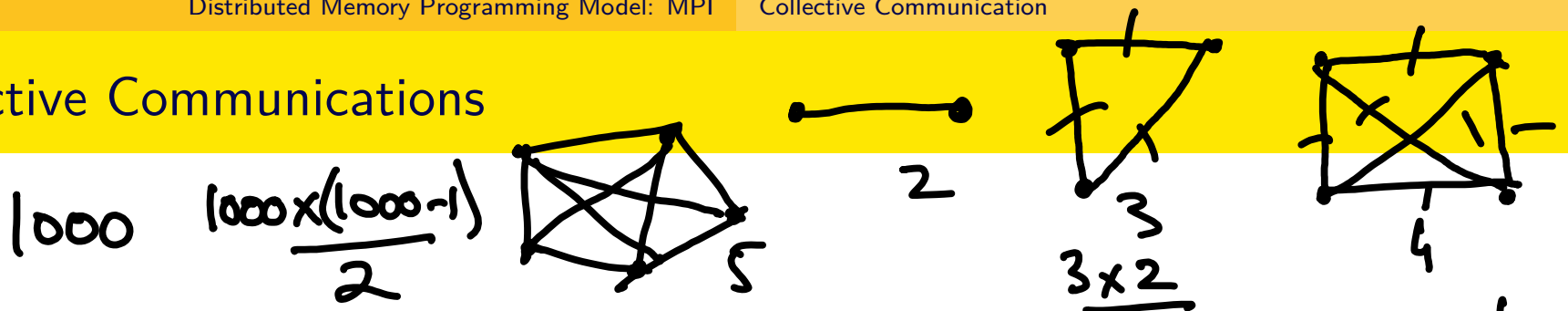
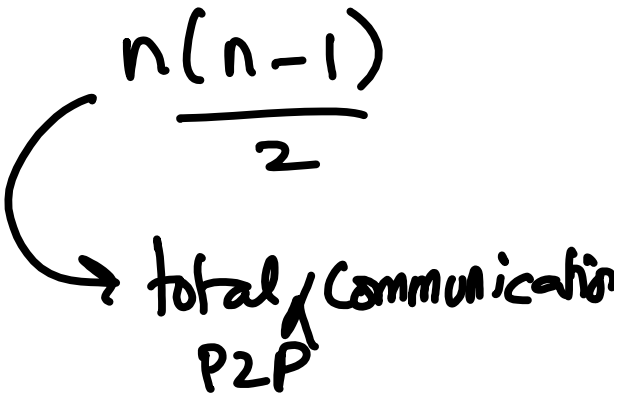
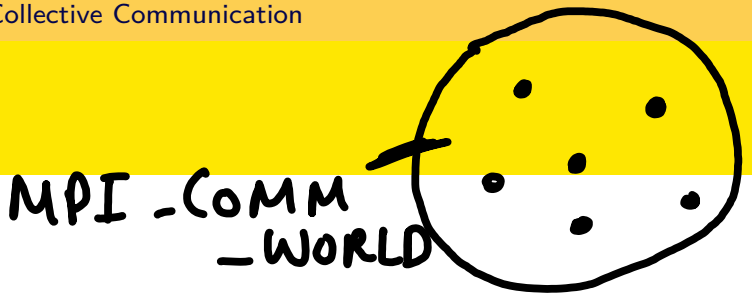
Collective Communications



- Point-to-Point: It is programmer's responsibility to ensure that all processes participate correctly in a given communication (Programmer's burden)
- MPI simplifies this using Collective Communication. Types are:
  - **Synchronization**:
    - Barriers: MPI_Barrier()
  - **Moving Data**:
    - Broadcasting: MPI_Bcast()
    - Scattering: MPI_Scatter()
    - Gathering: MPI_Gather()
  - **Collective Computation**:
    - Reduction: MPI_Reduce()
- Difference to point-to-point communications
  - No message tags
  - Most calls/versions support blocking communication only



