deep se
dependable evolvable pervasive software engineering group

Message Passing Interface

Eugenio Gianniti & Danilo Ardagna

Politecnico di Milano
name.lastname@polimi.it

POLITECNICO DI MILANO

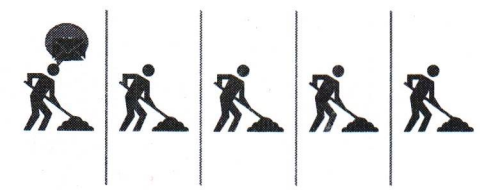
Content

- Collective Communication
- Reference

COLLECTIVE COMMUNICATION

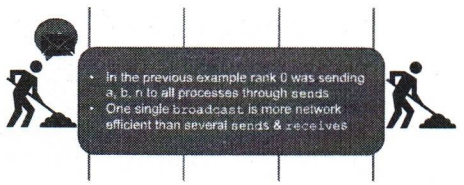
Or how to throw flyers from a plane

MPI in Pictures – Send & Receive



MPI Library Runtime

MPI in Pictures – Collective (broadcast)



- In the previous example rank 0 was sending a, b, n to all processes through sends
- One single broadcast is more network efficient than several sends & receives

MPI Lib[illegible]Runtime

— We want rank 0 to send a single message and everyone to receive it (we don't want rank 0 to send individual messages to everyone)

Collective Routines

- Involve all the processes in a communicator
- Only blocking routines
- Transmit only predefined MPI data types
- Cannot use tags to identify messages
- Attention! Be sure that every process in the communicator calls the collective function to avoid deadlocks

If every process is running a broadcast but one, everyone is blocked

Broadcast

- Delivers an exact copy of the data in `buffer` from `root` to all the processes in `comm`

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

universal pointer (void pointer): from the sender side the buffer will contain the data, from the receiver side the buffer will be where the data will be stored

number of elements of the specified type

source (who owns the data)

communicator

	buffer	
Rank	Before	After
0	A	A
1	?	A
2	?	A
3	?	A

Trapezoidal Rule — Broadcasting Inputs

```
void get_input (double & a, double & b, unsigned & n)
{
  int rank;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  if (rank == 0)
    std::cin >> a >> b >> n;

  MPI_Bcast (&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast (&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast (&n, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);
}
```

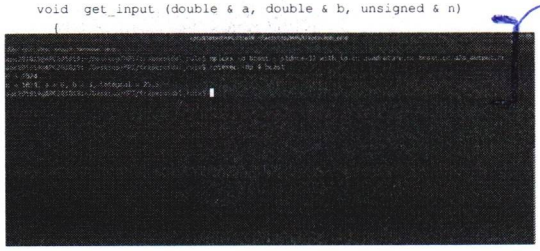
Only rank 0 can take inputs

now rank 0 sends the inputs to everyone with one command (no need for a for-loop)

DEMO

Trapezoidal Rule — Broadcasting Inputs

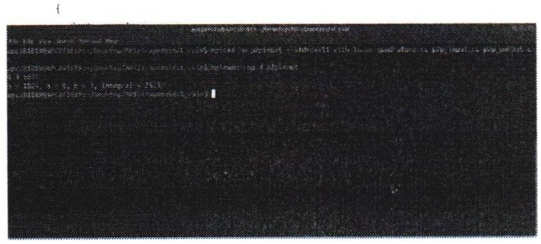
```
void get_input (double & a, double & b, unsigned & n)
{
```



```
$mpicxx -O bcast --std=c++11
  with_io.cc quadrature.cc
  bcast.cc p2p_output.cc
$mpiexec -np 4 bcast
0 3 1024
n=1024, a=0, b=3,
integral = 25.5
```


Trapezoidal Rule — Broadcasting Inputs

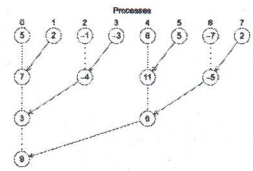
```
void get_input (double & a, double & b, unsigned & n)
{
```



Trapezoidal Rule - Splitting Sum Work

- We are not computing the "global sum" efficiently
- If we hire *N* workers to build a house, we might feel that we weren't getting our money's worth if *N* – *1* of the workers told the first what to do and then the *N* – *1* collected their pay and went home
  - But this is what we're doing in our global sum:
    - Each process with rank greater than 0 is "telling process 0 what to do" and then quitting
    - Process 0 is doing nearly all the work in computing the global sum

Trapezoidal Rule - Splitting Sum Work



- This is what is done for us by `MPI_Reduce`
- There are also some communication optimizations behind the curtain, but we don't enter the details

Reduce

- Applies `op` to portions of data in `sendbuf` from all the processes in `comm`, storing the result in `recvbuf` on `dest`

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

universal pointer to something that is on read only (because it's const)

number of elements involved (of type "datatype")

receiving buffer

communicator

destination process

specifies the operation

Example of Reduce

```
double local_partial = // some partial sum ;
double total;
MPI_Reduce (&local_partial, &total, 1,
            MPI_DOUBLE, MPI_SUM,
            0, MPI_COMM_WORLD);
```

Rank	local_partial	total
0	1	10
1	2	N/A
2	3	N/A
3	4	N/A

the initial data will be in the address of "local_partial", the destination will be "total", we're sending 1 double (we specify what we send locally) and the final receiver will be rank 0. Moreover we specify that we're doing the sum.

