Parallel Programming with MPI

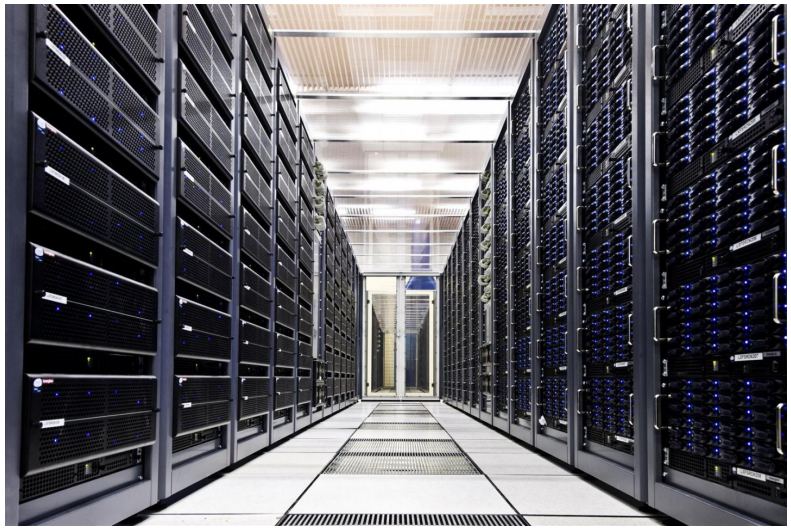
Antonio Falabella

INFN - CNAF (Bologna)

3^{rd} nternational Summer School on **IN**telligent Signal Processing for **F**ront**IE**r **R**esearch and **I**ndustry , 14-25 September 2015, Hamburg

Overview

- The aim of **Parallel Computing** is to reduce the time required to solve a computational problem using parallel computers that are computers with more that one processor or cluster of computers



- Parallelism can be exploited in two different way:
  - **Data parallelism**: Same task on different elements of a dataset
  - **Task or functional parallelism**: The same task on the same or different data
- Several ways to write parallel software:
  - Using a compiler that automatically detect the part of code that can be parallelized $\rightarrow$ not easy to achieve
  - One of the easiest ways is to add functions or compiler directives to an existing sequential language such that a user can explicitly specify which code should be performed in parallel: MPI (these lectures), OpenMP, etc...

Flynn's Taxonomy

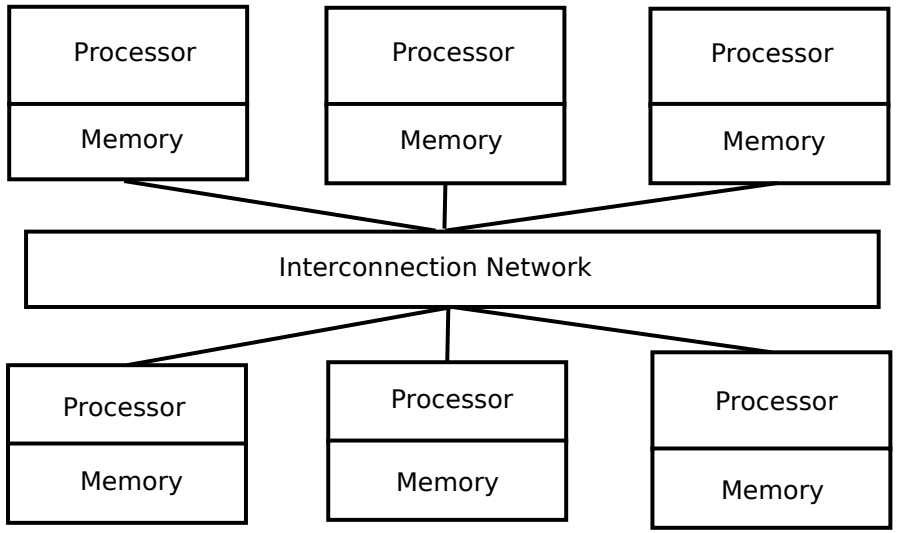
Categorization of parallel hardware based on the number of concurrent instruction and data streams:

