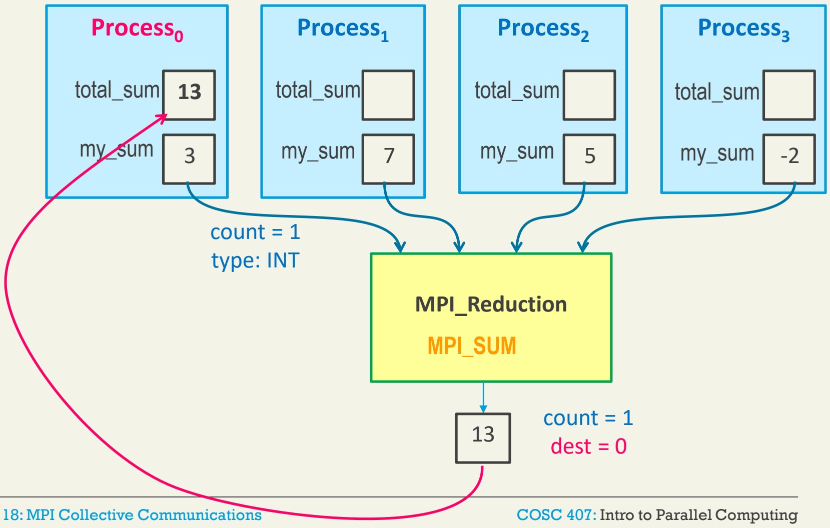
MPI_Reduce

```
int MPI_Reduce(
  void*          input_data_p,      /* in: pointer to input data */
  void*          output_data_p,     /* out: pointer to output data */

  int            count,             /* in: how many elements in i/p & o/p? */
  MPI_Datatype   datatype,          /* in: datatype of elements in i/p & o/p */

  MPI_Op         operator,          /* in: What type of reduction? */

  int            dest_process,      /* in: ONE process receives results */
  MPI_Comm       comm               /* in: which communicator? */
);
```

Example 1

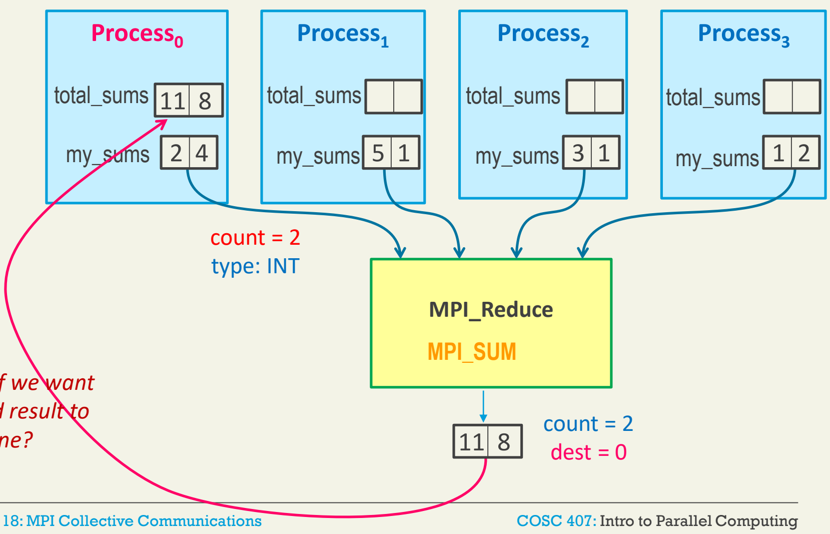
Count

```
int my_sum, total_sum;
my_sum = compute();
MPI_Reduce(&my_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```



Example 2

```
int my_sums[2], total_sums[2];
my_sums = compute();     //my_sums is an array of 2 elements
MPI_Reduce(&my_sums,&total_sums,2,MPI_INT,MPI_SUM,0, MPI_COMM_WORLD);
```



What if we want to send result to everyone?

MPI_Allreduce

