Message Passing Interface (part. II)

Distributed-Memory Parallel Programming

Orian Louant

In the previous episode...

Diffusion Equation in 1D

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

After discretization and by using a forward difference in time and a central difference in space, the previous equation reads

$$\frac{\partial}{\partial t} u(x_i, t_n) = \alpha \frac{\partial^2}{\partial x^2} u(x_i, t_n) \rightarrow \frac{u_i^{n+1} - u_i^n}{\Delta t} = \alpha \frac{u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n}{\Delta x^2}$$

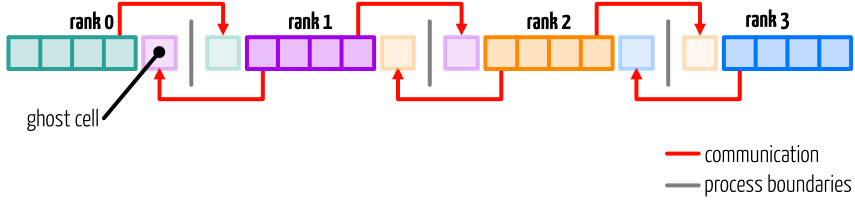
so that for each time step, u_i^{n+1} can be computed as

$$u_i^{n+1} = u_i^n + \alpha \frac{\Delta t}{\Delta x^2} \left(u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n \right)$$

Diffusion Equation in 1D

$$u_i^{n+1} = u_i^n + \alpha \frac{\Delta t}{\Delta x^2} \left(u_{i+1}^n - 2 \cdot u_i^n + u_{i-1}^n \right)$$

Each point depends on the values on the left and right: use ghost cells that are updated (via communication) at each time step



Diffusion 1D (C)

The communication part is written so that we do the left-right communication first and then right-left.

```
const int  left_rank = my_rank > 0            ? my_rank - 1 : MPI_PROC_NULL;
const int right_rank = my_rank < world_size-1 ? my_rank + 1 : MPI_PROC_NULL;

for (unsigned int iter = 1; iter <= NITERS; iter++) {
  MPI_Send(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0, /* ... */);
  MPI_Recv(&uold[0],       1, MPI_DOUBLE, left_rank,  0, /* ... */);

  MPI_Send(&uold[1],         1, MPI_DOUBLE, left_rank,  1, /* ... */);
  MPI_Recv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, /* ... */);

  for (unsigned int i = 1; i < my_size+1; i++)
    unew[i] = uold[i] + DIFF_COEF * dtdx2 * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);

  // ...
}
```

Diffusion 1D (Fortran)

The communication part is written so that we do the left-right communication first and then right-left.

```
 left_rank = merge(my_rank - 1, MPI_PROC_NULL, my_rank > 0)
right_rank = merge(my_rank + 1, MPI_PROC_NULL, my_rank < world_size-1)

do iter = 1, niters
  call MPI_Send(uold(my_size),  1, MPI_DOUBLE_PRECISION, right_rank, 0, ...)
  call MPI_Recv(uold(0),        1, MPI_DOUBLE_PRECISION, left_rank,  0, ...)

  call MPI_Send(uold(1),         1, MPI_DOUBLE_PRECISION, left_rank,  1, ...)
  call MPI_Recv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)

  do i = 1, my_size
    unew(i) = uold(i) + DIFF_COEF * dtdx2 * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
  end do

! ...
end do
```

Combined Sendrecv (C)

An alternative implementation could use combined send-receive

The send-receive operations combine in one operation the sending of a message to one destination and the receiving of another message, from another process

```
int MPI_Sendrecv(const void* sendbuf,          = address of the data you want to send
                 int sendcount,                = number of elements to send
                 MPI_Datatype sendtype,        = the type of data we want to send
                 int dest,                     = the recipient of the message (rank)
                 int sendtag,                  = identify the type of the message
                 void* recvbuf,                = where to receive the data
                 int recvcount,                = the receive buffer capacity
                 MPI_Datatype recvtype,        = the type of data we want to receive
                 int source,                   = the sender of the message (rank)
                 int recvtag,                  = identify the type of the message
                 MPI_Comm comm,                = the communicator used for this message
                 MPI_Status* status);          = informations about the message
```

Combined Sendrecv (Fortran)

The send-receive operations combine in one operation the sending of a message to one destination and the receiving of another message, from another process

```
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
             recvcount, recvtype, source, recvtag, comm, status, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in) :: sendcount, dest, sendtag, recvcount, source, recvtag
    type(MPI_Datatype), intent(in) :: sendtype, recvtype
    type(*), dimension(..) :: recvbuf
    type(MPI_Comm), intent(in) :: comm
    type(mpi_status) :: status
    integer, optional, intent(out) :: ierror
```

Diffusion 1D: Sendrecv version

The 1D diffusion communication code can be rewritten using two `MPI_Sendrecv`: one for left-right communication and a second call for the right-left communication.

```
MPI_Sendrecv(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0,
             &uold[      0], 1, MPI_DOUBLE, left_rank,  0, /* ... */);

MPI_Sendrecv(&uold[        1], 1, MPI_DOUBLE, left_rank,  1,
             &uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, /* ... */);
```

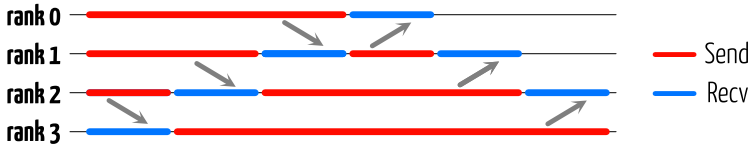
```
call MPI_Sendrecv(uold(my_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, &
&                 uold(      0), 1, MPI_DOUBLE_PRECISION, left_rank,  0, ...)

call MPI_Sendrecv(uold(        1), 1, MPI_DOUBLE_PRECISION, left_rank,  1, &
&                 uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)
```

Diffusion 1D: Communication Serialization

```
MPI_Send(&uold[my_size], 1, ..., right_rank, 0, ... );
MPI_Recv(&uold[0],       1, ..., left_rank,  0, ...);

MPI_Send(&uold[1],         1, ..., left_rank,  1, ...);
MPI_Recv(&uold[my_size+1], 1, ..., right_rank, 1, ...);
```

```
call MPI_Send(uold(my_size),  1, ..., right_rank, 0, ...)
call MPI_Recv(uold(0),        1, ..., left_rank,  0, ...)

call MPI_Send(uold(1),         1, ..., left_rank,  1, ...)
call MPI_Recv(uold(my_size+1), 1, ..., right_rank, 1, ...)
```