- **SISD** (Single instruction Single Data): Single uniprocessor machine that can execute a single instruction and fetch single data stream from memory
- **SIMD** (Single instruction Multiple Data): A computer that can issue the same single instruction to multiple data simultaneously such as GPUs
- **MISD** (Multiple instruction Single Data): Uncommon architecture used for fault tolerance. Several systems acts on the same data and must agree
- **MIMD** (Multiple instruction Multiple Data): Multiple processors executing different instruction on different data. These are distributed systems

- **SPMD** (Single Program Multiple Data): A single program is executed by multi-processor machine of cluster
- **MPMD** (Multiple Program Multiple Data): At least two programs are executed by multi-processor machine of cluster

MPI - Message Passing Interface

- MPI stands for Message Passing Interface
- The MPI model consist of a communication among processes through messages: *Inter-process communication*



- All the processes execute the same program, but each have an ID so you can select which part should be executed (SPMD - Single program Multiple Data)

MPI - Message Passing Interface

- It's a standard API to run application across several computers
- The latest version of the standard is Version 3.1 of 4th June 2015
- http://www.mpi-forum.org/
- Two main open source implementations both supporting version 3 of the standard:
  - MPICH
  - →https://www.mpich.org/
  - Open MPI
  - → http://www.open-mpi.org/
- Written in C, C++ and Fortran
- Used in several science and engineering fields:
  - Atmosphere, Environment
  - Applied-Physics such as condensed matter, fusion etc...
  - Bioscience
  - Molecular sciences
  - Seismology
  - Mechanical engineering
  - Electrical engineering

Tutorial Examples and Exercises

- **NOTE:** This tutorial is based on OpenMPI v1.8.2, you will use OpenMPI v1.6.5
- To check which version you are running issue:

```
$> ompi_info
```

- You can find the example and exercises at this repository:

```
$> https://afalabel@bitbucket.org/afalabel/mpi_tutorial.git
```

- to get the code:

```
$> git clone https://afalabel@bitbucket.org/afalabel/mpi_tutorial.git
```

Hello world! with MPI

- The source code is in :

```
$> mpi_tutorial/1_Hello_world/hello_world.c
```

- You can compile it using the issuing using the Makefile or in general :

```
$> make
```

- it calls *mpicc* which is a wrapper around the gcc compiler
- How to run the code:

```
/usr/bin/mpirun -n 4 ./hello_world
/usr/bin/mpirun -n 4  --host <host 1>,<host 2>,...<host N> ./
      hello_world
```

- For a full list of option of *mpirun* :
https://www.open-mpi.org/doc/v1.10/man1/mpirun.1.php

Hello world! with MPI

- Before running the code let's have a look at the code:

```
1  #include <mpi.h>
2  #include <stdio.h>
3  int main(int argc, char** argv) {
4
5    MPI_Init(NULL, NULL);
6
7    int size;
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9
10   int rank;
11   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12
13   char processor_name[MPI_MAX_PROCESSOR_NAME];
14   int name_len;
15   MPI_Get_processor_name(processor_name, &name_len);
16
17   printf("Hello world from processor %s, rank %d out of %d processors
       \n",
18          processor_name, rank, size);
19   MPI_Finalize();
20 }
```

Hello world with MPI

- Let's describe the code line by line:
- Includes the MPI functions and type declarations

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

- Perform the initial setup of the system. It must be called before any other MPI function

```
MPI_Init(NULL, NULL)
```

- After the initialization every process become a member of a communicator called *MPI_COMM_WORLD*. A **communicator** is an object that provides the environment for processes communication. Every process in a communicator has a unique **rank** that can be retrieved in this way

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

- While the total number of processes can be retrieved

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Hello world with MPI

- To get the name of the processor name (usually the hostname)

```
char processor_name[MPI_MAX_PROCESSOR_NAME];
int name_len;
MPI_Get_processor_name(processor_name, &name_len);
```

- To free the resources allocated for the MPI execution

```
MPI_Finalize()
```

Hello world with MPI

- To test our *hello_world* example we can run 4 processes on 2 machines

```
1 /usr/bin/mpirun -n 4 --host rd-xeon-02,rd-xeon-04  ./hello_world
2 Hello world from processor rd-xeon-02, rank 1 out of 4 processors
3 Hello world from processor rd-xeon-02, rank 0 out of 4 processors
4 Hello world from processor rd-xeon-04, rank 3 out of 4 processors
5 Hello world from processor rd-xeon-04, rank 2 out of 4 processors
```