```
int MPI_Allreduce(
  void*          input_data_p,    /* in:  address of input data */
  void*          output_data_p,   /* out: address of output data */
  int            count,           /* in:  how many elements in i/p & o/p? */
  MPI_Datatype   datatype,        /* in:  datatype of elements in i/p & o/p */
  MPI_Op         operator,        /* in:  What type of reduction? */
  MPI_Comm       comm             /* in:  which communicator? */
);
```

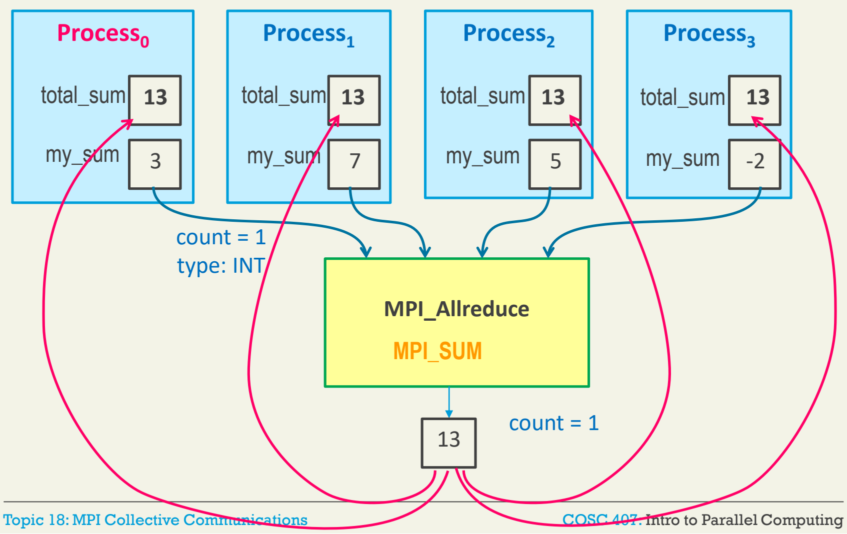
there is **no destination process**

Same as MPI_Reduce except that result is sent to all processes.

– e.g., all processes need global sum to complete some larger computation.

Example 3

```
int my_sum, total_sum; my_sum = compute();
MPI_Allreduce(&my_sum,&total_sum,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
```



Implementation of MPI_Allreduce

- Note that there are several possible ways of implementing reduction followed by distribution of data to all nodes. Again, it is left to the developer of the MPI implementation to code the best implementation of the operation.

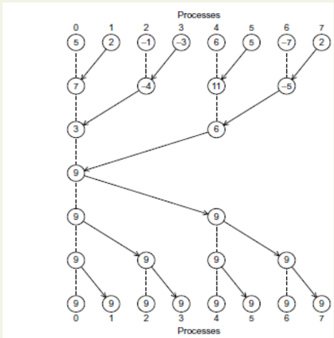


FIGURE 3.8
A global sum followed by distribution of the result

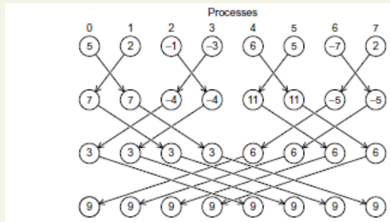


FIGURE 3.9
A butterfly-structured global sum

Reduction Operators

- Predefined **reduction operators** in MPI

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Compare with Version 2

```
int main() {          //version 2: without reduction
   int my_rank, comm_sz, my_n, source;
   double a, b, h, my_a, my_b, my_sum, total_sum;
   //initialize, get rank and comm_sz
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   read_input(my_rank, comm_sz, &a, &b, &n);
   //break down the problem to subproblems
   h = (b-a) / N;                 // the same for all processes
   my_n = N / comm_sz;                       // the same for all processes
   my_a = a + my_rank * my_n * h;            //unique to each process
   my_b = my_a + my_n * h;                   //unique to each process
   my_sum = Trap(my_a, my_b, my_n, h); //find partial result
   //manual reduction (by the programming)
   if (my_rank != 0) {                       //send my partial result to process 0
      MPI_Send(&my_sum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
   } else {                                  //process 0 combines the partial results
      total_sum = my_sum;
      for (source = 1; source < comm_sz; source++) {
        MPI_Recv(&my_sum,1,MPI_DOUBLE, source,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        total_sum += my_sum;
      }
   }
   if (my_rank == 0)  printf("%.15e\n", total_sum);
   MPI_Finalize();
   return 0;
}
```


Compare with Version 2

