Computação em Larga Escala

Message Passing Interface (*MPI*) *1*

António Rui Borges

Summary

- *Collective communication*
  - *Broadcast*
  - *Scatter*
  - *Gather*
  - *Reduce*
- *Suggested reading*

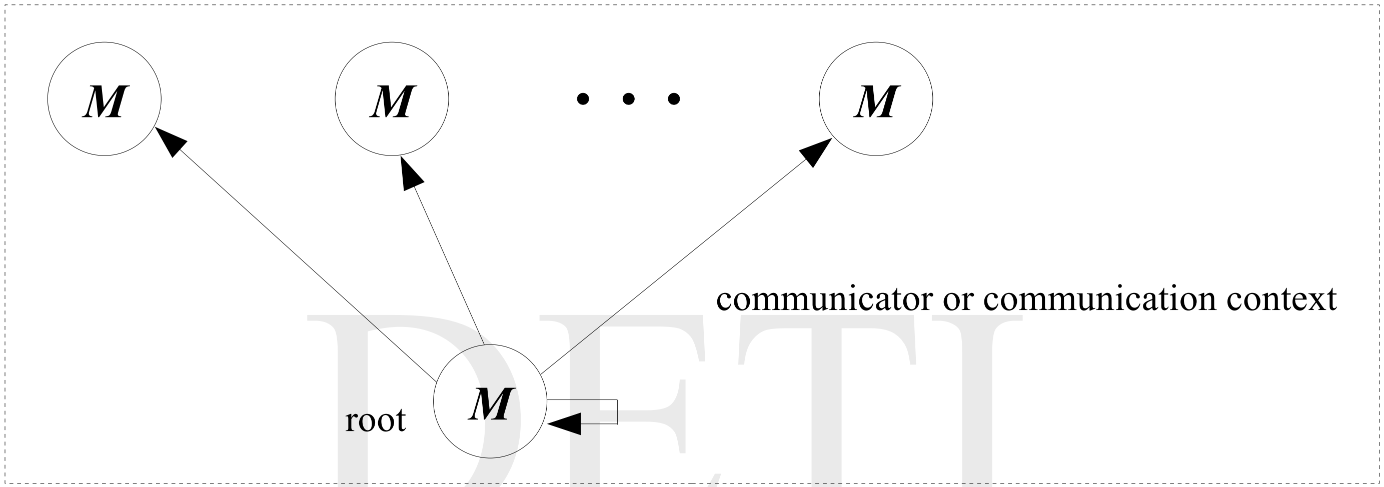
Collective communication

Collective communication addresses the case where multiple processes, organized in a group, or groups of participating processes, exchange messages.

There are several types of collective communication

- *broadcast* – a message is sent from a *root* process (the calling process of the communication group) to all the processes (belonging to the communication group), itself included
- *scatter* – specific messages of similar type are sent from a *root* process (the calling process of the communication group) to all the processes (belonging to the communication group), itself included
- *gather* – specific messages of similar type are sent from all the processes (belonging to the communication group) to a *root* process (the calling process of the communication group).

Broadcast - 1



- the message *M* is sent from the process with rank *root* to all the processes of the communication group, itself included
- in the standard case, *broadcast* is a blocking operation, that is, the processes block until the message is effectively received
- the *broadcast* operation can be thought of as the process with rank *root* to perform a *send* and all the processes in the group to perform next a *receive*

Broadcast - 2

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

buf - pointer to the memory region where the information content of the message is, or will be, stored

count - number of elements of the array which represents the information content of the message

datatype - MPI information data type

root - rank of the root process

comm - identification of the communication context (process group)

