dependable evolvable pervasive software engineering group

# Message Passing Interface

Eugenio Gianniti & Danilo Ardagna

Politecnico di Milano

name.lastname@polimi.it

07/12/2024

POLITECNICO DI MILANO

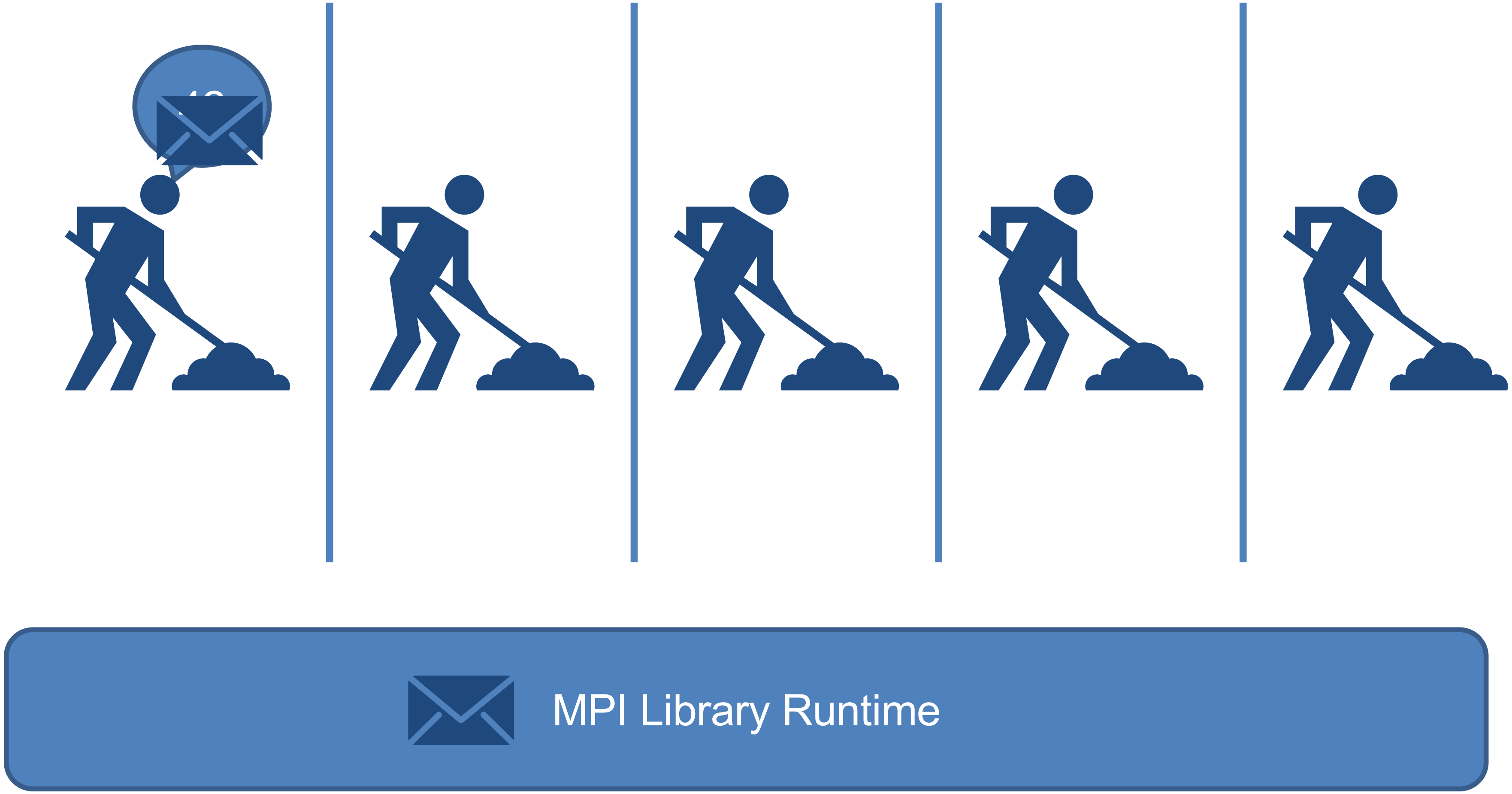
# Content

- Collective Communication
- Reference

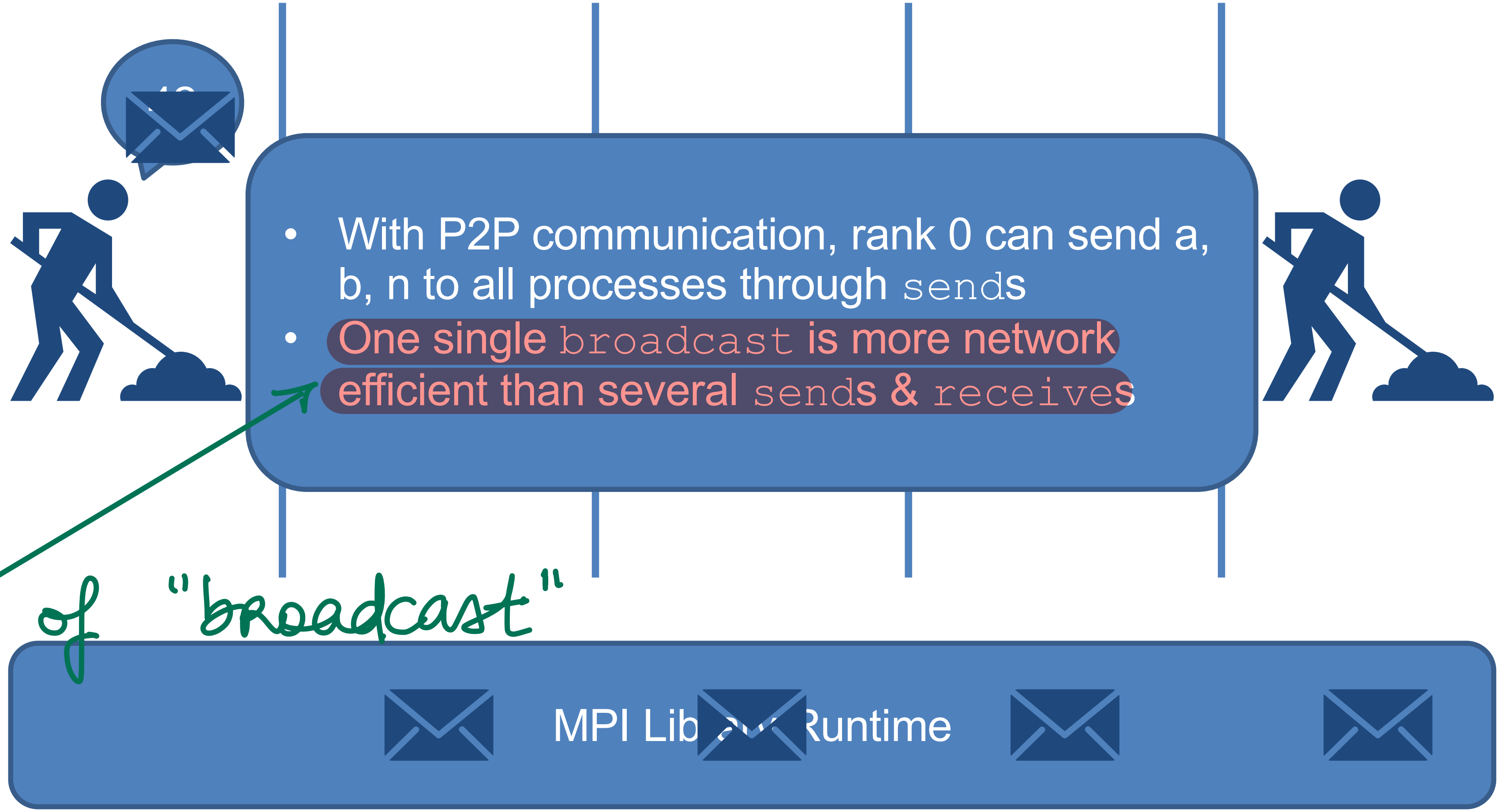
# COLLECTIVE COMMUNICATION

Or how to throw flyers from a plane

# MPI in Pictures – Send & Receive



# MPI in Pictures – Collective (broadcast)



- With P2P communication, rank 0 can send a, b, n to all processes through `send`s
- One single `broadcast` is more network efficient than several `send`s & `receive`s

Interest of "broadcast"