Note: because of the implementation all processes will run the code and every process will have "total", however only in rank 0 it'll be updated

Reduce Operators

- The standard provides several ready to use operators for reduce routines

MPI_MAX	MPI_LAND	MPI_LXOR
MPI_MIN	MPI_BAND	MPI_BXOR
MPI_SUM	MPI_LOR	MPI_MAXLOC
MPI_PROD	MPI_BOR	MPI_MINLOC

We are interested in only on these ones

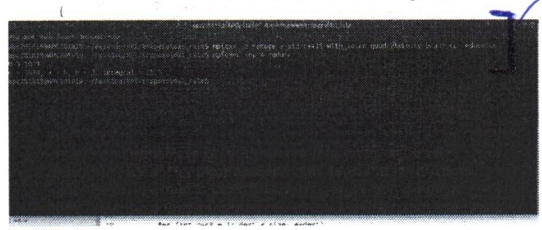
Trapezoidal Rule - Splitting Sum Work

```
void sum_and_print (double local_integral,
                    std::ostream & out,
                    double a, double b, unsigned n)
{
  int rank;
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  double total (0.);
  MPI_Reduce (&local_integral, &total, 1, MPI_DOUBLE,
              MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank == 0)
    {
      out << "n = " << n
          << ", a = " << a << ", b = " << b
          << ", integral = " << total << std::endl;
    }
}
```

DEMO

Trapezoidal Rule - Splitting Sum Work

```
void sum_and_print (double local_integral,
                    std::ostream & out,
                    double a, double b, unsigned n)
{
```



```
$mpicxx -o reduce --std=c++11
  with_io.cc quadrature.cc
  bcast.cc reduce.cc
$mpiexec -np 4 reduce
0 3 1024
n=1024, a=0, b=3,
integral = 25.5
```

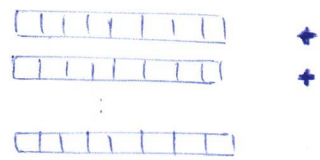

More on `MPI_Reduce`

- By using a count argument greater than 1, `MPI_Reduce` can operate on arrays instead of scalars
- The following code could thus be used to add a collection of *N*-dimensional vectors:

```
std::vector<double> local_x (N), sum (N);
/* partial computation on local_x */
MPI_Reduce (local_x.data (), sum.data (),
            N, MPI_DOUBLE, MPI_SUM,
            0, MPI_COMM_WORLD);
```

Returns the pointer to the vector elements

local_x :



Sum of all : sum :

With this code we perform the sums that we need and then we update the vector "sum" only in the rank 0 process (destination process).
What if we need the vector "sum" to be updated in all the processes?

we need to pass a pointer. In vector structure there is a method: data(). This method returns the pointer to the underlying array storing the N doubles (sequentially)

`MPI_Allreduce`

- In our trapezoidal rule program, we just print the result, so it's perfectly natural for only one process to get the result of the global sum
- In some situations (e.g., you are implementing a **parallel function**) ***all* of the processes** might need the result of a global sum in order to complete some larger computation
- MPI provides a variant of `MPI_Reduce` that will store the result on all the processes in the communicator

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

Same structure as MPI_Reduce but without the destination process → the final result will be available on all processes

Collective Communication "In Place"

- Collective communication routines generally use both a send and a receive buffer
  - When dealing with lots of data this implies that you are also occupying a lot of memory
- MPI provides the special placeholder `MPI_IN_PLACE` to enable the use of a single buffer for both input and output

```
double minimum (local_min);
MPI_Allreduce (MPI_IN_PLACE, &minimum, 1,
               MPI_DOUBLE, MPI_MIN,
               MPI_COMM_WORLD);
```

Suppose all processes have computed their minimum (locally) and they stored it in the local variable "minimum". After this call, "minimum" will store the global minimum

Collective Communication "In Place"