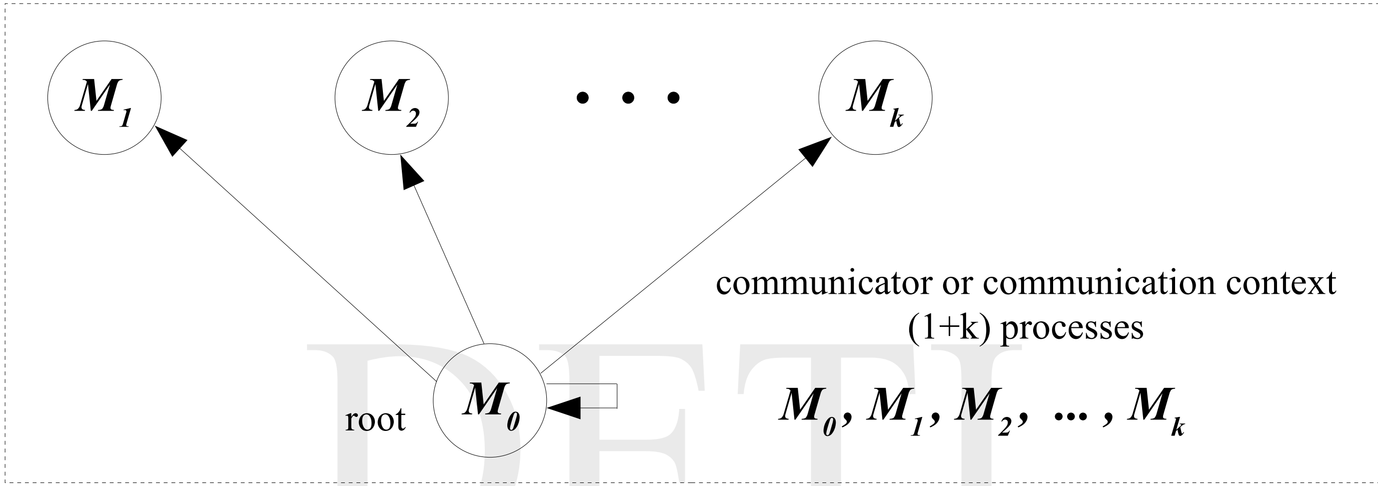
Broadcast - 3

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

int main (int argc, char *argv[])
{
   int rank;
   char *data = malloc ((strlen ("I, 0, am the leader!") + 1) * sizeof (char));
   int len = strlen ("I, 0, am the leader!") + 1;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   if (rank == 0)
      { data = "I, 0, am the leader!";
        printf ("Broadcast message: %s \n", data);
      }
   MPI_Bcast(data, len, MPI_CHAR, 0, MPI_COMM_WORLD);
   printf ("I, %d, received the message: %s \n", rank, data);
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Broadcast - 4

```
[ruib@ruib-laptop basic]$ mpicc -Wall -o broadCast broadCast.c

[ruib@ruib-laptop basic]$ mpiexec -n 10 ./broadCast
Broadcast message: I, 0, am the leader!
I, 0, received the message: I, 0, am the leader!
I, 1, received the message: I, 0, am the leader!
I, 4, received the message: I, 0, am the leader!
I, 5, received the message: I, 0, am the leader!
I, 6, received the message: I, 0, am the leader!
I, 7, received the message: I, 0, am the leader!
I, 2, received the message: I, 0, am the leader!
I, 3, received the message: I, 0, am the leader!
I, 8, received the message: I, 0, am the leader!
I, 9, received the message: I, 0, am the leader!
[ruib@ruib-laptop basic]$
```

Scatter - 1



- a collection of k+1 similar messages M_0 , M_1 , M_2 , ... , M_k , where k+1 is the group size, are sent from the process with rank *root* to all the processes of the communication group, itself included
- in the standard case, *scatter* is a blocking operation, that is, the processes block until the message is effectively received
- the *scatter* operation can be thought of as the process with rank *root* to perform a *send* and all the processes in the group to perform next a *receive*

