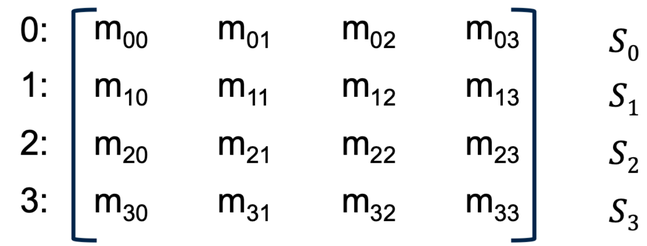
Many_to_Many With Bounded Traffic

- Generalized version of all-to-all

- Processors can have variable amount of data to send to all other processors (may be 0, in some cases)

  - m_{ij} : message from P_i to P_j
  - $|m_{ij}|$ = size of the message
    - $|m_{ij}| = 0$: message doesn't exist
  - $\max_i \sum_j |m_{ij}| \leq S$ — Total message size to send per processor
  - $\max_j \sum_i |m_{ij}| \leq R$ — Total message size to receive per processor



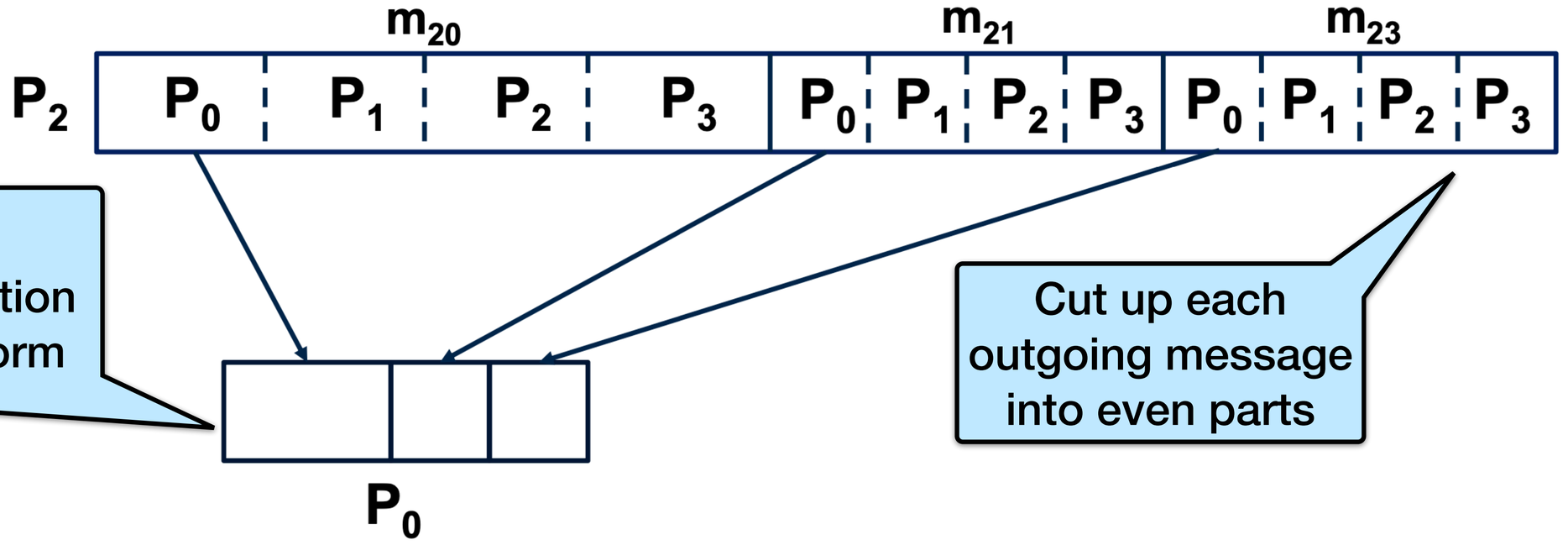
Many_to_Many

- Initial setup: Variable message size destined for each processor
- Example with p = 4

