Parallel Program Design

March 1, 2022

There is no one single way to have the best performance

Foster's methodology

1. Partitioning: divide the computation to be performed and the data operated on by the computation into small tasks.

   The focus here should be on identifying tasks that can be executed in parallel.

We are going to partition it into small tasks

Foster's methodology

2. Communication: determine what communication needs to be carried out among the tasks identified in the previous step.

ex: input & output between different cores

Foster's methodology

3. Agglomeration or aggregation: combine tasks and communications identified in the first step into larger tasks.

   For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.

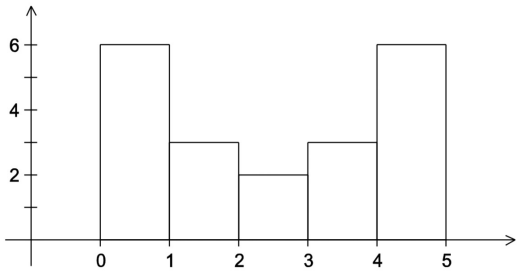
Foster's methodology

4. Mapping: assign the composite tasks identified in the previous step to processes/threads.

   This should be done so that communication is minimized, and each process/thread gets roughly the same amount of work.

Example - histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



we can try to make a serialized program to a parallelized program

Serial program - input

1. The number of measurements: data_count
2. An array of data_count floats: data
3. The minimum value for the bin containing the smallest values: min_meas
4. The maximum value for the bin containing the largest values: max_meas → defines range of numbers
5. The number of bins: bin_count

Serial program - output

1. bin_maxes : an array of bin_count floats – array of floating point number which defines the max
2. bin_counts : an array of bin_count ints – height of bar corresponding to each bin

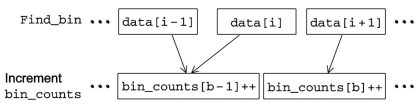
```
1 for (b = 0; b < bincount; b++)
2     bin_maxes[b] = min_meas + bin_width*(b+1);
```

```
1 for (i = 0; i < datacount; i++) {
2     bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
3     bin_counts[bin]++;
4 }
```

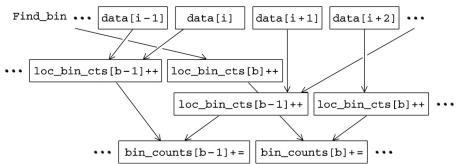
Now to convert it to a parallel program we use Forster's Terminology

9 of 14

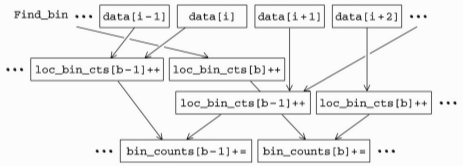
First two stages of Foster's Methodology



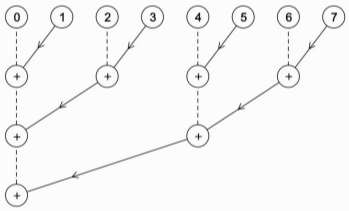
Alternative definition of tasks and communication



Alternative definition of tasks and communication



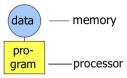
Adding the local arrays



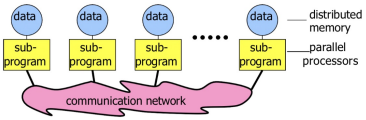
Parallel Programming with MPI

The Message-Passing Programming Paradigm

- **Sequential Programming Paradigm**

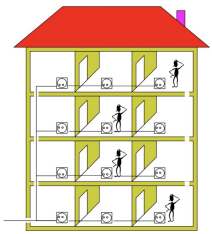


- **Message-Passing Programming Paradigm**



Analogy: Electric Installations in Parallel

- MPI sub-program
  = work of one electrician
  on one floor
- data
  = the electric installation
- MPI communication
  = real communication
  to guarantee that the wires
  are coming at the same
  position through the floor



Concluding Remarks (1)

Serial systems

The standard model of computer hardware has been the von Neumann architecture.

Parallel hardware

Flynn's taxonomy.

Parallel software

We focus on software for homogeneous MIMD systems, consisting of a single program that obtains parallelism by branching.

SPMD programs.

Concluding Remarks (2)

Input and Output

We'll write programs in which one process or thread can access stdin, and all processes can access stdout and stderr.

However, because of nondeterminism, except for debug output we'll usually have a single process or thread accessing stdout.

Concluding Remarks (3)

Performance

Speedup

Efficiency

Amdahl's law

Scalability

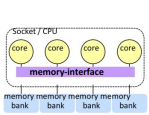
Parallel Program Design

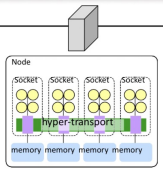
Foster's methodology

Parallel hardware architectures

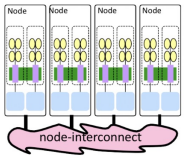
other ways MPI is also used

shared memory





distributed memory



Socket/CPU

→ memory interface

UMA (uniform memory access) SMP (symmetric multi-processing) All cores connected to all memory banks with same speed

Node

→ hyper-transport

ccNUMA (cache-coherent non-uniform memory access)

Shared memory programming is possible

Performance problems:

- Threads should be **pinned** to the physical sockets
- **First-touch** strategy is needed to minimize remote memory access

Cluster

→ node-interconnect

NUMA (non-uniform memory access)

!! fast access only on its own memory !!

Many programming options:

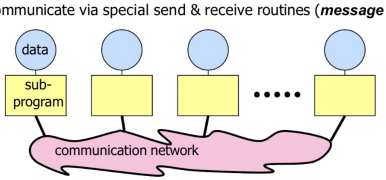
- Shared memory / symmetric multi-processing inside of each node
- distributed memory parallelization on the node interconnect
- **Or simply one MPI process on each core**

Shared memory programming with OpenMP

MPI works everywhere

The Message-Passing Programming Paradigm

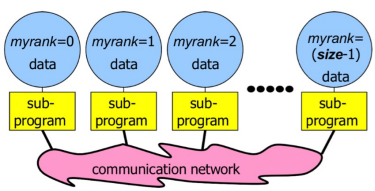
- Each processor in a message passing program runs a ***sub-program:***
  - written in a conventional sequential language, e.g., C, Fortran, or Python
  - typically the same on each processor (SPMD),
  - the variables of each sub-program have
    - **the same name**
    - **but different locations (distributed memory) and different data!**
    - **i.e., all variables are private**
  - communicate via special send & receive routines (***message passing***)



MPI Overview

Data and Work Distribution

- the value of ***myrank*** is returned by special library routine
- the system of ***size*** processes is started by special MPI initialization program (mpirun or mpiexec)
- all distribution decisions are based on ***myrank***
- i.e., which process works on which data




```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[])
{
  int n;    double result;
  int my_rank, num_procs;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

  if (my_rank == 0)
  { printf("Enter the number of elements (n): \n");
    scanf("%d",&n);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  result = 1.0 * my_rank * n;
  printf("I am process %i out of %i handling the %ith part of n=%i elements,result=%f\n",
                  my_rank,   num_procs,      my_rank,      n,          result);
  if (my_rank != 0)
  { MPI_Send(&result,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
  }
  else
  { int rank;
    printf("I'm proc 0: My own result is %f \n",result);
    for (rank=1; rank<num_procs; rank++)
    {
      MPI_Recv(&result,1,MPI_DOUBLE,rank,99,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("I'm proc 0: received result of
        process %i is %f \n", rank, result);
    }
  }
  MPI_Finalize();
}
```

Compiled, e.g., with: `mpicc first-example.c`
Started, e.g., with: `mpiexec -n 4 ./a.out`
Then, this code is running 4 times in parallel !

application-related data

MPI-related data

Now, each process knows who it is: number *my_rank* out of *num_procs* processes

reading the application data *n* from stdin only by process 0

process 0 is sender, all other processes are receivers

broadcasting the content of variable *n* in process 0 into variables *n* in all other processes

doing some **application work** in each process

send to process 0

sending some results from all processes (except 0) to process 0

Process 0: receiving all these messages and, e.g., printing them

receiving the message from process *rank*

```
Enter the number of elements (n): 100
I am process 0 out of 4 handling the 0th part of n=100 elements, result=0.0
I am process 2 out of 4 handling the 2th part of n=100 elements, result=200.0
I am process 3 out of 4 handling the 3th part of n=100 elements, result=300.0
I am process 1 out of 4 handling the 1th part of n=100 elements, result=100.0
I'm proc 0: My own result is 0.0
I'm proc 0: received result of process 1 is 100.0
I'm proc 0: received result of process 2 is 200.0
I'm proc 0: received result of process 3 is 300.0
```

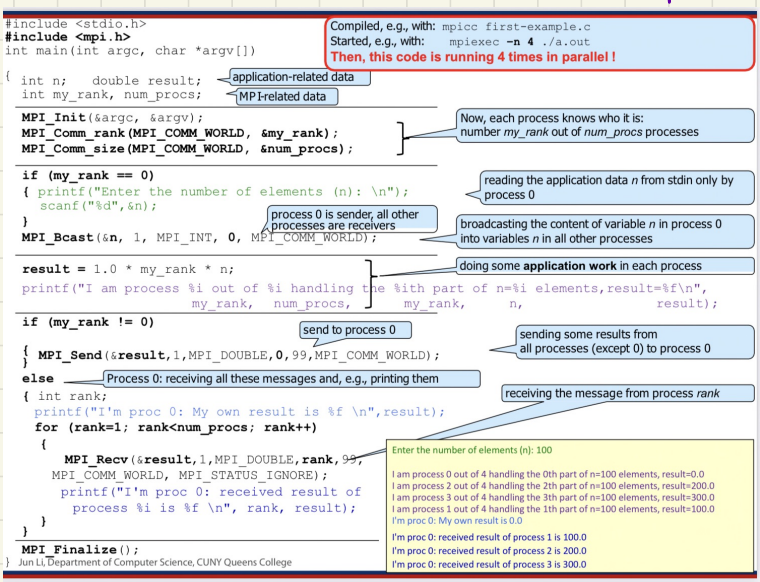
March 3, 2022

We're looking at the equivalent Python version

```
mpirun -n 4 echo "hello"
» hello
» hello
» hello
» hello
```

For different versions of MPI the command will be different. He prefers we use Python