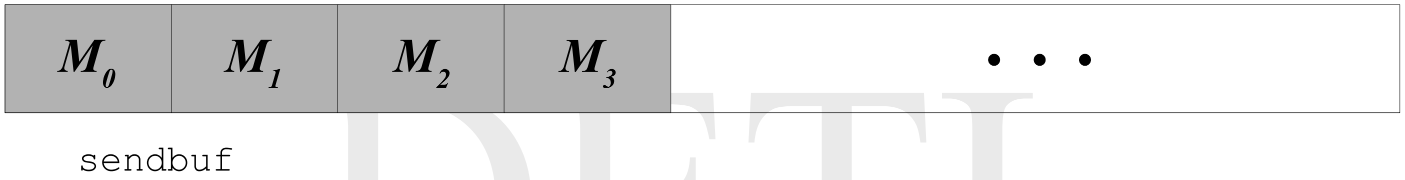
Scatter - 2

```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                 MPI_Comm comm);
int MPI_Scatterv (void *sendbuf, int *sendcounts, int *displs, MPI_Datatype sendtype,
                  void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                  MPI_Comm comm);
```

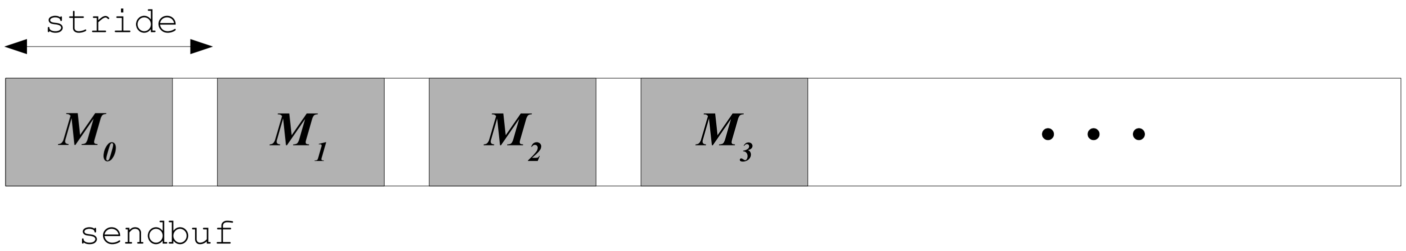
sendbuf – pointer to the memory region where the information content of the multiple messages to be sent are stored (significant only at root)
sendcount – number of elements sent to each process (significant only at root)
sendcounts – array with the number of elements sent to each process (significant only at root)
displs – array with the displacements (relative to the beginning of sendbuf) from which to take the sent data to each process
sendtype – MPI data type of send buffer elements (significant only at root)
recvbuf - pointer to the memory region where the information content of the received message is to be stored
recvcount – maximum number of elements received by each process
recvtype – MPI data type of receive buffer elements
root – rank of the root process
comm – identification of the communication context (process group)

Scatter - 3

In `MPI_Scatter`, all messages to be sent have the same length and are stored contiguously in the sent buffer.



In `MPI_Scatterv`, all messages to be sent may have different lengths and may not be stored contiguously in the sent buffer.



Scatter - 4

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

int main (int argc, char *argv[])
{
   int rank, size, i;
   int *sendData = NULL, *recData;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   recData = malloc (10 * sizeof (int));
   if (rank == 0)
      { sendData = malloc (10 * size * sizeof (int));
        printf ("Data to be scattered\n");
        for (i = 0; i < 10 * size; i++)
        { sendData[i] = i;
          printf ("%d ", sendData[i]);
        }
        printf ("\n");
      }
   MPI_Scatter (sendData, 10, MPI_INT, recData, 10, MPI_INT, 0, MPI_COMM_WORLD);
   printf ("Received data by process %d: ", rank);
   for (i = 0; i < 10; i++)
     printf ("%2d ", recData[i]);
   printf ("\n");
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Scatter - 5

```
[ruib@ruib-laptop basic]$ mpicc -Wall -o scatter1 scatter1.c

[ruib@ruib-laptop basic]$ mpiexec -n 4 ./scatter1
Data to be scattered
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 27 28 29 30 31 32 33 34 35 36 37 38 39
Received data by process 0:  0  1  2  3  4  5  6  7  8  9
Received data by process 2: 20 21 22 23 24 25 26 27 28 29
Received data by process 3: 30 31 32 33 34 35 36 37 38 39
Received data by process 1: 10 11 12 13 14 15 16 17 18 19
[ruib@ruib-laptop basic]$
```

