MPI - Collectives

Introduce concurrent communication amongst processes
Outline the structure of collective communications
Implement scatter-gather, reduction and scan patterns

Broadcasting | Scatter | Gather | Reduction | Scan | Exercises

MPI is not limited to point-to-point communications between two processes. Communications that involve more than two processes are called collective communications. During collective communications, we minimize communication time by reducing the number of idle compute nodes. MPI implements this through primitive concurrent support.

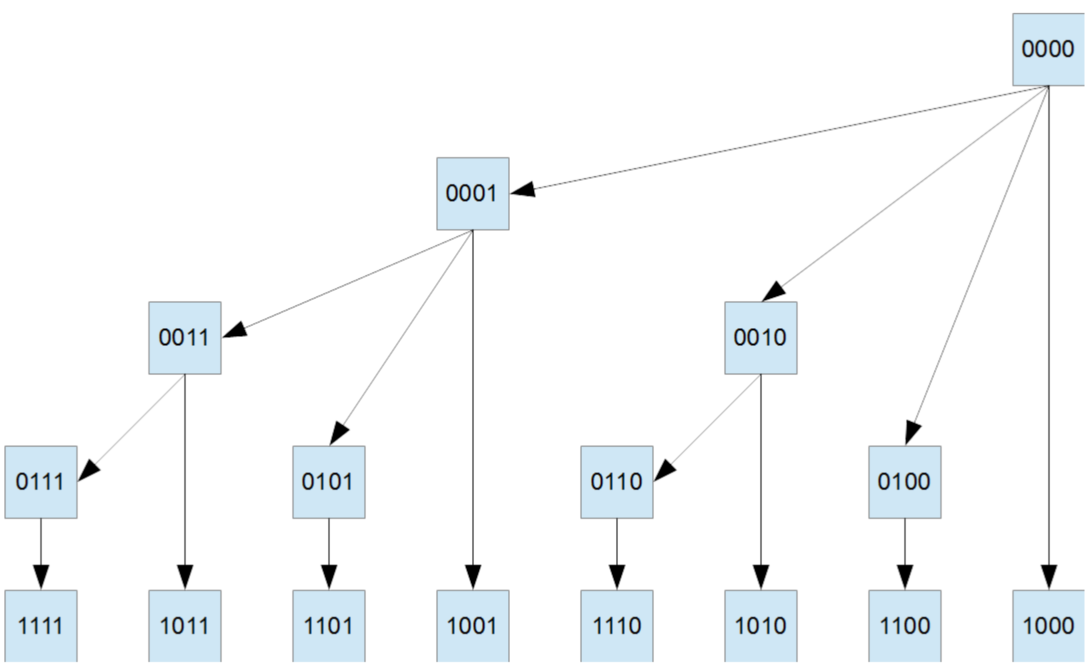
This chapter describes the MPI functions for broadcasting, scattering, gathering, reducing and scanning data across multiple processes.

BROADCASTING

In MPI, the master process - the process with rank 0 - has exclusive access to the standard input stream. The master process must necessarily broadcast data from the input stream to all of the other processes that will work on that data.

Ideal Broadcasting

Using the MPI point-to-point communication primitives, broadcasting would involve a serial algorithm of Θ(n) complexity. To reduce this complexity we can task each node with broadcasting the data that has received to a subset of idle processes. The hypercube algorithm in Figure 1 illustrates this concurrent solution.



Broadcast - Hypercube Algorithm

The hypercube algorithm distributes data to dimensions that haven't received data. Each node distributes to nodes with higher dimensions. We can identify the idle processes using the bit-wise representation of a process' rank. In any bit-wise representation the higher dimensions are those above the most significant bit. For instance, rank 2 of 16 (0010) distributes to ranks 6 (0110) and 10 (1010), while rank 3 of 16 (0011) distributes to ranks 7 (0111) and 11 (1011).

The hypercube solution is only optimal for a set of communication links that forms a complete graph. If the graph is incomplete, this solution is sub-optimal.

Irregular Links

MPI provides an optimized broadcasting implementation tuned for efficient performance on networks of heterogeneous components where the links may be irregular.