Communication performance is poor: the communication is serial if the MPI implementation choose to use the synchronous mode



The Serialization of Communication Problem

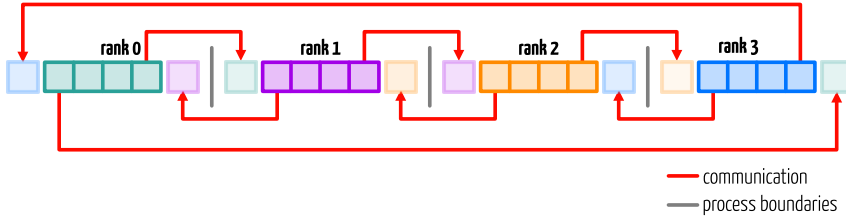
The problem with the serialization of the communication is that

- the processes we add, the longer the communication time
- it leads to a significant impact on the parallel efficiency

The best way to improve the performance is to use **non-blocking communication** where the send and/or receive calls return immediately without waiting for the communication to be completed

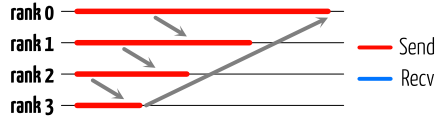
Back to Diffusion Equation

Let’s go back to the diffusion equation but this time with periodic boundary conditions



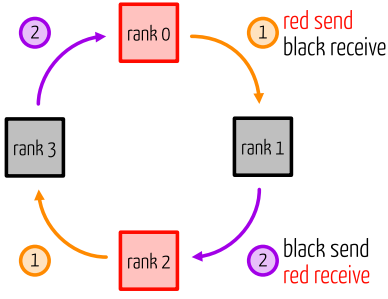
If we use the same communication pattern that we use for the non-periodic code, we end up with a deadlock

```
MPI_Send(..., left_rank, ...);
MPI_Recv(..., right_rank, ...);
```



Ring Communication: Odd-Even

One way to enable communication through a ring is to selectively reorder the send and receive calls



The communication is performed in two steps:

- even ranks send first to an odd rank process and then receive
- odd ranks receive first to an even rank process and then send

Diffusion Equation: Odd-Even

With the odd-even pattern for communication, we can write a left to right transfer of the periodic diffusion equation that do not dealock when using synchronous mode

```c
if (my_rank%2 == 0) {
  MPI_Send(&uold[1],         1, MPI_DOUBLE, left_rank,  1, ...);
  MPI_Recv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, ...);
} else {
  MPI_Recv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, ...);
  MPI_Send(&uold[1],         1, MPI_DOUBLE, left_rank,  1, ...);
}
```

```fortran
if (modulo(my_rank, 2) .eq. 0) then
  call MPI_Send(uold(1),         1, MPI_DOUBLE_PRECISION, left_rank,  1, ...)
  call MPI_Recv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)
else
  call MPI_Recv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, ...)
  call MPI_Send(uold(1),         1, MPI_DOUBLE_PRECISION, left_rank,  1, ...)
end if
```

Diffusion Equation: Odd-Even

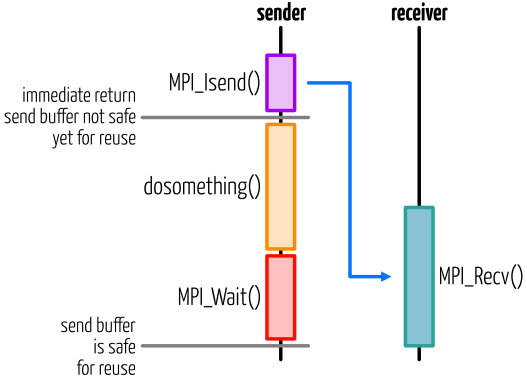
- with the odd-even communication pattern, our code is dealock free but the communication is done in two steps introducing unnecessary latency and synchronization
- on our simple diffusion example, the impact on performance is limited but for more complex application this can have a larger negative impact

Fortunatly, MPI provides an alternative to the blocking (or buffered) send and receive:

- non-blocking functions allow to perform asynchronous communication

Non-Blocking Communication

The other way in which these send and receive operations can be done is by using the "I" functions. The "I" stands for Immediate returns and allow to perform the communication in three phases:



- call the non-blocking function `MPI_Isend` or `MPI_Irecv`
- Do some work
- Wait for the non-blocking communication to complete

Non-Blocking Send

A non-blocking send is executed with the `MPI_Isend` function

```
MPI_Isend(void* buf,                = address of the data you want to send
          int count,                = number of elements to send
          MPI_Datatype datatype,    = the type of data we want to send
          int dest,                 = the recipient of the message (rank)
          int tag,                  = identify the type of the message
          MPI_Comm comm,            = the communicator used for this message
          MPI_Request* request)     = the handle on the non-blocking communication
```

```
MPI_Isend(buf, count, datatype, dest, tag, comm, request, ierror)
    type(*), dimension(..), intent(in), asynchronous :: buf
    integer, intent(in)                              :: count, dest, tag
    type(MPI_Datatype), intent(in)                   :: datatype
    type(MPI_Comm), intent(in)                       :: comm
    type(MPI_Request), intent(out)                   :: request
    integer, optional, intent(out)                   :: ierror
```

Non-Blocking Receive

A non-blocking receive is executed with the `MPI_Irecv` function

```
MPI_Irecv(void* buf,                = where to receive the data
          int count,                = number of elements to send
          MPI_Datatype datatype,    = the type of data we want to receive
          int source,               = the sender of the message (rank)
          int tag,                  = identify the type of the message
          MPI_Comm comm             = the communicator used for this message
          MPI_Request* request)     = the handle on the non-blocking communication
```

```
MPI_Irecv(buf, count, datatype, source, tag, comm, request, ierror)
    type(*), dimension(..), asynchronous :: buf
    integer, intent(in)                  :: count, source, tag
    type(MPI_Datatype), intent(in)       :: datatype
    type(MPI_Comm), intent(in)           :: comm
    type(MPI_Request), intent(out)       :: request
    integer, optional, intent(out)       :: ierror
```

Waiting for a Non-Blocking Communication

An important feature of the non-blocking communication is the `MPI_Request` handle.

