Introduction to MPI

Aravind Sukumaran Rajam

MPI send/receive

Function	Description
MPI_Barrier()	Blocks until all processes in the communicator have reached this routine
MPI_ISend ()	Send data asynchronously
MPI_IRecv ()	Receive data asynchronously
MPI_Wait()	Wait for an asynchronous operation to complete
MPI_Test()	Check whether an asynchronous operation is complete
MPI_Reduce	We already saw this
MPI_Scatter/MPI_Gather	Scatter and gather data
MPI_Scan	Prefix sum
MPI_Alltoall	Send and receive data

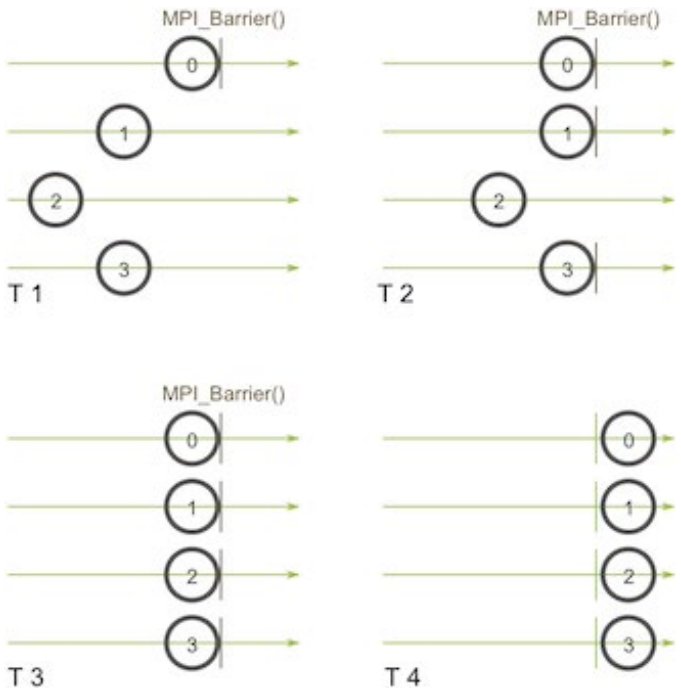
Remember our send/recv?

```c
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);


    char buffer[200];

    if (world_rank == 0)
    {
        char* rank0_msg = "Hello from rank 0";
        MPI_Send(rank0_msg, strlen(rank0_msg), MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
    else if (world_rank == 1)
    {
        MPI_Status status;
        MPI_Recv(buffer, 200, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
                 &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        printf("Process %d received '%s' with %d chars from process %d\n", world_rank,
                        buffer, count, status.MPI_SOURCE);
    }

    MPI_Finalize();
    return 0;
}
```

- One thing that we did not learn is whether we can write to the “buffer” after MPI_Send returns
- What is your opinion?

MPI Barrier

- Synchronizes all processes belonging to the given communicator
- Blocks the caller until all processes in the communicator have called it



Source:mpitutorial.com

MPI Barrier example

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    printf("Hi!!! I am rank %d. I am going to enter a barrier\n", my_rank);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Hello again from %d, glad that everyone got out of the  barrier\n", my_rank);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

Going back to send/receive

- Imagine that you have two processes
  - Rank 0: This is the main process which process the entire job; however, it in middle of its execution, it has to send data to Rank 1 for writing to database
  - Rank 1: This is a very slow computer and it is overloaded. It may take a lot of time to post a recv
  - This means, rank 0's send will wait for rank 1 to recv. However, rank 0 has more work to do, but it is not able to proceed till the send is complete.

Going back to send/receive

- The issue here is that MPI_send is blocking
- We need a non blocking version…

MPI_Isend, MPI_Irecv

int **MPI_Isend**(void *buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Communicator comm, MPI_Request *request);

int **MPI_Irecv**(void *buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Communicator comm, MPI_Request *request);

The I stands for immediate

MPI_Isend, MPI_Irecv

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 2)
    {
        printf("This application is meant to be run with 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    switch(my_rank)
    {
        case 0:
        {
            /// Process a lot of things


            /// Now send data to the db server
            int buffer_sent = 12345;
            MPI_Request request;
            printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
            MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Isend completes
        }
        case 1:
        {
            int received;
            MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("MPI process %d received value: %d.\n", my_rank, received);
            break;
        }
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- This code has an issue. Can anyone spot it?