Synchronization: Barrier

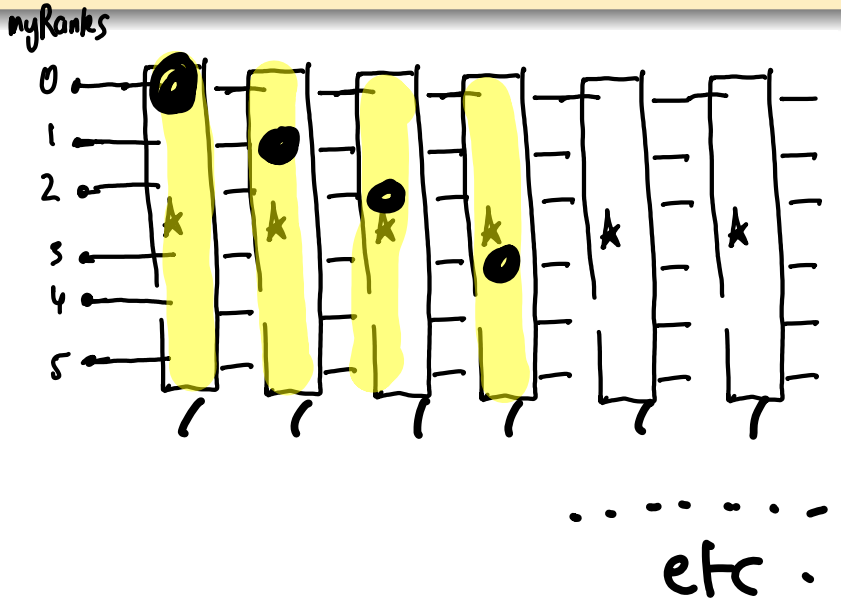


Suggestive.

```
int MPI_Barrier(MPI_Comm comm)
```

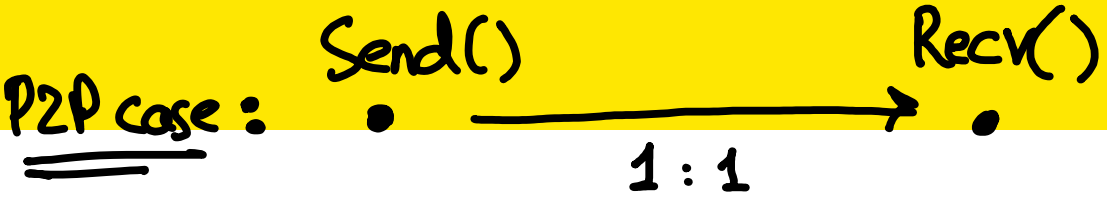
```
int x, myRank;
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

for (x = 0; x < 10; x++) {
    MPI_Barrier(MPI_COMM_WORLD);
    if (x == myRank) {
        printf("%d \n", myRank);
    }
}

MPI_Finalize();
```

1 iteration
5 iterations discarded.

