Outline of this lecture

Communication Modes

- Standard Send
- Synchronous Send
- Buffered send

Blocking call

- A call is blocking if the memory space used for the communication can be reused immediately after the exit of the call
- The data sent can be modified after the call.
- The data received can be read after the call.

Point-to-Point Send Modes

Mode	Blocking	Non-blocking
Standard Send	`MPI_SEND()`	`MPI_ISEND()`
Synchronous Send	`MPI_SSEND()`	`MPI_ISSEND()`
Buffered send	`MPI_BSEND()`	`MPI_IBSEND()`
Receive	`MPI_RECV()`	`MPI_IRECV()`

Table: Point-to-Point Send Modes

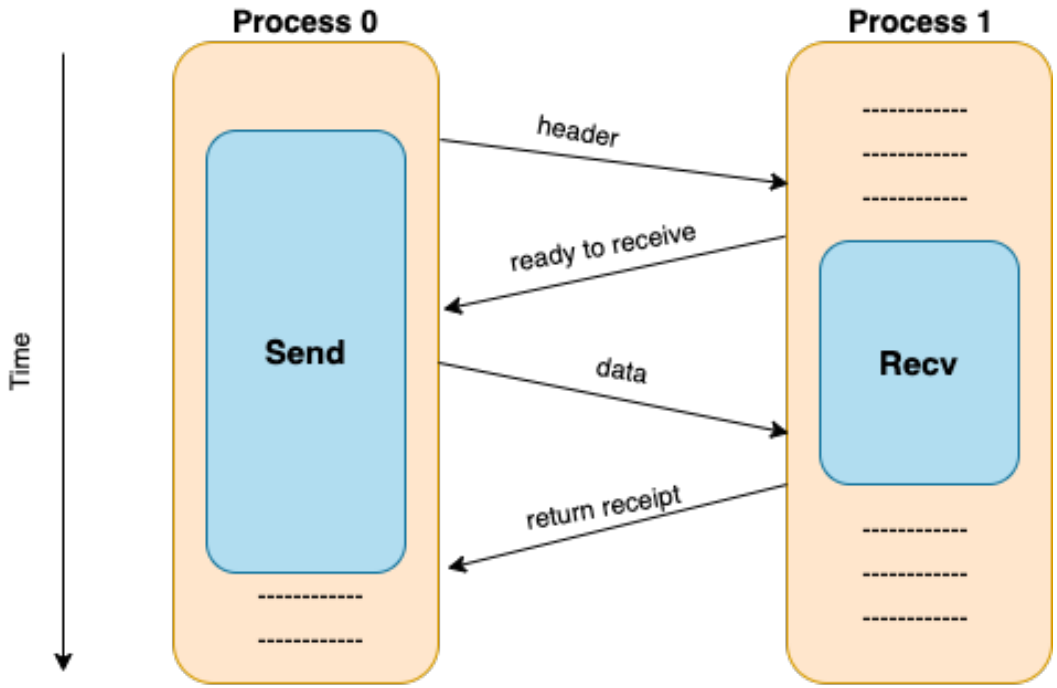
Communication Modes

Synchronous sends

- A synchronous send involves a synchronization between the involved processes. A send cannot start until its receive is posted. There can be no communication before the two processes are ready to communicate.

Rendezvous Protocol

- The rendezvous protocol is generally the protocol used for synchronous sends (implementation-dependent). The return receipt is optional.



Communication Modes

Interface of : MPI_SSEND()

```
MPI_SSEND(values, count, msgtype, dest, tag, comm, code)

type(*), intent(in) :: values
integer, intent(in) :: count, msgtype, dest, tag, comm

integer, intent(out) :: code
```

- Advantages of synchronous mode
  - Low resource consumption (no buffer)
  - Rapid if the receiver is ready (no copying in a buffer)
  - Knowledge of receipt through synchronization
- Disadvantages of synchronous mode
  - Waiting time if the receiver is not there/not ready
  - Risk of deadlocks

```
COMM.Ssend(self, data, int dest, int tag=0)
#or
COMM.ssend(self, data, int dest, int tag=0)
```