MPI_Isend, MPI_Irecv

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 2)
    {
        printf("This application is meant to be run with 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    switch(my_rank)
    {
        case 0:
        {
            /// Process a lot of things


            /// Now send data to the db server
            int buffer_sent = 12345;
            MPI_Request request;
            printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
            MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Isend completes
        }
        case 1:
        {
            int received;
            MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("MPI process %d received value: %d.\n", my_rank, received);
            break;
        }
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- At some point we need to make sure that the receiver received the data

MPI_Wait

- int MPI_Wait(MPI_Request *request, MPI_Status *status)
- Waits for an MPI request to complete

MPI_Isend, MPI_Irecv

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 2)
    {
        printf("This application is meant to be run with 2 processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    switch(my_rank)
    {
        case 0:
        {
            /// Process a lot of things


            /// Now send data to the db server
            int buffer_sent = 12345;
            MPI_Request request;
            printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
            MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

            // Do other things while the MPI_Isend completes

            // Let's wait for the MPI_Isend to complete before progressing further.
            MPI_Wait(&request, MPI_STATUS_IGNORE);
            break;
        }
        case 1:
        {
            int received;
            MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("MPI process %d received value: %d.\n", my_rank, received);
            break;
        }
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- This is the correct code

MPI_Test

- Imagine a situation where we have posted an Isend and the sender wants to find whether the send has been complete, but it doesn't want to get blocked
- A use case is when Rank0 want to know if the send succeeded, if yes it wants to give a new job to Rank1 otherwise, Rank 0 will process it
- Using MPI_Wait will block the sender … not ideal
- MPI_Test to rescue

MPI_Test

- int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
- Tests for the completion of a request

MPI_Test

```c
switch(my_rank)
{
    case 0:
    {
        /// Process a lot of things

        /// Now send data to the db server
        int buffer_sent = 12345;
        MPI_Request request;
        printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
        MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

        /// Do other things


        /// Lets check if the send succeeded without blocking
        int success;
        MPI_Test(&request, &success, MPI_STATUS_IGNORE);
        if(success)
            printf("Isend completed at this point\n");
        else
            printf("Isend did not completed at this point\n");

        /// Do other things

        /// Let's wait for the MPI_Isend to complete before progressing further.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        break;
    }
    case 1:
    {
        int received;
        MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("MPI process %d received value: %d.\n", my_rank, received);
        break;
    }
}

MPI_Finalize();

return EXIT_SUCCESS;
```

Reduction

- Remember our OpenMP reduction?
- Imagine that each process owns an integer. We add all these integers. How do we do that?

