



# Point-to-Point Communication with MPI

Part I: Blocking Communication

# Be Boulder.



University of Colorado **Boulder**



# Outline

- The message-passing paradigm
- Blocking communication
- Send/Receive Syntax
- Sample program
- Deadlock
- Exercises

# Useful MPI References

- General MPI (C++/Fortran):

<https://www.mpich.org/documentation/guides/>

- Mpi4py (Python):

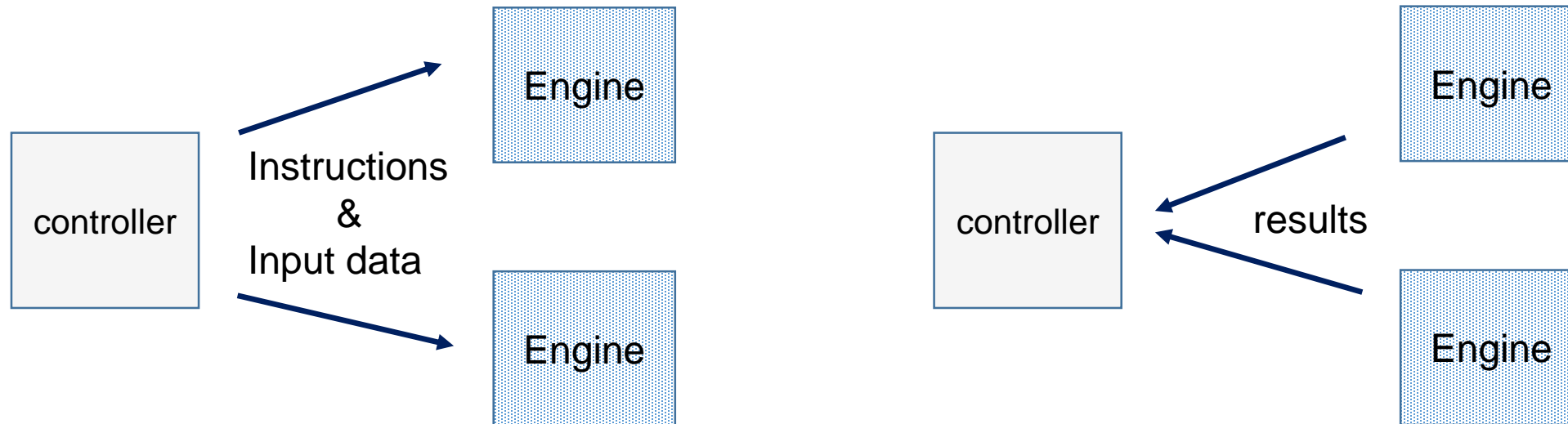
<http://mpi4py.scipy.org/docs/usrman/index.html>

- pdbMPI (R):

<https://cran.r-project.org/web/packages/pbdMPI/index.html>

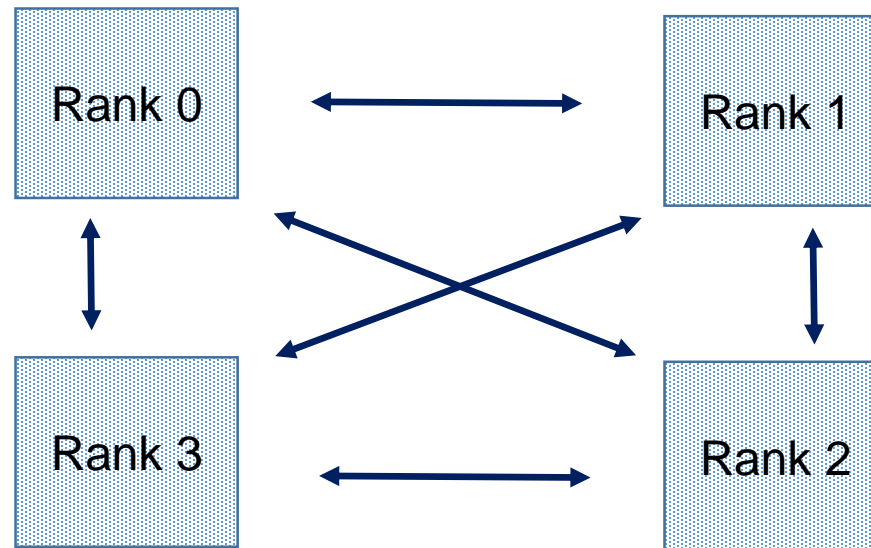
# Engine/Controller Paradigm

- Similar to what you've seen so far
- Master process controls workflow
- Well-suited to collective operations