The value of this handle

- is generated by the non-blocking communication function
- is used by the `MPI_Wait` or `MPI_Test` functions

When using non-blocking communication, you have to be careful and avoid to

- modify the send buffer before the send operation completes
- read the receive buffer before the receive operation completes

Waiting for a Non-Blocking Communication

The **MPI_Wait** returns when the operation identified by request is complete. This is a blocking function.

```
MPI_Wait(MPI_Request* request,   = handle of the non-blocking communication
         MPI_Status*  status)    = status of the completed communication
```

```
MPI_Wait(request, status, ierror)
    type(MPI_Request), intent(inout) :: request
    type(MPI_Status)                 :: status
    integer, optional, intent(out)   :: ierror
```

Wait on Multiple Requests

You can wait on multiple requests in one call with the `MPI_Waitall` function

```
MPI_Waitall(int count,                          = number of request handlers to wait on
            MPI_Request array_of_requests[],    = request handlers to wait on
            MPI_Status* array_of_statuses[])    = array in which write the statuses
```

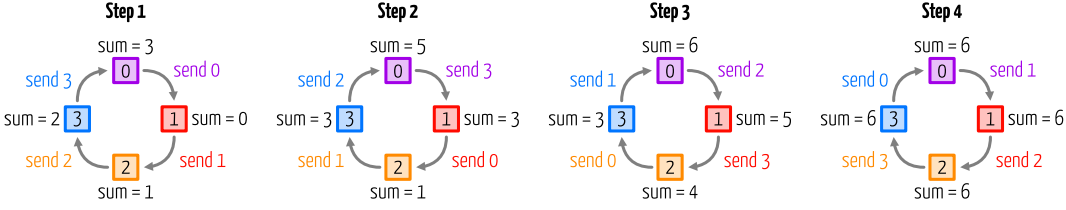
```
MPI_Waitall(count, array_of_requests, array_of_statuses, ierror)
    integer, intent(in)              :: count
    type(MPI_Request), intent(inout) :: array_of_requests(count)
    type(MPI_Status)                 :: array_of_statuses(*)
    integer, optional, intent(out)   :: ierror
```

The i-th entry in `array_of_statuses` is set to the return status of the i-th operation

Example: Ring Communication

As an example, let's consider the rotation of information inside a ring

- Each process stores its rank in the send buffer
- Each process sends its rank to its neighbour on the right
- Each process adds the received value to the sum
- Repeat



Example: Ring Communication (blocking)

The blocking version of the ring communication is similar to the diffusion code that we considered earlier

```c
sum = 0;
send_buf = rank;

for(int i = 0; i < world_size; ++i) {
  if (rank % 2 == 0) {
    MPI_Send(&send_buf, 1, MPI_INT, right_rank, ...);
    MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, ...);
  } else {
    MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, ...);
    MPI_Send(&send_buf, 1, MPI_INT, right_rank, ...);
  }

  send_buf = recv_buf;
  sum += recv_buf;
}
```

```f90
sum = 0
send_buf = rank

do i = 1,world_size
  if (modulo(rank, 2) .eq. 0) then
    call MPI_Send(send_buf, 1, MPI_INTEGER, right_rank, ...)
    call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, ...)
   else
    call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, ...)
    call MPI_Send(send_buf, 1, MPI_INTEGER, right_rank, ...)
  end if

  send_buf = recv_buf
  sum = sum + recv_buf
end do
```

Example: Ring Communication (non-blocking, C)

For the non-blocking version, we use `MPI_Isend` instead of `MPI_Send`

- no need to separate the processes in two groups for the code to be correct: `MPI_Isend` returns immediately so the blocking receive is called
- we use `MPI_Wait` to make sure that the send buffer is safe to be reused

```
sum = 0;
send_buf = rank;

for(int i = 0; i < world_size; ++i) {
  MPI_Isend(&send_buf, 1, MPI_INT, right_rank, ..., &request);
  MPI_Recv(&recv_buf, 1, MPI_INT, left_rank, ...);

  MPI_Wait(&request, MPI_STATUS_IGNORE);

  send_buf = recv_buf;
  sum += recv_buf;
}
```

Example: Ring Communication (non-blocking, Fortran)

For the non-blocking version, we use `MPI_Isend` instead of `MPI_Send`

- no need to separate the processes in two groups for the code to be correct: `MPI_Isend` returns immediately so the blocking receive is called
- we use `MPI_Wait` to make sure that the send buffer is safe to be reused

```
sum = 0
send_buf = rank

do i = 1,world_size
  call MPI_Isend(send_buf, 1, MPI_INTEGER, right_rank, ..., request)
  call MPI_Recv(recv_buf, 1, MPI_INTEGER, left_rank, ...)

  call MPI_Wait(request, MPI_STATUS_IGNORE)

  send_buf = recv_buf
  sum = sum + recv_buf
end do
```

Diffusion Using Non-Blocking Communication (C)

Like the communication around a ring, rewriting the periodic diffusion code with non-blocking avoid to do the communication in two steps

```c
for (int iter = 1; iter <= NITERS; iter++) {
  MPI_Irecv(&uold[0],         1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, &reqs[0]);
  MPI_Irecv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, &reqs[1]);

  MPI_Isend(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &reqs[2]);
  MPI_Isend(&uold[1],       1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD, &reqs[3]);

  MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);

  for (int i = 1; i <= my_size; i++) {
    unew[i] =  uold[i] + alpha * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);
  }

  SWAP_PTR(uold, unew);
}
```

Diffusion Using Non-Blocking Communication (Fortran)

Like the communication around a ring, rewriting the periodic diffusion code with non-blocking avoid to do the communication in two steps

```
do iter = 1, niters
  call MPI_Isend(uold(my_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD, reqs(0))
  call MPI_Isend(uold(1),       1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD, reqs(1))

  call MPI_Irecv(uold(0),         1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, reqs(2))
  call MPI_Irecv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, reqs(3))

  call MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE)

  do i = 1, my_size
    unew(i) =  uold(i) + alpha * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
  end do

  uold(1:my_size) = unew(1:my_size)
end do
```

Better Hide the Communications

As discussed before, communication comes at a cost. You should try to hide this cost has much as possible:

- if you want to use a large number of processes, you should try to hide this cost has much as possible
- achieved by overlaping communication and computation with non-blocking communication
- let the CPU do useful science instead of waiting for a communication to complete

The basic idea is to

- start the communication as soon as possible
- only wait at a point where you need to use the result of the communication or reuse a buffer involved in a communication