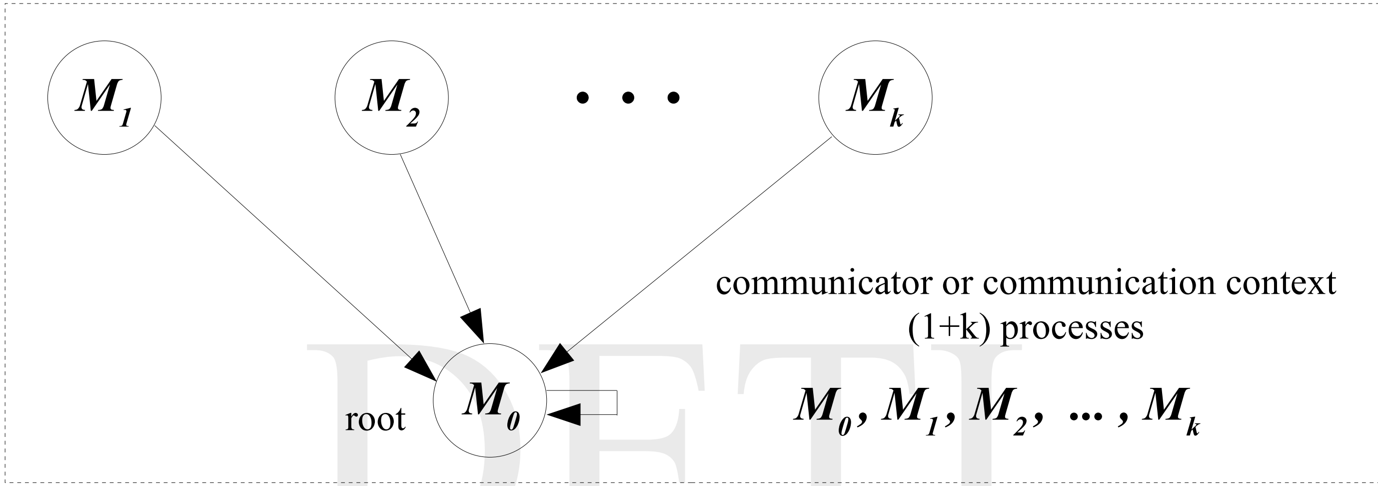
Scatter - 6

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

int main (int argc, char *argv[])
{
   int rank, size;
   int *sendData = NULL, *recData, *sendCount, *disp, recCount;
   int i;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   recData = malloc (10 * sizeof (int));
   sendCount = malloc (size * sizeof (int));
   disp = malloc (size * sizeof (int));
   for (i = 0; i < size; i++)
   { sendCount[i] = (i < 3) ? 2 * (i + 1) : 10;
     disp[i] = 20 * i;
   }
   if (rank == 0)
      { sendData = malloc (20 * size * sizeof (int));
        printf ("Data to be scattered\n");
        for (i = 0; i < 20 * size; i++)
        { sendData[i] = i;
          printf ("%2d ", sendData[i]);
        }
        printf ("\n");
      }
   recCount = (rank < 3) ? 2 * (rank + 1) : 10;
   MPI_Scatterv (sendData, sendCount, disp, MPI_INT, recData, recCount, MPI_INT, 0,
                 MPI_COMM_WORLD);
   printf ("Received data by process %d: ", rank);
   for (i = 0; i < recCount; i++)
     printf ("%2d ", recData[i]);
   printf ("\n");
   MPI_Finalize ();