# Message-Passing Paradigm

- No one is really in control.
- Everyone can communicate with each other
- **Nearly Entire code** runs on each process
- Processes referred to as MPI ranks



# Message passing

- Most natural and efficient paradigm for distributed-memory systems
- Two-sided, **send** and **receive** communication between processes
- Efficiently portable to shared-memory or almost any other parallel architecture:
  - “assembly language of parallel computing” due to universality and detailed, low-level control of parallelism

# More on message passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Sometimes deemed tedious and low-level, but thinking about locality promotes
  - good performance,
  - scalability,
  - Portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

# Sending and Receiving: Questions

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be stored on the receiving process?
- What amount of data is the receiving process prepared to accept?
- Failing to specify these **consistently** between sender and receiver leads to **problems!**



# Blocking Send (General Syntax)

- call MPI\_SEND(  
    message,  
    count,  
    data\_type,  
    destination,  
    tag,  
    communicator,  
    ierr  
)

e.g., my\_partial\_sum,

number of values in message

e.g., MPI\_DOUBLE\_PRECISION,

e.g., myid + 1

some info about msg, e.g., store it

e.g., MPI\_COMM\_WORLD,

error tag (return value)

All arguments are inputs (except ierr).

# Fortran MPI Data Types

MPI\_CHARACTER

MPI\_COMPLEX, MPI\_COMPLEX8, also 16 and 32

MPI\_DOUBLE\_COMPLEX

MPI\_DOUBLE\_PRECISION

MPI\_INTEGER

MPI\_INTEGER1, MPI\_INTEGER2, also 4 and 8

MPI\_LOGICAL

MPI\_LOGICAL1, MPI\_LOGICAL2, also 4 and 8

MPI\_REAL

MPI\_REAL4, MPI\_REAL8, MPI\_REAL16

Numbers = numbers of bytes

Somewhat different in C/C++ (next slide)

# C MPI Datatypes

MPI_CHAR	8-bit character
MPI_DOUBLE	64-bit floating point
MPI_FLOAT	32-bit floating point
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_DOUBLE	64-bit floating point
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_SHORT	16-bit integer
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_WCHAR	Wide (16-bit) unsigned character

# Python and R Data Types

- Abstracted away from user (less tedium!)
- Auto-detected when calling Send/Receive
- In Python, best to work with Numpy Arrays ('int32', 'float32', 'float64', 'i', 'd', etc.)