Reduction

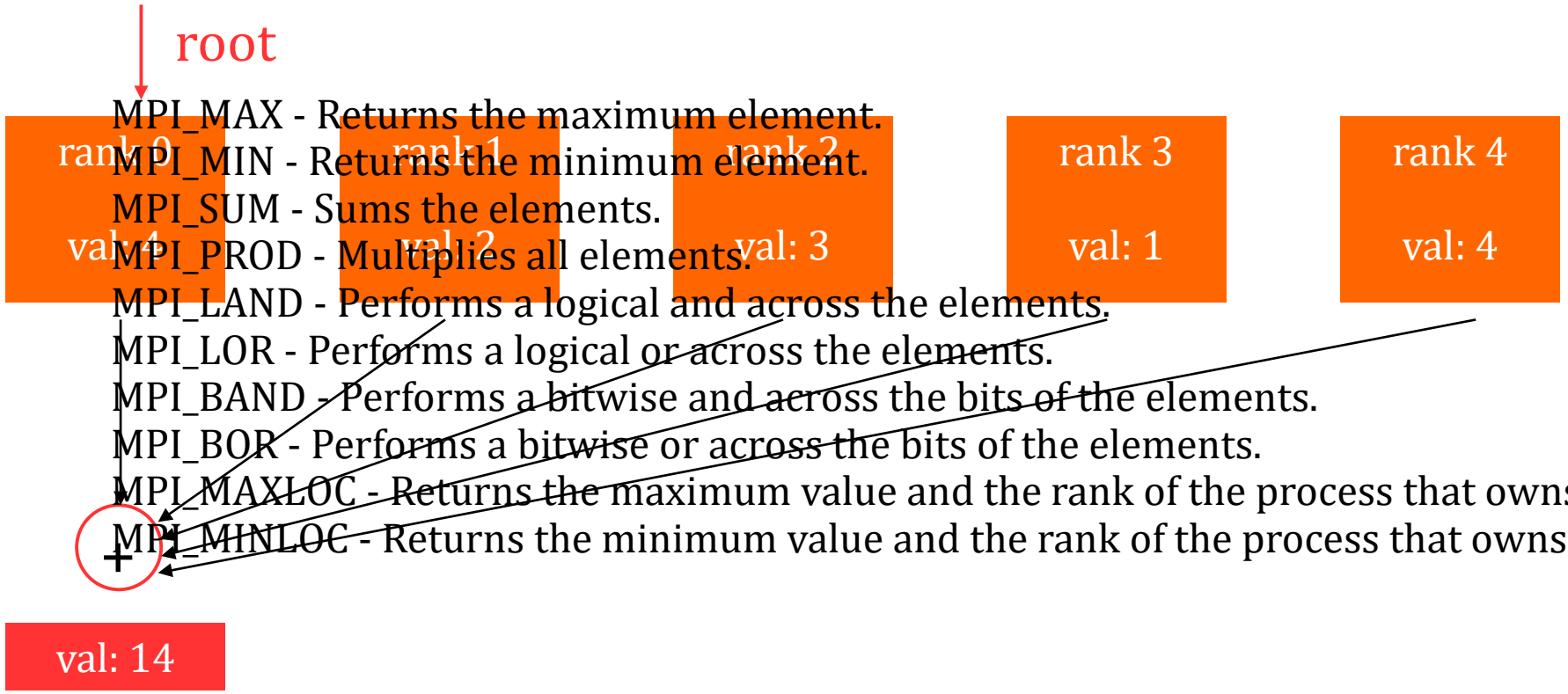
- Each node could send data to node 0, and node 0 can create the sum
- Is there an easier (and better) way?
- We will learn about why the above is inefficient later!!!

MPI_Reduce

- int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- Reduces values on all processes to a single value

MPI_reduce

```
int MPI_Reduce(void *send_data, void *recv_data,
               int count, MPI_Datatype type,
               MPI_op, int root, MPI_Comm comm);
```



MPI_MAX - Returns the maximum element.
MPI_MIN - Returns the minimum element.
MPI_SUM - Sums the elements.
MPI_PROD - Multiplies all elements.
MPI_LAND - Performs a logical and across the elements.
MPI_LOR - Performs a logical or across the elements.
MPI_BAND - Performs a bitwise and across the bits of the elements.
MPI_BOR - Performs a bitwise or across the bits of the elements.
MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

Reduction

- Now imagine that all nodes needs the reduced value.
- How will we do this?

MPI_Allreduce

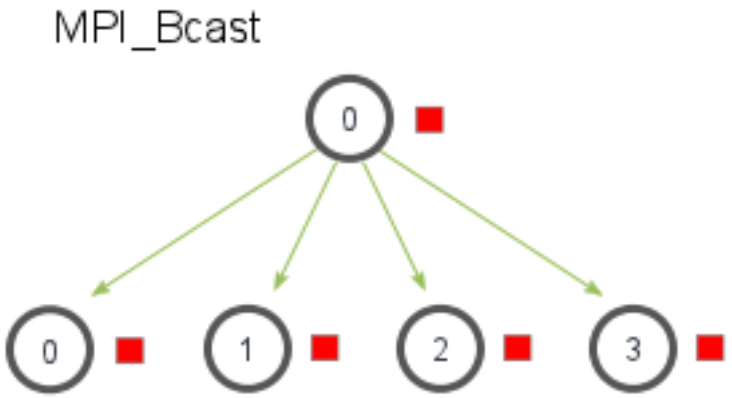
- int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- Combines values from all processes and distributes the result back to all processes