- It might be tempting to call `MPI_Reduce` using the same buffer for both input and output

```
MPI_Reduce(&x , &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

we write this if we have x in every process and we want to store the sum of all x's in x. However we cannot write it like this! That's an undefined behavior! We need to use "MPI_IN_PLACE"

- This call is illegal in MPI (undefined behavior), its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash, it might even produce a correct result
- It's illegal because it involves aliasing of an output argument
  - Two arguments are aliased if they refer to the same block of memory
  - MPI prohibits aliasing of arguments if one of them is an output or input/output argument

Collective Communication "In Place"

- It might be tempting to call `MPI_Reduce` using the same buffer for both input and output

```
MPI_Reduce(&x , &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

Use MPI_IN_PLACE instead:

```
MPI_Reduce(MPI_IN_PLACE, &x, 1, MPI_DOUBLE, MPI_SUM,
           0, comm);
```

- It's illegal because it involves aliasing of an output argument
  - Two arguments are aliased if they refer to the same block of memory
  - MPI prohibits aliasing of arguments if one of them is an output or input/output argument

Working With Vectors

Scatter & Gather

Data Distributions

- Suppose we want to write a function that computes a vector sum:

we want to do it in parallel

$\mathbf{x} + \mathbf{y} = (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1})$
$= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) = (z_0, z_1, \ldots, z_{n-1}) = \mathbf{z}$

```
// Assumption: x and y with the same size
vector<double> sum (const vector<double> & x,
                    const vector<double> & y)
{
      vector<double> z(x.size ());
      for (std::size_t i = 0; i < x.size (); ++i)
            z[i] = x[i] + y[i];
      return z;
}
```


Data Distributions

- If the number of components is *n* and we have comm_sz cores or processes, let's assume that *n* evenly divides comm_sz and define local_n = *n* / comm_sz
- We can parallelize the sum by assigning blocks of local_n consecutive components to each process

Process	Block			
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11

elements of a vector of size 12 (the blocks are consecutive)

- This is the so-called **block partition**

Data Distributions

- An alternative to a block partition is a **cyclic partition**, where components are assigned in a round robin fashion

Process	Cyclic			
0	0	3	6	9
1	1	4	7	10
2	2	5	8	11

```
for (size_t i=rank; i< v.size(); i+=size){
    // do something on v[i]
}
```


Data Distributions

- Given the partitioning scheme, each process simply adds its assigned components

- **Block partitioning** is used when **data source** is available on a **single process**

- **Cyclic partitioning** is used when data source is already available across all processes

With cyclic partitioning we don't have to worry that n/#cores is a perfect division or not

Data Distributions - Block Partitioning

- Each process will have `local_n` components of the vectors, and, in order to save on storage, we can just store these on each process as a vector of `local_n` elements

```
vector<double> parallel_sum (const vector<double> &
local_x, const vector<double> & local_y)
{     vector<double> local_z(local_x.size ());
      for (std::size_t i = 0; i < local_x.size (); ++i)
            local_z[i] = local_x[i] + local_y[i];
      return local_z;
}
```


Scatter - Block Partitioning

- To implement our vector addition function, we need to:
  - read the dimension of the vectors
  - then read in the vectors **x** and **y**
- Process 0 can prompt the user, read in the dimension, and broadcast the value to the other processes
- For the vectors:
  - process 0 reads them (no other option)
  - then process 0 sends the needed components to each of the other processes
- This is exactly what `MPI_Scatter` implements (**block partition** scheme)

Scatter

- Sends a portion of the data in `sendbuf` from `root` to all the processes in `comm`, storing it in `recvbuf`
- `sendbuf` holds `sendcount * size` elements (in other words, `sendcount` is the number of elements sent to individual processes)

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

send to individual processes (the number of the elements that we send overall is sendcount * size)

send buffer

number of elements that we send

type of the data we're sending

receiving buffer

how many elements need to be received

source of the data and communicator

receiving type

Example of Scatter – rank 0 is root

Rank	sendbuf	recvbuf
0	ABCDEFGH	AB
1	N/A	CD
2	N/A	EF
3	N/A	GH

read_vector

```
std::vector<double>
read_vector (unsigned n,
             std::string const & name,
             MPI_Comm const & comm)
{
  int rank, size;
  MPI_Comm_rank (comm, &rank);
  MPI_Comm_size (comm, &size);
  const unsigned local_n = n / size;
  std::vector<double> result (local_n);
```

Extremely important to place this here

size of the vector

communicator

we prepare for each process a vector "result" which stores local_n zeros

read_vector

```
  if (rank == 0)
    {
      std::vector<double> input (n);
      std::cout << "Enter " << name << "\n";
      for (double & e : input)
        std::cin >> e;
      MPI_Scatter (input.data (), local_n,
                   MPI_DOUBLE,
                   result.data (), local_n,
                   MPI_DOUBLE, 0, comm);
    }
```

size of the full vector that we want to use (initialized to n (as size), not local_n)

note that the vector "input" is available only in this part (scope)

we read the input data and we scatter it. We send local_n MPI_DOUBLEs, we'll store the data in result.data() (how many? local_n, of what? MPI_DOUBLE) The source is rank 0.