Many_to_Many

Converting to fixed-size all-to-alls

- **Stage 1**: example with $p = 4$



- All_to_All with max message size $\leq \frac{S}{p}$

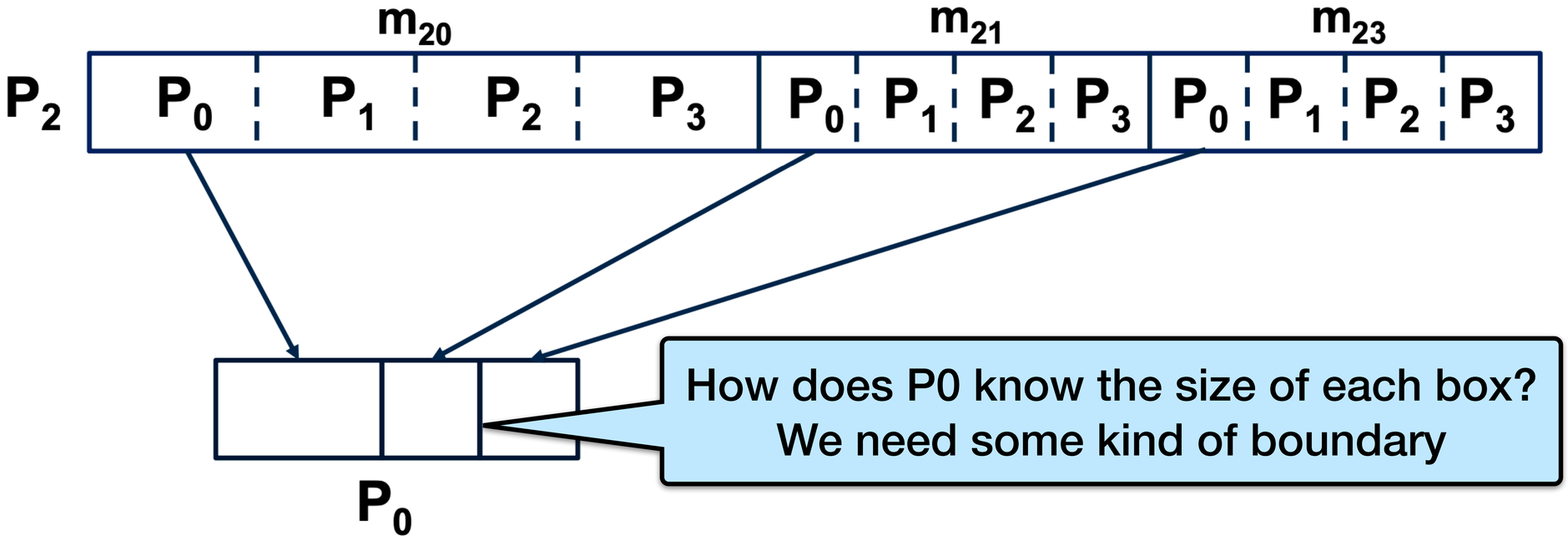
Many_to_Many

- **Stage 2**: Route the message fragments to actual destinations and assemble
  - All_to_All with max message size $\leq \frac{R}{p}$
- Runtime for All_to_All: $\Theta(\tau \cdot p + \mu \cdot m \cdot p)$
- Stage 1: $\Theta(\tau \cdot p + \mu \cdot \frac{S}{p} \cdot p)$
- Stage 2: $\Theta(\tau \cdot p + \mu \cdot \frac{R}{p} \cdot p)$
- Total Runtime: $\Theta(\tau \cdot p + \mu \cdot (R + S))$

From Srinivas Aluru

Many_to_Many

- **Stage 1**: example with $p = 4$



- All_to_All with max message size $\leq \frac{S}{p}$

Many_to_Many

- Stage 1: $\Theta\left(\tau \cdot p + \mu \cdot \left(\frac{S}{p} + p\right) \cdot p\right)$
- Stage 2: $\Theta\left(\tau \cdot p + \mu \cdot \left(\frac{R}{p} + p\right) \cdot p\right)$

Space required to mark the boundary is constant per processor

- Total: $\Theta(\tau \cdot p + \mu \cdot (R + S + p^2))$
- Total: $\Theta\left(\tau \cdot p + \mu \cdot (R + S)\right)$ provided $p^2 = O(S + R)$

Runtime Summary

- Broadcast
- Reduce
- AllReduce
- Scan

$\Theta(\tau \log p + \mu\, m \log p)$

- Gather
- AllGather
- Scatter

$\Theta(\tau \log p + \mu\, m\, p)$

- All_to_All
  - Arbitrary Permutations: $O(\tau \cdot p + \mu \cdot m \cdot p)$
  - Hypercubic Permutations: $O(\tau \cdot \log p + \mu \cdot m \cdot p \cdot \log p)$
- Many_to_Many: $\Theta\big(\tau \cdot p + \mu \cdot (R + S)\big)$ provided $p^2 = O(S + R)$

From Srinivas Aluru

CSE 6220/CX 4220
Introduction to HPC

Lecture 10: MPI Collective Communication

Helen Xu
hxu615@gatech.edu

Slides from UC Berkeley 267, Srinivas Aluru

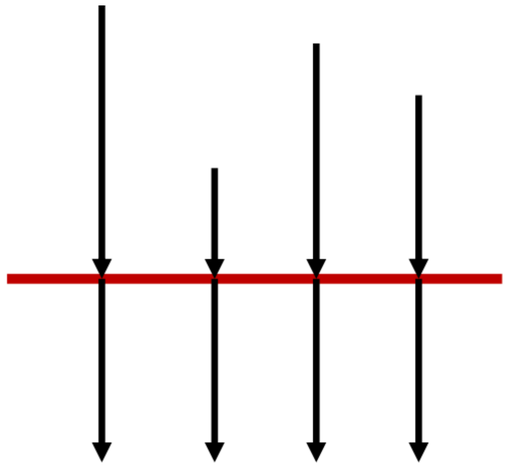
Recap: Synchronization

- Many parallel algorithms require that **no process proceeds before all the processes** have reached the same state at certain points of a program.

- Explicit synchronization :

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

- If one rank calls barrier, all of them must call it.



Recap: The Simplest MPI Send Command

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)
```

Returns only when communication is finished

This **blocking send function** returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

Recap: The Simplest MPI Receive Command

```
int MPI_Recv(void *buf, int count
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status);
```

- This **blocking receive function** waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or `MPI_ANY_SOURCE`), message tag (or `MPI_ANY_TAG`).
- Receiving fewer `datatype` elements than count is ok, but receiving more is an error.

Example Send / Recv

```
#include <mpi.h>
#include <iostream>

int main(int argc , char* argv []) {
    int rank;
    MPI_Init (&argc , &argv);
    MPI_Comm_rank (MPI_COMM_WORLD , &rank );
    const int N = 32;
    int tab[N];
    if (rank == 0) {
        tab[N - 1] = 13;
        MPI_Send(tab, N, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(tab, N, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
        std::cout << tab[N - 1] << std::endl;
    }
    return MPI_Finalize();
}
```

Send N integers from rank 0 to rank 1.
Blocks till everything is sent.

Receive N integers.
Blocks till everything is received.