```
int main() {          //version 3: using reduction
   int my_rank, comm_sz, my_n;
   double a, b, h, my_a, my_b, my_sum, total_sum;
   //initialize, get rank and comm_sz
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   read_input(my_rank, comm_sz, &a, &b, &n);
   //break down the problem to subproblems
   h = (b-a) / N;                 // the same for all processes
   my_n = n / comm_sz;                       // the same for all processes
   my_a = a + my_rank * my_n * h;            //unique to each process
   my_b = my_a + my_n * h;                   //unique to each process
   my_sum = Trap(my_a, my_b, my_n, h); //find partial result
   //reduction (by the MPI implementation)
   MPI_Reduce(&my_sum,&total_sum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

   if (my_rank == 0)  printf("%.15e\n", total_sum);
   MPI_Finalize();
   return 0;
}
```

Ask

Collective vs. Point-to-Point Communications

1. ALL the processes in the communicator must call the **same collective function**..
   – For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is **erroneous**, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "**compatible**."
   – For example, 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**, and, once again, the program is likely to hang or crash.
3. In case of MPI_Reduce, The output_data_p argument is only used on dest_process
   – However, all of the processes still need to pass in an actual argument corresponding to output_data_p, even if it's just NULL.

Collective vs. Point-to-Point Communications

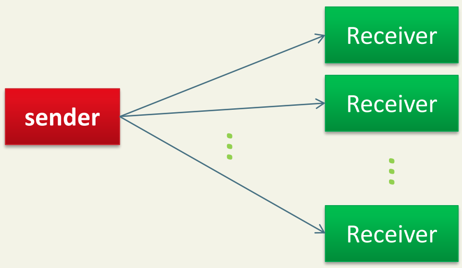
4. **BE CAREFUL**: **Point-to-point communications** are matched on the basis of **tags** and **communicators**.
   Collective communications don't use tags. They're matched solely on the basis of the communicator and the ***order*** in which they're called.
   – Example: Assume each process calls MPI_Reduce with MPI_SUM, and **destination** process is 0.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

- At first glance, it might seem that after the two calls to MPI_Reduce, b on Process 0 will be 3 (b=a+a+a), and the value of d will be 6 (d=c+c+c).
- However, the **names of the memory locations are irrelevant** to the matching of the calls to MPI_Reduce. The order of the calls will determine the matching so: **b** will be 1 + 2 + 1 = 4 ($b_{Proc0} = a_{Proc0} + c_{Proc1} + a_{Proc2}$)
  d will be 2 + 1 + 2 = 5 ($d_{Proc0} = c_{Proc0} + a_{Proc1} + c_{Proc2}$)

Broadcasting

- **Broadcast** is a collective communication in which data belonging to a single process is sent to all of the processes in the communicator.
- The distribution of data could be done in ***different ways***. Similarly to reduction, the *MPI implementation decides* the best way of doing this while balancing the workload.
- Conceptually, we are going to think of a broadcast as shown below:



MPI_Bcast

```
int MPI_Bcast(
  void*          data_p,      /* in/out: address of in/out data */

  int            count,       /* in: how many elements to send/receive */
  MPI_Datatype datatype,      /* in: datatype of elements */

  int            sourc_proc,  /* in: source process */
  MPI_Comm       comm         /* in: which communicator? */ );
```

source_proc sends its data to everyone else

Getting an Input With and Without Broadcast

Without

```
if (my_rank==0){   //only Process 0 reads input then sends it to everyone
    data = readData();
    for each node other than 0:
         MPI_Send(process 0 sends data to node);
} else {           //everyone else receive data from process 0
    MPI_Recv(node receives data from 0);
}
```

With

```
if (my_rank==0)    //only Process 0 reads input
    data = readData();
MPI_Bcast(process 0 sends data to everyone; everyone receive data from 0)
```


read_input Without Broadcasting

```
void read_input(int my_rank,int comm_sz,double* a_p,double* b_p,int* n_p){
   int dest;
   if (my_rank==0){         //only Process 0 reads input
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT,    dest, 0, MPI_COMM_WORLD);
      }
   } else {                 //everyone else receive data from process 0
      MPI_Recv(a_p,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Recv(b_p,1,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      MPI_Recv(n_p,1,MPI_INT,   0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
   }
}
```

read_input WITH Broadcasting