MPI continued



Run mpiexec -n 4 python first-example.py

```python
from mpi4py import MPI

# application-related data
n = None
result = None

comm_world = MPI.COMM_WORLD
# MPI-related data
my_rank = comm_world.Get_rank() # or my_rank = MPI.COMM_WORLD.Get_rank()
num_procs = comm_world.Get_size() # or ditto ...

if (my_rank == 0):
    # reading the application data "n" from stdin only by process 0:
    n = int(input("Enter the number of elements (n): "))

# broadcasting the content of variable "n" in process 0
# into variables "n" in all other processes:
n = comm_world.bcast(n, root=0)

# doing some application work in each process, e.g.:
result = 1.0 * my_rank * n
print(f"I am process {my_rank} out of {num_procs} handling the {my_rank}ith part of n={n}
  elements, result={result}")

if (my_rank != 0):
    # sending some results from all processes (except 0) to process 0:
    comm_world.send(result, dest=0, tag=99)
else:
    # receiving all these messages and, e.g., printing them
    rank = None
    print(f"I'm proc 0: My own result is {result}")
    for rank in range(1,num_procs):
        result = comm_world.recv(source=rank, tag=99)
        print(f"I'm proc 0: received result of process {rank} is {result}")
```

Access

MPI: addresses are the ranks of the MPI processes (sub-programs)

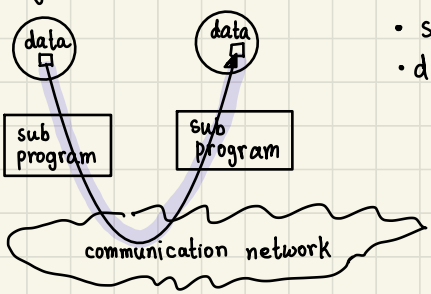
- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - mail box
  - phone line
  - fax machine
  - etc.
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e., all sub-programs of the program) must be started with the MPI startup tool

- unique
- easy to remember
- follows a range of 0 to n-1

Access

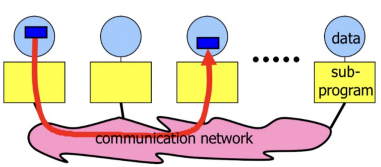
- A sub program needs to be connected to a message passing system
- MPI:
  - sub-program must be linked with an MPI library
  - sub-program must use include file of this MPI library
  - the total program (i.e. all sub-programs of the program) must be started with the MPI startup tool

Messages



- source data size - how much data will you send
- destination buffer size - how much space is available to receive data

Messages



- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process
  - source location
  - source data type
  - source data size
  - receiving process
  - destination location
  - destination data type
  - destination buffer size

i.e., the ranks

basic or derived datatypes

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - mail addresses
  - phone number
  - fax number
  - etc.
- MPI: addresses are ranks of the MPI processes (sub-programs)

Addressing

MPI: addresses are the ranks of the MPI processes (sub-programs)

- unique
- easy to remember
- follows a range of 0 to n-1

- Point to point communication - easiest way to send
  1. Synchronous send
  2. Asynchronous send through buffer

- Simplest form of message passing.
- One process sends a message to another.
- Different types of point-to-point communication:
  - synchronous send
  - buffered = asynchronous send

Synchronous Sends

- The sender gets an information that the message is received.
- Analogue to the *beep* or *okay-sheet* of a fax.

Buffered = Asynchronous Sends

- Only know when the message has left.

Blocking operations

- Non-blocking operations
  - Non-blocking procedures are not the same as sequential subroutine calls
    -the operation may continue while the application executes the next statements
  - You need to specify to the program that the resources have been freed.

Blocking Operations

- Operations are activities, such as
  - sending (a message)
  - receiving (a message)
- Some operations may **block** until another process acts:
  - synchronous send operation **blocks until** receive is posted;
  - receive operation **blocks until** message was sent.
- Relates to the completion of an operation.
- Blocking subroutine returns only when the operation has completed.

Nonblocking Operations

Nonblocking operations consist of:

- A nonblocking procedure call: it returns immediately and allows the sub-program to perform other work
- At some later time the sub-program must ***test*** or ***wait*** for the completion of the nonblocking operation

- All nonblocking procedures must have a matching wait (or test) procedure. (Some system or application resources can be freed only when the nonblocking operation is completed.)
- A nonblocking procedure immediately followed by a matching wait is equivalent to a blocking procedure.
- Nonblocking procedures are not the same as sequential subroutine calls:
  - the operation may continue while the application executes the next statements!

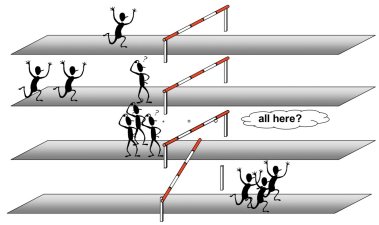
- Collective Communications
  - Several processes involved at once
  - Can be built out of point-to-point communication

Collective Communications

- Collective communication routines are higher level routines.
- Several processes are involved at a time.
- May allow optimized internal implementations, e.g., tree based algorithms.
- Can be built out of point-to-point communications.

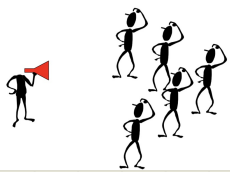
Barriers

- Synchronize processes.



Broadcast

- A one-to-many communication.



Reduction Operations

- Combine data from several processes to produce a single result.

- Reduction operation: combine data from several processes to produce a single result
- Barriers
  - useful if we want to synchronize different processes
  - we specify a barriers, we ask a process to wait at a certain point until all needed processes reach the barrier & then the process can start again

  Ex: this is useful when we may want to load data but not measure that as a part of runtime. we can set up barrier until all the data is loaded to all the processes.

Parallel File I/O

The I/O operations are hard to be parallelized. With more cores it takes more time.

With serial I/O in a parallel program, there is:

1) waste of resources
2) negative side effect for users

Parallel computation → need for parallel I/O
→ do parallel I/O

Process Model & Language Binding - how to setup for MPI program

MPI function format

- you don't want to use object-serialization for send & receive as it has a low performance. You want to use Python.

Initializing MPI : must be done at beginning

Exiting MPI :

- all communication terminated
- you should not expect too much work after this, you might only have return

Starting the MPI program

- mpirun -np number of processes ./executable
- mpiexec -n number of processes ./executable

Communicator MPI_COMM_WORLD

- all sub-programs combine here & communicate

Header files

C
- C / C++

```
#include <mpi.h>
```

Python
- Python

```
from mpi4py import MPI
```

MPI_Init() must be called before any other MPI routine
(only a few exceptions, e.g., MPI_Initialized)

C
- int MPI_Init(int *argc, char ***argv)

```
#include <mpi.h>
int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  ....
```

MPI-2.0 and higher:
Also
MPI_Init(NULL, NULL);

Python
- # MPI.Init()

This call is not needed, because automatically called at the import of MPI at the begin of the program

```
from mpi4py import MPI
# MPI.Init() is not needed
....
```

MPI Function Format

In C and Python: case sensitive

C
- C / C++: error = MPI_Xxxxxx(parameter, ...);
  MPI_Xxxxxx(parameter, ...);

Python
- Python: result_value_or_object = input_mpi_object.mpi_action(parameter, ...)

direct communication of numPy arrays (like in C)	Or with object-serialization:
comm_world = **MPI.COMM_WORLD**	
comm_world.**S**end((snd_buf, ...), ...)	comm_world.**s**end(snd_buf, ...)
comm_world.**R**ecv((rcv_buf, ...), ...)	rcv_buf = comm_world.**r**ecv(...)

Mixed cases:

- MPI procedures in C — MPI_Xxx_mixed
- MPI type declaration — MPI_Xxx_mixed
- MPI constant — MPI_XXX_UPPER

Exiting MPI

C
- C/C++: int MPI_Finalize()

Python
- Python: # MPI.Finalize()

This call is not needed, because automatically called at the end of the program

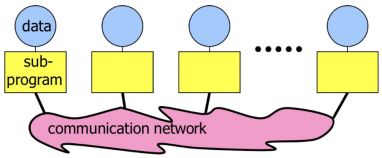
- **Must** be called last by all processes.
- User must ensure the completion of all pending communications (locally) before calling finalize
- After MPI_Finalize:
  - Further MPI-calls are forbidden
  - Especially re-initialization with MPI_Init is forbidden
  - **May** abort the calling process if its rank in MPI_COMM_WORLD is ≠0

Handles

- MPI object
-

Starting the MPI Program

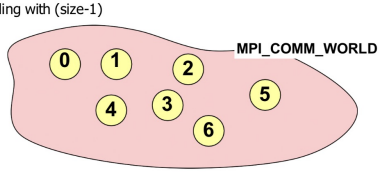
- Start mechanism is implementation dependent
- mpirun –np ***number_of_processes*** ./***executable*** (most implementations)
- mpiexec –n ***number_of_processes*** ./***executable*** (with MPI-2 and later)



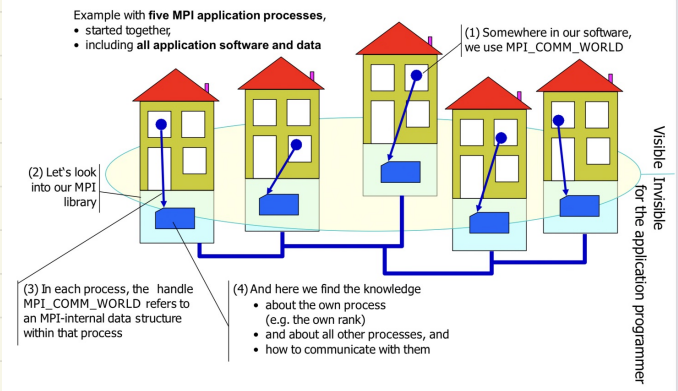
- The parallel MPI processes exist at least after MPI_Init was called.