- As you can see 4 processes are executed (2 per node)
- The number of processes are not in principle equally distruted among the nodes
- The rank doesn't reflect the creation order presented in the output

Parallel Program Design - Foster's Design Methodology

Now it's time to write parallel code. Unfortunately there is not a recipe to do it. In his book: *Designing and Building Parallel Programs* Ian Foster outlined some steps to help doing it

- **Partition**: The process of dividing the computation of the dataset in order to exploit the parallelism
- **Communication**: After the computation has been split in several tasks, it may be required a communication between the processes
- **Agglomeration or aggregation**: Tasks are grouped in order to simplify the programming while profiting of the parallelism
- **Mapping**: The procedure of assigning a task to a processor

MPI provide a lot of functions to ease the communication among parallel tasks. In this tutorial we will see a few of them.

Data Communication

- In MPI, communication consists of sending a copy of the data to another process
- On the sender side communication requires the following thing:
  - Who to send the data (the rank of the receiver process)
  - Data type and size (100 integer, 1 char etc.)
  - the location for the message
- The receiver side need to know
  - It is not required to know who is the sender
  - Data type and size
  - A storage location to put the resulting message

MPI datatypes

MPI defines elementary types, mainly for portability reasons

Name	Type
MPI_CHAR, MPI_SIGNED_CHAR	signed char
MPI_WCHAR	wide char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG	signed long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX,MPI_C_FLOAT_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex

MPI datatypes

Name	Type
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack / MPI_Unpack

- MPI standard allow also for the creation of *Derived Datatypes* (not covered here)

MPI_Send and MPI_Recv

- Performs a **blocking** send:

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

  - buf, count and datatype define the message buffer
  - dest and comm identify the destination process
  - tag is an optional tag for the message
  - When the function exits it means that the message has been sent and the buffer can be reused

- Blocking receive for a message :

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

  - This function waits for a message from source and comm is received into the buffer defined by buf, count and datatype
  - tag is an optional tag for the message
  - status contains additional information such as information, the sender, the actually number of bytes received

Data Communication Example

- Send and Receive example:

```
int main(int argc, char** argv) {
  // Initialize the MPI environment
  MPI_Init(NULL, NULL);
  // Find out rank, size
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int number;
  if (rank == 0) {
    number = 43523;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
     MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
  }
  MPI_Finalize();
}
```

Data Communication Example

- The program sends a number to from the root node to the second node:

```
int number;
if (rank == 0) {
  number = 43523;
  MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
  MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
   MPI_STATUS_IGNORE);
  printf("Process 1 received number %d from process 0\n", number);
}
```

- Selecting the part of the code to run given the rank is typical for MPI programs
- Here the root node (rank$= 0$) is the sender
- The other node (rank$= 1$) will receive the number and print the number
- If you run more than 2 processes the ones with rank> 1 do nothing

Exercise number 1

- Initialize an array of 100 elements
- Perform the sum of the elements sharing the load among 4 processes using MPI_Send and MPI_Recv

Solution to Exercise number 1

- Root processes part

```
1  fraction = (ARRAYSIZE / PROCESSES);
2  if (rank == ROOT){
3    sum = 0;
4    for(i=0; i<ARRAYSIZE; i++) {
5      data[i] =  i * 1.0;
6    }
7    offset = fraction;
8    for (dest=1; dest<PROCESSES; dest++) {
9      MPI_Send(&data[offset], fraction, MPI_FLOAT, dest, 0,
         MPI_COMM_WORLD);
10     printf("Sent %d elements to task %d offset= %d\n",fraction,dest,
         offset);
11     offset = offset + fraction;
12   }
13   offset = 0;
14   for (i=0; i<fraction; i++){
15     mysum += data[i];
16   }
17   for (i=1; i<PROCESSES; i++) {
18     source = i;
19     MPI_Recv(&tasks_sum, 1, MPI_FLOAT, source, 0, MPI_COMM_WORLD, &
         status);
20     sum+=tasks_sum;
21     printf("Received partial sum %f from task %d\n",i,tasks_sum);
22     }
23   printf("Final sum= %f \n",sum+mysum);
24   }
```