   return EXIT_SUCCESS;
}
```

Scatter - 7

```
[ruib@ruib-laptop basic]$ mpicc -Wall -o scatter2 scatter2.c

[ruib@ruib-laptop basic]$ mpiexec -n 4 ./scatter2
Data to be scattered
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79
Received data by process 0:  0  1
Received data by process 3: 60 61 62 63 64 65 66 67 68 69
Received data by process 1: 20 21 22 23
Received data by process 2: 40 41 42 43 44 45
[ruib@ruib-laptop basic]$
```

Gather - 1



- a collection of k+1 similar messages M_0 , M_1 , M_2 , ... , M_k , where k+1 is the group size, are sent from all the processes of the communication group, *root* included, to the process with rank *root*
- in the standard case, *gather* is a blocking operation, that is, the process with rank *root* blocks until all the messages are effectively received
- the *gather* operation can be thought of as all the processes in the group to perform a *send* and the process with rank *root* to perform next a *receive*

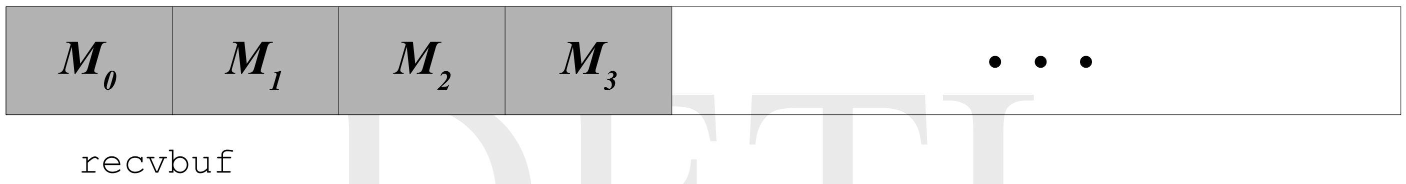
Gather - 2

```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm);
int MPI_Gatherv (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int *recvcounts, int *displs, MPI_Datatype recvtype, int root,
                 MPI_Comm comm);
```

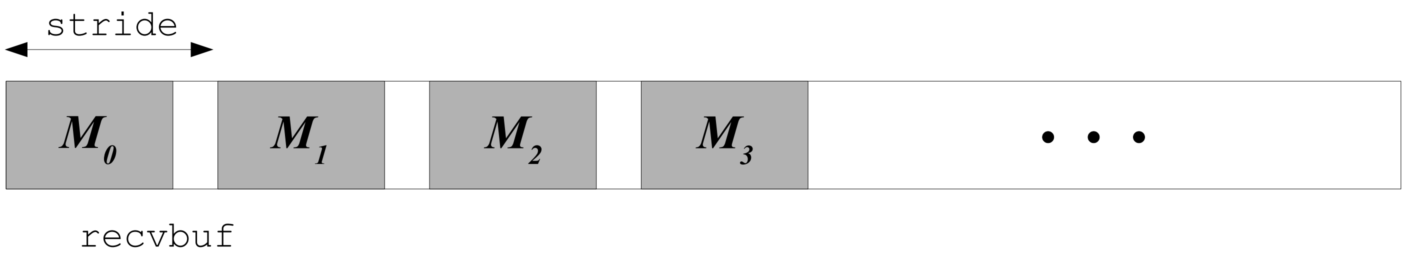
sendbuf - pointer to the memory region where the information content of the message to be sent is stored
sendcount - number of elements sent by each process
sendtype - MPI data type of send buffer elements
recvbuf - pointer to the memory region where the information content of the received messages are to be stored (significant only at root)
recvcount - maximum number of elements received from each process (significant only at root)
recvcounts - array with the number of elements received from each process (significant only at root)
displs - array with the displacements (relative to the beginning of recvbuf) from which to store the sent data from each process (significant only at root)
recvtype - MPI data type of receive buffer elements (significant only at root)
root - rank of the root process
comm - identification of the communication context (process group)

Gather - 3

In `MPI_Gather`, all messages to be received have the same length and are stored contiguously in the receive buffer.



In `MPI_Gatherv`, all messages to be received may have different lengths and may not be stored contiguously in the received buffer.



Gather - 4

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

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

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   sendData = malloc (10 * sizeof (int));
   recData = malloc (10 * size * sizeof (int));
   printf ("Data to be sent by process %d: ", rank);
   for (i = 0; i < 10; i++)
   { sendData[i] = 10 * rank + i;
     printf ("%d ", sendData[i]);
   }
   printf ("\n");
   MPI_Gather (sendData, 10, MPI_INT, recData, 10, MPI_INT, 0, MPI_COMM_WORLD);
   if (rank == 0)
      { printf ("Gathered data\n");
        for (i = 0; i < 10 * size; i++)
          printf ("%d ", recData[i]);
        printf ("\n");
      }
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Gather - 5