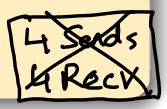


Communication – Computation

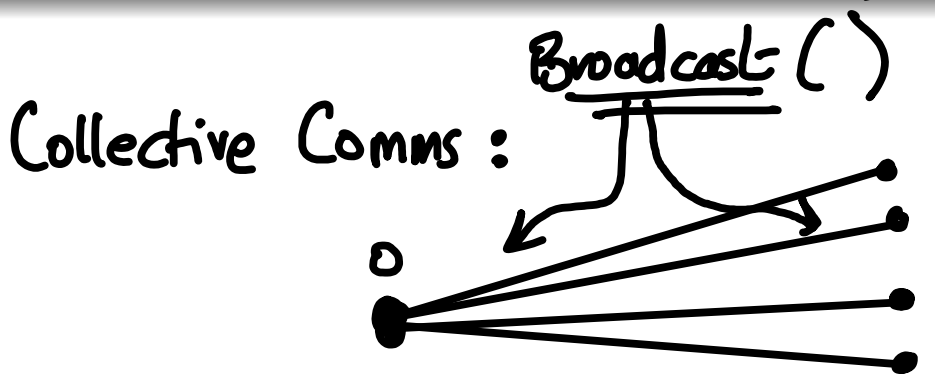
Moving Data: Broadcast




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



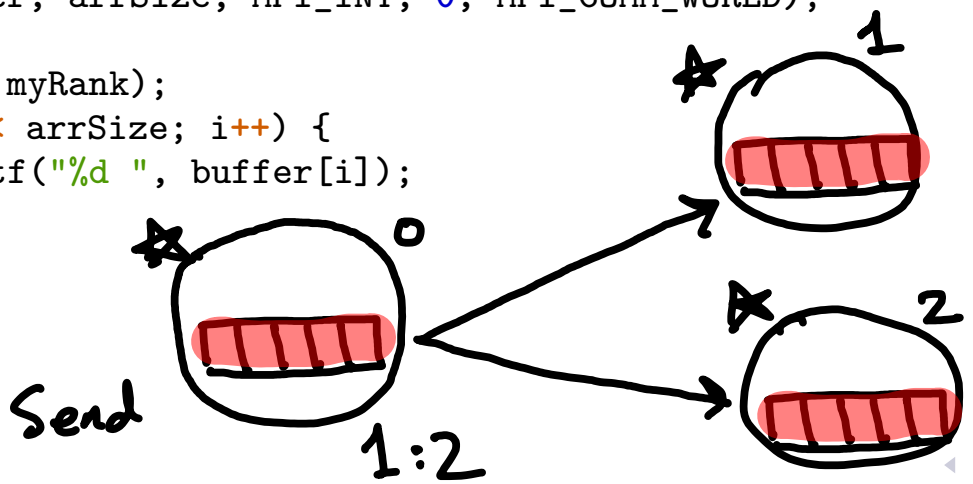

```
int arrSize = 10;
int *buffer = malloc(sizeof(int)*arrSize);

if (myRank == 0) {
    for (i = 0; i < arrSize; i++) {
        buffer[i] = i;
    }
}

MPI_Bcast(buffer, arrSize, MPI_INT, 0, MPI_COMM_WORLD);
```



Same API Call but sender/receiver treats it differently.

1 : M

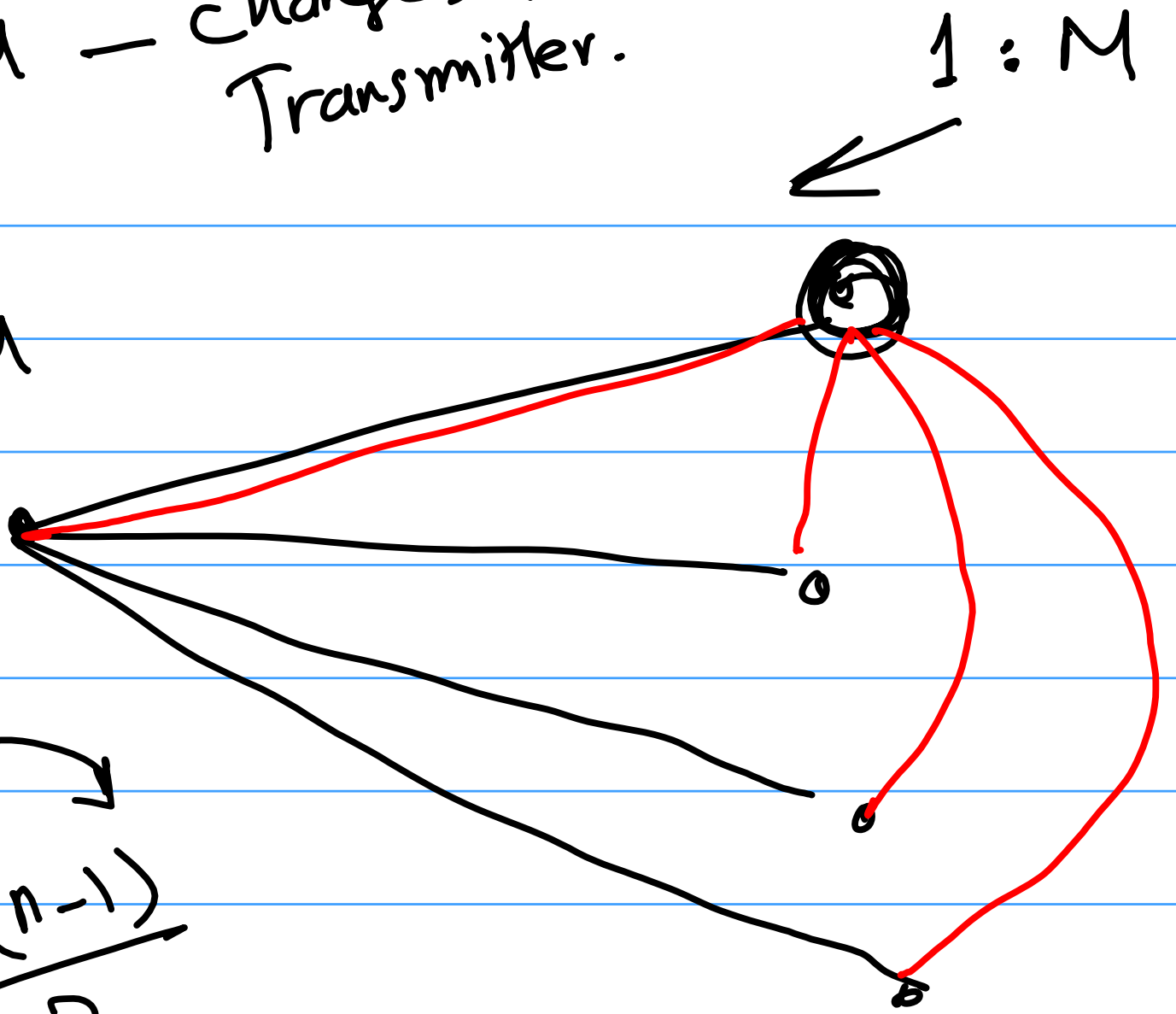
```
if( )
printf("%d: ", myRank);
for (i = 0; i < arrSize; i++) {
          printf("%d ", buffer[i]);
}
printf("\n");
```