Solution to Exercise number 1

- other tasks code

```
1  if (rank > ROOT) {
2
3    source = ROOT;
4    MPI_Recv(&data[offset], fraction, MPI_FLOAT, source, 0,
         MPI_COMM_WORLD, &status);
5
6    for (i=offset; i<(offset+fraction); i++){
7      tasks_sum += data[i];
8    }
9
10   dest = ROOT;
11   MPI_Send(&tasks_sum, 1 , MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
12   }
```

Blocking vs Non-Blocking Communications

- MPI_Send and MPI_Recv are blocking communication routines
  - Return after completion
  - When they return their buffer can be reused
- MPI_Isend and MPI_Irecv are the non-blocking version
  - Return immediately, the completion must be checked separately
  - Computation and communication can be overlapped

```
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int
     dest, int tag,
              MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source
     ,
 int tag, MPI_Comm comm, MPI_Request *request)
```

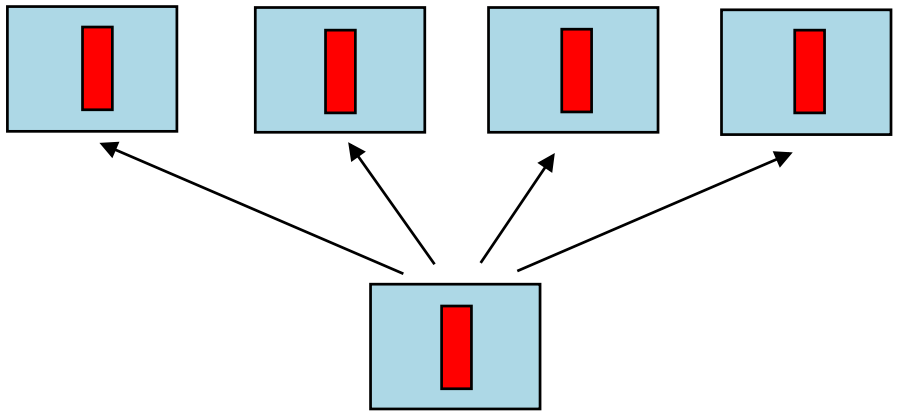
- In addition to the blocking calls you have an request object as a parameter

Collective Communication Routines

- Collective communication involve all the processes within the scope of a communicator
- All processes are by default in the MPI_COMM_WORLD communicator, but groups of communicators can be defined
- Collective operations usually require:
  - Synchronization: e.g. MPI_Barrier
  - Data Movement: e.g. MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather, MPI_Alltoall
  - Collective Computation: e.g. MPI_Reduce

Broadcasting messages

- Broadcast a message from the process with rank root to all other processes of the communicator including the root process




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

Broadcasting messages


```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <mpi.h>
4  #include <math.h>
5
6  #define BUFFERSIZE 4
7
8  int main(int argc, char** argv) {
9      int i,size,rank;
10     int buffer[BUFFERSIZE];
11     MPI_Init(&argc,&argv);
12     MPI_Comm_size(MPI_COMM_WORLD,&size);
13     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
14
15     char processor_name[MPI_MAX_PROCESSOR_NAME];
16     int name_len;
17     MPI_Get_processor_name(processor_name, &name_len);
18     //root node
19     if(rank == 0){
20       printf("\nInitializing array on node %s with rank %d\n",
       processor_name,rank);
21       for(i=0;i<BUFFERSIZE;i++)
22         buffer[i]=i;
23     }
```

Broadcasting messages


```
24      MPI_Bcast(buffer,BUFFERSIZE,MPI_INT,0,MPI_COMM_WORLD);
25      printf("\nReceived buffer on node %s with rank %d\n",processor_name,
          rank);
26      for(i=0;i<BUFFERSIZE;i++)
27        printf("%d ",buffer[i]);
28      printf("\n");
29      MPI_Finalize();
30  }
```

Exercise 2

- Generate an array of size N containing random values and create an histogram with $M < N$ bins