The `MPI_Bcast()` function broadcasts the same information to all nodes in a network optimally.

```
int                     // error code
MPI_Bcast(
    void *buffer,       // address of data buffer
    int count,          // number of elements in data buffer
    MPI_Datatype type,  // MPI type of elements in buffer
    int src,            // rank of the source process
    MPI_Comm            // communicator
)
```

Requirements

The requirements for using this function include:

- the broadcast applies to all nodes of a communicator. The operation does not distinguish by tag.
- related calls are identical for both source and sink processes - the type and number of input items are identical to the type and number of output items

Example

The master process in the following program broadcasts the contents of `data` to all other processes. The full set of processes generates the output listed on the right

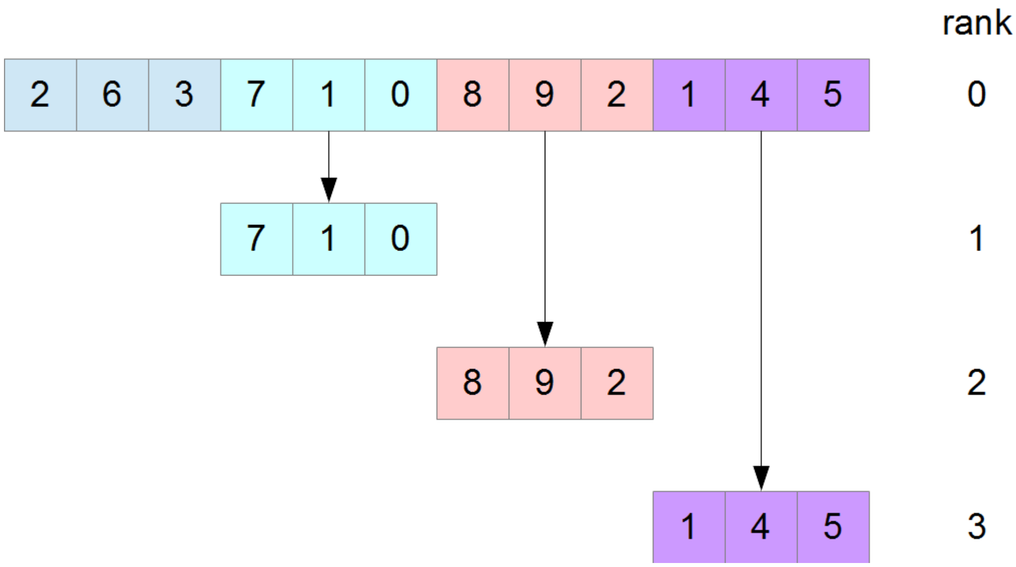
```
// MPI Collectives
// mpi_broadcast.c
#include <stdio.h>
#include <mpi.h>
#define D_SIZE 5

int main(int argc, char** argv) {
    int rank, np, i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == 0) {
        double data[D_SIZE];
        for (i = 0; i < D_SIZE; i++)
            data[i] = (double)i / D_SIZE;
        MPI_Bcast(data, D_SIZE,
         MPI_DOUBLE, 0, MPI_COMM_WORLD);
        printf("All sent!!\n");                All sent!!
    }
    else {
        double data[D_SIZE];
        MPI_Bcast(data, D_SIZE,
         MPI_DOUBLE, 0, MPI_COMM_WORLD);
        printf("#%d rcvd ", rank);
        for (i = 0; i < D_SIZE; i++)           #1 rcvd 0.00 0.20 0.40 0.60 0.80
            printf("%.2lf ", data[i]);         #2 rcvd 0.00 0.20 0.40 0.60 0.80
```

```
        printf("\n");
    }
    MPI_Finalize();
    return 0;
}
```

```
#4 rcvd 0.00 0.20 0.40 0.60 0.80
#5 rcvd 0.00 0.20 0.40 0.60 0.80
#7 rcvd 0.00 0.20 0.40 0.60 0.80
#3 rcvd 0.00 0.20 0.40 0.60 0.80
#6 rcvd 0.00 0.20 0.40 0.60 0.80
```

SCATTER

Scatter algorithms for a distributed memory system deal out the elements of a data set across the different processes with each process receiving its own subset. An in-place scatter algorithm leaves the subset assigned to the source node in place and avoids the overhead of copying that subset back to the source node.



An In-Place Scatter Algorithm

The **`MPI_Scatter()`** function uses two buffers: a source buffer and a destination buffer:

```
int                     // error code
MPI_Scatter(
    // source data
    void *send,         // address of send buffer
    int count,          // number of elements in send buffer per process
    MPI_Datatype stype, // MPI type of elements in send buffer

    // destination data
    void *recv,         // address of receive buffer
    int count,          // number of elements in receive buffer
    MPI_Datatype rtype, // MPI type of elements in receive buffer

    // process information
    int src,            // rank of the source process
    MPI_Comm            // communicator
)
```

As with **`MPI_Bcast()`** the calls are identical for all particpating processes - sources and sinks.