Drawback

1 : M — changes.

Transmitter.



N = M - 1

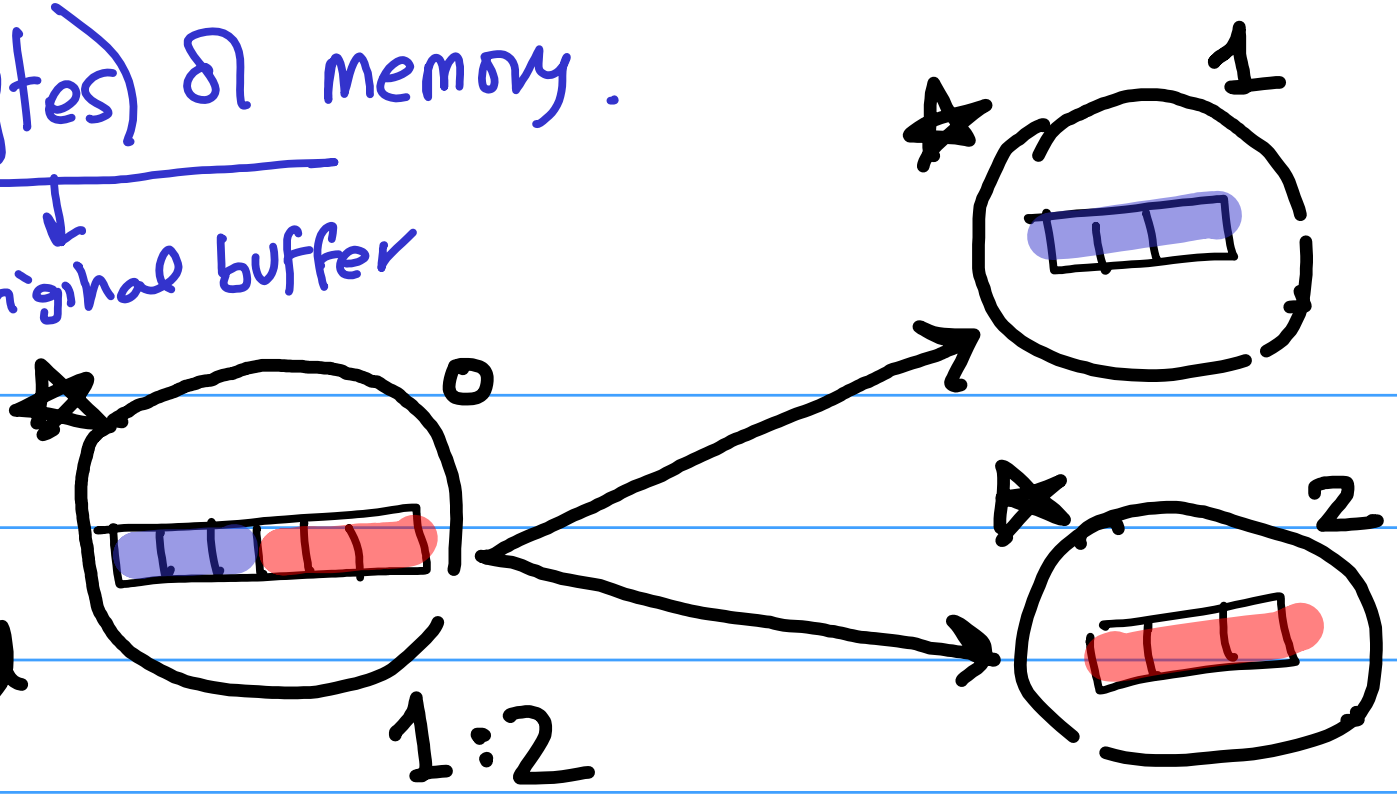
M : N

$\frac{n(n-1)}{2}$

(m x n bytes) of memory.