Overlap Computation and Communication (C)

The 1D diffusion can be restructured so that overlap computation and communication

```c
for (int iter = 1; iter <= NITERS; iter++) {
  MPI_Irecv(&uold[0],         1, MPI_DOUBLE, left_rank,  0, MPI_COMM_WORLD, &recv_reqs[0]);
  MPI_Irecv(&uold[my_size+1], 1, MPI_DOUBLE, right_rank, 1, MPI_COMM_WORLD, &recv_reqs[1]);

  MPI_Isend(&uold[my_size], 1, MPI_DOUBLE, right_rank, 0, MPI_COMM_WORLD, &send_reqs[0]);
  MPI_Isend(&uold[1],       1, MPI_DOUBLE, left_rank,  1, MPI_COMM_WORLD, &send_reqs[1]);

  for (int i = 2; i <= my_size-1; i++) {
    unew[i] =  uold[i] + alpha * (uold[i+1] - 2.0 * uold[i] + uold[i-1]);
  }

  MPI_Waitall(2, recv_reqs, MPI_STATUSES_IGNORE);

  unew[      1] =  uold[      1] + alpha * (uold[        2] - 2.0 * uold[      1] + uold[        0]);
  unew[my_size] =  uold[my_size] + alpha * (uold[my_size+1] - 2.0 * uold[my_size] + uold[my_size-1]);

  MPI_Waitall(2, send_reqs, MPI_STATUSES_IGNORE);
  SWAP_PTR(uold, unew);
}
```

Overlap Computation and Communication (Fortran)

The 1D diffusion can be restructured so that overlap computation and communication

```
do iter = 1, niters
  call MPI_Isend(uold(my_size), 1, MPI_DOUBLE_PRECISION, right_rank, 0, MPI_COMM_WORLD, send_reqs(1))
  call MPI_Isend(uold(1),       1, MPI_DOUBLE_PRECISION, left_rank,  1, MPI_COMM_WORLD, send_reqs(2))

  call MPI_Irecv(uold(0),         1, MPI_DOUBLE_PRECISION, left_rank,  0, MPI_COMM_WORLD, recv_reqs(1))
  call MPI_Irecv(uold(my_size+1), 1, MPI_DOUBLE_PRECISION, right_rank, 1, MPI_COMM_WORLD, recv_reqs(2))

  do i = 2, my_size-1
    unew(i) =  uold(i) + alpha * (uold(i+1) - 2.0 * uold(i) + uold(i-1))
  end do

  call MPI_Waitall(2, recv_reqs, MPI_STATUSES_IGNORE)

  unew(      1) = uold(      1) + alpha * (uold(        2) - 2.0 * uold(      1) + uold(        0))
  unew(my_size) = uold(my_size) + alpha * (uold(my_size+1) - 2.0 * uold(my_size) + uold(my_size-1))

  call MPI_Waitall(2, send_reqs, MPI_STATUSES_IGNORE)
  uold(1:my_size) = unew(1:my_size)
end do
```

Collective Communication

Collective Communication

So far, we have covered the topic of point-to-point communication: with a message that is exchanged between a sender and a receiver. However, in a lot of applications, collective communication may be required:

- **All-To-All:** All processes contribute to the result and all processes receive the result
- **All-To-One:** all processes contribute to the result and one process receives the result
- **One-To-All:** one process contributes to the result and all processes receive the result

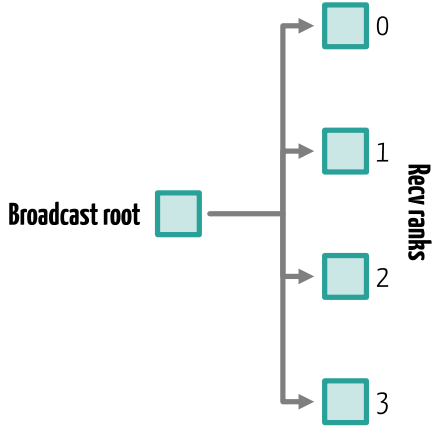
Collective Communication

- the key argument of the collective communication routine is the communicator that define the group of participating processes
- the amount of data sent must exactly match the amount of data specified by the receiver

- **Broadcast:** Send data to all the processes
- **Scatter:** Distribute data between the processes
- **Gather:** Collect data from multiple processes to one process
- **Reduce:** Perform a reduction

Broadcast

During a broadcast, one process (the root) sends the same data to all processes in a communicator.



Broadcast

```
MPI_Bcast(void* buffer,              = address of the data you want to broadcast
          int count,                 = number of elements to broadcast
          MPI_Datatype datatype,     = the type of data we want to broadcast
          int root,                  = rank of the broadcast root
          MPI_Comm comm)             = the communicator used for this broadcast
```

```
MPI_Bcast(buffer, count, datatype, root, comm, ierror)
    type(*), dimension(..)          :: buffer
    integer, intent(in)             :: count, root
    type(MPI_Datatype), intent(in) :: datatype
    type(MPI_Comm), intent(in)      :: comm
    integer, optional, intent(out) :: ierror
```

Broadcast Example

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

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int bcast_root = 0;

int value;
if(rank == bcast_root) {
    value = 12345;
    printf("I am the broadcast root with rank %d "
           "and I send value %d.\n", rank, value);
}

MPI_Bcast(&value, 1, MPI_INT, bcast_root, MPI_COMM_WORLD);

if(rank != bcast_root) {
    printf("I am a broadcast receiver with rank %d "
           "and I obtained value %d.\n", rank, value);
}

MPI_Finalize();
```

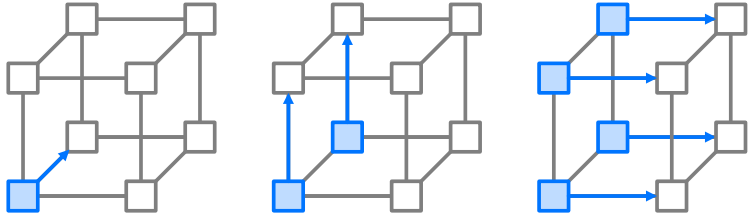
```fortran
      integer :: i, value, ierror
      integer :: rank, bcast_root