Example

The master process in the following program scatters the contents of **`data`** amongst processes. The results generated by this program are listed on the right

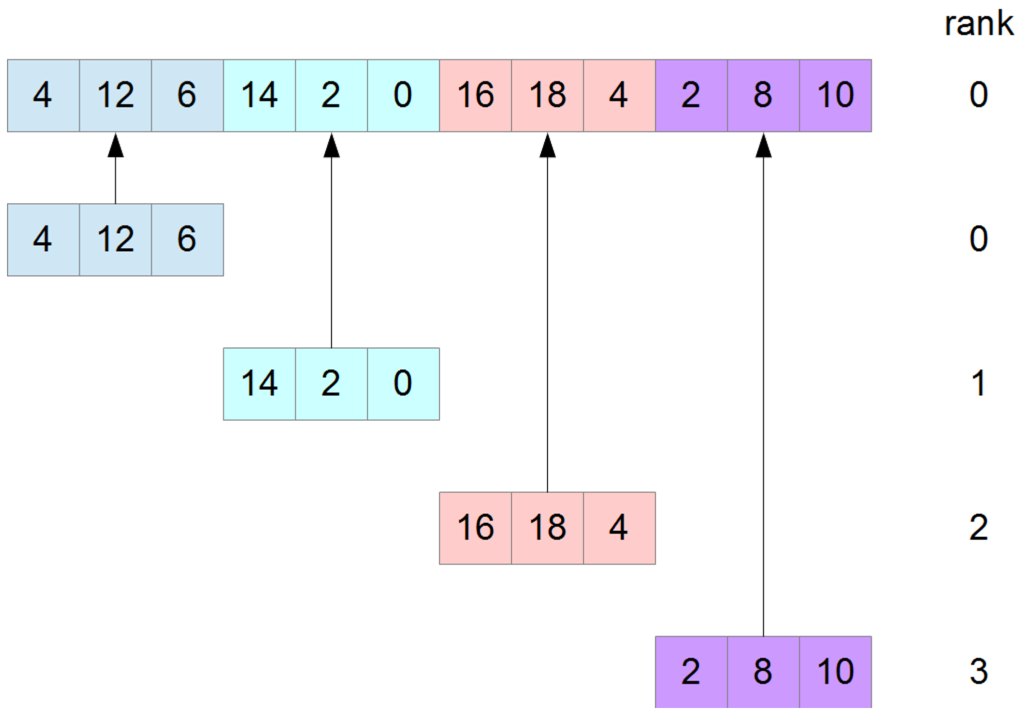
```
// MPI Collectives
// mpi_scatter.c
#include <stdio.h>
#include <mpi.h>
#define SIZE 12
#define NPR 3
#define TAG 1

void op(int *data, int size) {
    int i;
    for (i = 0; i < size; i++)
        data[i] *= 2;
}

int main(int argc, char** argv) {
    int rank, np, i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == 0) {
        int data[SIZE] = {2, 6, 3, 7, 1, 0, 8, 9, 2, 1, 4, 5};
        MPI_Scatter(data, 3, MPI_INT, MPI_IN_PLACE,
         0, MPI_INT, 0, MPI_COMM_WORLD);
        op(data, NPR);
        for (i = 1; i < np; i++)
            MPI_Recv(data + i * NPR, NPR, MPI_INT, i, TAG,
             MPI_COMM_WORLD, &status);
        for (i = 0; i < SIZE; i++)
            printf("%3d\n", data[i]);
    }
    else {
        int data[NPR];
        MPI_Scatter(NULL, NPR, MPI_INT, data, NPR, MPI_INT,
         0, MPI_COMM_WORLD);
        printf("Rcvd by #%d\n", rank);
        op(data, NPR);
        MPI_Send(data, NPR, MPI_INT, 0, TAG, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

Run 4
processes

```
Rcvd by #1
  4
 12
  6
 14
  2
  0
 16
 18
  4
  2
  8
 10
Rcvd by #2
Rcvd by #3
```

GATHER

Gather algorithms for a distributed memory system collect data from all of a communicator's processes into the destination process.



Gather Algorithm

The **MPI_Gather()** function uses two buffers: a source buffer and a destination buffer:

```
int                      // error code
MPI_Gather(
    // source data
    void *send,          // address of send buffer
    int count,           // number of elements in send buffer per process
    MPI_Datatype stype,  // MPI type of elements in send buffer

    // destination data
    void *recv,          // address of receipt buffer
    int count,           // number of elements in receive buffer
    MPI_Datatype rtype,  // MPI type of elements in receive buffer

    // process information
    int dest,            // rank of the destination process
    MPI_Comm             // communicator
)
```

As with **MPI_Bcast()** the calls are identical for all particpating processes - sources and sinks.

Example

The master process in the following program scatters the contents of **data** amongst the processes, each process doubles each data value in its own subset and finally the master process gathers the results. The results generated by this program are listed on the right