```
Output: 13
```

From Srinivas Aluru

MPI Tags

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype,
             int dest, int tag,
             MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count
             MPI_Datatype datatype,
             int source, int tag,
             MPI_Comm comm,
             MPI_Status *status);
```

Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.

Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive.

https://mpitutorial.com/tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/

Message Matching

- Each send operation **must be matched** by a corresponding receive operation
- Messages are matched via their **envelope:**
  - communicator
  - source rank
  - tag
  - **NOT** by type or size
- Thus: for a receive to succeed, a message needs to match the communicator, tag, and rank

Blocking:

- An `MPI_Recv` operation **blocks** until a matching message is received
- An `MPI_Send` operation **may block** until the message has been received by the destination process

Example: Message Matching

```
#include <mpi.h>

int main(int argc , char* argv []) {
    int rank;
    MPI_Init(&argc , &argv);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    int x;
    if (rank == 0) {
        x = 13;
        MPI_Send(&x, 1, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(&x, 1, MPI_INT,
                 0, 222, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

What's wrong?

Tags are not matching
111 != 222

MPI program will deadlock!

From Srinivas Aluru

Example: Message Matching

```
#include <mpi.h>
int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int x[16];
    float y[16];
    if (rank == 0) {
        x[13] = 13;
        y[13] = 0.13;
        MPI_Send(x, 16, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
        MPI_Send(y, 16, MPI_FLOAT,
                 1, 222, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(y, 16, MPI_FLOAT,
                 0, 222, MPI_COMM_WORLD, &stat);
        MPI_Recv(x, 16, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

What's wrong?

How to fix it?

Blocks until message with tag 111 is received.

Blocks until message with tag 222 is sent.

MPI program will deadlock!

From Srinivas Aluru

Example: Message Matching

```c
#include <mpi.h>
int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int x[16];
    float y[16];
    if (rank == 0) {
        x[13] = 13;
        y[13] = 0.13;
        MPI_Send(x, 16, MPI_INT,
                 1, 111, MPI_COMM_WORLD);
        MPI_Send(y, 16, MPI_FLOAT,
                 1, 222, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Status stat;
        MPI_Recv(y, 16, MPI_FLOAT,
                 0, 222, MPI_COMM_WORLD, &stat);
        MPI_Recv(x, 16, MPI_INT,
                 0, 111, MPI_COMM_WORLD, &stat);
    }
    return MPI_Finalize();
}
```

What's wrong?

How to fix it?

Reorder receives to x and y

Blocks until message with tag 111 is received.

Blocks until message with tag 222 is sent.

MPI program will deadlock!

From Srinivas Aluru

Any

- Useful trick:
  - We can use any combination of **`tag = MPI_ANY_TAG`** and/or **`source = MPI_ANY_SOURCE`** in **`MPI_Recv`** to receive any message from any processor
  - We can use the **`MPI_Status`** structure to check envelope of the received message

- Example:

```
int x;
MPI_Status stat;
MPI_Recv(&x, 1, MPI_INT,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
int source = stat.MPI_SOURCE;
int tag = stat.MPI_TAG;
```

Example: any tag, any src

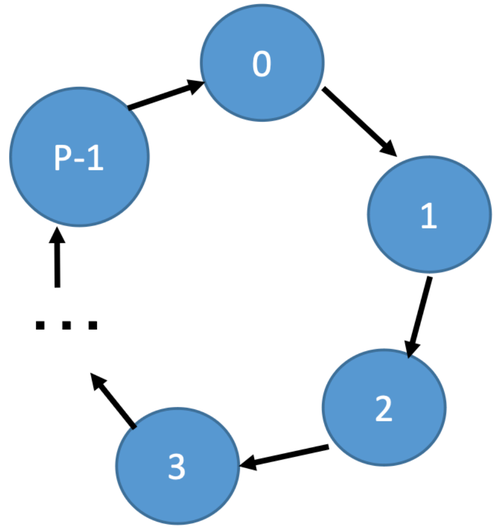
```
#include <mpi.h>
#include <iostream>

int main(int argc , char* argv []) {
    int rank , size;
    MPI_Init (&argc , &argv );
    MPI_Comm_rank (MPI_COMM_WORLD , &rank );
    MPI_Comm_size (MPI_COMM_WORLD , &size );
    int x;
    if (rank == 0) {
        MPI_Status stat;
        for (int i = 1; i < size; ++i) {
            MPI_Recv(&x, 1, MPI_INT,
                     MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                     &stat);
            std::cout << "From: " << stat.MPI_SOURCE << " "
            << "Tag: " << stat.MPI_TAG << std::endl;
        }
    } else {
        x = rank;
        MPI_Send(&x, 1, MPI_INT,
                 0, size - rank, MPI_COMM_WORLD);
    }
    return MPI_Finalize();
}
```

Receives any one of the messages. No order guaranteed.

From Srinivas Aluru

Cyclic Dependencies

- Let's say we want to send messages according to right shift permutation:
  - Processor `i` sends a message to `(i+1) mod p`



Example:

```
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
```

What's wrong?

May block till message is received, MPI_Recv never called

Reverse?

```
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
```

Same problem.

Non-blocking Communication

- Enter: Non-blocking Communication
  - **`MPI_Isend`** and **`MPI_Irecv`** will initiate transfers but **not block** until they succeed
  - Enables:
    - Dead-lock avoidance
    - Hiding latency by overlapping computation and communication
  - Same parameters as `MPI_Send`, `MPI_Recv` but additional output parameter:

```
int MPI_Isend(void* buf, int count, MPI_Datatype type,
              int dest, int tag, MPI_Comm comm,
              MPI_Request* req);

int MPI_Irecv(void* buf, int count, MPI_Datatype type,
              int source, int tag, MPI_Comm comm,
              MPI_Request* req);
```

From Srinivas Aluru

`MPI_Wait` and `MPI_Test`

```
int MPI_Wait(MPI_Request* request,
             MPI_Status* status);
```

`MPI_Wait` **waits** for a non-blocking operation to complete and will block until the underlying operation is done.