MPI_Allreduce

```c
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
    int size, rank;

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


    int local_val;
    int global_sum;

    local_val = rank * 2; //some random value


    MPI_Allreduce( &local_val, &global_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD );


    printf("I am rank %d. My local value was %d and the global_sum is %d\n", rank, local_val, global_sum);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Bcast

- Lets think about another scenario. We want one node to send data to all other nodes.
- Obviously we can use our plain old send and recv.
- The question is can we do it easier and better
- We will learn about why the above is inefficient later!!!



Source:mpitutorial.com

MPI_Bcast

- int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- Broadcasts a message from the process with rank "root" to all other processes of the communicator

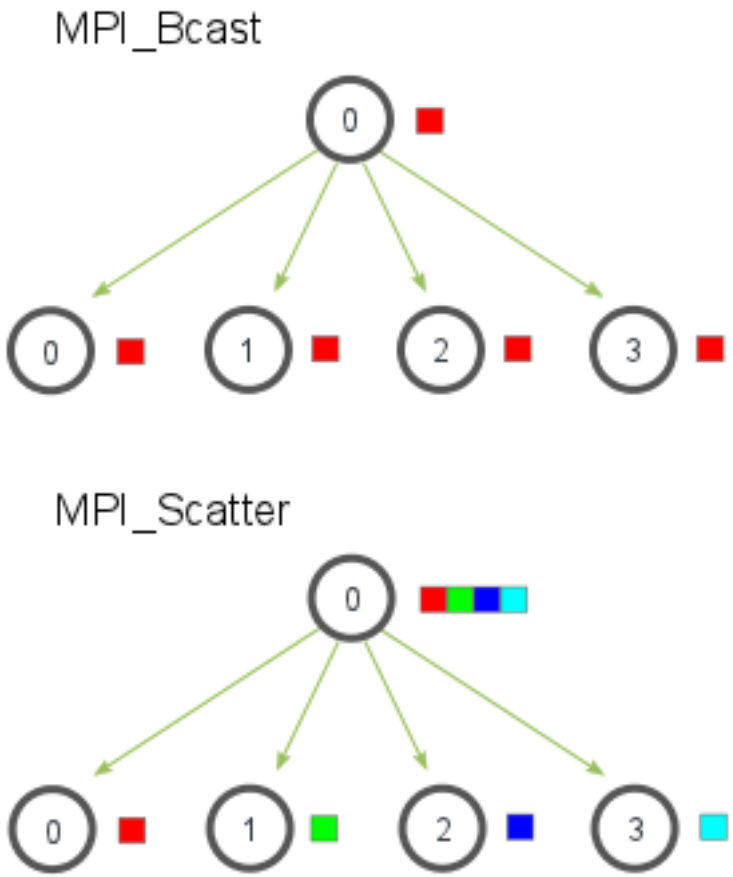
MPI_Bcast

```c
int main(int argc, char* argv[])
{

    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    //lets assume node 1 is the root
    int root = 1;
    int data = world_rank;

    MPI_Bcast(&data, 1, MPI_INT, root, MPI_COMM_WORLD);


    /// If we dont use Bcast, we will have to use the below code

    /*
    if (world_rank == root)
    {
        for (int i = 0; i < world_size; i++)
        {
            if (i != world_rank)
            {
                MPI_Send(data, count, datatype, i, 0, MPI_COMM_WORLD);
            }
        }
    }
    else
    {
        // If we are a receiver process, receive the data from the root
        MPI_Recv(data, count, datatype, root, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
    */

    printf("Hi!!! I am rank %d. my data is %d\n", world_rank, data);

    MPI_Finalize();