# 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 **same collective function** to avoid deadlocks

# Broadcast → WHAT DOES IT DO ?

- 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)
```

|  | `buffer` |  |
|---|---|---|
| **Rank** | **Before** | **After** |
| 0 | A | A |
| 1 | ? | A |
| 2 | ? | A |
| 3 | ? | A |

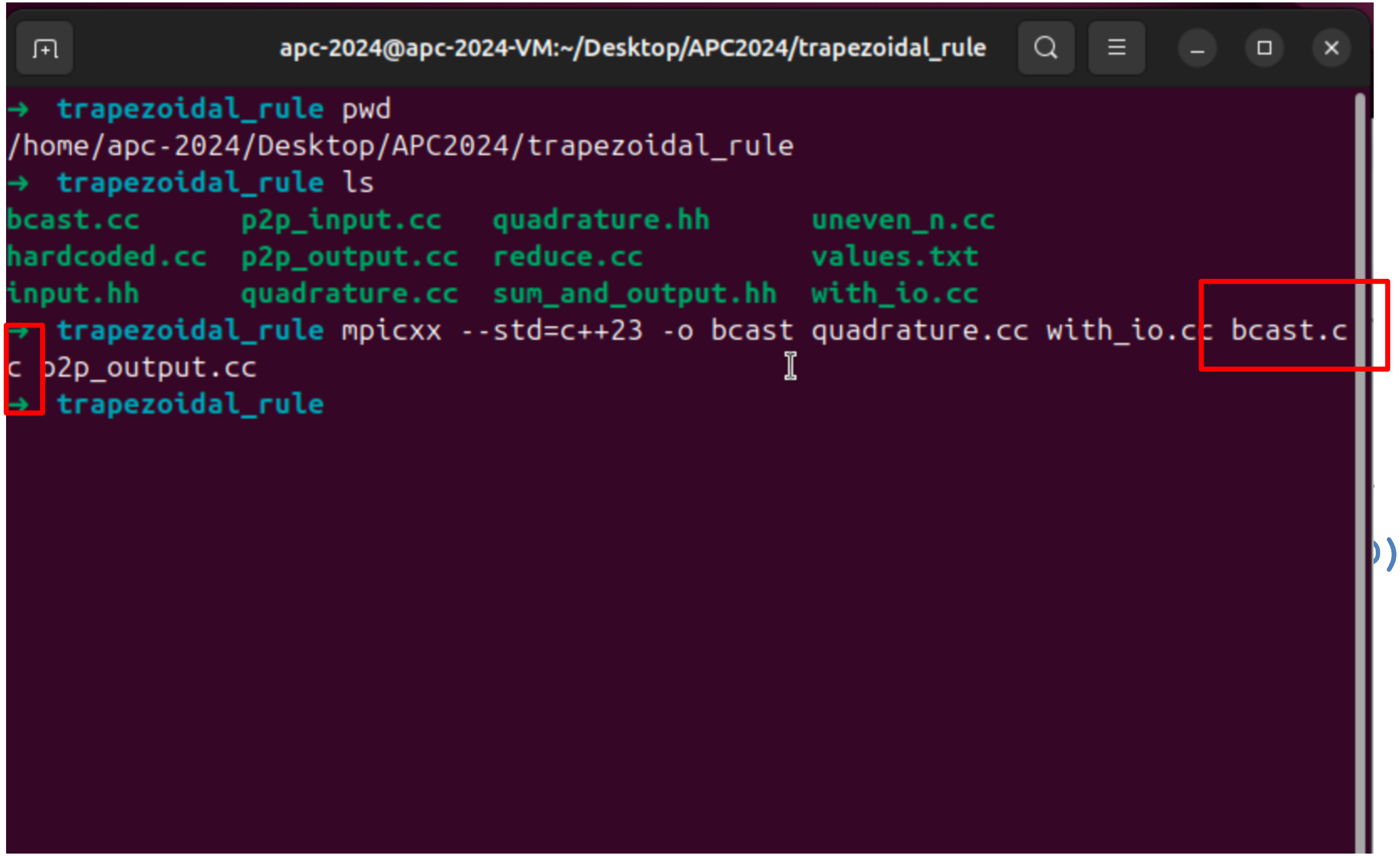
# Trapezoidal Rule — Broadcasting Inputs