read_vector

```
  else
    {
      /* Here are only receiving ranks,
       * no need for the send buffer. */
      MPI_Scatter (nullptr, local_n,
                   MPI_DOUBLE,
                   result.data (), local_n,
                   MPI_DOUBLE, 0, comm);
    }
  return result;
}
```


Gather

dual of the Scatter

It composes small pieces into one piece

- Joins portions of data in `sendbuf` from all the processes in `comm` to `root`, storing them all in `recvbuf`
- `recvcount` values received from each process

```
int MPI_Gather (const void *sendbuf,
```

- `MPIAll_gather` provides the destination buffer to all processes
- Same parameters as `MPI_Gather` only root is missing

Gather

- Joins portions of data in `sendbuf` from all the processes in `comm` to `root`, storing them all in `recvbuf`
- `recvcount` values received from each process

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

send buffer
number of elements of the send buffer
type of elements of the send buffer
receive buffer
number of elements that needs to be received
type of received elements
! root = destination
root and communicator

Example of Gather – rank 0 is root

Rank	sendbuf	recvbuf
0	AB	ABCDEFGH
1	CD	N/A
2	EF	N/A
3	GH	N/A

print_vector

```
void
print_vector (std::vector<double> const &
              local_v, unsigned n,
              std::string const & title,
              MPI_Comm const & comm)
{
  int rank, size;
  MPI_Comm_rank (comm, &rank);
  MPI_Comm_size (comm, &size);
  const unsigned local_n = local_v.size ();
```

(local value, total size, title, communicator)

print_vector

```
  if (rank > 0)
    {
      /* Here are only sending ranks,
       * no need for the receive buffer. */
      MPI_Gather (local_v.data (), local_n,
                  MPI_DOUBLE,
                  nullptr, local_n,
                  MPI_DOUBLE, 0, comm);
    }
```

Where is the data? In local_v. How many? local_n. What's the type? MPI_DOUBLE. Do we have the destination? NO, we don't → nullptr. The destination needs to receive local_n MPI_DOUBLE. What is the process of the destination? 0.

print_vector

```
  else
    {
      std::vector<double> global (n);
      MPI_Gather (local_v.data (), local_n,
                  MPI_DOUBLE,
                  global.data (), local_n,
                  MPI_DOUBLE, 0, comm);
      std::cout << title << "\n";
      for (double value : global)
        std::cout << value << " ";
      std::cout << std::endl;
    }
```

This will be the destination value. We have to prepare it.

Final Remarks on Collective Communications

- All the processes in the communicator must call the same collective function
  - If a program attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process it is erroneous and probably will hang or crash.
- The arguments passed by each process to an MPI collective communication must be "compatible"
  - If one process passes in 0 as the dest process and another passes in 1, then the outcome of a call to, e.g., `MPI_Reduce` is erroneous and the program is likely to hang or crash
- The `recvbuf` argument is only used on `dest` process. However, all of the processes still need to pass in an actual argument corresponding to `recvbuf`, even if it's just `nullptr`

Final Remarks on Collective Communications

- Point-to-point communications are matched based on tags and communicators. Collective communications don't use tags, so they're matched solely based on the communicator and the order in which they're called

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2;	a = 1; c = 2;	a = 1; c = 2;
1	MPI_Reduce(&a,&b,..., 0,comm);	MPI_Reduce(&c,&d,..., 0,comm);	MPI_Reduce(&a,&b,..., 0,comm);
2	MPI_Reduce(&c,&d,..., 0,comm);	MPI_Reduce(&a,&b,..., 0,comm)	MPI_Reduce(&c,&d,..., 0,comm)

Final Remarks on Collective Communications

- Point-to-point communications are matched based on tags and communicators. Collective communications don't use tags, so they're matched solely based on the communicator and the order in which they're called

Time	Process 0	Process 1	Process 2
0	a = 1; c = 2;	a = 1; c = 2;	a = 1; c = 2;
1	MPI_Reduce(&a,&b,..., 0,comm);	MPI_Reduce(&c,&d,..., 0,comm);	MPI_Reduce(&a,&b,..., 0,comm);
2	MPI_Reduce(&c,&d,..., 0,comm);	MPI_Reduce(&a,&b,..., 0,comm)	MPI_Reduce(&c,&d,..., 0,comm)

b = 1 + 2 + 1 = 4
d = 2 + 1 + 2 = 5

Reference

- The Open MPI documentation: https://www.open-mpi.org/doc/current/
- The MPI tutorial by the Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/mpi/
- Pacheco Chapter 3