Communicator MPI_COMM_WORLD

- All processes (= sub-programs) of one MPI program are combined in the **communicator MPI_COMM_WORLD**.
- MPI_COMM_WORLD is a predefined **handle** in
  - mpi.h
  - Each process has its own **rank** in a communicator:
  - starting with 0
  - ending with (size-1)



Handles refer to internal MPI data structures



Handles

- Handles identify MPI objects.
- For the programmer, handles are
  - **predefined constants** in C include file **mpi.h** or **MPI module of mpi4py**
    - Example: MPI_COMM_WORLD or MPI.COMM_WORLD
    - Can be used in initialization expressions or assignments.
    - They are link-time constants, i.e., need not to be compile-time constants.
  - **values returned** by some MPI routines, to be stored in variables, that are defined as
    - C: special MPI typedefs, e.g., MPI_Comm sub_comm;
    - Python: Type of object defined by the creating function, e.g., sub_comm = MPI.COMM_WORLD.Split(…)
- Handles refer to internal MPI data structures

C
Python

Point to Point Communication

Sending a Message

- In C we shouldn't specify the buffer type in the signature since it could be any type

Receiving a Message

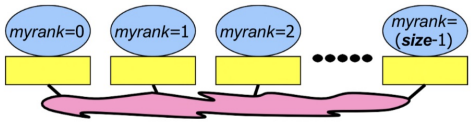
- Format in send & receive is pretty similar

Rank

- The rank identifies different processes.
- The rank is the basis for any work and data distribution.

C
- C/C++: int MPI_Comm_rank(MPI_Comm comm, int *rank)

Python
- Python: *rank* = comm.Get_rank()



Size

- How many processes are contained within a communicator?

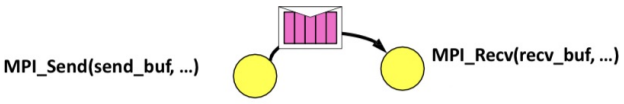
C
- C/C++: int MPI_Comm_size(MPI_Comm comm, int *size)

Python
- Python: *size* = comm.Get_size()

Fortran & C: Interface definitions
- On these slides & in the MPI standard
- You have to write the corresponding procedure **calls**
- E.g., in C:
  MPI_Comm_size (MPI_COMM_WORLD, &size);

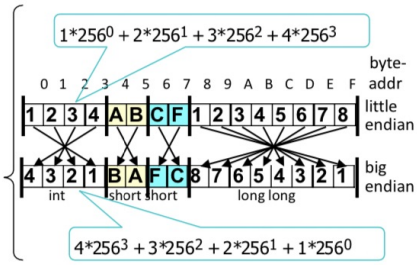
Python: Mix of usage of the interface and typed argument list
- See MPI for Python (mpi4py.github.io), and
  MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io), and
  The API reference (mpi4py.github.io/apiref/index.html)

Major decisions: performance and functionality



Important questions / decisions

- How to address the destination process?
- Which message content?
  - How to handle strided data?
  - Data conversion in inhomogeneous cluster



- Which protocol?
  - When must the **send** be completed?
    - **As soon as possible → buffered protocol → low latency ☺ / bad bandwidth ☹**
    - **Only after the receive is called → synchronous protocol → sending process blocked by receiver → high latency ☹ / good bandwidth ☺**
    - **Application guarantees that corresponding receive is already called → direct protocol → sending process is not blocked / good bandwidth ☺**
    - **Automatic decision → may be blocked / low latency ☺ / good bandwidth ☺**

Messages

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic datatype.
  - Derived datatypes
- Derived datatypes can be built up from basic or derived datatypes.
- Datatype handles are used to describe the type of the data in the memory.

Example: message with 5 integers

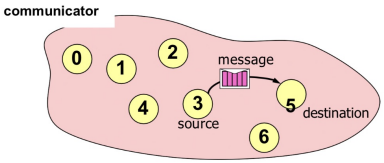
2345	654	96574	-12	7676

Python: Python: messages can be stored in

- Objects → using send(...), recv(...), ... mpi4py routines (Lower-case methods) → slow object serialization
- Buffers as numPy arrays → using Send(...), Recv(...), ... → fast communication (Upper-case methods)

Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator, e.g., MPI_COMM_WORLD.
- Processes are identified by their ranks in the communicator.



Receiving a Message

C

- C/C++: int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Python

- Python: comm.Recv(buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
  obj = comm.recv(buf=None, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)
  buf is only a temporary buffer, deprecated since version 3.0.0

- buf/count/datatype describe the receive buffer.
- Receiving the message sent by process with rank source in comm.
- Envelope information is returned in status.
- On can pass MPI_STATUS_IGNORE instead of a status argument.
- Output arguments are printed blue-cursive.
- **Message matching rule:** receives only if comm, source, and tag match.
- Python: Send requires that the matching receive is a Recv / ditto for send and recv

count, datatype is **not** part of this matching rule

Sending a Message

C

- C/C++: int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

Python

- Python: comm.Send(buf, int dest, int tag=0)
  comm.send(obj, int dest, int tag=0)

- buf is the starting point of the message with count elements, each described with datatype.
- dest is the rank of the destination process within the communicator comm.
- tag is an additional nonnegative integer piggyback information, additionally transferred with the message.
- The tag can be used by the program to distinguish different types of messages.
- Python: – buf must implement the Python buffer protocol, e.g., numPy arrays
  - buf can be buf or (buf, datatype) or (buf, count, datatype)
  - with C datatypes in Python syntax, e.g., MPI.INT, MPI.FLOAT, ...
  – obj is any Python object that can be serialized with the pickle method

Requirements for Point-to-Point Communications

For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Type matching:

```
float sndbuf[n];                          float rcvbuf[n];
MPI_Send(sndbuf, n, MPI_FLOAT,...)        MPI_Recv(rcvbuf, n, MPI_FLOAT,...)
```

  1. Send-buffer's C/Python type must match with the send datatype handle Send
  2. datatype handle must match with the receive datatype handle
  3. Receive datatype handle must match with receive-buffer's C/Python type

- Tags must match → typical usage: **different tags for different data**

```
#define TAG_velocity 111                  #define TAG_velocity 111
MPI_Send( velocity_sndbuf, ... TAG_velocity, ...)   MPI_Recv( velocity_rcvbuf, ... TAG_velocity, ...)
```

  → The velocity message will never be received in, e.g., a temperature array

- Receiver's buffer must be large enough.

Wildcarding

- Receiver can wildcard.
- To receive from any source — source = MPI_ANY_SOURCE
- To receive from any tag — tag = MPI_ANY_TAG
- Actual source and tag are returned in the receiver's *status* parameter.

- With info assertions **New in MPI-4.0**
  - "mpi_assert_no_any_source" = "true" and/or
  - "mpi_assert_no_any_tag" = "true"

  stored on the communicator using MPI_Comm_set_info(),
  - an MPI application can tell the MPI library that it will never use MPI_ANY_SOURCE and/or MPI_ANY_TAG on this communicator

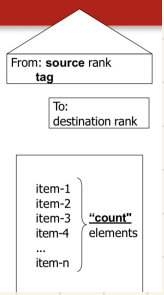
  → may enable lower latencies.
- Other assertions:
  - "mpi_assert_exact_length" = "true" → receive buffer must have exact length
  - "mpi_assert_allow_overtaking" = "true" → message order need not to be preserved

Communication Envelope

- Envelope information is returned from MPI_RECV in *status*.

C
- C/C++: MPI_Status status;
  status.MPI_SOURCE
  status.MPI_TAG
  status.MPI_ERROR

Python
- Python: status.Get_source()
  status.Get_tag(),
  status.Get_error()…



Receive Message Count

C
- C/C++: int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)

Python
- Python: *count* = status.Get_count(Datatype datatype=BYTE)

Caution:

```
buf = np.zeros((100,), dtype=np.double)
comm.Send((buf, 5, MPI.DOUBLE), ....)
comm.Recv((buf, 100, MPI.DOUBLE), ...., status)
count = status.Get_count(MPI.DOUBLE) # → 5
count = status.Get_count() # → 40
```

Communication Modes

- Send communication modes:
  - synchronous send → MPI_**S**SEND
  - buffered [asynchronous] send → MPI_**B**SEND
  - standard send → MPI_**SEND**
  - Ready send → MPI_**R**SEND
- Receiving all modes → MPI_**RECV**

Communication Modes — Definitions

Sender mode	Definition	Notes
Synchronous send **MPI_SSEND**	Only completes when the receive has started	
Buffered send **MPI_BSEND**	Always completes (unless an error occurs), irrespective of receiver	needs application-defined buffer to be declared with MPI_BUFFER_ATTACH
Standard send **MPI_SEND**	Either synchronous or buffered	uses an internal buffer
Ready send **MPI_RSEND**	May be started **only** if the matching receive is already posted!	highly dangerous!
Receive **MPI_RECV**	Completes when a message has arrived	same routine for all communication modes

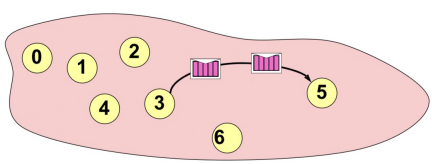
Rules for the communication modes

- Standard send (**MPI_SEND**)
  - minimal transfer time
  - may block due to synchronous mode

  → all risks of synchronous send
- Synchronous send (**MPI_SSEND**)
  - risk of deadlock
  - risk of serialization
  - risk of waiting —> idle time
  - high latency / best bandwidth
- Buffered send (**MPI_BSEND**)
  - low latency / bad bandwidth
- Ready send (**MPI_RSEND**)
  - use **never**, except you have a *200% guarantee* that Recv is already called in the current version and all future versions of your code,
  - may be the fastest

Message Order Preservation

- Rule for messages on the same connection, i.e., same communicator, source, and destination rank:
- **Messages do not overtake each other.**
- This is true even for non-synchronous sends.



- If both receives match both messages, then the order is preserved.