      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      bcast_root = 0

      if (rank .eq. bcast_root) then
        value = 12345
        print 100, rank, value
100     format('I am the broadcast root with rank ', i0, &
&               ' and I send value ', i0)
      end if

      call MPI_Bcast(value, 1, MPI_INT, bcast_root, &
&                    MPI_COMM_WORLD, ierror)

      if (rank .ne. bcast_root) then
        print 200, rank, value
200     format('I am a broadcast receiver with rank ', i0, &
&               ' and I obtained value ', i0)
      end if

      call MPI_Finalize(ierror)
```

Broadcast Example

```
 $ mpicc -o broadcast broadcast.c
 $ mpirun -np 4 ./broadcast
I am the broadcast root with rank 0 and I send value 12345.
I am a broadcast receiver with rank 2 and I obtained value 12345.
I am a broadcast receiver with rank 1 and I obtained value 12345.
I am a broadcast receiver with rank 3 and I obtained value 12345.
```

Broadcast hypercube

A one to all broadcast can be visualized on a hypercube of d dimensions with $d = \log_2 p$.

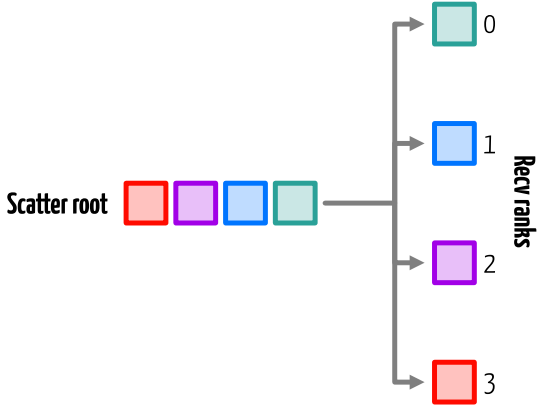


The broadcast procedure involves $\log_2 p$ point-to-point simple message transfers.

$$T_{broad} = \left(T_{latency} + \frac{n}{B_{peak}} \right) \log_2 p$$

MPI Scatter

During a scatter, the elements of an array are distributed in the order of process rank.



MPI Scatter

```
MPI_Scatter(void* sendbuf,          = address of the data you want to scatter
            int sendcount,          = number of elements sent to each process
            MPI_Datatype sendtype,  = the type of data we want to scatter
            void* recvbuf,          = where to receive the data
            int recvcount,          = number of elements to receive
            MPI_Datatype recvtype,  = the type of data we want to receive
            int root,               = rank of the scatter root
            MPI_Comm comm)          = the communicator used for this scatter
```

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in)                :: sendcount, recvcount, root
    type(MPI_Datatype), intent(in)     :: sendtype, recvtype
    type(*), dimension(..)             :: recvbuf
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

Scatter Example

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

int size, rank, value, scatt_root = 0;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

int* data = NULL;
if(rank == scatt_root) {
    data = (int*)malloc(sizeof(int)*size);

    printf("Values to scatter from process %d:", rank);
    for (int i = 0; i < size; i++) {
        data[i] = 100 * i;
        printf(" %d", data[i]);
    }
    printf("\n");
}

MPI_Scatter(data, 1, MPI_INT, &value, 1, MPI_INT,
            scatt_root, MPI_COMM_WORLD);
printf("Process %d received value %d.\n", rank, value);

if(rank == scatt_root) free(data);

MPI_Finalize();
```

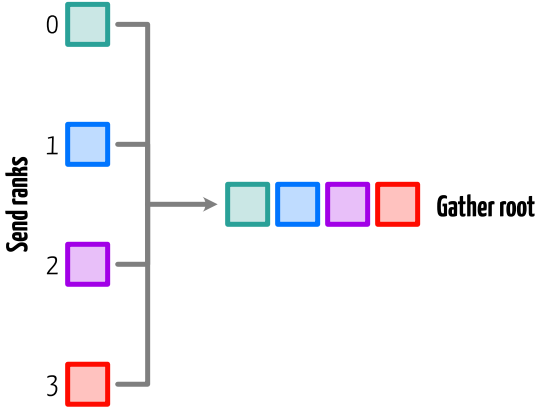
```fortran
      integer :: size, rank, ierror
      integer :: value, scatt_root, i
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      if (rank .eq. scatt_root) then
          allocate(buffer(size))
          do i = 1,size
            buffer(i) = 100 * i
          end do

          print 100, rank, data
100       format('Values to scatter from process ', i0, &
&                 ':', *(1x,i0))
      end if

      call MPI_Scatter(buffer, 1, MPI_INTEGER, value, 1,
           MPI_INTEGER, &
&                     scatt_root, MPI_COMM_WORLD, ierror)

      print 200, rank, value
200   format('Process ', i0, ' received value ', i0)

      if (rank .eq. scatt_root) deallocate(buffer)

      call MPI_Finalize(ierror)
```

Scatter Example

```
 $ mpicc -o scatter scatter.c
 $ mpirun -np 4 ./scatter
Values to scatter from process 0: 0 100 200 300
Process 1 received value 100.
Process 2 received value 200.
Process 0 received value 0.
Process 3 received value 300.
```

MPI Gather

A gathering is taking elements from each process and gathers them to the root process.



MPI Gather

```
MPI_Gather(void* sendbuf,           = address of the data you want to gather
           int sendcount,           = number of elements to gather
           MPI_Datatype sendtype,   = the type of data we want to gather
           void* recvbuf,           = where to receive the data
           int recvcount,           = number of elements to receive
           MPI_Datatype recvtype,   = the type of data we want to receive
           int root,                = rank of the gather root
           MPI_Comm comm)           = the communicator used for this gather
```

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             root, comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in)                :: sendcount, recvcount, root
    type(MPI_Datatype), intent(in)     :: sendtype, recvtype
    type(*), dimension(..)             :: recvbuf
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

Gather Example

```c
int gath_root = 0;

int size, rank, ierror, value;
int *buffer;

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

value = rank * 100;
printf("Process %d has value %d.\n", rank, value);

if (rank == gath_root) buffer = (int*)malloc(sizeof(int)*size);
MPI_Gather(&value, 1, MPI_INT, buffer, 1, MPI_INT,
           gath_root, MPI_COMM_WORLD);

if (rank == gath_root) {
  printf("Values collected on process %d:", rank);
  for (int i = 0; i < size; ++i) printf(" %d", buffer[i]);
  printf(".\n");

  free(buffer);
}

MPI_Finalize();
```

```fortran
      integer, parameter :: gath_root = 0

      integer :: size, rank, ierror, value
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      value = rank * 100

      print 100, rank, value
100   format('Process ', i0, ' has value ', i0, '.')

      if (rank .eq. gath_root) allocate(buffer(size))
      call MPI_Gather(value, 1, MPI_INTEGER, buffer, 1, &
&          MPI_INTEGER, gath_root, MPI_COMM_WORLD, ierror)

      if (rank .eq. gath_root) then
        print 200, rank, buffer
200     format('Values collected on process ', i0, &
&             ': ', *(1x,i0), '.')

        deallocate(buffer)
      end if

      call MPI_Finalize(ierror)
```

Gather Example

```
 $ mpicc -o gather gather.c
 $ mpirun -np 4 ./gather
Process 2 has value 200.
Process 0 has value 0.
Process 3 has value 300.
Process 1 has value 100.
Values collected on process 0: 0 100 200 300.
```

Back to the Sum of Integer

If we go back to the communication part of the sum of integers.