```
void read_input(int my_rank,int comm_sz,double* a_p,double* b_p,int* n_p){
   int dest;
   if (my_rank==0){      //only Process 0 reads input
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT,    0, MPI_COMM_WORLD);
}
```

Everyone MUST execute MPI_Bcast so that it can be matched.
Process 0 sends, everyone else receive.

Data Partitioning Options

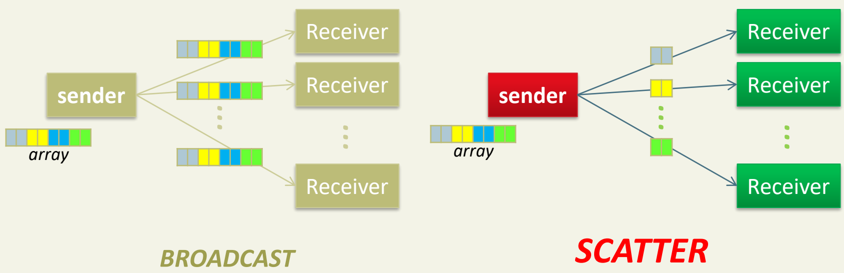
- We can partition the data and send them over to different processes. We might consider these options
  - Block partitioning (THE EASIEST – **We shall use this one**)
    - Assign blocks of consecutive components to each process.
  - Cyclic partitioning
    - Assign components in a round robin fashion.
  - Block-cyclic partitioning
    - Use a cyclic distribution of blocks of components.

Example: partitioning 12-component vector among 3 processes

Process	Components: Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Scattering

- **MPI Scattering** is a collective routine that similar to broadcasting. The main difference is that:
  - ***broadcasting*** sends the *same* piece of data to all processes,
  - ***scattering*** sends ***chunks*** *of an array* to different processes.



MPI_Scatter

```
int MPI_Scatter(
   void*          send_buffer_p,    /* in:  array of data on src process */
   int            send_count,       /* in:  # elements to send to EACH process*/
   MPI_Datatype   send_type,        /* in:  datatype of elements */

   void*          recv_buffer_p,    /* in:  this is where to received data */
   int            recv_count,       /* in:  capacity of recv buffer*/
   MPI_Datatype   recv_type,        /* in:  datatype of elements */

   int            src_process,      /* in:  who sent the data*/
   MPI_Comm       comm              /* in:  which communicator? */ );
```

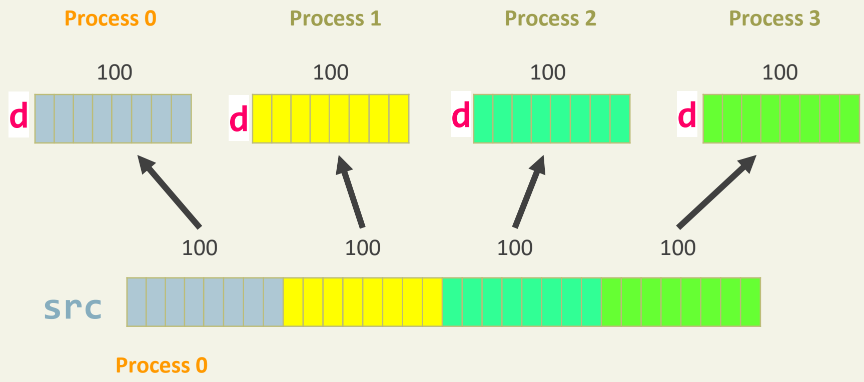
MPI_Scatter **uses block partitioning** with the assumption that the length of data is divisible by comm_sz. With this assumption,

send_count = send_receive = N / comm_sz

In this course, we will not consider other cases!

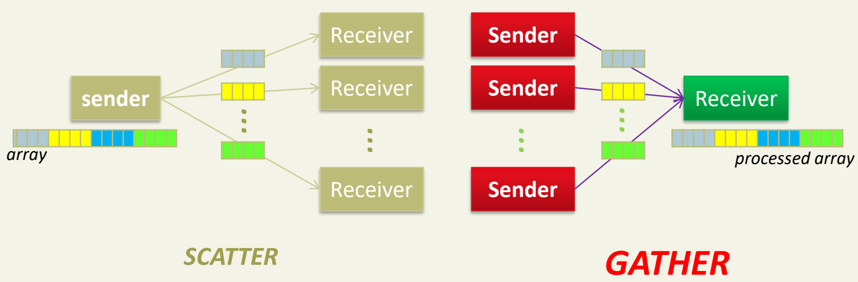
Scatter Example