Blocking Routines -> Risk of Deadlocks & Serializations

For cyclic boundary:

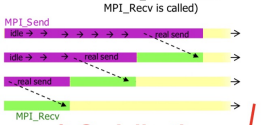
```
MPI_Send(..., right_rank, ...)
MPI_Recv(   ..., left_rank,  ...)
```

→ **Deadlock**

(If the MPI library chooses the synchronous protocol, i.e. MPI_Send waits until MPI_Recv is called)

For non-cyclic boundary:

```
if (myrank < size-1)
  MPI_Send(..., left, ...);
if (myrank > 0)
  MPI_Recv(    ..., right,  ...);
```

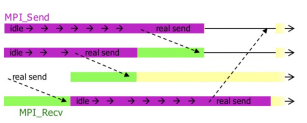


→ **Serialization**

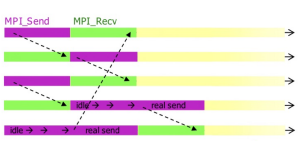
(If the MPI library chooses the synchronous protocol)

Cyclic communication – other bad ideas

```
if (myrank < size-1)
  { MPI_Send(..., left, ...);
  MPI_Recv(    ..., right,  ...);
}else {
  MPI_Recv(    ..., right,  ...);
  MPI_Send(..., left, ...);
}
```




```
if (myrank%2 == 0)
  { MPI_Send(..., left, ...);
  MPI_Recv(    ..., right,  ...);
}else {
  MPI_Recv(    ..., right,  ...);
  MPI_Send(..., left, ...);
}
```



→ **Serialization**

(If the MPI library chooses the synchronous protocol)

Non-Blocking Communications

Separate communication into **three phases**:

- Initiate nonblocking communication
  - returns immediately
  - routine name starting with MPI_**I**...

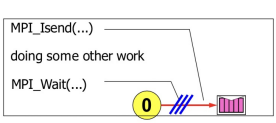
  → it is local,
  i.e., it returns independently of any other process' activity

"I" stands for
- Immediately (=local) and
- Incomplete (=nonblocking[1])

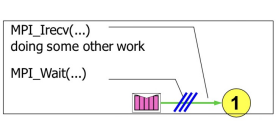
- Do some work (perhaps involving other communications?)
- Wait for nonblocking communication to **complete**, i.e.,
  - the send buffer is read out, or
  - the receive buffer is filled in

Non-Blocking Examples

- Nonblocking **send**



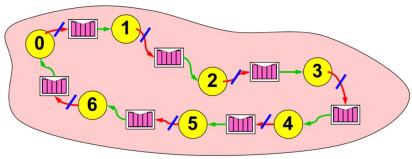
- Nonblocking **receive**



/// = waiting until operation locally completed

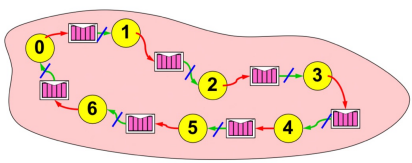
Non-Blocking Send

- Initiate nonblocking send
  → in the ring example: Initiate nonblocking send to the right neighbor
- Do some work:
  → in the ring example: Receiving the message from left neighbor
- Now, the message transfer can be completed
- Wait for nonblocking send to complete

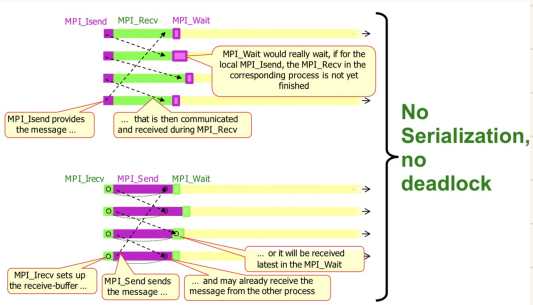


Non-Blocking Receive

- Initiate nonblocking receive
  → in the ring example: Initiate nonblocking receive from left neighbor
- Do some work:
  → in the ring example: Sending the message to the right neighbor
- Now, the message transfer can be completed
- Wait for nonblocking receive to complete



Timelines for both solutions



Request Handles

Request handles

- are used for nonblocking communication
- **must** be stored in local variables
  - in C/C++: MPI_Request
  - in Python: automatically

the value
- **is generated** by a nonblocking communication routine
- **is used** (and freed) in the MPI_WAIT routine

Nonblocking Synchronous Send

- C/C++: MPI_Issend(&buf, count, datatype, dest, tag, comm, [OUT] &*request_handle*);

  MPI_Wait([INOUT] &request_handle, &*status*);
- Python: *request* = comm_world.Issend(...) / status = MPI.Status(); request.Wait(*status*)

- buf must not be modified between Issend and Wait (in all progr. languages)
  (In MPI-2.1, this restriction was stronger: "should not access", see MPI-2.1, page 52, lines 5-6)
- "Issend + Wait directly after Issend" is equivalent to blocking call (Ssend)
- Nothing returned in status (because send operations have no status)

Nonblocking Receive

- C/C++: MPI_Irecv (*buf*, count, datatype, source, tag, comm, [OUT] &*request_handle*);

  MPI_Wait([INOUT] &request_handle, &*status*);
- Python: *request* = comm_world.Irecv(...) / status = MPI.Status(); request.Wait(*status*)

- buf must not be used between Irecv and Wait (in all progr. languages)
- Message status is returned in Wait

Blocking and Non-Blocking

- Send and receive can be blocking or nonblocking.
- A blocking send can be used with a nonblocking receive, and vice-versa.
- Nonblocking sends can use any mode
  - standard – MPI_ISEND
  - synchronous – MPI_ISSEND
  - buffered – MPI_IBSEND
  - ready – MPI_IRSEND
- Synchronous mode affects completion, i.e. MPI_Wait / MPI_Test, not initiation, i.e., MPI_I....

Completion

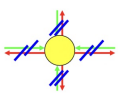
- C/C++:
  MPI_Wait(&request_handle, &*status*);
  MPI_Test(&request_handle, &*flag*, &*status*);
- Python:
  status = MPI.Status(); request_handle.Wait(*status*)
  status = MPI.Status(); *flag* = request_handle.Test(*status*)

- one must
  - WAIT or
  - loop with TEST until request is completed,
    i.e., flag == non-zero (C) or True (Python)

Multiple Non-Blocking Communications

You have several request handles:

- Wait or test for completion of **one** message
  - **MPI_Waitany / MPI_Testany**
- Wait or test for completion of **all** messages
  - **MPI_Waitall / MPI_Testall *)**
- Wait or test for completion of **at least one** messages
  - **MPI_Waitsome / MPI_Testsome *)**



*) Each status contains an additional error field.
This field is only used if MPI_ERR_IN_STATUS is returned (also valid for send operations).

Other MPI features: Send-Receive in one routine

- MPI_Sendrecv & MPI_Sendrecv_replace
  - Combines the triple "MPI_Irecv + Send + Wait" into one routine

New in MPI-4.0

- Nonblocking MPI_Isendrecv & MPI_Isendrecv_replace
  - Whereas blocking MPI_Sendrecv was used to prevent
    - **serializations and**
    - **deadlocks,**
  - the nonblocking MPI_Isendrecv can be used, e.g., to parallelize the existing communication calls in multiple directions → e.g., to minimize idle times if only some neighbors are delayed

Performance options

Which is the fastest neighbor communication?
- MPI_Irecv + MPI_Send
- MPI_Irecv + MPI_Isend
- MPI_Isend + MPI_Recv
- MPI_Isend + MPI_Irecv
- MPI_Sendrecv

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming but efficiency of MPI application-programming is not portable!

Use cases for nonblocking operations

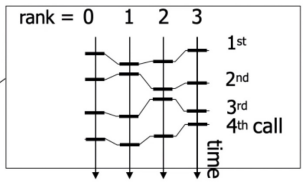
- To prevent serializations and deadlocks
  (as if overlapping of communication with other communication)
- Real overlapping of
  - several communications
  - communication and computation

Internally: tree based algorithm

March 15, 2022

Characteristics of Collective Communication

- Collective action over a communicator.
- All process of the communicator must communicate, i.e., must call the collective routine.
- On a given communicator, the n-th collective call must match on all processes of the communicator.



- In MPI-1.0 – MPI-2.2, all collective operations are blocking. Nonblocking versions since MPI-3.0.
- No tags.

very important

- For each message, the amount of data sent must exactly match the amount of data specified by the receiver

  → It is forbidden to provide receive buffer count arguments that are too long (and also too short, of course)

  Exception with Python (mpi4py): if a buffer argument represents #processes of messages (e.g. snd_buf in comm.Scatter) and the argument count is to be derived from the buffer argument (i.e. is not explicitly defined in the argument list), then this count argument is derived from the inferred number of elements of the buffer divided by the size of the communicator.

e.g., when passing snd_buf, or (snd_buf, datatype).
For buffer options such as BufSpec, BufSpecV, …, see e.g. mpi4py/typing.pyi at master · mpi4py/mpi4py (github.com)

Barrier Synchronization

C

- C/C++: int MPI_Barrier(MPI_Comm comm)

Python

- Python: comm.Barrier() or comm.barrier()

- MPI_Barrier is normally never needed:
  - all synchronization is done automatically by the data communication:
    - **a process cannot continue before it has the data that it needs.**
  - if used for debugging:
    - **please guarantee, that it is removed in production.**
  - for profiling: to separate time measurement of
    - **Load imbalance of computation [MPI_Wtime(); MPI_Barrier(); MPI_Wtime()]**
    - **communication epochs [MPI_Wtime(); MPI_Allreduce(); …; MPI_Wtime()]**

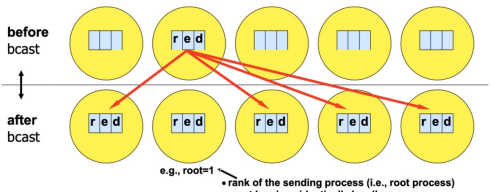
Broadcast

C

- C/C++: int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

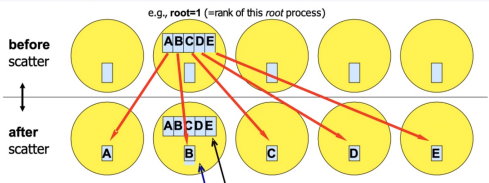
Python

- Python: comm.Bcast(buf, int root=0) or comm.bcast(obj, int root=0)



Example:
MPI_Bcast(buf, 3, MPI_CHAR, 1, MPI_COMM_WORLD);

Scatter



C

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

Python

comm.Scatter(sendbuf *or* None, recvbuf, int root=0)
recvobj = comm.scatter(sendobj *or* None, int root=0)

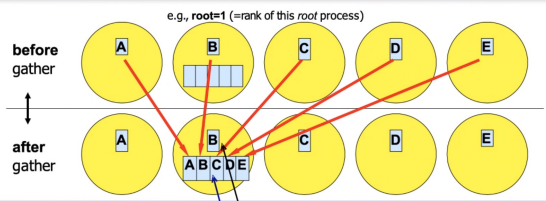
See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

sendcount describes only one message

Example: MPI_Scatter(sbuf, 1, MPI_CHAR, *rbuf*, 1, MPI_CHAR, 1, MPI_COMM_WORLD);