```
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);
}
```

DEMO

# Trapezoidal Rule — Broadcasting Inputs

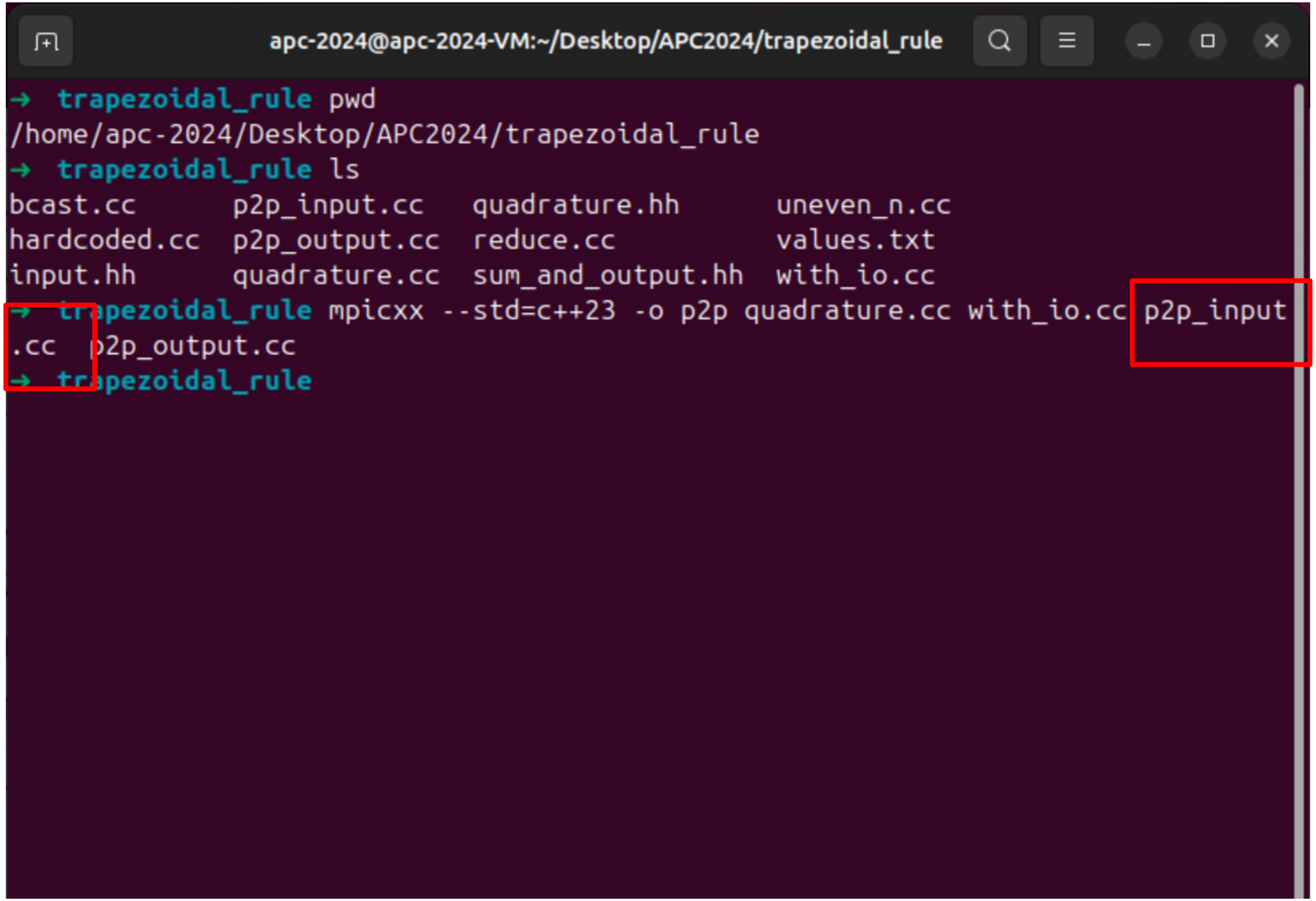
```
void  get_input (double & a, double & b, unsigned & n)
```