Solution to exercise 2


```
31 #include <stdio.h>
32 #include <stdlib.h>
33 #include <mpi.h>
34 #include <math.h>
35
36 #define BUFFERSIZE 1000
37 #define MAX 100
38 #define MIN 0
39 #define BINS 10
40
41 int main(int argc, char** argv) {
42
43     int i,counter=0;
44     int size,rank;
45     float buffer[BUFFERSIZE];
46     float bin_size=(float)(MAX-MIN)/BINS;
47
48     MPI_Init(&argc,&argv);
49     MPI_Comm_size(MPI_COMM_WORLD,&size);
50     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
51     if (size==BINS){
52     char processor_name[MPI_MAX_PROCESSOR_NAME];
53     int name_len;
54     MPI_Get_processor_name(processor_name, &name_len);
55     if(rank == 0){
56       printf("\nInitializing array on node %s with rank %d\n",
       processor_name,rank);
57       for(i=0;i<BUFFERSIZE;i++){
58         buffer[i]= rand() % (MAX - MIN - 1) + MIN;
59       }
60     }
```

Solution to exercise 2

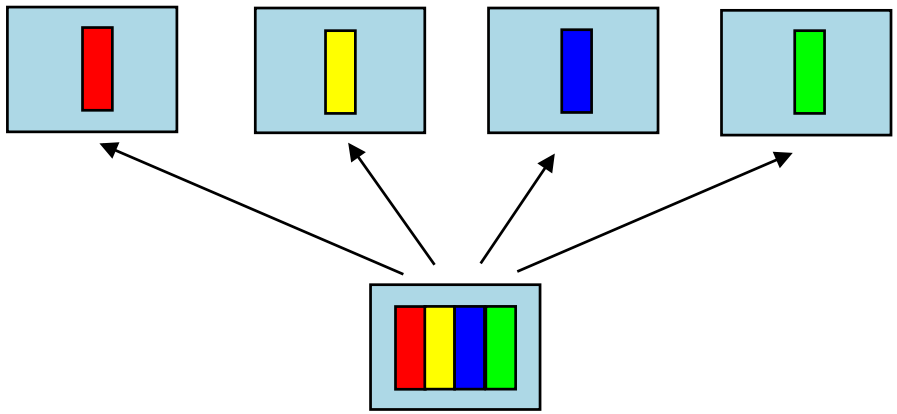

```
61      MPI_Bcast(buffer,BUFFERSIZE,MPI_FLOAT,0,MPI_COMM_WORLD);
62      printf("\nReceived buffer on node  %s with rank %d\n",processor_name,
          rank);
63
64      for(i=0;i<BUFFERSIZE;i++){
65        if (buffer[i]>=rank*bin_size && buffer[i]<(rank+1)*bin_size)
66          counter++;
67      }
68      printf("\n Bin %d has %d entries\n",rank,counter);
69      }
70      MPI_Finalize();
71  }
```

Solution to exercise 2 - Output


```
> /usr/bin/mpirun -n 10 --host rd-xeon-02,rd-xeon-04 ./broadcast_histo
Initializing array on node rd-xeon-02.cnaf.infn.it with rank 0
Received buffer on node rd-xeon-02.cnaf.infn.it with rank 0
 Bin 0 has 98 entries
Received buffer on node rd-xeon-02.cnaf.infn.it with rank 1
 Bin 1 has 95 entries
Received buffer on node rd-xeon-02.cnaf.infn.it with rank 2
 Bin 2 has 90 entries
Received buffer on node rd-xeon-02.cnaf.infn.it with rank 3
 Bin 3 has 95 entries
Received buffer on node rd-xeon-02.cnaf.infn.it with rank 4
 Bin 4 has 103 entries
Received buffer on node rd-xeon-04.cnaf.infn.it with rank 6
 Bin 6 has 115 entries
Received buffer on node rd-xeon-04.cnaf.infn.it with rank 7
 Bin 7 has 86 entries
Received buffer on node rd-xeon-04.cnaf.infn.it with rank 8
 Bin 8 has 106 entries
Received buffer on node rd-xeon-04.cnaf.infn.it with rank 9
 Bin 9 has 87 entries
Received buffer on node rd-xeon-04.cnaf.infn.it with rank 5
 Bin 5 has 125 entries
```