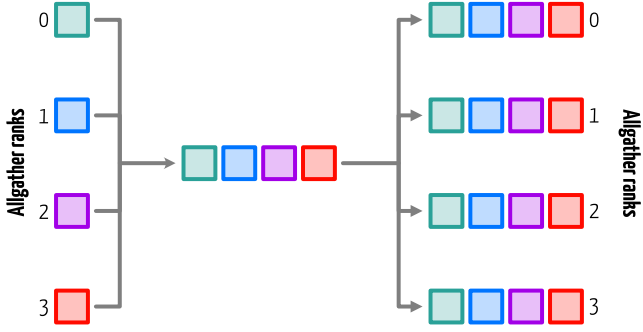
```
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

We can rewrite this part of the code with a gather

```
unsigned int* remote_sums;
if(rank == 0) remote_sums = (unsigned int*)malloc(sizeof(int)*world_size);

MPI_Gather(&proc_sum, 1, MPI_UNSIGNED, remote_sums, 1, MPI_UNSIGNED, 0, MPI_COMM_WORLD);

if(rank == 0) {
  unsigned int sum = 0;
  for(int i = 0; i < world_size; ++i)
    sum += remote_sums[i];
}
```

MPI All Gather

You can perform an all gather operation where all the pieces of data are gathered in the order of the ranks and then, the result is broadcast to all the processes in the communicator.



MPI All Gather

```
MPI_Allgather(void* sendbuf,           = address of the data you want to gather
              int sendcount,           = number of elements to gather
              MPI_Datatype sendtype,   = the type of data we want to gather
              void* recvbuf,           = where to receive the data
              int recvcount,           = number of elements to receive
              MPI_Datatype recvtype,   = the type of data we want to receive
              MPI_Comm comm)           = the communicator used for this gather
```

```
MPI_Allgather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
              comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in)                :: sendcount, recvcount
    type(MPI_Datatype), intent(in)     :: sendtype, recvtype
    type(*), dimension(..)             :: recvbuf
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

All Gather Example

```c
int gath_root = 0;
int size, rank, ierror, value;
int *buffer;

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

value = rank * 100;

printf("Process %d has value %d.\n", rank, value);

buffer = (int*)malloc(sizeof(int)*size);
MPI_Allgather(&value, 1, MPI_INT, buffer, 1,
              MPI_INT, MPI_COMM_WORLD);

printf("Values collected on process %d:", rank);
for (int i = 0; i < size; ++i) printf(" %d", buffer[i]);
printf(".\n");

free(buffer);

MPI_Finalize();
```

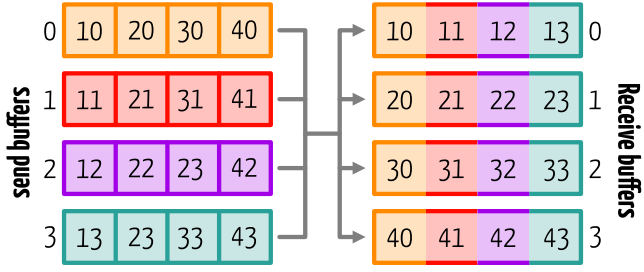
```fortran
      integer, parameter :: gath_root = 0

      integer :: size, rank, ierror, value
      integer, dimension(:), allocatable :: buffer

      call MPI_Init(ierror)
      call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      value = rank * 100
      print 100, rank, value
100   format('Process ', i0, ' has value ', i0, '.')

      allocate(buffer(size))

      call MPI_Allgather(value, 1, MPI_INTEGER, buffer, 1, &
&                        MPI_INTEGER, MPI_COMM_WORLD, ierror)

      print 200, rank, buffer
200   format('Values collected on process ', i0, &
&            ': ', *(1x,i0), '.')

      deallocate(buffer)

      call MPI_Finalize(ierror)
```

All Gather Example

```
 $ mpicc -o allgather allgather.c
 $ mpirun -np 4 ./allgather
Process 2 has value 200.
Process 0 has value 0.
Process 3 has value 300.
Process 1 has value 100.
Values collected on process 1: 0 100 200 300.
Values collected on process 3: 0 100 200 300.
Values collected on process 0: 0 100 200 300.
Values collected on process 2: 0 100 200 300.
```

MPI All to All

All to all Scatter/Gather communication allows for data distribution to all processes: the `j`-th block sent from rank `i` is received by rank `j` and is placed in the `i`-th block of the receive buffer.



MPI All to All

```
MPI_Alltoall(void* sendbuf,          = address of the data you want to send
             int sendcount,          = number of elements to send
             MPI_Datatype sendtype,  = the type of data we want to send
             void* recvbuf,          = where to receive the data
             int recvcount,          = number of elements to receive
             MPI_Datatype recvtype,  = the type of data we want to receive
             MPI_Comm comm)          = the communicator used
```

```
MPI_Alltoall(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype,
             comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in)                :: sendcount, recvcount
    type(MPI_Datatype), intent(in)     :: sendtype, recvtype
    type(*), dimension(..)             :: recvbuf
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

Vector Variants

Ok, but what if I do not want to transfer the same number of elements from each process?

- all the functions presented previously have a ”v” variant
- these variants allow the messages received to have different lengths and be stored at arbitrary locations

On the root process, instead of specifying the number of elements to send, the vector variants take:

- an array where the entry `i` specifies the number of elements to send to rank `i`
- an array where the entry `i` specifies the displacement from which to take the data to send to rank `i`

Vector Variants: Scatterv

For example, the vector variant of the `MPI_Scatter` function is `MPI_Scatterv`