Completely ignored at all processes except *root*

Gather



C

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

Python

comm.Gather(sendbuf, recvbuf *or* None, int root=0)
recvobj = comm.gather(sendobj, int root=0)

See, e.g., Tutorial — MPI for Python 3.1.1 documentation (mpi4py.readthedocs.io)

CALL MPI_Gather(sbuf, 1, MPI_CHARACTER, *rbuf*, 1, MPI_CHARACTER, 1, MPI_COMM_WORLD);

Completely ignored at all processes except *root*

recvcount describes only one message

Global Reduction Operations

- To perform a global reduce operation across all members of a group.
- $d_0 \circ d_1 \circ d_2 \circ d_3 \circ \dots \circ d_{s-2} \circ d_{s-1}$
  - d_i = data in process rank i
    - **single variable, or**
    - **vector**
  - $\circ$ = associative operation
  - Example:
    - **global sum or product**
    - **global maximum or minimum**
    - **global user-defined operation**
- floating point rounding may depend on usage of associative law:
  - $[(d_0 \circ d_1) \circ (d_2 \circ d_3)] \circ [\dots \circ (d_{s-2} \circ d_{s-1})]$
  - $((((((d_0 \circ d_1) \circ d_2) \circ d_3) \circ \dots) \circ d_{s-2}) \circ d_{s-1})$
  - May be even worse through partial sums in each process:
    $\bigcirc_{i=0}^{n-1} x_i \rightarrow [[[(\bigcirc_{i=0}^{n/s-1} x_i \circ \bigcirc_{i=n/s}^{2n/s-1} x_i) \circ (\dots \circ \dots)] \circ [\dots \circ (\dots \circ \dots)]]]$

E.g., with $n=10^8$ rounding errors may modify last 3 or 4 digits!

Example of Global Reduction

- Global integer sum.
- Sum of all inbuf values should be returned in *resultbuf.*

C

- C/C++: root=0;
  MPI_Reduce(&inbuf, &*resultbuf*, 1, MPI_INT,
  MPI_SUM, root, MPI_COMM_WORLD);

Python

- Python: comm_world = MPI.COMM_WORLD
  snd_buf = np.array(value, dtype=np.intc)
  resultbuf = np.empty((), dtype=np.intc)
  comm_world.Reduce(snd_buf, *resultbuf*, op=MPI.SUM)
- The result is only placed in *resultbuf* at the root process.

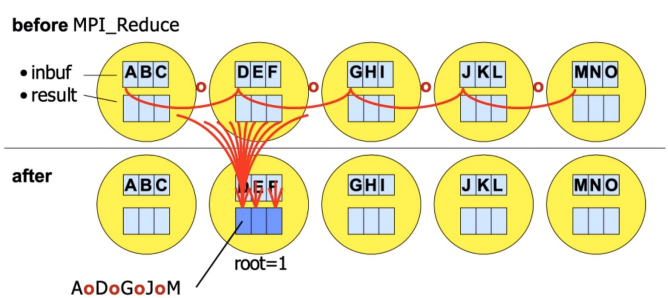
op=MPI.SUM and root=0 are defaults

The buffer can be added up & this way we can use reduce function for sum

Predefined Reduction Operation Handles

Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_Reduce



Reduce can be applied to arrays & vectors.

User-Defined Reduction Operations

- Operator handles
  - predefined
    - see table above
  - user-defined
- User-defined operation ▫:
  - associative
  - user-defined function must perform the operation vector_A ▫ vector_B
  - syntax of the user-defined function → MPI standard
- Registering a user-defined reduction function:

C

  - C/C++: MPI_Op_create(MPI_User_function *func, int commute, MPI_Op *op)

Python

  - Python: op = MPI.Op.Create(func, commute=True *or* False)

- COMMUTE tells the MPI library whether FUNC is commutative.

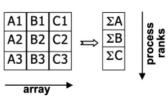
As a programmer you must ensure the function follows associative property.
It has two inputs

Variants of Reduction Operations

- MPI_Allreduce
  - no root,
  - returns the result in all processes

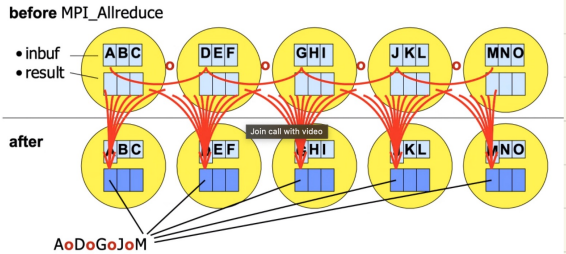
New in MPI-2.2

- MPI_Reduce_scatter_block and MPI_Reduce_scatter
  - result vector of the reduction operation is scattered to the processes into the real result buffers

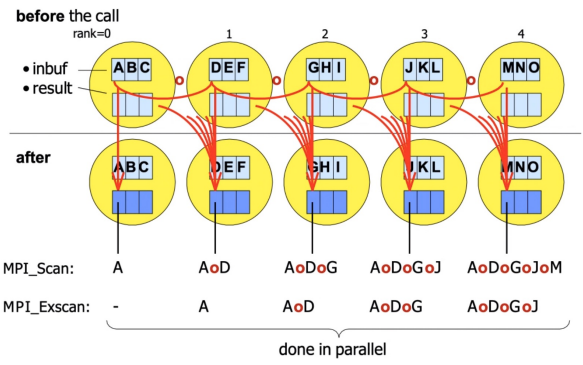


- MPI_Scan
  - prefix reduction
  - result at process with rank i := reduction of inbuf-values from rank 0 to rank i
- MPI_Exscan
  - result at process with rank i := reduction of inbuf-values from rank 0 to rank i-1

MPI_Allreduce



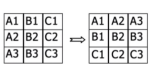
MPI_Scan and MPI_Exscan



Other Collective Communication Routines

- MPI_Allgather → similar to MPI_Gather, but all processes receive the result vector

- MPI_Alltoall → each process sends messages to all processes



- MPI_.....v (Gatherv, Scatterv, Allgatherv, Alltoallv, Alltoallw)
  - → Each message has a different count and displacement
  - → array of counts and array of displs (Alltoallw: also array of types)
  - → interface does **not scale** to thousands of MPI processes!
  - → Recommendation: One should try to use data structures with same communication size on all ranks.

Nonblocking Collective Communication Routines

New in MPI-3.0

MPI_I..... **Nonblocking** variants of all collective communication:
MPI_Ibarrier, MPI_Ibcast, ...

- **Nonblocking** collective operations do **not match** with **blocking** collective operations
- Collective initiation and completion are separated

With point-to-point message passing, such matching is allowed

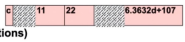
- **MPI_I...** calls are **local** (i.e., not synchronizing), whereas the **corresponding MPI_Wait** collectively **synchronizes** in same way as corresponding blocking collective procedure
- May have multiple outstanding collective communications on same communicator
- Ordered initialization on each communicator

You can have communication with overlapping computations.

MPI Datatypes

March 17, 2022

- In previous slides:
  - A messages was a contiguous sequence of elements of basic types:
  - buf, count, datatype_handle
- New goals in this part:
  - Transfer of any data in memory in one message
    - **Strided data (portions of data with holes between the portions)**
    - **Various basic datatypes within one message**



  - No multiple messages → **no multiple latencies** → requires initialization & terminal so additional overhead.
  - No copying of data into contiguous scratch arrays → **no waste of memory bandwidth**
- Method: **Datatype handles**
  - Memory layout of send / receive buffer
  - Basic types / **derived types**:
    - **vectors**
    - **subarrays**
    - **structs**
    - **others**

Message passing:
- **Goal and reality may differ !!!**

Parallel file I/O:
- Derived datatypes are **important** to express I/O patterns

In order to squeeze different datatypes in one message we need a user defined datatype.

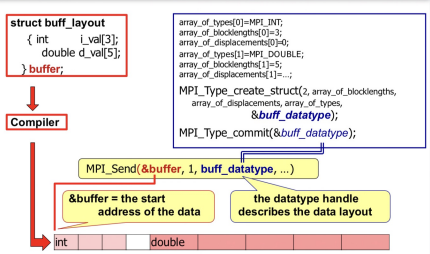
Data Layout & the Describing Datatype Handle

- the memory is a multiple of 8, we may need to keep a hole before adding anything else in memory.
- in MPI to use a custom datatype you need a create_struct & commit to use it

MPI_Type_Create_struct(2, array_of_blocklengths, array_of_displacements, &buff_datatype)

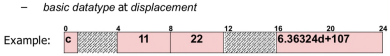
- 2 → num types of data
- array_of_blocklengths → sum of size of arrays
- array_of_displacements → where does it start in memory → 0
- &buff_datatype → array_of_types → array that holds all the types

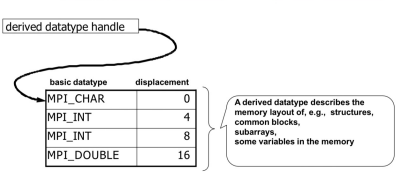
Data Layout and the Describing Datatype Handle



Derived Datatypes — Type Maps

- A derived datatype is logically a pointer to a list of entries:
  - *basic datatype* at *displacement*





Derived Datatypes - Type Maps

- we need to describe the offset
- it is logically a pointer to a list of entries

Contiguous Data

After you create a datatype you must commit it

After you finish using the derived datatype - you have to free up the space.

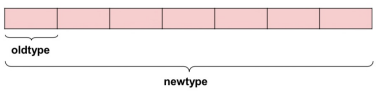
Vector Datatype

Example:

★ why is there a gap in the middle?
why is the stride 16?

Contiguous Data

- The simplest derived datatype
- Consists of a number of contiguous items of the same datatype



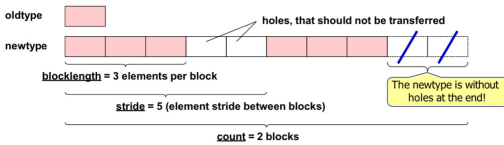
C

- C/C++: int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)

Python

- Python: newtype = oldtype.Create_contiguous(int count)

Vector DataType



C

- C/C++: int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Python

- Python: newtype = oldtype.Create_vector(int count, int blocklength, int stride)

Committing and Freeing a Datatype

- Before a datatype handle is used in message passing communication, **it needs to be committed with MPI_TYPE_COMMIT.**
- This need be done only once (by each MPI process). (Using more than once ☹ corresponds to additional no-operations.)