# Blocking?

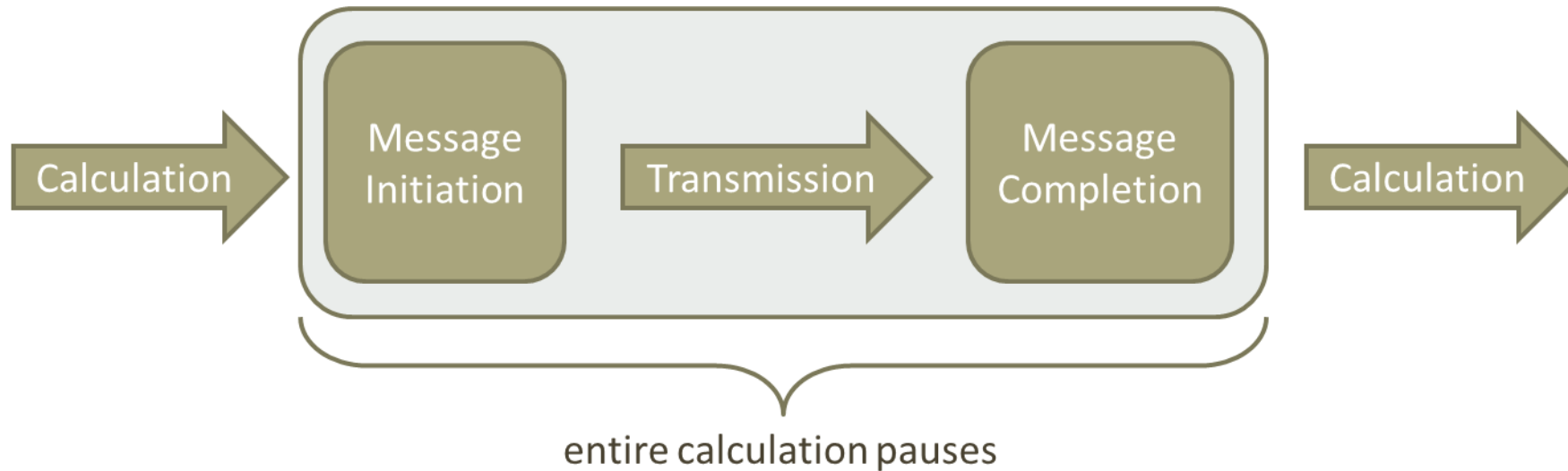
- MPI\_send

- Does not return until the message data and envelope have been buffered in matching receive buffer or temporary system buffer.
- Can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver.
- MPI buffers or not, depending on availability of space
- **Take-Away 1:** small message sends *may not block...*
- **Take-Away 2:** successful completion of the send operation may depend on the occurrence of a matching receive.



# Blocking Communication: Program Flow

- Programs written using blocking sends & receives possess portions similar to schematic below:



# Blocking receive

- Process must wait until message is received to return from call.
- Unlike Send, Recv *always* blocks.
- Stalls progress of program BUT
  - blocking sends and receives enforce process synchronization
  - so enforce consistency of data

# Blocking Receive (General Syntax)

- call MPI\_RECV(  
    message,                   e.g., my\_partial\_sum,  
    count,                    number of values in msg  
    data\_type,                e.g., MPI\_DOUBLE\_PRECISION,  
    source,                   e.g., myid - 1  
    tag,                      some info about msg, e.g., store it  
    communicator,            e.g., MPI\_COMM\_WORLD,  
    status,                   info on size of message received  
    ierr  
)

# The arguments

- outputs: `message`, `status`
- `count * size of data_type` determines size of receive buffer:
  - too large message received gives error,
  - too small message is ok (maybe...won't crash)
- status must be decoded if needed
  - `MPI_Get_Count(status, datatype, ierror)`
  - `status(MPI_SOURCE)`      `status.MPI_SOURCE`
  - `status(MPI_TAG)`      `status.MPI_TAG`
  - `status(MPI_ERROR)`      `status.MPI_ERROR`

# Wildcards

- MPI\_ANY\_SOURCE
- MPI\_ANY\_TAG
  
- Send must send to specific receiver
- Receive can receive from arbitrary sender



# Example Program

- Simple example of point-to-point communication:

`.../Point_to_Point/mpi_messages.{f90,cpp,py,R}`

- Let's examine the source code
- Building the Code (C++/Fortran):
  - `module load intel/psxe-2018u1`
  - `mpi{f90,cc} mpi_messages.{f90,cpp} -o mpi_messages.out`

# Running the Program

- For Fortran/C++:
  - `mpiexec -np 2 ./mpi_messages.out`
- For Python:
  - `mpiexec -np 2 python mpi_messages.py`
- For R:
  - `mpiexec -np 2 Rscript mpi_messages.R`
- Uncomment the appropriate line in `job.sh` and submit the code: `sbatch job.sh`

# Deadlock

## Good Code

```
If (my_rank == 0):  
    send( to rank N)  
    receive( from rank N)  
If (my_rank == N):  
    receive( from rank 0)  
    send( to rank 0)
```

## Bad Code

Communication hangs

```
If (my_rank == 0):  
    send( to rank N)  
    receive( from rank N-1)  
If (my_rank == N):  
    send( to rank 0)  
    receive( from rank 0)
```

*Common source of error and frustration*  
*Many possible sources...*

# Deadlock

## *Quick Exercise:*

Induce a deadlock in mpi\_message by swapping the order of one send/receive pair.

```
If (my_rank == 0):  
    send( to rank N)  
    receive( from rank N)  
If (my_rank == N):  
    receive( from rank 0)  
    send( to rank 0)
```

```
If (my_rank == 0):  
    send( to rank N)  
    receive( from rank N-1)  
If (my_rank == N):  
    send( to rank 0)  
    receive( from rank 0)
```

# Next Up: Hands-on Exercises

- Exercises 1-3: Clearing deadlocks
- Exercise 5: Write your own parallel routine