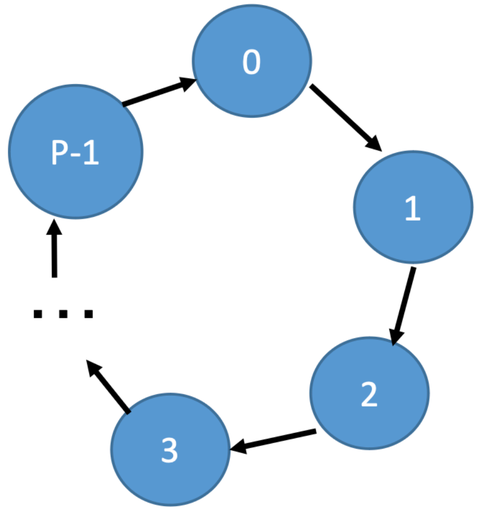
```
int MPI_Test(MPI_Request* request,
             int* flag,
             MPI_Status* status);
```

`MPI_Test` checks if a non-blocking operation is complete at a given time.
It will not wait for the underlying non-blocking operation to complete.

https://rookiehpc.org/mpi/docs/mpi_wait/index.html
https://rookiehpc.org/mpi/docs/mpi_test/index.html

Cyclic Dependencies

- Let's say we want to send messages according to right shift permutation:
  - Processor `i` sends a message to `(i+1) mod p`



Example:

```
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Recv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &stat);
```

What's wrong?

May block till message is received, MPI_Recv never called

Using Non-blocking receive:

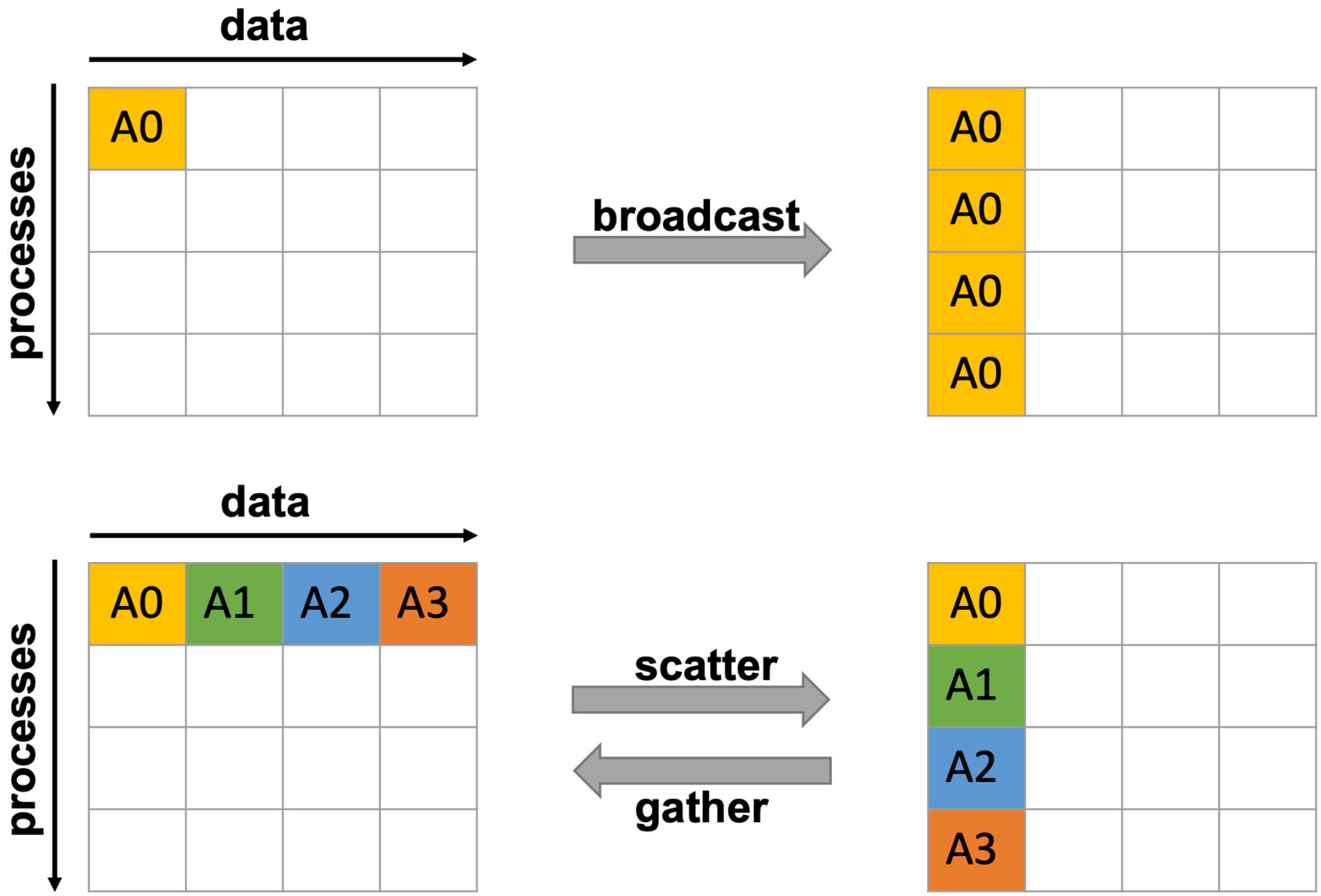
```
MPI_Request req;
MPI_Irecv(&y, 1, MPI_INT,
         MPI_ANY_SOURCE, 13, MPI_COMM_WORLD, &req);
MPI_Send(&x, 1, MPI_INT,
         (rank+1) % size, 13, MPI_COMM_WORLD);
MPI_Wait(&req, &stat);
```

Correct!