```
➜  trapezoidal_rule pwd
/home/apc-2024/Desktop/APC2024/trapezoidal_rule
➜  trapezoidal_rule ls
bcast.cc       p2p_input.cc    quadrature.hh       uneven_n.cc
hardcoded.cc   p2p_output.cc   reduce.cc           values.txt
input.hh       quadrature.cc   sum_and_output.hh   with_io.cc
➜  trapezoidal_rule mpicxx --std=c++23 -o bcast quadrature.cc with_io.cc bcast.c
c p2p_output.cc
➜  trapezoidal_rule
```

) ;

# Trapezoidal Rule — Based on P2P



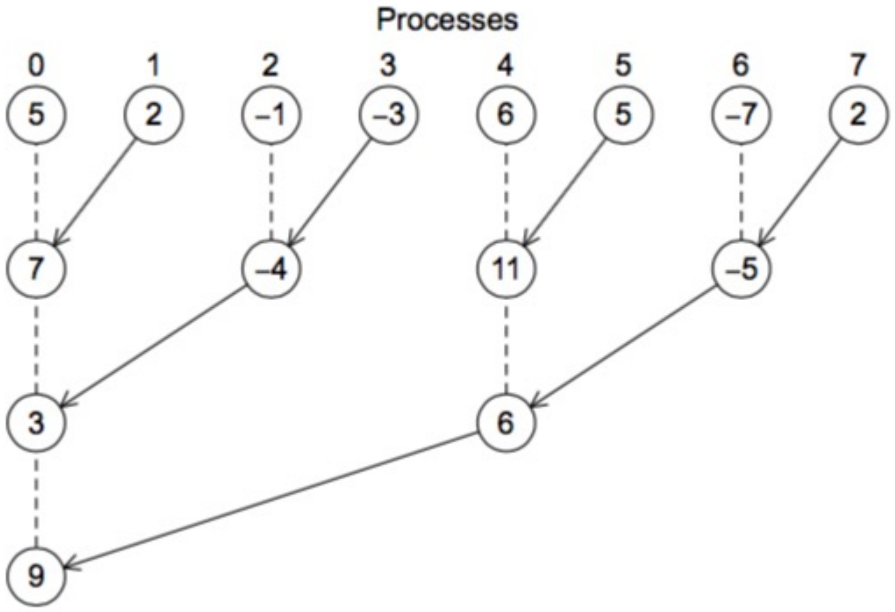
```
→  trapezoidal_rule pwd
/home/apc-2024/Desktop/APC2024/trapezoidal_rule
→  trapezoidal_rule ls
bcast.cc       p2p_input.cc   quadrature.hh      uneven_n.cc
hardcoded.cc   p2p_output.cc  reduce.cc          values.txt
input.hh       quadrature.cc  sum_and_output.hh  with_io.cc
→  trapezoidal_rule mpicxx --std=c++23 -o p2p quadrature.cc with_io.cc p2p_input
.cc  p2p_output.cc
→  trapezoidal_rule
```

# 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` (cf next slide)
- There are also some communication optimizations behind the curtain, but we don't enter the details

# Reduce

SYNTAX

- 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)
```

cf example @ next slide.