    return 0;
}
```

Scatter

- Lets consider another situation, where one node has an array. We want to divide it and send each piece to all nodes
- As an example, consider that Rank 0 has a list of jobs that should be split evenly across all nodes. We have to send the job information to each node
- How do we do this?

Scatter



Source:mpitutorial.com

MPI_Scatter

- int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
- Sends data from one process to all other processes in a communicator

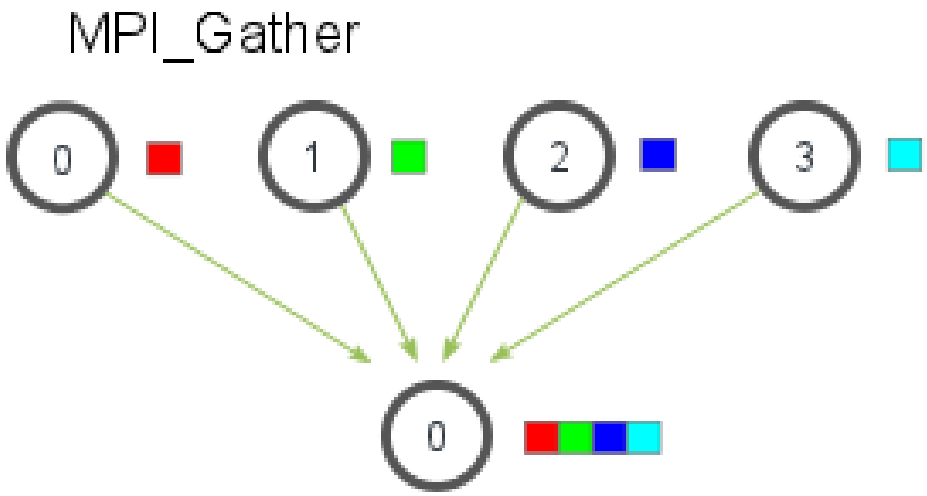
MPI_Scatter

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

int main(int argc, char* argv[])
{

    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    /// lets assume node 1 is the root
    int root = 1;
    /// 5 jobs per node
    int *jobs = malloc(world_size*sizeof(int)*5);

    if(world_rank == root )
    for (int i = 0; i < world_size*5; ++i) {
        jobs[i] = i*42; //random init
    }

    int *recv= malloc(sizeof(int)*5);

    /// scatter
    MPI_Scatter(jobs, 5, MPI_INT, recv, 5, MPI_INT, root, MPI_COMM_WORLD );



    for (int i = 0; i < 5; ++i) {
        printf("Hi!!! I am rank %d. I got job %d\n", world_rank, recv[i]);
    }

    MPI_Finalize();

    return 0;
}
```

- There is a version of MPI_Scatter called as MPI_Scatterv which allows you to send different amount of data to different ranks

Gather

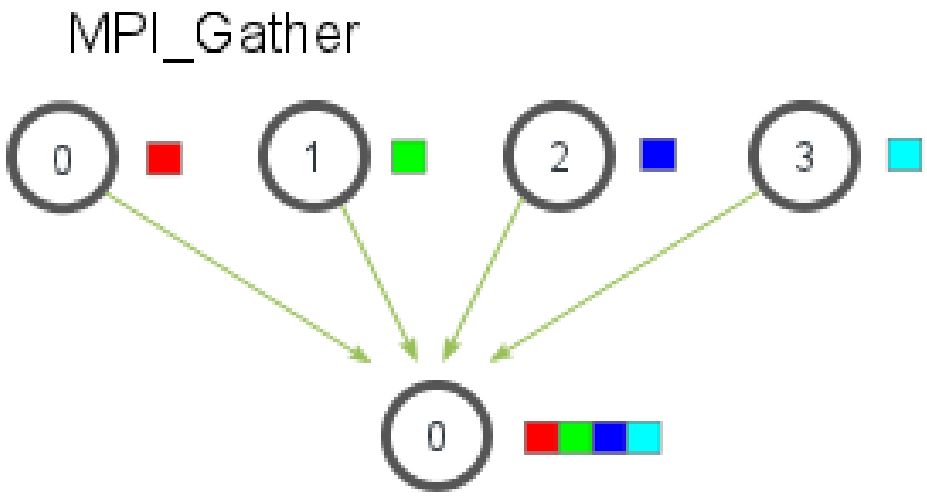
- What if we wanted to do the reverse of Scatter



Source:mpitutorial.com

Gather

- What if we wanted to do the reverse of Scatter



Source:mpitutorial.com

MPI_Gather

- int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

MPI_Gather

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

int main(int argc, char* argv[])
{

    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    /// lets assume node 1 is the root
    int root = 1;
    /// 5 jobs per node
    int *ldata = malloc(sizeof(int) * 5);

    for (int i = 0; i < 5; ++i)
    {
        ldata[i] = world_rank; //random init
    }

    int *recv = malloc(world_size * sizeof(int) * 5);

    /// Gather
    MPI_Gather(ldata, 5, MPI_INT, recv, 5, MPI_INT, root, MPI_COMM_WORLD );

    if(world_rank == root)
        for (int i = 0; i < 5 * world_size; ++i)
        {
            printf("Hi!!! I am rank %d. I got data %d\n", world_rank, recv[i]);
        }