Scatter messages

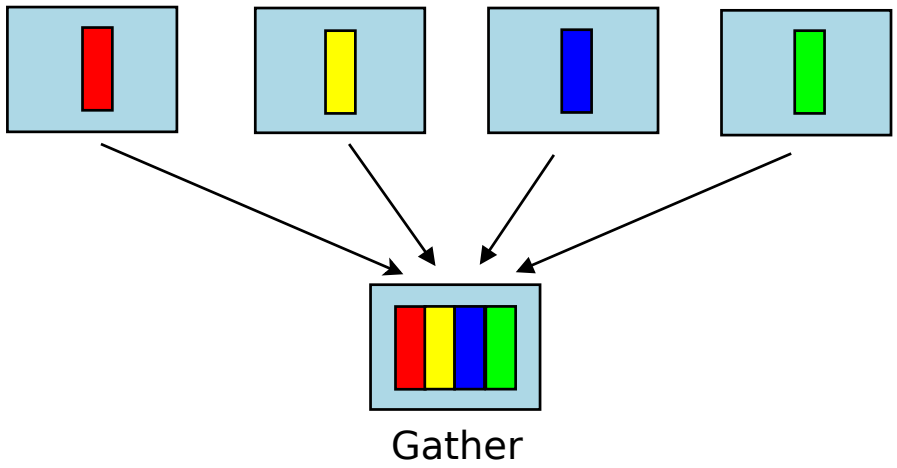
- When you need to send a fraction of data to each of your nodes you can use *MPI_Scatter*




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

Gather messages

- When you need to retrieve a portion of data from different processes *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)
```

Scatter Gather example


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

#define ROOT 0
#define DATA_SIZE 4

int main(int argc,char *argv[]){
  int nodes,rank;
  int *partial,*send_data,*recv_data;
  int size,mysize,i,k,j,partial_sum,total;


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

  partial=(int*)malloc(DATA_SIZE*sizeof(int));
  //create the data to be sent
  if(rank == ROOT ){
    size=DATA_SIZE*nodes;
    send_data=(int*)malloc(size*sizeof(int));
    recv_data=(int*)malloc(nodes*sizeof(int));
    for(i=0;i<size;i++)
      send_data[i]=i;
  }
  //send different data to each process
  MPI_Scatter(send_data, DATA_SIZE, MPI_INT,
              partial, DATA_SIZE, MPI_INT,
             ROOT, MPI_COMM_WORLD);
```

Scatter Gather example


```
  partial_sum=0;
  for(i=0;i<DATA_SIZE;i++)
    partial_sum=partial_sum + partial[i];

  printf("rank= %d total= %d\n",rank,partial_sum);

  //send the local sums back to the root
  MPI_Gather(&partial_sum,    1,  MPI_INT,
             recv_data,       1,  MPI_INT,
      ROOT, MPI_COMM_WORLD);

  if(rank == ROOT){
    total=0;
    for(i=0;i<nodes;i++)
      total=total+recv_data[i];
    printf("Total is = %d \n",total);
  }
  MPI_Finalize();
}
```

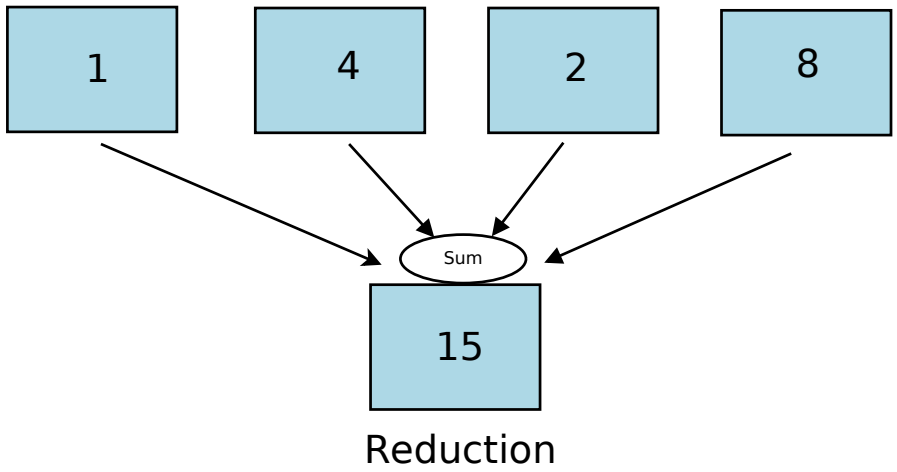
Scatter Gather example output

- For example running the code for two processes

```
/usr/bin/mpirun -n 2 --host rd-xeon-02,rd-xeon-04 ./scatter_gather
rank= 0 total= 6
 Total is = 28
rank= 1 total= 22
```