Communication Modes

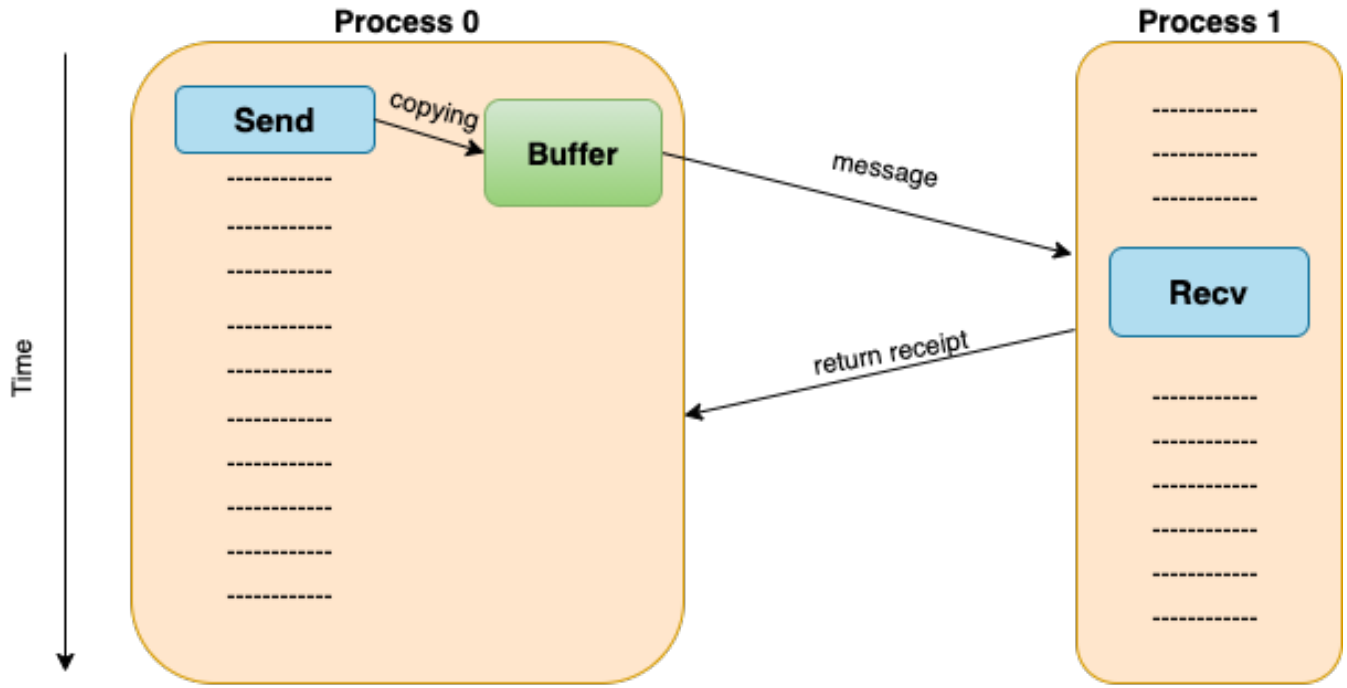
Deadlock example

```
1  from mpi4py import MPI
2
3  COMM = MPI.COMM_WORLD
4  RANK = COMM.Get_rank()
5
6  tag = 99
7
8  sendbuf = 1000
9  COMM.ssend(sendbuf, dest=1, tag=tag)
10
11 recvbuf = COMM.recv(source=1, tag=tag)
12 print("I, process 1, I received ",recvbuf," from the process 2.")
```

```
mpirun -n 2 python ssendrecv.py
```

Communication Modes

- **Buffered sends**
  A buffered send implies the copying of data into an intermediate memory space.
- **Protocol with user buffer on the sender side**
  In this approach, the buffer is on the sender side and is managed explicitly by the application. A buffer managed by MPI can exist on the receiver side. Many variants are possible. The return receipt is optional.