# 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 |

# Reduce Operators

MPI_Op

- 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

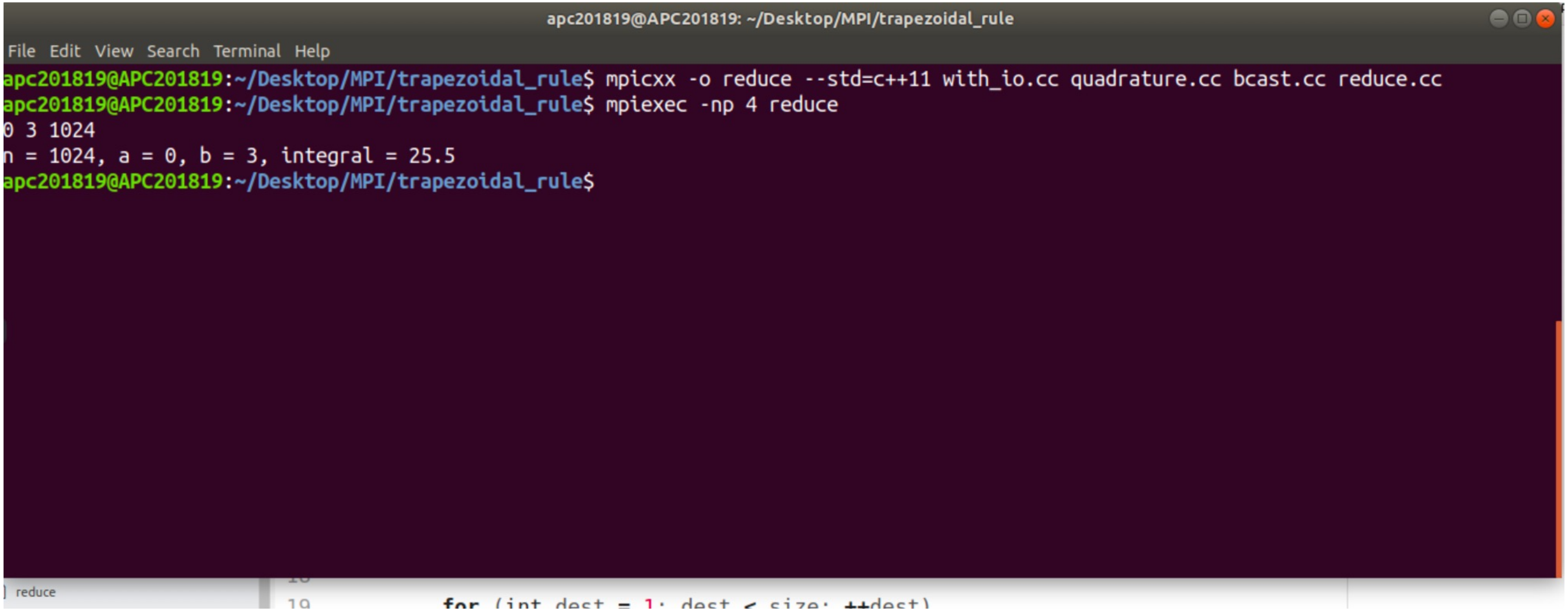
# Trapezoidal Rule - Splitting Sum Work

```
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)
  {
```



apc201819@APC201819: ~/Desktop/MPI/trapezoidal_rule
File Edit View Search Terminal Help
reduce

```
apc201819@APC201819:~/Desktop/MPI/trapezoidal_rule$ mpicxx -o reduce --std=c++11 with_io.cc quadrature.cc bcast.cc reduce.cc
apc201819@APC201819:~/Desktop/MPI/trapezoidal_rule$ mpiexec -np 4 reduce
0 3 1024
n = 1024, a = 0, b = 3, integral = 25.5
apc201819@APC201819:~/Desktop/MPI/trapezoidal_rule$
```

```
        for (int dest = 1; dest < size; ++dest)
```

# More on MPI_Reduce

about "count"

- 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

# MPI_Allreduce

context ↓ new context

- 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)
```

# 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

only one buffer.

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

# 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);
```

- 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

Instead, use MPI_Allreduce (cf previous slide).

- 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

and next slide.

# 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);
```

[redacted]

- 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:

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

```
// 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;
}
```

→ classical way to do it (no MPI).

# 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 |

- 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 |

Here : "+3"

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

DEMO

# Data Distributions

- Back to the vector sum problem, given the partitioning scheme, each process simply adds its assigned components

- **Block partitioning** is used when **data source** is available on a **single process**
  - We have a primitive for this (`MPI_Scatter`)

- **Cyclic partitioning** is used when data source is already available across all processes
  - This is general, we do not need even any assumptions on the number of elements to be processed

# Data Distributions - Block Partitioning

↳ 1st method: cf slide 25

- 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 (size_t i = 0; i < local_x.size (); ++i)
              local_z[i] = local_x[i] + local_y[i];
       return local_z;
}
```

of size local_n { we store only local_x & local_y on each process.

# 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)

MPI_Scatter = block partition implementation

# Scatter WHAT DOES IT DO ?

- 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)
```

# 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 |

clearly it's the implementation of block partition.

# read_vector 1/3

```
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

# read_vector 2/3

```
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);
  }
```

# read_vector

3/3

```
  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

WHAT is iT ?

- 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

WHAT iS iT ?

- 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)
```

cf next slide for example.

# 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 1/3

```
void
print_vector (vector<double> const &
              local_v, unsigned n,
              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 ();
```

# print_vector 2/3

```
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);
  }
```

# print_vector

```
else                //Rank 0 receives everything
  {
    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;   //Reads all the
  }                           //values of the vector.
```

# 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