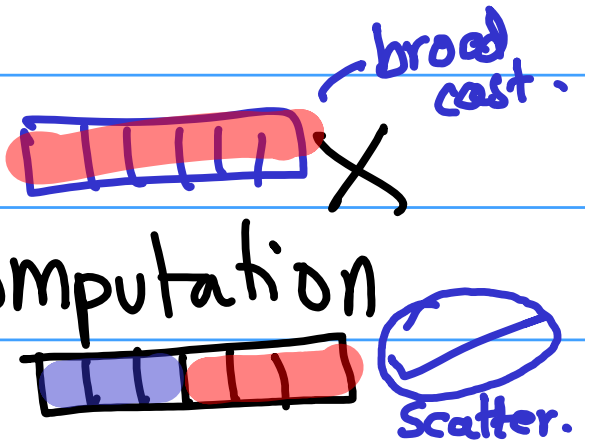
broad cast operation.

original buffer

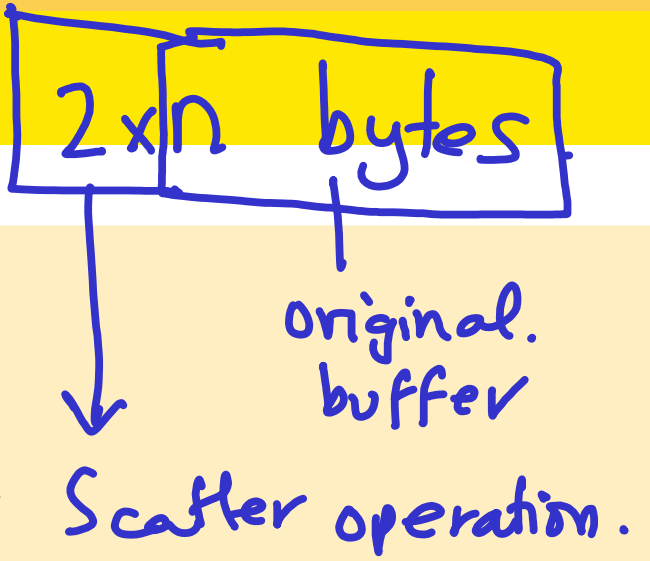


Why?

Sharing data ✓



Moving Data: Scatter




```
int MPI_Scatter(
      void *sendbuf,          // full size buffer
      int sendcount,          // addresses to send
      MPI_Datatype sendtype, // data type to send
      void *recvbuf,          // chunk size buffer
      int recvcount,          // how many are to be received
      MPI_Datatype recvtype, // datatype of receive type
      int root,               // who is the source
      MPI_Comm comm);         // communication world
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;

int *buffer  = malloc(sizeof(int)*fullSize);
int *buf     = malloc(sizeof(int)*reducedSize);

if (myRank == 0)     for (i = 0; i < fullSize; i++)     buffer[i] = i;

MPI_Scatter(buffer, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

printf("%d: ", myRank);
for (i = 0; i < reducedSize; i++)          printf("%d ", buf[i]);
printf("\n");
```

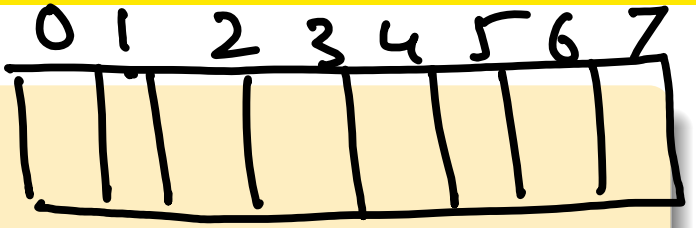
Moving Data: Scatter

1 : M

Map-Reduce

$n_r = 1$

$n = 8$ int

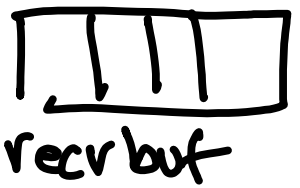


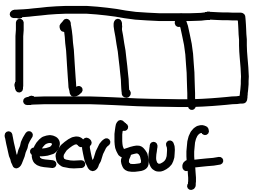

```
int MPI_Scatter(
    void *sendbuf,           // full size buffer
    int sendcount,           // addresses to send
    MPI_Datatype sendtype,   // data type to send
    void *recvbuf,           // chunk size buffer
    int recvcount,           // how many are to be received
    MPI_Datatype recvtype,   // datatype of receive type
    int root,                // who is the source
    MPI_Comm comm);          // communication world
```