Communication Modes

Interface of : MPI_BSEND()

- **Buffered sends**
  The buffers have to be managed manually (with calls to MPI_BUFFER_ATTACH() and MPI_BUFFER_DETACH()). Message header size needs to be taken into account when allocating buffers (by adding the constant MPI_BSEND_OVERHEAD() for each message occurrence).
- **Interfaces**

```
1  MPI_BUFFER_ATTACH (buf, typesize, code)
2  MPI_BSEND (values, count, msgtype, dest, tag, comm, code)
3  MPI_BUFFER_DETACH (buf, typesize, code)
4
5  TYPE(*), intent(in) :: values
6  integer, intent(in) :: count, msgtype, dest, tag, comm
7  integer, intent(out) :: code
8  TYPE(*) :: buf
9
10 integer :: typesize
```

```
1  COMM.Bsend(self, data, int dest, int tag=0)
2  #or
3  COMM.bsend(self, data, int dest, int tag=0)
```

Communication Modes

- Advantages of buffered mode
  - No need to wait for the receiver (copying in a buffer)
  - No risk of deadlocks
- Disadvantages of buffered mode
  - Uses more resources (memory use by buffers with saturation risk)
  - The send buffers in the MPI_BSEND() or MPI_IBSEND() calls have to be managed manually (often difficult to choose a suitable size)
  - Slightly slower than the synchronous sends if the receiver is ready
  - No knowledge of receipt (send-receive decoupling)
  - Risk of wasted memory space if buffers are too oversized
  - Application crashes if buffer is too small
  - There are often hidden buffers managed by the MPI implementation on the sender side and/or on the receiver side (and consuming memory resources)

Communication Modes

Interface of : MPI_SEND()

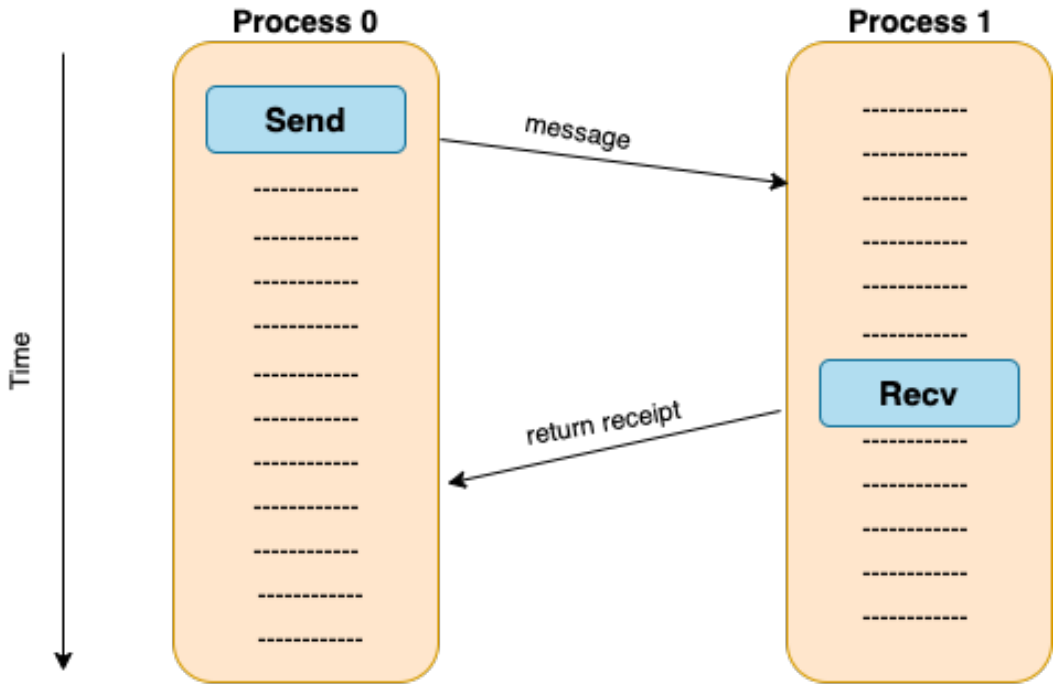
- **Standard sends**
  A standard send is made by calling the MPI_SEND() subroutine. In most implementations, the mode is buffered (eager) for small messages but is synchronous for larger messages.
- **Interfaces**

```
1 MPI_SEND(values, count, msgtype, dest, tag, comm, code)
2
3 TYPE(*), intent(in) :: values
4 integer, intent(in) :: count, msgtype, dest, tag, comm
5
6 integer, intent(out) :: codee
```

```
1 COMM.Send(self, data, int dest, int tag=0)
2 #or
3 COMM.send(self, data, int dest, int tag=0)
```