From Srinivas Aluru

MPI Collectives

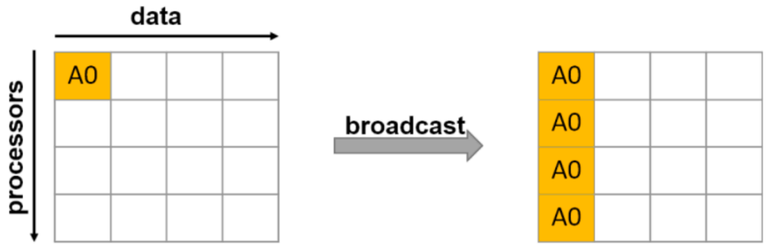
Broadcast, Scatter, Gather



From Srinivas Aluru

Broadcast

Broadcast a message from one process(=root) to all other processes




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

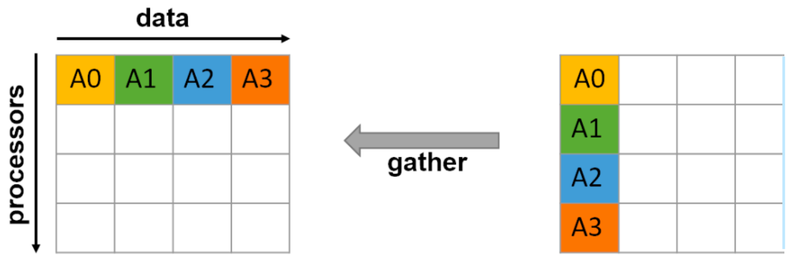
Example:

```
#include <mpi.h>
#include <iostream>
int main(int argc , char* argv []) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    double x = 0.0;
    if (rank == 0) x = -13.13;
    MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    std::cout << x << std::endl;
    return MPI_Finalize ();
}
```

Prints -13.13 on every process.

Gather

Gather data from all processes on one process(=root)



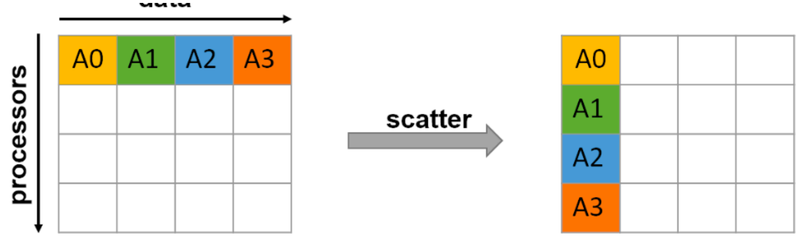

```
int MPI_Gather(void* sbuf, int scount, MPI_Datatype stype,
               void* rbuf, int rcount, MPI_Datatype rtype,
               int root, MPI_Comm comm);
```

```
#include <mpi.h>
#include <iostream >
#include <vector >
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::vector<int> buf;
    if (rank == 0) buf.resize(size, -1);
    MPI_Gather(&rank  , 1, MPI_INT,
               &buf[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank == 0) std::cout << buf.back() << std::endl;
    return MPI_Finalize();
}
```

At this point, what is the content of the vector buf?

Scatter

Scatter data from one process to all processes




```
int MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype,
                void* rbuf, int rcount, MPI_Datatype rtype,
                int root, MPI_Comm comm);
```

```
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    float x = 13;
    std::vector<float> buf;
    if (rank == 0) {
        buf.resize(size, -1);
        for (int i = 0; i < size; ++i) buf[i] = 1.0 / (1 + i);
    }
    MPI_Scatter(&buf[0], 1, MPI_FLOAT,
                &x, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    std::cout << rank << " " << x << std::endl;
    return MPI_Finalize();
}
```

What is the content of the variable x on each process?

From Srinivas Aluru

Scatterv

```
int MPI_Scatterv(void* sbuf, int *scounts, int *displs,
                 MPI_Datatype stype,
                 void* rbuf, int rcount, MPI_Datatype rtype,
                 int root, MPI_Comm comm);
```

Array of send counts

Array of start indices

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>


#define N 4


int main(int argc, char *argv[])
{
  int rank, size;    // this process' rank, and the number of processes
  int *sendcounts;   // number of elements to send to each process
  int *displs;       // displacements where each segment begins
  int sum=0;         // Sum of counts. Used to calculate displacements
  char rec_buf[100]; // buffer where the received data should be stored