processors = 2

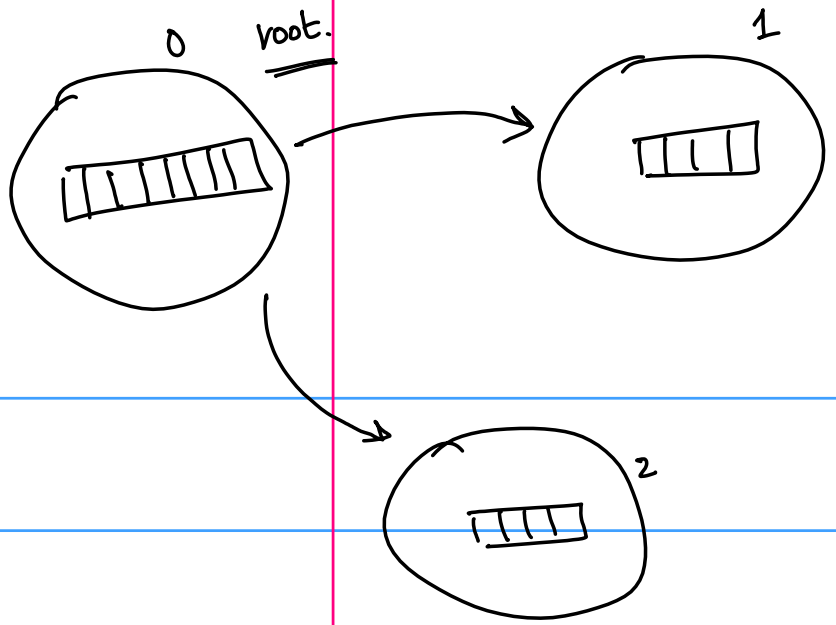
$n_r = n/\text{processors} = 4$

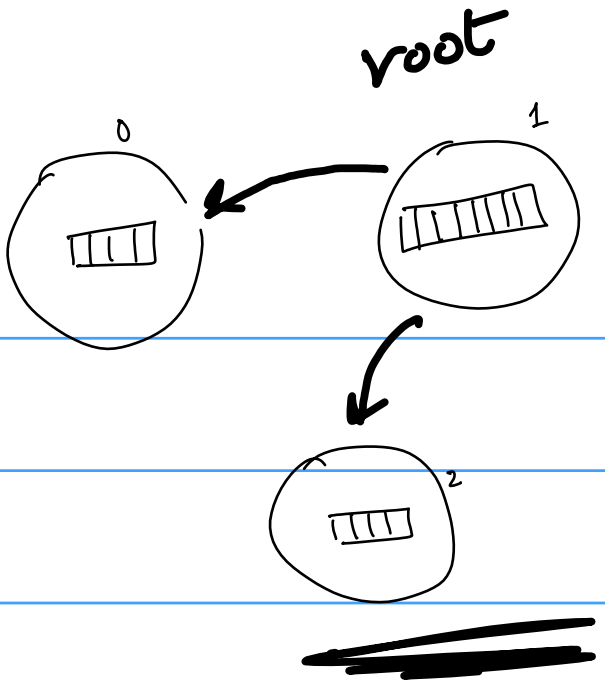
```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;

int *buffer  = malloc(sizeof(int)*fullSize);
int *buf     = malloc(sizeof(int)*reducedSize);

if (myRank == 0)     for (i = 0; i < fullSize; i++)     buffer[i] = i;

MPI_Scatter(buffer, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

printf("%d: ", myRank);
for (i = 0; i < reducedSize; i++)          printf("%d ", buf[i]);
printf("\n");
```





Tx Rx





etc.

Moving Data: Scatter & Gather

Map - Reduce

Broadcast

```
int MPI_Gather(
      void *sendbuf,          // chunk size buffer
      int   sendcount, MPI_Datatype sendtype,
      void *recvbuf,          // full size buffer
      int   recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)   for (i = 0; i < fullSize; i++)   buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)    buf[i]*=2;
MPI_Gather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    printf("%d: ", myRank);
    for (i = 0; i < fullSize; i++)
        printf("%d ", bufferR[i]);
}
```