Communication Modes

- **The eager protocol**
  The eager protocol is often used for standard sends of small-size messages. It can also be used for sends with MPI_BSEND() for small messages (implementation-dependent) and by bypassing the user buffer on the sender side. In this approach, the buffer is on the receiver side. The return receipt is optional.



Communication Modes

- Advantages of standard mode
  - Often the most efficient (because the constructor chose the best parameters and algorithms)
  - The most portable for performance
- Disadvantages of standard mode
  - Little control over the mode actually used (often accessible via environment variables)
  - Risk of deadlocks depending on the mode used
  - Behavior can vary according to the architecture and problem size

Communication Modes

- **Example :**

```
1  if RANK == 2:
2      sendbuf = 1000
3      COMM.send(sendbuf, dest=1, tag=tag)
4
5  if RANK == 5
6      recvbuf = COMM.recv(source=2, tag=tag)
7      print("I, process 5, I received ",recvbuf," from the process 2.")
```

```
mpirun -n 6 python bsendrecv.py

I, process 5, I received 1000 from the process 2.
```

Communication Modes

Non blocking communication

- **Presentation**

  The overlap of communications by computations is a method which allows executing communications operations in the background while the program continues to operate.

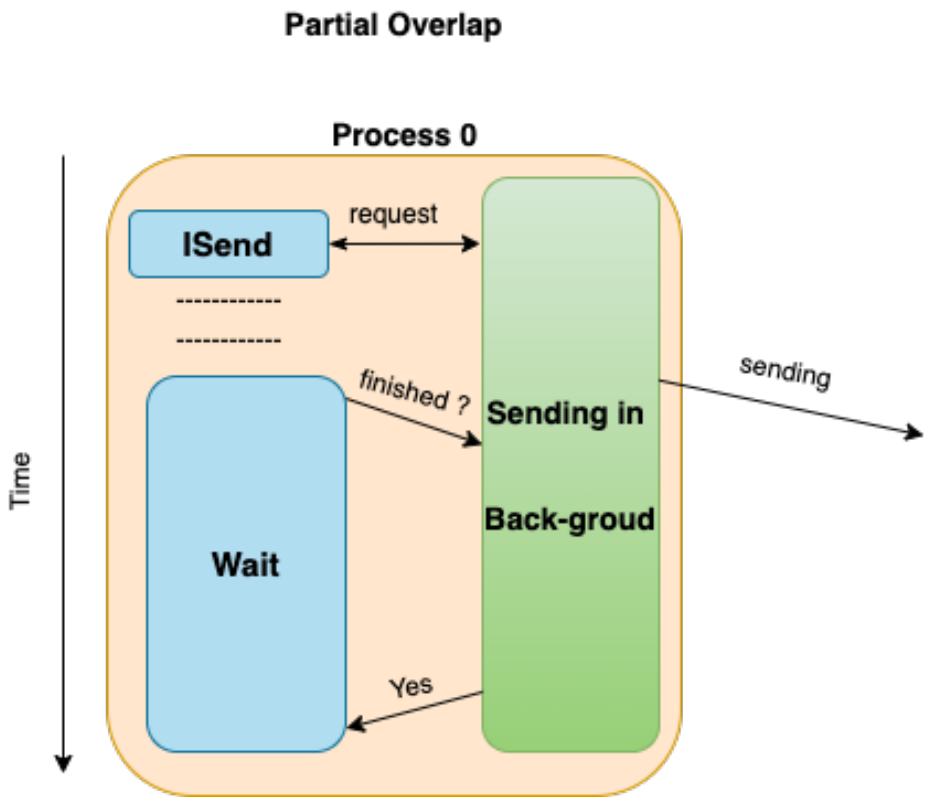
  On Ada, the latency of a communication internode is 1.5 µs, or 4000 processor cycles.