```
MPI_Scatterv(void* sendbuf,           = address of the data you want to send
             int sendcounts[],        = the number of elements to send to each process
             int displs[],            = the displacement to the message sent to each process
             MPI_Datatype sendtype,   = the type of data we want to send
             void* recvbuf,           = where to receive the data
             int recvcount,           = number of elements to receive
             MPI_Datatype recvtype,   = the type of data we want to receive
             int root,                = rank of the root proces
             MPI_Comm comm)           = the communicator used
```

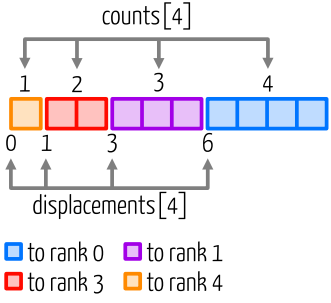
Vector Variants: Scatterv

For example, the vector variant of the `MPI_Scatter` function is `MPI_Scatterv`

```
MPI_Scatterv(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount,
             recvtype, root, comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    integer, intent(in)                :: sendcounts(*), displs(*)
    integer, intent(in)                :: recvcount, root
    type(MPI_Datatype), intent(in)     :: sendtype, recvtype
    type(*), dimension(..)             :: recvbuf
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

Scatterv Example (C)

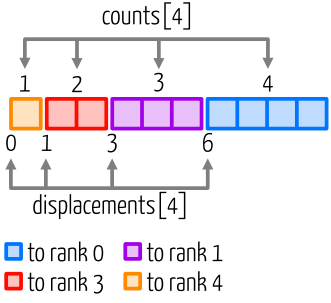
- process with rank 0 is the root, it fills an array and dispatches the values to all the processes
- the processes receive `rank + 1` elements




```c
int*  displs = NULL;
int*  nelems = NULL;
int* sendbuf = NULL;
int* recvbuf = (int*)malloc(sizeof(int)*(rank+1));

if (rank == 0) {
  int n = world_size*(world_size+1)/2;

  sendbuf = (int*)malloc(sizeof(int)*n);
  displs  = (int*)malloc(sizeof(int)*world_size);
  nelems  = (int*)malloc(sizeof(int)*world_size);

  for (int i = 0; i < n; ++i) sendbuf[i] = 100*(i+1);

  for (int i = 0; i < world_size; ++i) {
    displs[i] = i*(i+1)/2;
    nelems[i] = i+1;
  }
}

MPI_Scatterv(sendbuf, nelems, displs, MPI_INT,
             recvbuf, rank+1, MPI_INT, ...);

printf("Process %d received values:", rank);
for(int i = 0; i < rank+1; i++) printf(" %d", recvbuf[i]);
printf("\n");
```

Scatterv Example (Fortran)

- process with rank 0 is the root, it fills an array and dispatches the values to all the processes
- the processes receive `rank + 1` elements



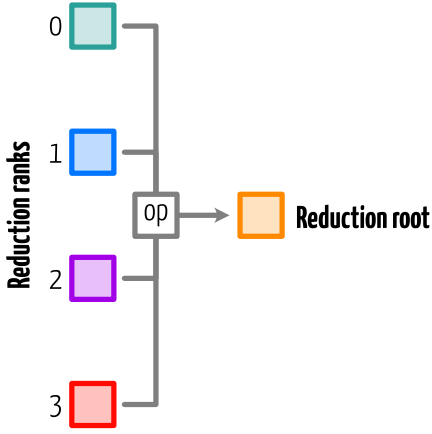

```
    integer, dimension(:), allocatable :: sendbuf, recvbuf
    integer, dimension(:), allocatable :: displs, nelems

    allocate(recvbuf(rank+1))

    if (rank .eq. 0) then
      n = world_size*(world_size+1)/2

      allocate(sendbuf(n))
      allocate(displs(world_size), nelems(world_size))

      do i = 1,n
        sendbuf(i) = 100*i
      end do

      do i = 1,world_size
        displs(i) = i*(i-1)/2
        nelems(i) = i
      end do
    end if

    call MPI_Scatterv(sendbuf, nelems, displs, MPI_INTEGER, &
&                     recvbuf, rank+1, MPI_INTEGER, ...)

    print 100, rank, recvbuf
100 format('Process ', i0, ' received values:', *(1x,i0))
```

Scatterv Example

```
 $ mpicc -o scatterv scatterv.c
 $ mpirun -np 4 ./scatterv
Process 0 received values: 100
Process 1 received values: 200 300
Process 3 received values: 700 800 900 1000
Process 2 received values: 400 500 600
```

MPI Reduce

Data reduction is reducing a set of numbers into a smaller set of numbers. For example, summing elements of an array or find the min/max value in an array.



MPI Reduce

```
MPI_Reduce(void* sendbuf,           = address of the data you want to reduce
           void* recvbuf,           = address of where to store the result
           int count,               = the number of data elements
           MPI_Datatype datatype,   = the type of data we want to reduce
           MPI_Op op,               = the type operation to perform
           int root,                = rank of the reduction root
           MPI_Comm comm)           = the communicator used for this reduction
```

```
MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    type(*), dimension(..)             :: recvbuf
    integer, intent(in)                :: count, root
    type(MPI_Datatype), intent(in)     :: datatype
    type(MPI_Op), intent(in)           :: op
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

MPI Reduction Operators

MPI has a number of elementary reduction operators, corresponding to the operators of the C programming language.

MPI Op	Operation	MPI Op	Operation
`MPI_MIN`	`min`	`MPI_LAND`	`&&`
`MPI_MAX`	`max`	`MPI_LOR`	`||`
`MPI_SUM`	`+`	`MPI_BAND`	`&`
`MPI_PROD`	`*`	`MPI_BOR`	`|`

In addition you can create your own custom operator type.

Back to the Sum of Integers

If we go back to the communication part of the sum of integers.

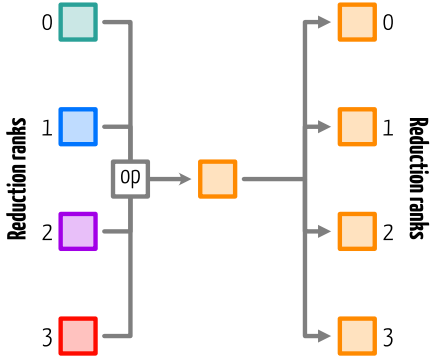
```
if (rank > 0) {
  MPI_Send(&proc_sum, 1, MPI_UNSIGNED, 0, 1, MPI_COMM_WORLD);
} else {
  unsigned int remote_sum;
  for(int src = 1; src < world_size; ++src) {
    MPI_Recv(&remote_sum, 1, MPI_UNSIGNED, src, 1, MPI_COMM_WORLD, &status);
    proc_sum += remote_sum;
  }
}
```

We can rewrite this part of the code with a reduction