  char data[N][N]={   // the data to be distributed
     {'a','b','c','d'}, {'e','f','g','h'},
     {'i','j','k','l'}, {'m','n','o','p'}
  };
```

From Srinivas Aluru

Example modified from https://gist.github.com/ehamberg/1263868.

Scatterv

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

sendcounts = (int*) malloc(sizeof(int)*size);
displs = (int*) malloc(sizeof(int)*size);

// calculate send counts and displacements
int rem = (N*N) % size;// elements remaining after division among processes
for (int i=0; i<size; i++) {
   sendcounts[i] = (N*N) / size;
   if (rem-- > 0)
      sendcounts[i]++;
   displs[i] = sum;
   sum += sendcounts[i];
}

// print calculated send counts and displacements for each process
if (0 == rank) {
   for (int i=0; i<size; i++)
      printf("sendcounts[%d] = %d\tdispls[%d] = %d\n", i, sendcounts[i],
           i, displs[i]);
}

// divide the data among processes as described by sendcounts and displs
MPI_Scatterv(&data, sendcounts, displs, MPI_CHAR,
         &rec_buf, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Treat it as a 1D vector

From Srinivas Aluru

Scatterv

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

sendcounts = (int*) malloc(sizeof(int)*size);
displs = (int*) malloc(sizeof(int)*size);

// calculate send counts and displacements
int rem = (N*N) % size;// elements remaining after division among processes
for (int i=0; i<size; i++) {
   sendcounts[i] = (N*N) / size;
   if (rem-- > 0)
      sendcounts[i]++;
   displs[i] = sum;
   sum += sendcounts[i];
}

// print calculated send counts and displacements for each process
if (0 == rank) {
   for (int i=0; i<size; i++)
      printf("sendcounts[%d] = %d\tdispls[%d] = %d\n", i, sendcounts[i],
           i, displs[i]);
}

// divide the data among processes as described by sendcounts and displs
MPI_Scatterv(&data, sendcounts, displs, MPI_CHAR,
         &rec_buf, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Treat it as a 1D vector

Find displacements, some may need extra if n % p =/= 0

From Srinivas Aluru

Scatterv

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

sendcounts = (int*) malloc(sizeof(int)*size);
displs = (int*) malloc(sizeof(int)*size);

// calculate send counts and displacements
int rem = (N*N) % size;// elements remaining after division among processes
for (int i=0; i<size; i++) {
   sendcounts[i] = (N*N) / size;
   if (rem-- > 0)
      sendcounts[i]++;
   displs[i] = sum;
   sum += sendcounts[i];
}

// print calculated send counts and displacements for each process
if (0 == rank) {
   for (int i=0; i<size; i++)
      printf("sendcounts[%d] = %d\tdispls[%d] = %d\n", i, sendcounts[i],
           i, displs[i]);
}

// divide the data among processes as described by sendcounts and displs
MPI_Scatterv(&data, sendcounts, displs, MPI_CHAR,
         &rec_buf, 100, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Treat it as a 1D vector

Find displacements, some may need extra if n % p =/= 0

Prefix sum of send counts

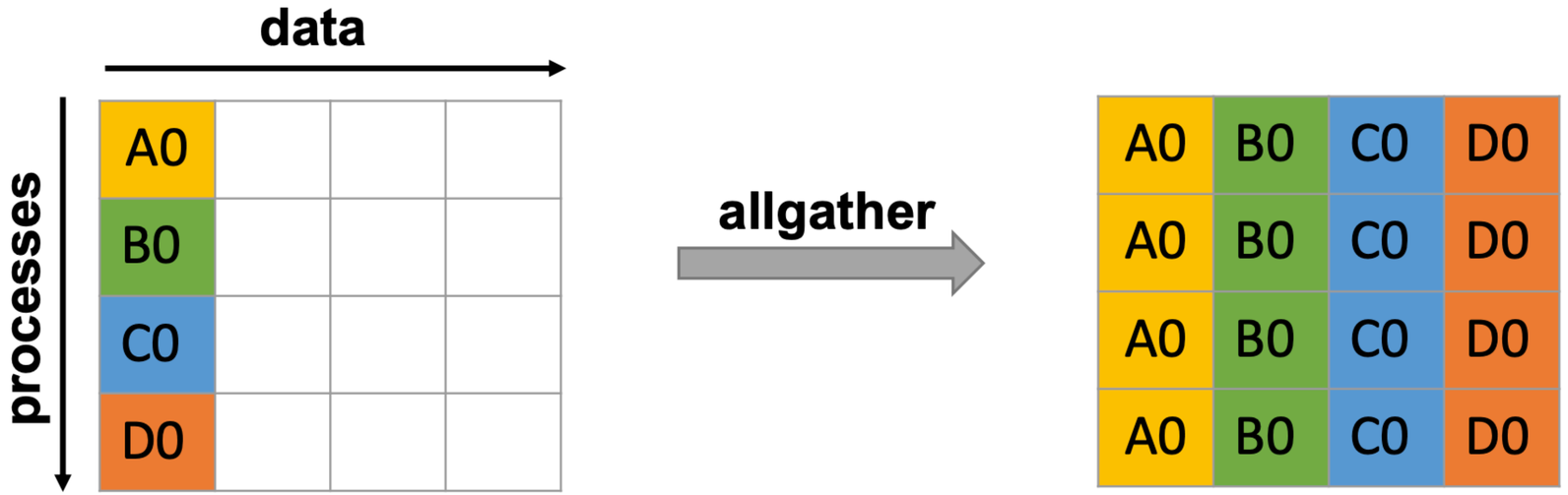
From Srinivas Aluru

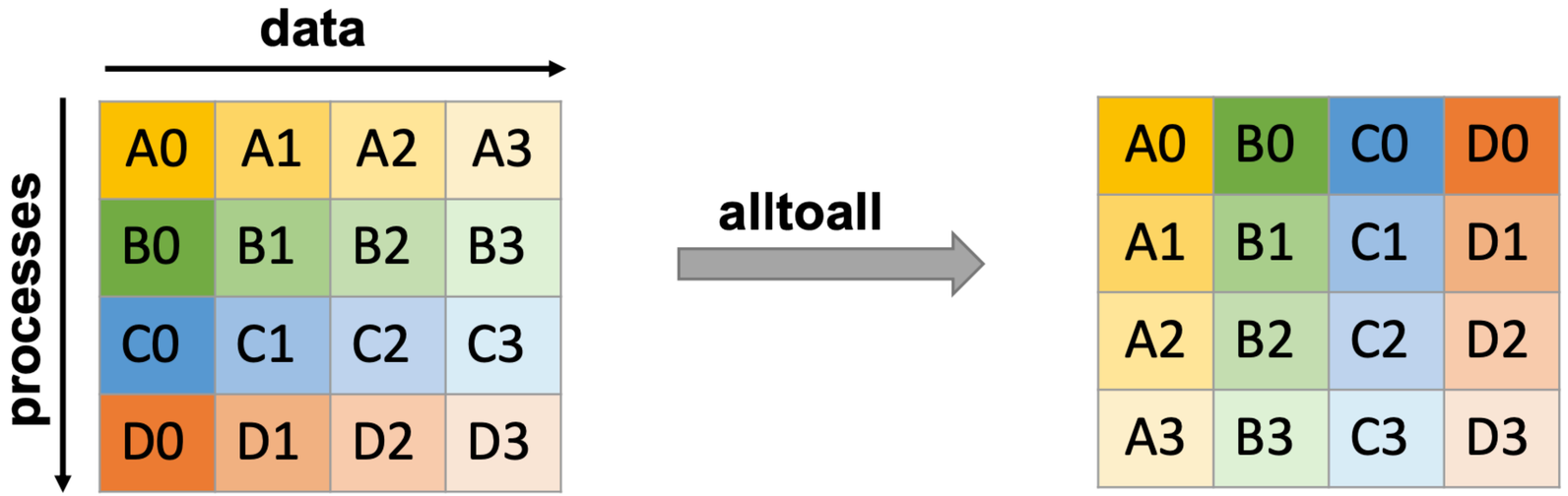
Scatterv