```
[ruib@ruib-laptop basic]$ mpicc -Wall -o gather1 gather1.c

[ruib@ruib-laptop basic]$ mpiexec -n 4 ./gather1
Data to be sent by process 0:  0  1  2  3  4  5  6  7  8  9
Data to be sent by process 3: 30 31 32 33 34 35 36 37 38 39
Data to be sent by process 1: 10 11 12 13 14 15 16 17 18 19
Data to be sent by process 2: 20 21 22 23 24 25 26 27 28 29
Gathered data
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39
[ruib@ruib-laptop basic]$
```

Gather - 6

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

int main (int argc, char *argv[])
{
   int rank, size;
   int *sendData, sendCount, *recData, *recCount, *disp, offset;
   int i;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   recCount = malloc (size * sizeof (int));
   disp = malloc (size * sizeof (int));
   offset = 0;
   for (i = 0; i < size; i++)
   { recCount[i] = (i < 3) ? 2 * (i + 1) : 10;
     disp[i] = offset;
     offset += recCount[i];
   }
   sendData = malloc (recCount[rank] * sizeof (int));
   recData = malloc (offset * size * sizeof (int));
   printf ("Data to be sent by process %d: ", rank);
   sendCount = (rank < 3) ? 2 * (rank + 1) : 10;
   for (i = 0; i < sendCount; i++)
   { sendData[i] = 10 * rank + i;
     printf ("%2d ", sendData[i]);
   }
   printf ("\n");
   MPI_Gatherv (sendData, sendCount, MPI_INT, recData, recCount, disp, MPI_INT, 0,
                MPI_COMM_WORLD);
   if (rank == 0)
      { printf ("Gathered data\n");
        for (i = 0; i < offset; i++)
          printf ("%d ", recData[i]);
        printf ("\n");
      }
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Gather - 7

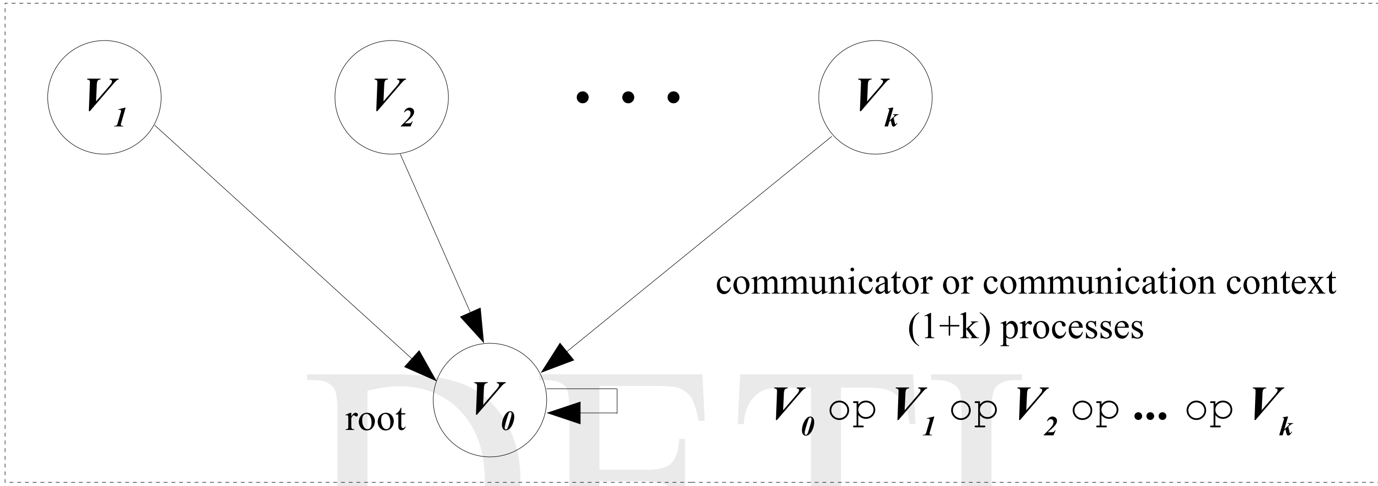
```
[ruib@ruib-laptop basic]$ mpicc -Wall -o gather2 gather2.c