    MPI_Finalize();

    return 0;
}
```

- There is a version of MPI_Gather called as MPI_Gatherv which allows you to receive different amount of data from different ranks

MPI_Allgather

- int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- Gathers data from all tasks and distribute the combined data to all tasks
- In otherwords, the block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

Scan

- MPI also has inbuild support for prefix sum
- Unfortunately this is one of the most misunderstood part of MPI

MPI_Scan

- int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,MPI_Op op, MPI_Comm comm)
- Computes the scan (partial reductions) of data on a collection of processes

MPI_Scan

- int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,MPI_Op op, MPI_Comm comm)
- Computes the scan (partial reductions) of data on a collection of processes

MPI_Scan

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get the sum of all ranks up to mine and print it
    int total;
    MPI_Scan(&my_rank, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    printf("[MPI process %d] Total = %d.\n", my_rank, total);

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

MPI_Scan

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int arr[] = {1, 1, 1, 1};
    int prefix[] = {-1, -1, -1, -1};

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get the sum of all ranks up to mine and print it
    MPI_Scan(&arr, &prefix, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (int i = 0; i < 4; ++i)
    {
        printf("[MPI process %d] Total = %d.\n", my_rank, prefix[i]);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- Seems reasonable
- But this is wont do what you expect it to do

MPI_Scan

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int arr[] = {1, 1, 1, 1};
    int prefix[] = {-1, -1, -1, -1};

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get the sum of all ranks up to mine and print it
    MPI_Scan(&arr, &prefix, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (int i = 0; i < 4; ++i)
    {
        printf("[MPI process %d] Total = %d.\n", my_rank, prefix[i]);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- Seems reasonable
- But this is wont do what you expect it to do

```
[MPI process 0] Total = 1.
[MPI process 0] Total = 1.
[MPI process 0] Total = 1.
[MPI process 0] Total = 1.
[MPI process 1] Total = 2.
[MPI process 1] Total = 2.
[MPI process 1] Total = 2.
[MPI process 1] Total = 2.
[MPI process 2] Total = 3.
[MPI process 2] Total = 3.
[MPI process 2] Total = 3.
[MPI process 2] Total = 3.
[MPI process 3] Total = 4.
[MPI process 3] Total = 4.
[MPI process 3] Total = 4.
[MPI process 3] Total = 4.
```

MPI_Scan

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

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);

    int arr[] = {1, 1, 1, 1};
    int prefix[] = {-1, -1, -1, -1};

    // Get my rank
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // Get the sum of all ranks up to mine and print it
    MPI_Scan(&arr, &prefix, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

    for (int i = 0; i < 4; ++i)
    {
        printf("[MPI process %d] Total = %d.\n", my_rank, prefix[i]);
    }

    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

- Inside a node you have do the prefix sum yourself
- The MPI Scan with multiple elements means that your are doing prefix sum on multiple elements simultaneously

MPI_Scan

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

int arr[] = {1, 1, 1, 1};
int prefix[] = {-10, -10, -10, -10};

// Get my rank
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

prefix[0] = arr[0];

for (int i = 1; i < 4; ++i)
{
    prefix[i] = prefix[i - 1] + arr[i];
}

int from_left = 0;

// Get the sum of all ranks up to mine and print it
MPI_scan(&prefix[3], &from_left, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

for (int i = 0; i < 4; ++i)
{
    prefix[i] += from_left;
}

for (int i = 0; i < 4; ++i)
{
    printf("[MPI process %d] Total = %d.\n", my_rank, prefix[i]);
}

MPI_Finalize();
```

- Take a look at the o/p. We are right and wrong ☺
- In most cases this is not what you want

```
[MPI process 0] Total = 5.
[MPI process 0] Total = 6.
[MPI process 0] Total = 7.
[MPI process 0] Total = 8.
[MPI process 1] Total = 9.
[MPI process 1] Total = 10.
[MPI process 1] Total = 11.
[MPI process 1] Total = 12.
[MPI process 2] Total = 13.
[MPI process 2] Total = 14.
[MPI process 2] Total = 15.
[MPI process 2] Total = 16.
[MPI process 3] Total = 17.
[MPI process 3] Total = 18.
[MPI process 3] Total = 19.
[MPI process 3] Total = 20.
```