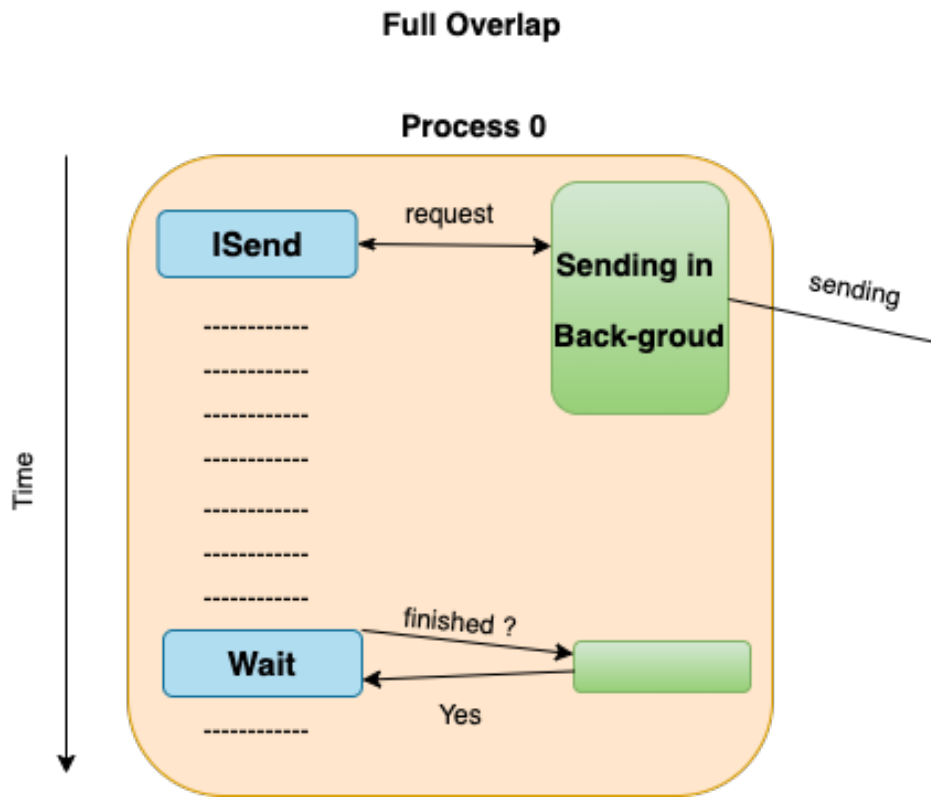
  - It is thus possible, if the hardware and software architecture allows it, to hide all or part of communications costs.
  - The computation-communication overlap can be seen as an additional level of parallelism.
  - This approach is used in MPI by using nonblocking subroutines (i.e.MPI_ISEND(), MPI_IRECV() and MPI_WAIT()).

- **Non blocking communication**

  A nonblocking call returns very quickly but it does not authorize the immediate re-use of the memory space which was used in the communication. It is necessary to make sure that the communication is fully completed (with MPI_WAIT(), for example) before using it again.

Communication Modes





Communication Modes

- Advantages of non blocking mode
  - Possibility of hiding all or part of communications costs (if the architecture allows it)
  - No risk of deadlock
- Disadvantages of non blocking mode
  - Greater additional costs (several calls for one single send or receive, request management)
  - Higher complexity and more complicated maintenance
  - Less efficient on some machines (for example with transfer starting only at the MPI_WAIT() call)
  - Risk of performance loss on the computational kernels (for example, differentiated management between the area near the border of a domain and the interior area, resulting in less efficient use of memory caches)
  - Limited to point-to-point communications (it is extended to collective communications in MPI 3.0)