[ruib@ruib-laptop basic]$ mpiexec -n 4 ./gather2
Data to be sent by process 1: 10 11 12 13
Data to be sent by process 3: 30 31 32 33 34 35 36 37 38 39
Data to be sent by process 2: 20 21 22 23 24 25
Data to be sent by process 0:  0  1
Gathered data
0 1 10 11 12 13 20 21 22 23 24 25 30 31 32 33 34 35 36 37 38 39
[ruib@ruib-laptop basic]$
```

Reduce - 1

A further type of collective communication is

- *reduce* – allows one to perform a global calculation by aggregating (reducing) the values of a variable from all the processes (belonging to the communication group) using a commutative binary operator and sending the result to the *root* process (the calling process of the communication group).

Reduce - 2



- a collection of k+1 similar value arrays V_0, V_1, V_2, ... , V_k, where k+1 is the group size, are sent from all the processes of the communication group to the process with rank *root* and are computed together, element by element, through the application of a commutative binary operator
- in the standard case, *reduce* is a blocking operation, that is, the process with rank *root* blocks until the computation is effectively concluded
- the *reduce* operation can be thought of as all the processes in the group to perform a *send* and the process with rank *root* to perform next a *receive*

Reduce - 3

MPI predefined reduction operations

MPI operation	Meaning
MPI_MAX	maximum value
MPI_MIN	minimum value
MPI_SUM	sum of values
MPI_PROD	product of values
MPI_LAND	logical and of values
MPI_BAND	bitwise and of values
MPI_LOR	logical or of values
MPI_BOR	bitwise or of values
MPI_LXOR	logical x-or of values
MPI_BXOR	bitwise x-or of values

Reduce - 4

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

sendbuf – pointer to the memory region where the information content of the message is stored

recvbuf – pointer to the memory region where the information content of the message will be stored (significant only at root)

count – number of elements of the array which represents the information content of the messages to be sent

datatype – MPI information data type

op – reduce operation

root – rank of the root process

comm – identification of the communication context (process group)

Reduce - 5

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

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

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   srandom (getpid ());
   val = ((double) rand () / RAND_MAX) * 1000;
   printf ("Value generated by process %d = %d \n", rank, val);
   MPI_Reduce (&val, &maxVal, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
   if (rank == 0)
      printf ("Largest value = %d\n", maxVal);
   MPI_Finalize ();
   return EXIT_SUCCESS;
}
```

Reduce - 6

```
[ruib@ruib-laptop basic2]$ mpicc -Wall -o reduce reduce.c

[ruib@ruib-laptop basic2]$ mpiexec -n 10 ./reduce
Value generated by process 1 = 393
Value generated by process 2 = 755
Value generated by process 3 = 115
Value generated by process 5 = 328
Value generated by process 0 = 540
Value generated by process 4 = 478
Value generated by process 6 = 688
Value generated by process 7 = 546
Value generated by process 8 = 908
Value generated by process 9 = 270
Largest value = 908
[ruib@ruib-laptop basic2]$
```

Suggested reading

- *Introduction to HPC with MPI for Data Science*, Nielsson F., Springer International, 2016
  - Chapter 2: *Introduction to MPI: The Message Passing Interface*
- *MPI: A Message-Passing Interface Standard* (*Version 3.1*), Message Passing Interface Forum, 2015