C

- C/C++: int MPI_Type_commit(MPI_Datatype *datatype);

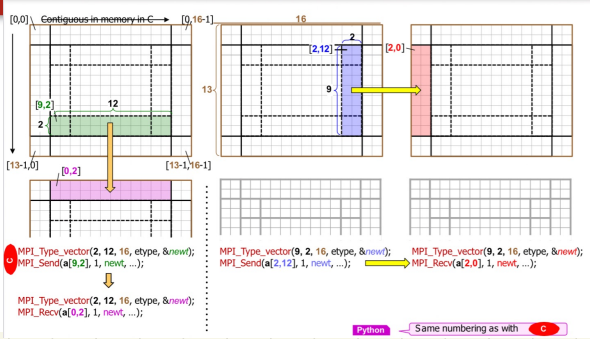
Python

- Python: datatype.Commit()

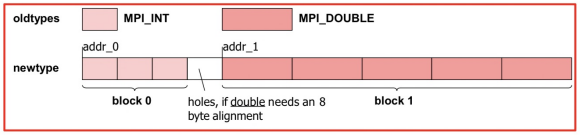
IN-OUT argument (although handle is not modified)

- If usage is over, one may call MPI_TYPE_FREE() to free a datatype and its internal resources.

Example with MPI_Type_vector



Struct Datatype



C

- C/C++: int MPI_Type_create_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types, MPI_Datatype *newtype)

Python

- Python: newtype = MPI.Datatype.Create_struct(array_of_blocklengths, array_of_displacements, array_of_types)

count = 2
array_of_blocklengths = (3, 5)
array_of_displacements = (0, addr_1 – addr_0) 1)
array_of_types = (MPI_INT, MPI_DOUBLE)

1) Via MPI_Get_address and MPI_Aint_diff, see following slides

How to compute the displacement (1)

- array_of_displacements[i] := address(block_i) – address(block_0)

Retrieve an absolute address:

C

– C/C++: int MPI_Get_address(void* location, MPI_Aint *address)

Python

– Python: address = MPI.Get_address(location)

How to compute the displacement (2)

New in MPI-3.1

Relative displacement:= absolute address 1 – absolute address 2

C

– C/C++: MPI_Aint MPI_Aint_diff(MPI_Aint addr1, MPI_Aint addr2)

Python

– Python: int MPI.Aint_diff(addr1, addr2)

Python's int allows 64 bit

New in MPI-3.1

New absolute address := existing absolute address + relative displacement:

C

– C/C++: MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)

Python

– Python: int MPI.Aint_add(base, disp)

Advice to users. Users are cautioned that displacement arithmetic can overflow in variables of type MPI_Aint and result in unexpected values on some platforms. The MPI_AINT_ADD and MPI_AINT_DIFF functions can be used to safely perform address arithmetic with MPI_Aint displacements. (*End of advice to users.*)

Example for array_of_displacements[i] := address(block_i) – address(block_0)

C

```
struct buff
   {     int i[3];
         double d[5];
   } snd_buf;
MPI_Aint iaddr0, iaddr1, disp;
MPI_Get_address( &snd_buf.i[0], &iaddr0);  // the address value &snd_buf.i[0] is stored into variable iaddr0
MPI_Get_address(&snd_buf.d[0], &iaddr1);
disp = MPI_Aint_diff(iaddr1, iaddr0);                    // MPI-3.0 & former: disp = iaddr1–iaddr0
```

New in MPI-3.1

Python

```
np_dtype = np.dtype([('i', np.intc, 3), ('d', np.double, 4)])
snd_buf = np.empty((),dtype=np_dtype)
addr0 = MPI.Get_address(snd_buf['i'])
addr1 = MPI.Get_address(snd_buf['d'])
disp = MPI.Aint_diff(addr1, addr0)
```

How to calculate displacement (1+2) → formulas here

- difference between the current type of data & the first type of data

In practice,

in Python there is no native struct

We can calculate difference

Scope & Performance Options

- There is no guarantee of performance

Scope of MPI derived datatypes:

- Fixed memory layout
- but not a linked list/tree,
  i.e., if the location of data portions depend on data (pointers/indexes) in this list

→ C++ data structures often require external libraries for flattening such data

→ E.g., Boost serialization methods

Which is the fastest neighbor communication with strided data?

- **Copying** the strided data in a contiguous scratch send-buffer, communicating this send-buffer into a contiguous recv-buffer, and copying the recv-buffer back into the strided application array
- Using derived datatype handles
- And which of the communication routines should be used?

No answer by the MPI standard, because:

MPI targets portable and efficient message-passing programming
but
efficiency of MPI application-programming is not portable!

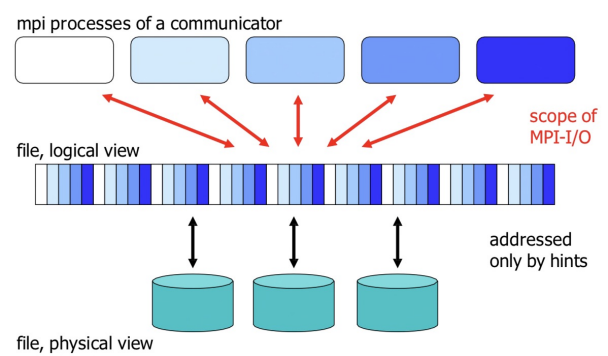
Parallel I/O

MPI-I/O Features

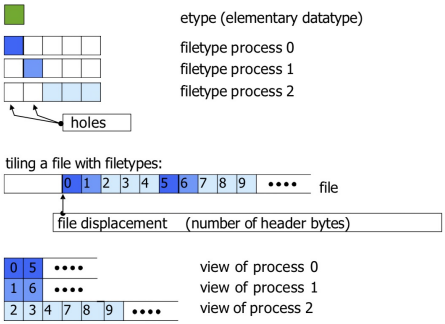
MPI-I/O Features

- Provides a high-level interface to support
  - data file partitioning among processes
  - transfer global data between memory and files (collective I/O)
  - asynchronous transfers
  - strided access
- MPI derived datatypes used to specify common data access patterns for maximum flexibility and expressiveness

Logical View / Physical View



Definitions



file - an ordered collection of typed data items

etypes
- is the unit of data access and positioning / offsets
- can be any basic or derived datatype
  (with non-negative, monotonically non-decreasing, non-absolute displacem.)
- generally contiguous, but need not be
- typically same at all processes

filetypes
- the basis for partitioning a file among processes
- defines a template for accessing the file
- different at each process
- the etype or derived from etype (displacements: non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)

view
- each process has its own view, defined by: a displacement, an etype, and a filetype.
- The filetype is repeated, starting at **displacement**

offset - position relative to current view, in units of etype

Opening an MPI filename

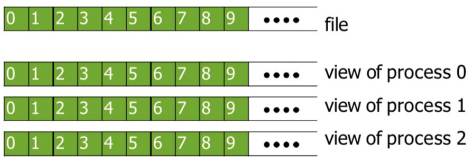
- Same filename could correspond to the same data

Default View

```
MPI_File_open(comm, filename, amode, info, fh)
```

- Default:
  - displacement = 0
  - etype = MPI_BYTE
  - filetype = MPI_BYTE

  } each process has access to the whole file



- Sequence of MPI_BYTE matches with any datatype
- Binary I/O (no ASCII text I/O)

Closing and Deleting a File

- Close: collective

```
MPI_File_close(fh)
```

- Delete:
  - automatically by MPI_FILE_CLOSE if **amode=MPI_DELETE_ON_CLOSE | ...** was specified in MPI_FILE_OPEN
  - deleting a file that is not currently opened:

```
MPI_File_delete(filename, info)
```

[*same implementation-dependent rules as in MPI_FILE_OPEN*]

Access Modes

- same value of **amode** on all processes in **MPI_File_open**
- Bit vector OR of integer constants
  - MPI_MODE_RDONLY - read only
  - MPI_MODE_RDWR - reading and writing
  - MPI_MODE_WRONLY - write only
  - MPI_MODE_CREATE - create if file doesn't exist
  - MPI_MODE_EXCL - error creating a file that exists
  - MPI_MODE_DELETE_ON_CLOSE - delete on close
  - MPI_MODE_UNIQUE_OPEN - file not opened concurrently
  - MPI_MODE_SEQUENTIAL - file only accessed sequentially: mandatory for sequential stream files (pipes, tapes, ...)
  - MPI_MODE_APPEND - all file pointers set to end of file [*caution: reset to zero by any subsequent MPI_FILE_SET_VIEW*]

Set/Get File View

- Set view
  - changes the process's view of the data
  - local and shared file pointers are reset to zero
  - collective operation
  - etype and filetype must be committed
  - datarep argument is a string that specifies the format in which data is written to a file: "native", "internal", "external32", or user-defined
  - same etype extent and same datarep on all processes
- Get view
  - returns the process's view of the data

```
MPI_File_set_view(fh, disp, etype, filetype, datarep, info)
MPI_File_get_view(fh, disp, etype, filetype, datarep)
```

File Views

- Provides a visible and accessible set of data from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view

Data Representation

- Data is represented the same way in the underlying structure
- can be user defined

- "native"
  - data stored in file identical to memory
  - on homogeneous systems no loss in precision or I/O performance due to type conversions
  - on heterogeneous systems loss of interoperability
  - no guarantee that MPI files accessible from C/Fortran
- "internal"
  - data stored in implementation specific format
  - can be used with homogeneous or heterogeneous environments
  - implementation will perform type conversions if necessary
  - no guarantee that MPI files accessible from C/Fortran

- "external32"
  - follows standardized representation (IEEE)
  - all input/output operations are converted from/to the "external32" representation
  - files can be exported/imported between different MPI environments
  - due to type conversions from (to) native to (from) "external32" data precision and I/O performance may be lost
  - "internal" may be implemented as equal to "external32"
  - can be read/written also by non-MPI programs
- user-defined

No information about the default,
i.e., datarep without MPI_File_set_view() is not defined