```
  // print what each process received
  printf("%d: ", rank);
  for (int i=0; I < sendcounts[rank]; i++) {
     printf("%c\t", rec_buf[i]);
  }
  printf("\n");

  MPI_Finalize();

  free(sendcounts);
  free(displs);

  return 0;
}
```

```
% mpirun -np 3 scatterv
sendcounts[0] = 6 displs[0] = 0
sendcounts[1] = 5 displs[1] = 6
sendcounts[2] = 5 displs[2] = 11
0: a     b     c     d     e     f
1: g     h     i     j     k
2: l     m     n     o     p
```

Allgather, Alltoall

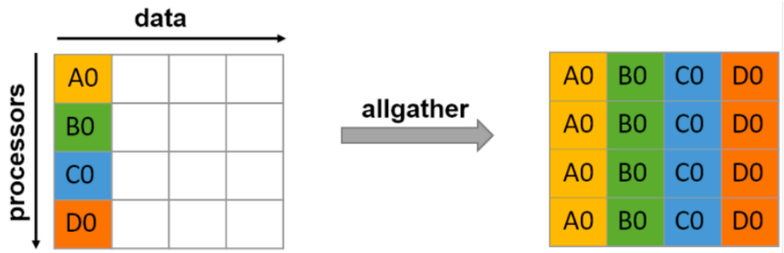




From Srinivas Aluru

Allgather

Allgather: corresponds to each process broadcasting its data




```
int MPI_Allgather(void* sbuf, int scount, MPI_Datatype stype,
                  void* rbuf, int rcount, MPI_Datatype rtype,
                  MPI_Comm comm);
```

Example:

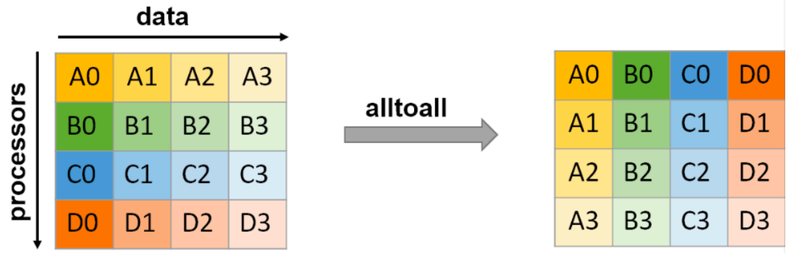
```
int main(int argc, char* argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::vector<int> buf;
    buf.resize(size, -1);
    MPI_Allgather(&rank  , 1, MPI_INT,
                  &buf[0], 1, MPI_INT, MPI_COMM_WORLD);
    return MPI_Finalize();
}
```

At this point, what is the content of the vector buf?

From Srinivas Aluru

Alltoall

Alltoall: every process has one message for every other process




```
int MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype,
                 void* rbuf, int rcount, MPI_Datatype rtype,
                 MPI_Comm comm);
```

```
int main(int argc, char* argv []) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    std::vector<float> send(size);
    std::vector<float> recv(size, -1);
    for (int i = 0; i < size; ++i) send[i] = rank;
    MPI_Alltoall(&send[0], 1, MPI_FLOAT,
                 &recv[0], 1, MPI_FLOAT, MPI_COMM_WORLD);
    return MPI_Finalize();
}
```

At this point, what is the content of the vector recv?

From Srinivas Aluru

Collective Communication

Summary:

- All processes within a communicator must participate
- All collective operations are **blocking**
- Except of `MPI_Barrier`, no synchronization can be assumed
- All collective operations are guaranteed to not interfere with point-to-point messages
- Collective operations can be implemented using only point-to-point calls

Operations:

- barrier, broadcast, scatter, gather, allgather, alltoall

Reductions

- Let's assume we want to compute a sum over n elements:

```
int sum = 0;
for (int i = 0; i < n; ++i) sum = sum + A[i];
```

- Can we do this in parallel? -> Parallel Reductions

```
int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm);
```

Example:

```
// assume each processor has one element:
int a = A[rank];
int sum = 0;
MPI_Reduce(&a, &sum, 1, MPI_INT,
           MPI_SUM , 0, MPI_COMM_WORLD);
std::cout << sum << std::endl;
```

Reductions

- Let's assume we want to compute a sum over n elements:

```
int sum = 0;
for (int i = 0; i < n; ++i) sum = sum + A[i];
```

- Can we do this in parallel? -> Parallel Reductions

```
int MPI_Reduce(void* sbuf, void* rbuf, int count, MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm);
```

Example:

```
// assume each processor has one element:
int a = A[rank];
int sum = 0;
MPI_Reduce(&a, &sum, 1, MPI_INT,
           MPI_SUM , 0, MPI_COMM_WORLD);
std::cout << sum << std::endl;
```

Reductions

- What about other binary operations?