MPI_Scan

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

int arr[] = {1, 1, 1, 1};
int prefix[] = {-10, -10, -10, -10};

// Get my rank
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

prefix[0] = arr[0];

for (int i = 1; i < 4; ++i)
{
    prefix[i] = prefix[i - 1] + arr[i];
}

int from_left = 0;

// Get the sum of all ranks up to mine and print it
MPI_Exscan(&prefix[3], &from_left, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

for (int i = 0; i < 4; ++i)
{
    prefix[i] += from_left;
}

for (int i = 0; i < 4; ++i)
{
    printf("[MPI process %d] Total = %d.\n", my_rank, prefix[i]);
}

MPI_Finalize();
```

- Finally!!!
- We had to use exclusive scan

```
[MPI process 0] Total = 1.
[MPI process 0] Total = 2.
[MPI process 0] Total = 3.
[MPI process 0] Total = 4.
[MPI process 1] Total = 5.
[MPI process 1] Total = 6.
[MPI process 1] Total = 7.
[MPI process 1] Total = 8.
[MPI process 2] Total = 9.
[MPI process 2] Total = 10.
[MPI process 2] Total = 11.
[MPI process 2] Total = 12.
[MPI process 3] Total = 13.
[MPI process 3] Total = 14.
[MPI process 3] Total = 15.
[MPI process 3] Total = 16.
```

All to all

- There are situations in which you want to send a portion of data that you have to others and you want to receive a portion of data that others have

array u								Rank	array v							
10	11	12	13	14	15	16	17	0	10	11	20	21	30	31	40	41
20	21	22	23	24	25	26	27	1	12	13	22	23	32	33	42	43
30	31	32	33	34	35	36	37	2	14	15	24	25	34	35	44	45
40	41	42	43	44	45	46	47	3	16	17	26	27	36	37	46	47

Source:TUdelft

All to all

- int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
- Sends data from all to all processes

All to all

```c
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int size, my_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(size != 4)
    {
        printf("This application is meant to be run with 4 MPI processes.\n");
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    int my_values[8], recv[8];

    for(int i = 0; i < 8; i++)
        my_values[i] = my_rank * 100 + i;

    printf("Process %d, my values = %d, %d, %d, %d, %d, %d, %d, %d\n", my_rank,
                    my_values[0], my_values[1], my_values[2], my_values[3],
                    my_values[4], my_values[5], my_values[6], my_values[7]);
    /// This is the core
    MPI_Alltoall(&my_values, 2, MPI_INT, recv, 2, MPI_INT, MPI_COMM_WORLD);

    printf("Process %d, received = %d, %d, %d, %d, %d, %d, %d, %d\n", my_rank,
                    recv[0], recv[1], recv[2], recv[3], recv[4], recv[5],
                    recv[6], recv[7]);


    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

```
Process 1, my values = 100, 101, 102, 103, 104, 105, 106, 107
Process 3, my values = 300, 301, 302, 303, 304, 305, 306, 307
Process 0, my values = 0, 1, 2, 3, 4, 5, 6, 7
Process 2, my values = 200, 201, 202, 203, 204, 205, 206, 207
Process 3, received = 6, 7, 106, 107, 206, 207, 306, 307
Process 0, received = 0, 1, 100, 101, 200, 201, 300, 301
Process 1, received = 2, 3, 102, 103, 202, 203, 302, 303
Process 2, received = 4, 5, 104, 105, 204, 205, 304, 305
```

- MPI_Alltoallv allows to customize the amount data send to each process

Summary

Process 0	Process 1*	Process 2	Process 3	Function Used	Process 0	Process 1*	Process 2	Process 3
a	b	c	d	MPI_Gather	-	a,b,c,d	-	-
a	b	c	d	MPI_Allgather	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
-	a,b,c,d	-	-	MPI_Scatter	a	b	c	d
a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	MPI_Alltoall	a,e, i,m	b,f, j,n	c,g, k,o	d,h, l,p
-	b	-	-	MPI_Bcast	b	b	b	b
Send Buffer	Send Buffer	Send Buffer	Send Buffer		Receive Buffer	Receive Buffer	Receive Buffer	Receive Buffer

*** is our root**

Source:bu.edu