Communication Modes

Interface of : MPI_ISEND(), MPI_ISSEND(), MPI_IBSEND(), MPI_IRECV()

```
1 MPI_ISEND(values, count, datatype, dest, tag, comm, req, code)
2 MPI_ISSEND(values, count, datatype, dest, tag, comm, req, code)
3 MPI_IBSEND(values, count, datatype, dest, tag, comm, req, code)
4
5 TYPE(*), intent(in) :: values
6
7 integer, intent(in) ::count, datatype, dest, tag, comm
8 integer, intent(out) :: req, code
```

```
1 MPI_IRECV(values, count, msgtype, source, tag, comm, req, code)
2
3 TYPE(*), intent(in) :: values
4 integer, intent(in) :: count, msgtype, source, tag, comm
5
6 integer, intent(out) :: req, code
```

```
1  COMM.Isend(self, data, int dest, int tag=0)
2  COMM.Issend(self, data, int dest, int tag=0)
3  COMM.Ibsend(self, data, int dest, int tag=0)
4
5
6  data = COMM.Irecv(self, source, int tag=0)
7  #or
8  Comm.IRecv(self, buf, int source, int tag=0, Status status=None)
```

Communication Modes

Interface of : MPI_WAIT()

MPI_WAIT() wait for the end of a communication, MPI_TEST() is the nonblocking version.

```
1 MPI_WAIT(req, statut, code)
2 MPI_TEST(req, flag, statut, code)
3
4 integer, intent(inout) :: req
5 integer, dimension(MPI_STATUS_SIZE), intent(out) :: statut
6 integer, intent(out) :: code
7
8 logical, intent(out) :: flag
```

```
1 req = COMM.Isend(self, data, int dest, int tag=0)
2 req.Wait(self, Status status=None)
3 req.Test(self, Status status=None)
```

Communication Modes

Interface of : MPI_WAIT()

MPI_WAITALL() (MPI_TESTALL()) await the end of all communications.

```
1 MPI_WAITALL(taille, reqs, statuts, code)
2 MPI_TESTALL(taille, reqs, statuts, flag, code)
3
4 integer, intent(in) :: count
5 integer, dimension(count) :: reqs
6 integer, dimension(MPI_STATUS_SIZE,count), intent(out) :: statuts
7 integer, intent(out) :: code
8
9 logical, intent(out) :: flag
```

```
1 req = COMM.Isend(self, data, int dest, int tag=0)
2 req.Waitall(type cls, requests, statuses=None)
3 req.Test(type cls, requests, statuses=None)
```

Communication Modes

- **Request management**
  - After a call to a blocking wait function (MPI_WAIT(), MPI_WAITALL(),...), the request argument is set to MPI_REQUEST_NULL.
  - The same for a nonblocking wait when the flag is set to true.
  - A wait call with a MPI_REQUEST_NULL request does nothing.
- Example :

```
1  data = np.ones(1)
2
3  if RANK == 2:
4      data = np.random.random_sample(1)
5      print("I, process 2, I send", data[0], "to process 5")
6      req = COMM.Isend(data, dest=5)
7      req.Wait()
8      req.Test()
9
10 if RANK == 5:
11     req = COMM.Irecv(data, source=2)
12     req.Wait()
13     print("I, process 5, I received ",data[0]," from the process 2.")
```

```
mpirun -n 6 python isendirecv.py

I, process 2, I send 0.7493708552194022 to process 5
I, process 5, I received  0.7493708552194022  from the process 2.
```

Communication Modes

Number of received elements

```
1 MPI_RECV(buf,count,datatype,source,tag,comm,msgstatus,code)
2
3 <type>:: buf
4 integer :: count, datatype
5 integer :: source, tag, comm, code
6
7 integer, dimension(MPI_STATUS_SIZE) :: msgstatus
```

- In MPI_RECV() or MPI_IRECV() call, the count argument in the standard is the number of elements in the buffer buf.
- This number must be greater than the number of elements to be received.
- When it is possible, for increased clarity, it is adviced to put the number of elements to be received.
- We can obtain the number of elements received with MPI_GET_COUNT() and the msgstatus argument returned by the MPI_RECV() or MPI_WAIT() call.

```
1 MPI_GET_COUNT(msgstatus,msgtype,count,code)
2
3 integer, INTENT(IN) :: msgtype
4 integer, INTENT(OUT) :: count, code
5
6 integer, dimension(MPI_STATUS_SIZE), INTENT(IN) :: msgstatus
```