```
// MPI Collectives
// mpi_gather.c
#include <stdio.h>
#include <mpi.h>
#define D_SIZE 12
#define NPR 3
```

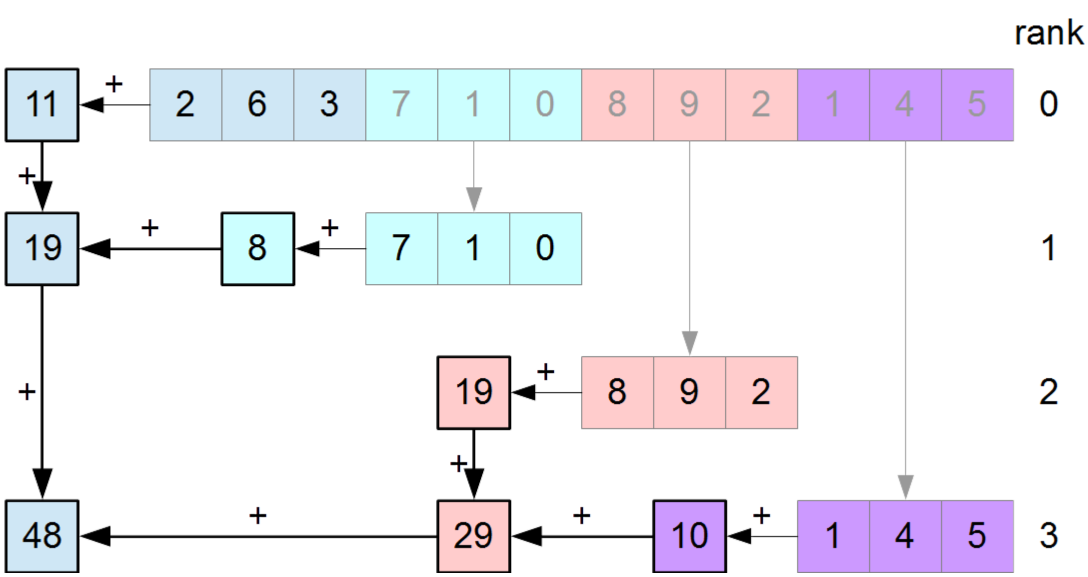
Run 4 processes

```c
void op(int *data, int size) {
    int i;
    for (i = 0; i < size; i++)
        data[i] *= 2;
}

int main(int argc, char** argv) {
    int rank, np, i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == 0) {
        int data[D_SIZE] = { 2, 6, 3, 7, 1, 0, 8, 9, 2, 1, 4, 5 };
        int result[D_SIZE];
        MPI_Scatter(data, NPR, MPI_INT, MPI_IN_PLACE,
         0, MPI_INT, 0, MPI_COMM_WORLD);
        op(data, NPR);
        MPI_Gather(data, NPR, MPI_INT, result, NPR, MPI_INT,
         0, MPI_COMM_WORLD);
        for (i = 0; i < D_SIZE; i++)
            printf("%3d\n", result[i]);
    }
    else {
        int data[NPR];
        MPI_Scatter(NULL, NPR, MPI_INT, data, NPR, MPI_INT,
         0, MPI_COMM_WORLD);
        printf("Rcvd by #%d\n", rank);
        op(data, NPR);
        MPI_Gather(data, NPR, MPI_INT, NULL, NPR, MPI_INT,
         0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

```
Rcvd by #1
Rcvd by #2
Rcvd by #3
  4
 12
  6
 14
  2
  0
 16
 18
  4
  2
  8
 10
```

REDUCTION

Reduction algorithms for a distributed memory system collect the data values generated by each process and apply an operation to reduce those values to a single value.



Reduction Algorithm - Sum Operation

The `MPI_Reduce()` function uses two buffers: a source buffer and a destination buffer:

```
int                     // error code
MPI_Reduce(
    void *sbuf,         // address of source buffer
    void *dbuf,         // address of destination buffer
    int count,          // number of elements in source buffer per process
    MPI_Datatype type,  // MPI type of elements in source buffer
    MPI_Op op,          // type of reduction to perform
    int src,            // rank of destination process
    MPI_Comm            // communicator
)
```

As with `MPI_Bcast()` the calls are identical for all particpating processes - sources and sinks.