Moving Data: Scatter & Gather

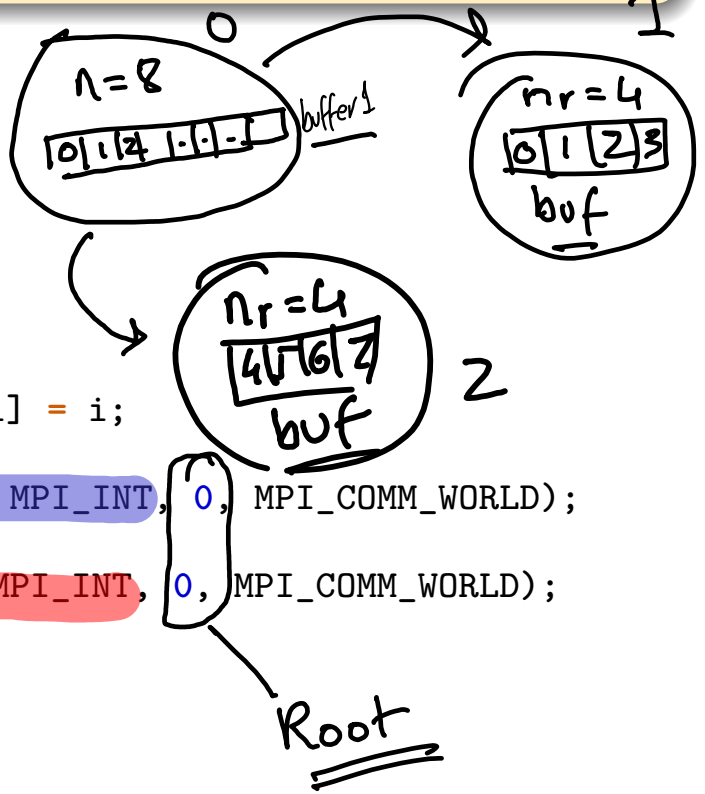
1:M M:1

p=2

n=8

```
int MPI_Gather(
      void *sendbuf,          // chunk size buffer
      int   sendcount, MPI_Datatype sendtype,
      void *recvbuf,          // full size buffer
      int   recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 8, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)    for (i = 0; i < fullSize; i++)    buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)     buf[i]*=2;
MPI_Gather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    printf("%d: ", myRank);
    for (i = 0; i < fullSize; i++)
        printf("%d ", bufferR[i]);
}
```



Moving Data: Scatter & Gather

```
int MPI_Gather(
      void *sendbuf,          // chunk size buffer
      int   sendcount, MPI_Datatype sendtype,
      void *recvbuf,          // full size buffer
      int   recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)   for (i = 0; i < fullSize; i++)   buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)    buf[i]*=2;
MPI_Gather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    printf("%d: ", myRank);
    for (i = 0; i < fullSize; i++)
        printf("%d ", bufferR[i]);
}
```

Moving Data: All Gather

Root → Any Root

```
int MPI_Allgather(
      void *sendbuf,          // chunk size buffer
      int   sendcount, MPI_Datatype sendtype,
      void *recvbuf,          // full size buffer
      int   recvcount, MPI_Datatype recvtype, MPI_Comm comm) // Note: No root
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size(MPI_COMM_WORLD, &totalProcesses);

int fullSize = 100, reducedSize = fullSize/totalProcesses;
int *buffer1  = malloc(sizeof(int)*fullSize);
int *buffer2  = malloc(sizeof(int)*fullSize);
int *buf      = malloc(sizeof(int)*reducedSize);

if (myRank == 0)   for (i = 0; i < fullSize; i++)   buffer1[i] = i;

MPI_Scatter(buffer1, reducedSize, MPI_INT, buf, reducedSize, MPI_INT, 0, MPI_COMM_WORLD);
for (i = 0; i < reducedSize; i++)    buf[i]*=2;
MPI_Allgather(buf, reducedSize, MPI_INT, buffer2, reducedSize, MPI_INT, MPI_COMM_WORLD);

if (myRank == 0) {                          // Now can be changed to any rank !!
    printf("%d: ", myRank);
    for (i = 0; i < fullSize; i++)
        printf("%d ", bufferR[i]);
}
```

Root = 0

Doesn't matter.