All Data Access Routines

positioning	synchronism	coordination		
		noncollective	collective	split collective
explicit offsets	blocking	READ_AT	READ_AT_ALL	READ_AT_ALL_BEGIN
		WRITE_AT	WRITE_AT_ALL	READ_AT_ALL_END
	nonblocking	IREAD_AT	IREAD_AT_ALL	WRITE_AT_ALL_BEGIN
		IWRITE_AT	IWRITE_AT_ALL	WRITE_AT_ALL_END
individual file pointers	blocking	READ	READ_ALL	READ_ALL_BEGIN
		WRITE	WRITE_ALL	READ_ALL_END
	nonblocking	IREAD	IREAD_ALL	WRITE_ALL_BEGIN
		IWRITE	IWRITE_ALL	WRITE_ALL_END
shared file pointer	blocking	READ_SHARED	READ_**ORDERED**	READ_**ORDERED**_BEGIN
		WRITE_SHARED	WRITE_**ORDERED**	READ_**ORDERED**_END
	nonblocking	IREAD_SHARED	N/A	WRITE_**ORDERED**_BEGIN
		IWRITE_SHARED		WRITE_**ORDERED**_END

Read e.g. **MPI_FILE**_READ_AT

New in MPI-3.1

Writing with Explicit Offsets

MPI_File_write_at(fh, offset, buf, count, datatype, *status*)

- writes **`count`** elements of **`datatype`** from memory **`buf`** to the file
- starting **`offset`** * units of **`etype`** from begin of view
- the elements are stored into the locations of the current view
- the sequence of basic datatypes of **`datatype`** (= signature of **`datatype`**) must match contiguous copies of the **`etype`** of the current view

Reading with Explicit Offsets

e.g. MPI_File_read_at(fh, offset, *buf*, count, datatype, *status*)

- attempts to read **`count`** elements of **`datatype`**
- starting **`offset`** * units of **`etype`** from begin of view (= **`displacement`**)
- the sequence of basic datatypes of **`datatype`** (= signature of **`datatype`**) must match contiguous copies of the **`etype`** of the current view
- EOF can be detected by noting that the amount of data read is less than **`count`**
  - i.e. EOF is no error!
  - use MPI_Get_count(status, datatype, *recv_count*)

Individual File Pointer, I.

e.g. MPI_File_read(fh, *buf*, count, datatype, *status*)

- same as *"Explicit Offsets"*, except:
- the offset is the current value of the **individual file pointer** of the calling process
- the individual file pointer is updated by

$$\text{new\_fp} = \text{old\_fp} + \frac{\text{elements(datatype)}}{\text{elements(etype)}} * \text{count}$$

i.e. it points to the next etype after the last one that will be accessed (*if EOF is reached, then recv_count is used, see previous slide*)

Individual File Pointer, II.

MPI_File_seek(fh, offset, whence)

- set individual file pointer fp:
  - set fp to offset – if whence=MPI_SEEK_SET
  - advance fp by offset – if whence=MPI_SEEK_CUR
  - set fp to EOF+offset – if whence=MPI_SEEK_END

MPI_File_get_position(fh, *offset*)

MPI_File_get_byte_offset(fh, offset, disp)

- to inquire offset
- to convert offset into byte displacement
  [*e.g. for* `disp` *argument in a new view*]

Shared File Pointer

- we want to give different files access to the same file without needing to maintain data consistency

- same view at all processes mandatory!
- the offset is the current, *global* value of the **shared file pointer** of `fh`
- multiple calls [*e.g. by different processes*] **behave as** if the calls were **serialized**
- non-collective, e.g.

```
MPI_File_read_shared(fh, buf, count, datatype, status)
```

- collective calls are *serialized* in the **order** of the processes' ranks, e.g.:

```
MPI_File_read_ordered(fh, buf, count, datatype, status)
```

```
MPI_File_seek_shared(fh, offset, whence)
MPI_File_get_position_shared(fh,offset)
MPI_File_get_byte_offset(fh, offset, disp)
```

- same rules as with individual file pointers

Jun Li, Department of Computer Science, CUNY Queens College

Nonblocking Data Access

```
e.g. MPI_File_iread(fh, buf, count, datatype, request)
     MPI_Wait(request, status)
     MPI_Test(request, flag, status)
```

- analogous to MPI-1 nonblocking

Application Scenery, I.

- Scenery A:
  - Task: Each process has to read the whole file
  - Solution: **MPI_File_read_all**
    = collective with individual file pointers,
    with same view (displacement+etype+filetype)
    on all processes
    [*internally: striped-reading by several process, only once from disk, then distributing with bcast*]
- Scenery B:
  - Task: The file contains a list of tasks,
    each task requires different compute time
  - Solution: **MPI_File_read_shared**
    =non-collective with a shared file pointer
    (same view is necessary for shared file p.)

- Scenery C:
  - Task: The file contains a list of tasks,
    each task requires **the same** compute time
  - Solution: **MPI_File_read_ordered**
    = **collective** with a **shared** file pointer
    (same view is necessary for shared file p.)
  - or: **MPI_File_read_all**
    = **collective** with **individual** file pointers,
    different views: *filetype* with
    MPI_Type_create_subarray(1, nproc,
    1, myrank, ..., datatype_of_task, *filetype*)
    [*internally: both may be implemented the same and equally with following scenery D*]

- Scenery D:
  - Task: The file contains a matrix,
    block partitioning,
    each process should get a block
  - Solution: generate different filetypes with
    MPI_Type_create_darray or **..._subarray**, the view on each process represents the block that should be read by this process,
    MPI_File_read_at_all with offset=0
    (= collective with explicit offsets)
    reads the whole matrix collectively
    [*internally: striped-reading of contiguous blocks by several process, then distributed with "alltoall"*]

Scenery – Nonblocking or Split Collective

- Scenery E:
  - Task: Each process has to read the whole file
  - Solution:
    - MPI_File_iread_all or MPI_File_read_all_begin
      = collective with individual file pointers,
      with same view (displacement+etype+filetype)
      on all processes
      [*internally: starting asynchronous striped-reading by several process*]
    - then computing some other initialization,
    - MPI_Wait or MPI_File_read_all_end.
      [*internally: waiting until striped-reading finished, then distributing the data with bcast*]

Error handling → "assembler for parallel computing"

Once there is error code, the behavior becomes unpredictable

Most important aspects:

- The communication should be reliable (same rule as for processor and memory)
- If the MPI program is erroneous → no warranties:
  - by default: abort, if error detected by MPI library otherwise, unpredictable behavior
    i.e., error handler MPI_ERRORS_ARE_FATAL is the default
  - C/C++: MPI_Comm_set_errhandler (comm, MPI_ERRORS_RETURN);

    Python: comm.Set_errhandler(MPI.ERRORS_RETURN)
    Newly added in MPI-4.0
  - directly after MPI_INIT with both comm = MPI_COMM_WORLD and MPI_COMM_SELF, then
    - **ierror returned by each MPI routine (except MPI window and MPI file routines)**
    - **undefined state after an erroneous MPI call has occurred (only MPI_Abort(…) should be still callable)**
  - Exception: MPI-I/O has default MPI_ERRORS_RETURN
    - Default can be changed through MPI_FILE_NULL:
    - MPI_File_set_errhandler (MPI_FILE_NULL, MPI_ERRORS_ARE_FATAL)

      Python: MPI.FILE_NULL.Set_errhandler(MPI.ERRORS_ARE_FATAL)

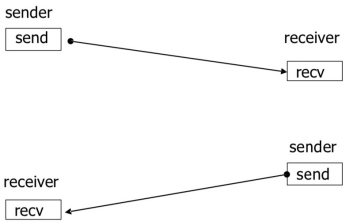
      — MPI_ERRORS_ARE_FATAL aborts the process and all connected processes
      — MPI_ERRORS_ABORT aborts only all processes of the related communicator
      New in MPI-4.0

Shared Memory

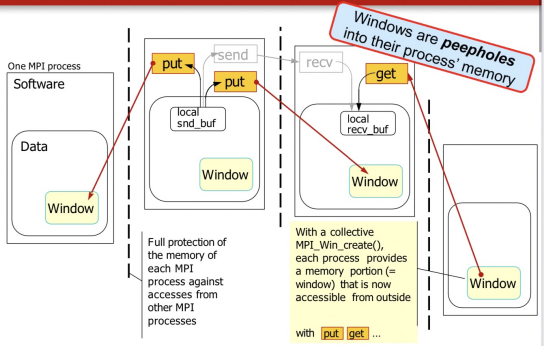
In general, MPI is a distributed memory system but it can take advantage of shared memory model

Cooperative Communication

- MPI-1 supports cooperative or 2-sided communication
- Both sender and receiver processes must participate in the communication

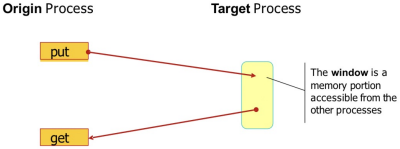


Typically, all processes are both, origin and target processes



One-sided Communication

- Communication parameters for both the sender and receiver are specified by one process (origin)
- User must impose correct ordering of memory accesses



One-sided Operations

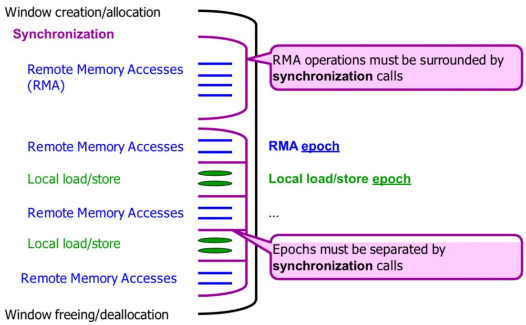
Three major sets of routines:

- Window creation or allocation
  - Each process in a group of processes **(defined by a communicator)**
  - defines a chunk of own memory – named ***window***,
  - which can be afterwards accessed by all other processes of the group.
- **R**emote **M**emory **A**ccess (RMA, nonblocking) routines
  - Access to remote windows:
    - **put, get, accumulate, …**
- Synchronization
  - The RMA routines are nonblocking and
  - must be surrounded by synchronization routines,
  - which guarantee
    - **that the RMA is locally and remotely finished**
    - **and that all necessary cache operation are implicitly done.**

Sequence of one-sided operations

For different IMA they are surrounded by different synchronization to make sure corresponding operations can be included.