```
MPI_Scatter(src,100,MPI_INT,d,100,MPI_INT,0,MPI_COMM_WORLD)
```



Gathering

- **MPI Gathering** is a collective routine that performs the inverse of scattering. Gathering combines data chunks from other processes into one big piece of data.
  - The chunks are ordered by the rank of the sending process.



MPI_Gather

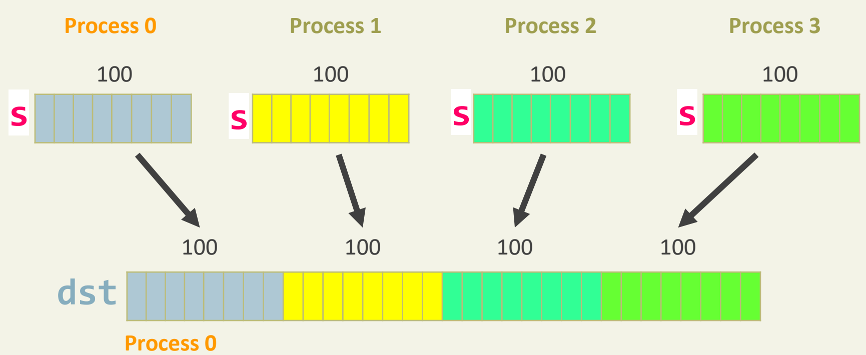
```
int MPI_Gather(
  void*          send_buffer_p,   /* in: array of data on src process */
  int            send_count,      /* in: # elements in send buffer*/
  MPI_Datatype send_type,         /* in: datatype of elements */

  void*          recv_buffer_p,   /* in: this is where to received data */
  int            recv_count,      /* in: #elements received from each process */
  MPI_Datatype recv_type,         /* in: datatype of elements */

  int            dest_process,    /* in: who collects data */
  MPI_Comm       comm             /* in: which communicator? */
);
```

Assuming number of elements in array is N, then

send_count = send_receive = N / comm_sz

Gather Example

```
MPI_Gather(s,100,MPI_INT,dst,100,MPI_INT,0,MPI_COMM_WORLD)
```



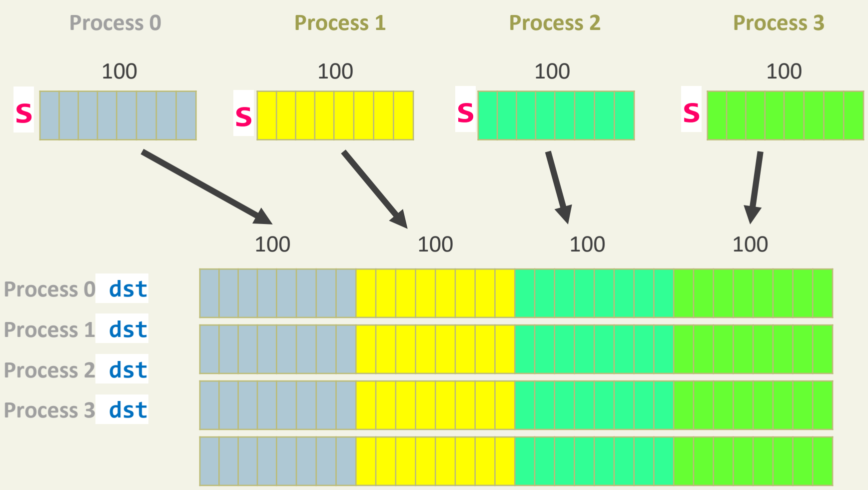
MPI_Allgather

Same as gather, but all processes receive a copy of the concatenated vector
- there is no id for receiver process.

```
int MPI_Allgather(
   void*          send_buffer_p,   /* in: array of data on src process */
   int            send_count,      /* in: # elements to send to DEST process*/
   MPI_Datatype   send_type,       /* in: datatype of elements */

   void*          recv_buffer_p,   /* in: this is where to received data */
   int            recv_count,      /* in: #elements received from each process */
   MPI_Datatype   recv_type,       /* in: datatype of elements */

   MPI_Comm       comm             /* in: which communicator? */ );
```


Gather Example

```
MPI_Allgather(s,100,MPI_INT,dst,100,MPI_INT,MPI_COMM_WORLD)
```



Conclusion

Today:

- MPI: Collective communication

Next day:

- Examples
- MPI Wrap-up
- Course Conclusion