```
unsigned int final_sum;
MPI_Reduce(&proc_sum, &final_sum, 1, MPI_UNSIGNED, MPI_SUM, 0, MPI_COMM_WORLD);
```

MPI All Reduce

You can also use an all reduce operation so that the result is available to all the processes in the communicator.



MPI All Reduce

```
MPI_Allreduce(void* sendbuf,          = address of the data you want to reduce
              void* recvbuf,          = address of where to store the result
              int count,              = the number of data elements
              MPI_Datatype datatype,  = the type of data we want to reduce
              MPI_Op op,              = the type operation to perform
              MPI_Comm comm)          = the communicator used for this reduction
```

```
MPI_Allreduce(sendbuf, recvbuf, count, datatype, op, comm, ierror)
    type(*), dimension(..), intent(in) :: sendbuf
    type(*), dimension(..)             :: recvbuf
    integer, intent(in)                :: count
    type(MPI_Datatype), intent(in)     :: datatype
    type(MPI_Op), intent(in)           :: op
    type(MPI_Comm), intent(in)         :: comm
    integer, optional, intent(out)     :: ierror
```

Reduce and Allreduce Example

As an example of the `MPI_Allreduce` and `MPI_Reduce` functions, we will consider the computation of standard deviation

$$\sigma = \sqrt{\frac{\sum_i (x_i - \mu)^2}{N}}$$

- x_i: value of the population
- μ: mean of the population
- N: size of the population

Reduce and Allreduce Example (C)

```c
for (int i = 0; i < nelems_per_rank; ++i)
  local_sum += values[i];

MPI_Allreduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
mean = global_sum / (nelems_per_rank * world_size);

for (int i = 0; i < nelems_per_rank; ++i)
  local_sq_diff += (values[i] - mean) * (values[i] - mean);

MPI_Reduce(&local_sq_diff, &global_sq_diff, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (rank == 0) {
  double stddev = sqrt(global_sq_diff / (nelems_per_rank * world_size));
  printf("Mean = %lf, Standard deviation = %lf\n", mean, stddev);
}
```

- for the mean as all ranks need it for the subsequent step we use `MPI_Allreduce`
- we use `MPI_Reduce` to sum the squared difference on rank 0 and compute the final result

Reduce and Allreduce Example (Fortran)

```
    do i = 1, nelems_per_rank
      local_sum = local_sum + values(i)
    enddo

    call MPI_Allreduce(local_sum, global_sum, 1, MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD)
    mean = global_sum / (nelems_per_rank * size)

    do i = 1, nelems_per_rank
      local_sq_diff = local_sq_diff + (values(i) - mean) * (values(i) - mean)
    end do

    call MPI_Reduce(local_sq_diff, global_sq_diff, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, MPI_COMM_WORLD)

    if (rank .eq. 0) then
      stddev = dsqrt(global_sq_diff / dble(nelems_per_rank * size))
      print 100, mean, stddev
100   format('Mean = ', f12.6, ', Standard deviation = ', f12.6, f12.6)
    end if
```

- for the mean as all ranks need it for the subsequent step we use `MPI_Allreduce`
- we use `MPI_Reduce` to sum the squared difference on rank 0 and compute the final result

MPI Barrier

A barrier can be used to synchronize all processes in a communicator. Each process wait until all processes reach this point before proceeding.

```
MPI_Barrier(MPI_Comm communicator)
```

For example:

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

int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

printf("Process %d: I start waiting at the barrier.\n",
        rank);

MPI_Barrier(MPI_COMM_WORLD);

printf("Process %d: I'm on the other side of the barrier.\n",
        rank);

MPI_Finalize();
```

```fortran
      integer :: rank, ierror

      call MPI_Init(ierror)
      call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

      print 100, rank
100   format('Process ', i0, ': I start waiting at the barrier.')

      call MPI_Barrier(MPI_COMM_WORLD, ierror);

      print 200, rank
200   format('Process ', i0, &
&           ': I am on the other side of the barrier.')

      call MPI_Finalize(ierror)
```

Wrapping up

Summary

Today, we covered the following topics:

- Point-to-point communication
- Non-blocking communication
- Collective communication

But we only scratch the surface: the possibilities offered by MPI are much broader than what we have discussed.

Going further

The possibilities offered by MPI are much broader than what we have discussed.

- User-defined datatype
- Persistent communication
- One-sided communication
- File I/O
- Topologies

Performance

Performance

- In order to extract the best performance from MPI, you need to hide the communication. This means that your code should spend most of its time doing the actual computation rather than communication.
- The same is true with the communication. Small messages are dominated by the latency. Try to have a message large enough to hide it.

Performance: Case Study

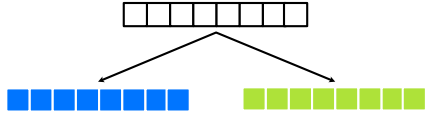
We will consider an hypothethical 1D stencil calculation where each cells are updated this way.

$$u_i^{new} = u_{i+1}^{old} - u_{i-1}^{old}$$

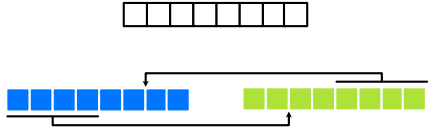
In order to update a cell, we require the value of the left and right cells.

Performance: Case Study

One way to do it, is to replicate the entire array to each of the processes.

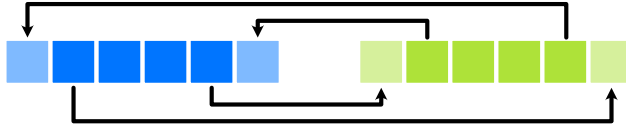


But then, at each step of the calculation, we will copy a large chunk of the data. Another problem is that the communication size increase with the size of the problem.



Performance: Case Study

The best way to solve this is to use a halo cells which contain the neighbour's value required for the calculation. Only these cells need to be updated through communication at each step.



This way, the size of the message will be the same whatever the problem size. Hopefully, the time spend in the calculation part, will hide the latency cost of the small message size.