Reduce

- When you need to retrieve a portion of data from different processes and you want to manipulate them *MPI_Reduce*




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

MPI Reduction Operations

MPI Operation		C Data Types
MPI_MAX	maximum	integer, float
MPI_MIN	minimum	integer, float
MPI_SUM	sum	integer, float
MPI_PROD	product	integer, float
MPI_LAND	logical AND	integer
MPI_BAND	bit-wise AND	integer, MPI_BYTE
MPI_LOR	logical OR	integer
MPI_BOR	bit-wise OR	integer, MPI_BYTE
MPI_LXOR	logical XOR	integer
MPI_BXOR	bit-wise XOR	integer, MPI_BYTE
MPI_MAXLOC	max value and location	float, double and long double
MPI_MINLOC	min value and location	float, double and long double

Reduce example

- The problem is the same as the scatter gather example

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

#define ROOT 0
#define DATA_SIZE 4

int main(int argc,char *argv[]){
  int nodes,rank;
  int *partial,*send_data,*recv_data;
  int size,mysize,i,k,j,partial_sum,total;

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

  partial=(int*)malloc(DATA_SIZE*sizeof(int));
  //create the data to be sent on the root
  if(rank == ROOT ){
    size=DATA_SIZE*nodes;
    send_data=(int*)malloc(size*sizeof(int));
    recv_data=(int*)malloc(nodes*sizeof(int));
    for(i=0;i<size;i++)
      send_data[i]=i;
  }

  MPI_Scatter(send_data, DATA_SIZE, MPI_INT,
              partial, DATA_SIZE, MPI_INT,
              ROOT, MPI_COMM_WORLD);
```

Reduce example

- The problem is the same as the scatter gather example

```
  partial_sum=0;
  for(i=0;i<DATA_SIZE;i++)
    partial_sum=partial_sum + partial[i];

  printf("rank= %d total= %d\n",rank,partial_sum);

  MPI_Reduce(&partial_sum, &total, 1, MPI_INT,
             MPI_SUM, ROOT, MPI_COMM_WORLD);

  if(rank == ROOT){
    printf("Total is = %d \n",total);
  }
  MPI_Finalize();
}
```

- It produces the same output:

```
/usr/bin/mpirun -n 2 --host rd-xeon-02,rd-xeon-04 ./reduce
rank= 0 total= 6
 Total is = 28
rank= 1 total= 22
```

Exercise 3

- Compute an approximate value of π
- Hint: Use the Taylor series expansion for the $arctan$

$$arctan(\frac{x}{a}) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \frac{x}{a}^{2n+1}$$

- Note: This is not the fastest way to approximate π

Solution to exercise 3


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

#define ROOT 0

double taylor(const int i, const double x, const double a){
  if (i>=-1 && a !=0){
    int sign = pow(-1,i);
    double num = pow(x,2*i+1);
    double den=  a*(2*i +1);
    return(sign*num/den);
  }
  else{
    printf("\nCheck your parameters!\n");
    return (-1);
  }
}


int main(int argc,char *argv[]){
  int nodes,rank;
  double* partial;
  double res;
  double total=0;

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

Solution to exercise 3


```
1
2     res = taylor(rank,1,1);
3     printf("rank= %d total= %f\n",rank,res);
4
5     MPI_Reduce(&res, &total, 1, MPI_DOUBLE,
6         MPI_SUM, ROOT, MPI_COMM_WORLD);
7
8     if(rank == ROOT){
9       printf("Total is = %f \n",4*total);
10    }
11    MPI_Finalize();
12  }
```

References

- M.J. Quinn. Parallel Programming in C with MPI and OpenMP. McGraw-Hill Higher Education, 2004
- https://www.mpich.org
- http://www.open-mpi.org/