```
int sum = A[0];
for (int i = 1; i < n; ++i) sum = op(sum, A[i]);
```

- Can we do this in parallel?
  - only if **op(x,y)** is associative: `op(x, op(y, z)) == op(op(x, y), z)`
- `MPI_Reduce` takes the combination operator as `MPI_Op`:

MPI_Op	Meaning	C operator
MPI_MIN	minimum	`x < y ? x : y`
MPI_MAX	maximum	`x < y ? y : x`
MPI_SUM	sum	`x + y`
MPI_PROD	product	`x * y`
MPI_LAND, MPI_LOR	Logical and/or	`x && y, x || y`
MPI_BAND, MPI_BOR	Binary and/or	`x & y, x | y`

Reductions

- MPI_Allreduce = MPI_Reduce + MPI_Bcast

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

Example:

```
while (true) {
        // compute ...
        // check termination
        int terminate = ...;
        int result;
        MPI_Allreduce(&terminate, &result, 1, MPI_INT,
                      MPI_LAND, MPI_COMM_WORLD);
        if (result == 1) break;
    }
```

From Srinivas Aluru

Reductions

Reduction Example:

```
float A[p];
float master_sum = 0.0f;
if (rank == 0)
    for (int i = 0; i < p; ++i) {
        A[i] = 1.0f*rand()/RAND_MAX;
        master_sum += A[i];
    }

float value;
MPI_Scatter(A, 1, MPI_FLOAT, &value, 1,
            MPI_FLOAT, 0, MPI_COMM_WORLD);

float sum;
MPI_Reduce(&value, &sum, 1, MPI_FLOAT,
           MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0 && master_sum != sum) {
    printf("ERROR!\n");
}
```

What happens?

Generates array of random numbers and sums them up

Sends one random number to each process

Parallel sum over random numbers

Why might this print “Error!”?

Answer: floating point math is not associative!

From Srinivas Aluru

Parallel Prefix

- Let's consider a prefix sum operation:

```
// prefix sum:
int x[n], y[n];
y[0] = x[0];
for (int i = 1; i < n; ++i)
    y[i] = op(y[i-1], x[i]);
```

- With MPI:

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

Example:

```
// assume each processor has one element:

int prefix = 0;
MPI_Scan(&rank, &prefix, 1, MPI_INT,
          MPI_SUM , MPI_COMM_WORLD);
std::cout << prefix << std::endl;
```

From Srinivas Aluru

Communicator Constructors

MPI provides functions to create a new communicator from an existing one:

Duplicate the same communicator

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm);

int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm* newcomm);
```

Communicators can be released when no longer needed:

```
int MPI_Comm_free(MPI_Comm* comm);
```

Communicator Constructors

MPI provides functions to create a new communicator from an existing one:

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm);
```

Duplicate the same communicator

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,
                   MPI_Comm* newcomm);
```

Each process gets a new communicator depending on color

All processes with the same color will belong to the same new communicator, ordered by key

Communicators can be released when no longer needed:

```
int MPI_Comm_free(MPI_Comm* comm);
```

Communicator Split Example

```
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int color = rank % 2;
    MPI_Comm comm;

    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
    int S = 0;
    MPI_Reduce(&rank, &S, 1, MPI_INT, MPI_SUM, 0, comm);
    MPI_Comm_free(&comm);

    std::cout << rank << " " << S << std::endl;
    return MPI_Finalize();
}
```

New communicator

Reduce on even and odd

Communicator Split Example

MPI_WORLD = {P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7}

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
rank	0	1	2	3	4	5	6	7
color	0	1	0	1	0	1	0	1
key	0	1	2	3	4	5	6	7

comm = {P_0, P_2, P_4, P_6} or { P_1, P_3, P_5, P_7}

	P_0	P_2	P_4	P_6	P_1	P_3	P_5	P_7
rank	0	1	2	3	0	1	2	3

Communicator Split Example

```
#include <mpi.h>
#include <iostream>

int main(int argc, char* argv[]) {
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int color = rank % 2;
    MPI_Comm comm;

    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
    int S = 0;
    MPI_Reduce(&rank, &S, 1, MPI_INT, MPI_SUM, 0, comm);
    MPI_Comm_free(&comm);

    std::cout << rank << " " << S << std::endl;
    return MPI_Finalize();
}
```

```
5 0
2 0
7 0
3 0
6 0
4 0
0 12
1 16
```

What is the output for p

Other Default Communicators

- Communicator constructors are collective operations
- MPI provides two additional predefined communicators `MPI_COMM_NULL` and `MPI_COMM_SELF`

If you want to avoid some ranks participating in collective operations

MPI References

- MPI Standard: https://www.mpi-forum.org/
- Other information: https://www.mcs.anl.gov/research/projects/mpi/index.htm
- LLNL tutorial: https://hpc-tutorials.llnl.gov/mpi/
- https://mpitutorial.com/