Sequence of One-sided Operations



MPI_Put

- Performs an operation equivalent to a **send** by the **origin process** and a matching **receive** by the target process
- The origin process specifies the arguments for both origin and target
- **Nonblocking call** → finished by subsequent synchronization call
  → don't modify the origin (=send) buffer until next syncron

Where is the recv_buf in the target process ?

- The target buffer is at address
  target_addr = win_base$_{target_process}$
  + target_disp$_{origin_process}$ * disp_unit$_{target_process}$

As provided in MPI_Win_create or _all at the target proces

Like **send_buf, count, datatype** in MPI_Send

```
MPI_Put( origin_address, origin_count, origin_datatype,
         target_rank, target_disp_origin_process,
         target_count, target_datatype, win)
```

Like **dest** in MPI_Send

Like **count, datatype** in an MPI_Recv at the target process

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

Window Creation

- Specifies the region in memory (already allocated) that can be accessed by remote processes
- **Collective** call over all processes in the intracommunicator
- Returns an opaque object of type `MPI_Win` which can be used to perform the remote memory access (RMA) operations

A normal buffer argument — byte size, MPI_Aint

```
MPI_Win_create( win_base_addr_target, win_size_target,
                disp_unit_target, info, comm, win)
```

byte size, int

A window handle represents:
- all about the communicator
- and its processes,
- the location of the windows in all processes,
- the disp_units in all processes

MPI_Put

C

- C/C++: int MPI_Put(const void *origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_co MPI_Datatype target_datatype, MPI_Win win)
  int MPI_Put_c(const void *origin_addr, MPI_Count origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, MPI_Count target_count, MPI_Datatype target_datatype, MPI_Win wi

Large count version, new in MPI-4.0

Python

- Python: win.Put((origin_buf, origin_count, origin_datatype), target_rank, (target_disp, target_count, target_datatype))

Window Creation

If you want to share a window, they need to be in same communicator

Window Creation with MPI_Win_create

C

- C/C++: int MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)
  int MPI_Win_create_c(void *base, MPI_Aint size, MPI_Aint disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

Large count version, new in MPI-4.0

Python

- Python: win = MPI.Win.Create(memory, disp_unit, info, comm)

e.g., a numpy array

All Memory Allocation with modern C-Pointer

C

```
float *buf;  MPI_Win win; int max_length; max_length = ...;
MPI_Win_allocate( (MPI_Aint)(max_length*sizeof(float)), sizeof(float),
                  MPI_INFO_NULL, MPI_COMM_WORLD,  &buf,  &win);
// the window elements are buf[0] .. buf[max_length-1]
```

Python

```
np_dtype = np.single  # = C type float → MPI.FLOAT
max_length = ...
win = MPI.Win.Allocate(np_dtype(0).itemsize*max_length, np_dtype(0).itemsize, MPI.INFO_NULL,
                       MPI.COMM_WORLD)
buf = np.frombuffer(win, dtype=np_dtype)
# the window elements are buf[0] .. buf[max_length-1]
#buf = np.reshape(buf,())  # in case of max_length==1 and using buf as a normal variable instead of a 1-dim array
```

MPI_Get

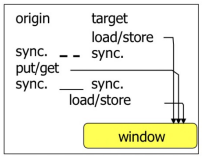
- Similar to the put operation, except that data is transferred from the target memory to the origin process
- To complete the transfer a synchronization call must be made on the window involved
- The local buffer should not be accessed until the synchronization call is completed

```
MPI_Get( origin_address, origin_count, origin_datatype,
         target_rank, target_disp, target_count,
         target_datatype, win)
```

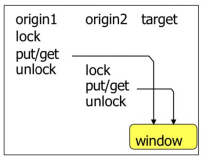
Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

Synchronization Calls (1)

- Active target communication
  - communication paradigm similar to message passing model
  - target process participates only in the synchronization
  - fence or post-start-complete-wait



- Passive target communication
  - communication paradigm closer to shared memory model
  - only the origin process is involved in the communication
  - lock/unlock



MPI_Accumulate

- Accumulates the contents of the origin buffer to the target area specified using the predefined operation `op`
- User-defined operations cannot be used
- Accumulate is **elementwise atomic**: many accumulates can be done by many origins to one target -> [*may be expensive*]

```
MPI_Accumulate(origin_address, origin_count,
               origin_datatype, target_rank, target_disp,
               target_count, target_datatype, op, win)
```

Heterogeneous platforms: Use only basic datatypes or derived datatypes without byte-length displacements!

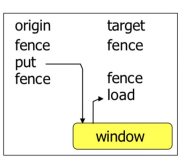
Synchronization Calls (2)

- Active target communication
  - MPI_Win_fence (like a barrier)
  - MPI_Win_post, MPI_Win_start, MPI_Win_complete, MPI_Win_wait/test
- Passive target communication
  - MPI_Win_lock, MPI_Win_unlock,
  - New in MPI-3.0 — MPI_Win_lock_all, MPI_Win_unlock_all,
  - New in MPI-3.0 — MPI_Win_flush(_all), MPI_Win_flush_local(_all), MPI_Win_sync

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- **Like a barrier**
- Used for active target communication
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation,
  - enables the optimization of internal cache operations
  - Integer 0 = no assertions
  - Several assertions with ***bitwise or*** operation

  E.g., in C: `MPI_MODE_NOSTORE | MPI_MODE_... | MPI_MODE_...`




```
MPI_Win_fence(assert, win)
```

Synchronization Calls (1)

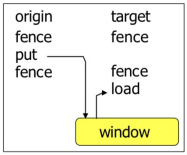
- Active Target Communication
  - We have both sources of communication
  -

Time goes down in the code

- Passive Target
  - Two origin & one target
  - Target does not need to do anything
  - Lock/unlock guarantees completion & returns the lock

MPI_Win_fence

- Synchronizes RMA operations on specified window
- Collective over the window
- **Like a barrier**
- Used for active target communication
- Should be used before and after calls to put, get, and accumulate
- The `assert` argument is used to provide optimization hints to the implementation,
  - enables the optimization of internal cache operations
  - Integer 0 = no assertions
  - Several assertions with ***bitwise or*** operation
    E.g., in C: `MPI_MODE_NOSTORE | MPI_MODE_... | MPI_MODE_...`



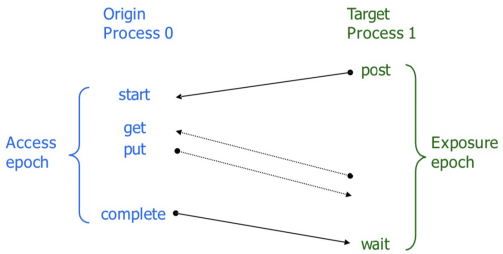

```
MPI_Win_fence(assert, win)
```

Start/Complete & Post/Wait, I

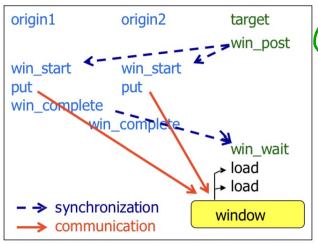
The more info you give to MPI the more weight it can assign

Start/Complete and Post/Wait, I.

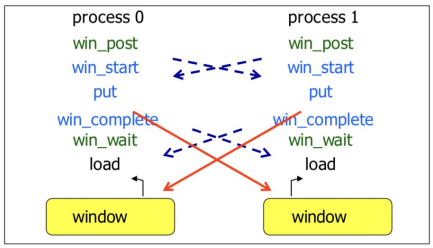
- Used for active target communication to restrict synchronization to a minimum



- RMA (put, get, accumulate) are finished
  - locally after win_complete
  - at the target after win_wait
- local buffer must not be reused before RMA call locally finished
- communication partners must be known
- no atomicity for overlapping "puts"
- assertions may improve efficiency
  --> give all information you have



- symmetric communication possible, only win_start and win_wait may block

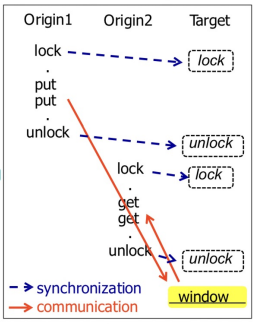


- Here, all processes are in the role of **target** *and* **origin**, i.e.
  - **expose** a window *and*
  - **access** windows per RMA

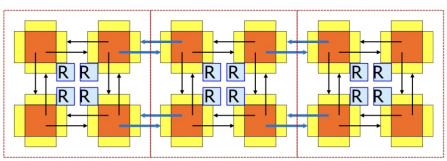
Lock/Unlock

- Does not guarantee a sequence
- agent may be necessary on systems without (virtual) shared memory
- Portable programs can use lock calls to windows in memory allocated **only** by `MPI_Alloc_mem`, `MPI_Win_allocate`, or

  `MPI_Win_attach` or `MPI_Win_allocate_shared` (New in MPI-4.0)
- RMA completed after `MPI_Unlock` at both origin and target

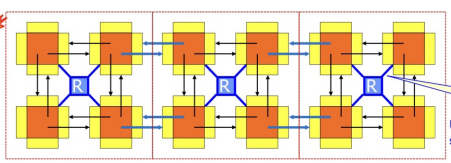


Programming opportunities with MPI shared memory: 1) Reducing memory space for replicated data



R = Replicated data in each MPI process

Example:
Cluster of 3 SMP nodes **without** using MPI shared memory methods



R = Shared memory → replicated data only once within each SMP node

Direct loads & stores, no library calls

Using MPI shared memory methods

MPI shared memory can be used to **significantly reduce the memory needs** for **replicated data.**

Programming opportunities with MPI shared memory: 2) Hybrid shared/cluster programming models

- MPI on each core (not hybrid)
  - Halos between all cores
  - MPI uses internally shared memory and cluster communication protocols
- MPI+OpenMP
  - Multi-threaded MPI processes
  - Halos communica. only between MPI processes
- new! MPI cluster communication + MPI shared memory communication
  - Same as "MPI on each core", but
  - within the shared memory nodes, halo communication through direct copying with C or Fortran statements
- new! MPI cluster comm. + MPI shared memory access
  - Similar to "MPI+OpenMP", but
  - shared memory programming through work-sharing between the MPI processes within each SMP node