Example

The master process in the following program scatters the contents of `data` amongst all processes. Each process reduces its data to a single result and returns that result to the master process for final reduction. The results generated by this program are listed on the right

```
// MPI Collectives
// mpi_reduce.c
#include <stdio.h>
#include <mpi.h>
#define D_SIZE 12
#define NPR 3

int sum(int *data, int size) {
    int i, sum = 0;
    for (i = 0; i < size; i++)
        sum += data[i];
    return sum;
}

int main(int argc, char** argv) {
    int rank, np, i;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    if (rank == 0) {
        int data[D_SIZE] =
         { 2, 6, 3, 7, 1, 0, 8, 9, 2, 1, 4, 5 };
        MPI_Scatter(data, 3, MPI_INT, MPI_IN_PLACE, 0,
         MPI_INT, 0, MPI_COMM_WORLD);
        printf("#%d Data: ", rank);
        for (i = 0; i < NPR; i++)
            printf("%2d ", data[i]);
        printf("\n");
        int part_sum = sum(data, NPR);
        printf("#%d Part sum = %3d\n", rank, part_sum);
        int grand_sum;
        MPI_Reduce(&part_sum, &grand_sum, 1, MPI_INT,
         MPI_SUM, 0, MPI_COMM_WORLD);
        printf("Grand sum   = %3d\n", grand_sum);
    }
    else {
        int data[NPR];
```

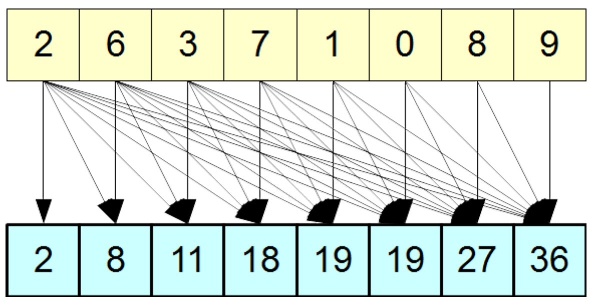
```
Run 4 processes
```

```
        MPI_Scatter(NULL, NPR, MPI_INT, data, NPR,
         MPI_INT, 0, MPI_COMM_WORLD);
        printf("#%d Data: ", rank);
        for (i = 0; i < NPR; i++)
            printf("%2d ", data[i]);
        printf("\n");
        int part_sum = sum(data, NPR);
        printf("#%d Part sum = %3d\n", rank, part_sum);
        MPI_Reduce(&part_sum, NULL, 1, MPI_INT,
         MPI_SUM, 0, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

```
#1 Data:  7  1  0
#1 Part sum =   8
#3 Data:  1  4  5
#3 Part sum =  10
#2 Data:  8  9  2
#2 Part sum =  19
#0 Data:  2  6  3
#0 Part sum =  11
Grand sum   =  48
```

SCAN

Scan algorithms for a distributed memory system incorporate contributions from processes of lower rank.



Scan Algorithm - Summation Operation

The **`MPI_Scan()`** function uses two buffers: a source buffer and a destination buffer:

```
int                     // error code
MPI_Scan(
    void *sbuf,         // address of source buffer
    void *dbuf,         // address of destination buffer
    int count,          // number of elements in source buffer per process
    MPI_Datatype type, // MPI type of elements in source buffer
    MPI_Op op,          // type of reduction to perform
    MPI_Comm            // communicator
)
```

Example

The master process in the following program scatters the contents of **`data`** amongst processes. The results generated by this program are listed on the right

```
// MPI Collectives
// mpi_scan.c
#include <stdio.h>
#include <mpi.h>
#define D_SIZE 8

int main(int argc, char** argv) {
    int rank, np;

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

```
Run 8
Processes
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    int data[D_SIZE] = { 2, 6, 3, 7, 1, 0, 8, 9 };
    int result;
    int local = 1;                                        #0 gives  2
    if (rank < D_SIZE) local = data[rank];                #6 gives 27
    MPI_Scan(&local, &result, 1, MPI_INT, MPI_SUM,        #2 gives 11
     MPI_COMM_WORLD);                                     #1 gives  8
    printf("#%d gives %2d\n", rank, result);              #3 gives 18
    MPI_Finalize();                                       #7 gives 36
    return 0;                                             #4 gives 19
}                                                         #5 gives 19
```

EXERCISES

- MPI Forum (2012). MPI: A Message-Passing Interface Standard, Version 3.0
- Barney, B. (2014). "Message Passing Interface (MPI)". Lawrence Livermore National Laboratory. Retrieved Jan 19 2015. Last Modified Nov 12 2014.
- Microsoft (2015). Microsoft MPI
- Barlas, G. (2015). Multicore and GPU Programming - An Integrated Approach. Morgan-Kaufmann. 978-0-12-417137-4. pp.261-279.