Barrier

Broadcast

Scatter

Gather

All gather

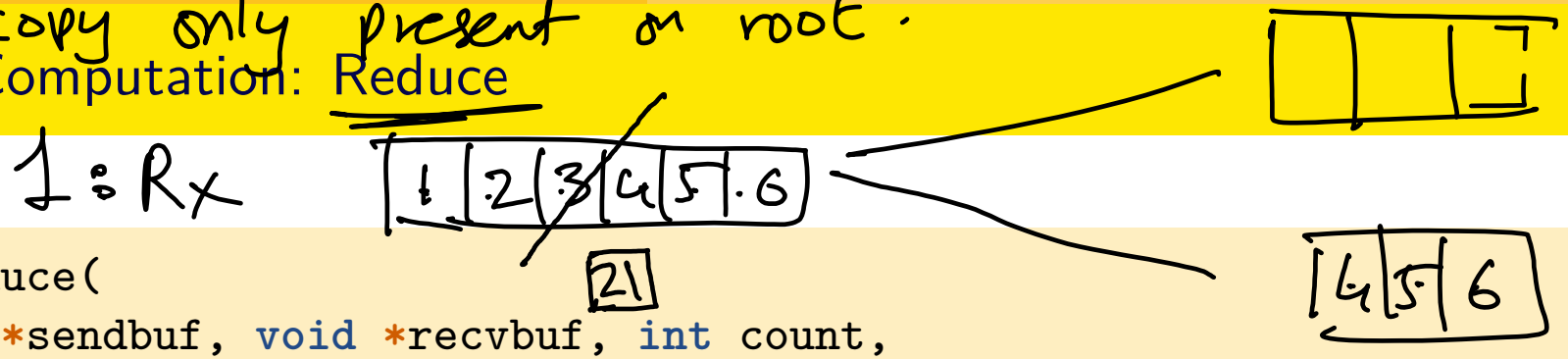
Reduce

} Communication API Calls

} Communication + Computation

Collective Computation: Reduce

Reduce copy only present on root.




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

Where, *op* is any of: (operation)

MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Summation	MPI_PROD	Product
MPI_LAND	Logical And	MPI_BAND	Bitwise And
MPI_LOR	Logical Or	MPI_BOR	Bitwise Or
MPI_LXOR	Logixal Xor	MPI_BXOR	Bitwise Xor

```
MPI_Init(&argc, &argv);

int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int buffer;
MPI_Reduce(&myRank, &buffer, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    printf("Final Sum: %d\n", buffer);
}

MPI_Finalize();
```

gather. Sum

Logical and
3 and 5
0011 and 0101

bitwise and
0011
0101
0 and 0
0 and 1
1 and 0
1 and 1

Collective Computation: All Reduce

Reduced copy present on all processor

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

Where, *op* is any of:

MPI_MAX	Maximum	MPI_MIN	Minimum
MPI_SUM	Summation	MPI_PROD	Product
MPI_LAND	Logical And	MPI_BAND	Bitwise And
MPI_LOR	Logical Or	MPI_BOR	Bitwise Or
MPI_LXOR	Logixal Xor	MPI_BXOR	Bitwise Xor

```
MPI_Init(&argc, &argv);

int myRank;
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int buffer;
MPI_Allreduce(&myRank, &buffer, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

if (myRank == 0) {                    // now can be any rank
    printf("Final Sum: %d\n", buffer);
}

MPI_Finalize();
```

Collective Computation: Reduction (on buffer)

```
int myRank, mySize, i;
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &mySize);

int *buffer = malloc(sizeof(int)*2);
buffer[0] = myRank;
buffer[1] = 1;

int *result = malloc(sizeof(int)*2);

MPI_Reduce(buffer, result, 2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

if (myRank == 0) {
    printf("Final Sum: %d\n", result[0]);
    printf("Final Sum: %d\n", result[1]);
}
```

Multiple Groups and Communicators

- Useful if collective tasks are performed on a subset of processes
- **New communicators** created by splitting **old communicators**
- Grouping made on bases of **color** and **key**
- The old communicator will still be present

```
int MPI_Comm_split(
 MPI_Comm comm,      // The communicator to be split
 int color,          // to which color each process will belong?
 int key,            // rank order in new group
 MPI_Comm *newcomm) // the name of the new communicator
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

int myColor = myRank / 4;

MPI_Comm aRow;
MPI_Comm_split(MPI_COMM_WORLD, myColor, myRank, &aRow);

int myRankInRow;
MPI_Comm_rank(aRow, &myRankInRow);
```

Block Transfers

```
int MPI_Type_create_subarray(
    int              ndims,
    int              array_of_sizes[],
    int                    array_of_subsizes[],
    int              array_of_starts[],
    int              order,  // MPI_Order_C
    MPI_Datatype oldtype,
    MPI_Datatype *newtype);

int MPI_Type_commit(MPI_Datatype *datatype)
```