MESSAGE-PASSING PARALLEL PROGRAMMING IN PYTHON USING MPI4PY

Outline

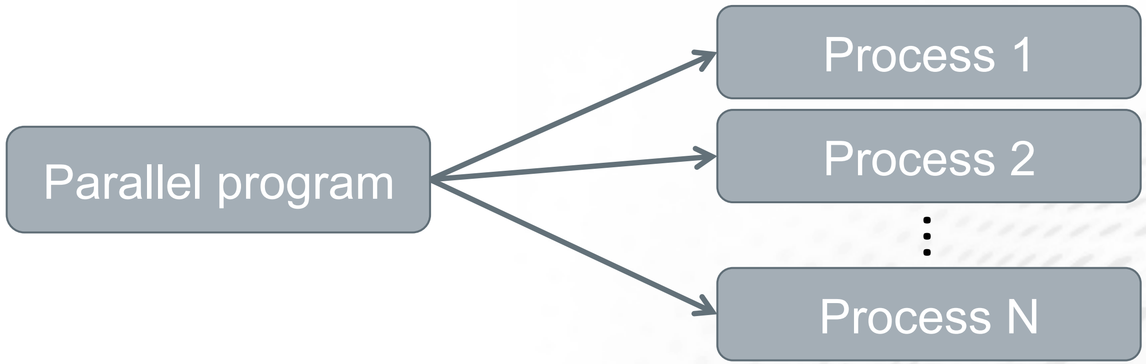
- Brief introduction to message passing interface (MPI)
- Python interface to MPI – mpi4py
- Performance considerations

Message passing interface

- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
- MPI is flexible and comprehensive
  - large (over 120 procedures)
  - concise (often only 6 procedures are needed)
- MPI standard defines C and Fortran interfaces
- **mpi4py** provides (an unofficial) Python interface

Execution model in MPI

- Parallel program is launched as set of **independent, identical processes**



- All the processes contain the same program code and instructions
- Processes can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  – mpirun, mpiexec, aprun, poe, ...
- When using Python, one launches N Python interpreters
  – mpirun -np 32 python parallel_script.py

MPI Concepts

- Rank: ID number given to a process
  - it is possible to query for rank
  - processes can perform different tasks based on their rank

mpi.py

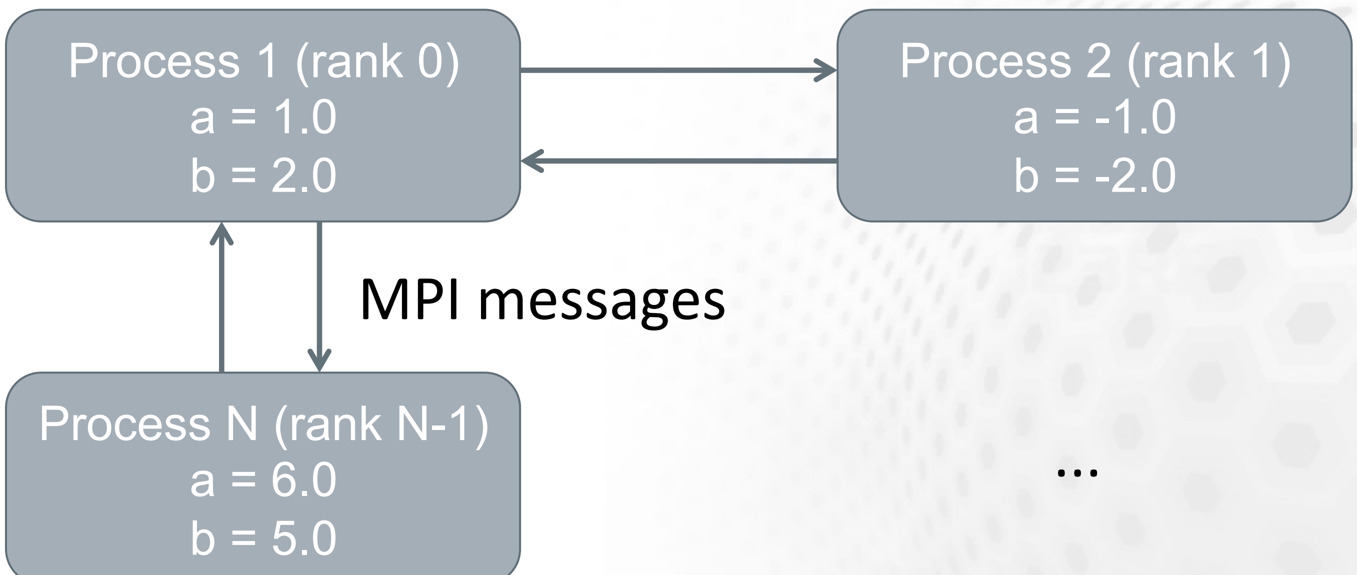
```
if (rank == 0):
    # do something
elif (rank == 1):
    # do something else
else:
    # all other processes do something different
```

MPI Concepts

- Communicator: a group containing all the processes that will participate in communication
  - in mpi4py all MPI calls are implemented as methods of a communicator object
  - **MPI_COMM_WORLD** contains all processes (MPI.COMM_WORLD in mpi4py)

Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



Using mpi4py

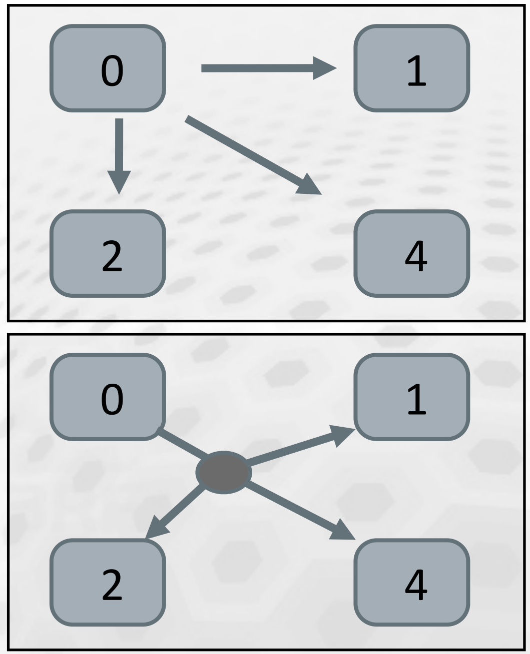
- Basic methods of communicator object
  - Get_size()
    Number of processes in communicator
  - Get_rank()
    rank of this process

mpi.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD  # communicator object containing all processes
size = comm.Get_size()
rank = comm.Get_rank()

print "I am rank %d in group of %d processes" % (rank, size)
```

MPI communication

- MPI processes are independent, they communicate to coordinate work
- Point-to-point communication
  – Messages are sent between two processes
- Collective communication
  – Involving a number of processes at the same time



MPI point-to-point operations

- One process *sends* a message to another process that *receives* it
- Sends and receives in a program should match – one receive per send

Sending and receiving data

- Sending and receiving a dictionary

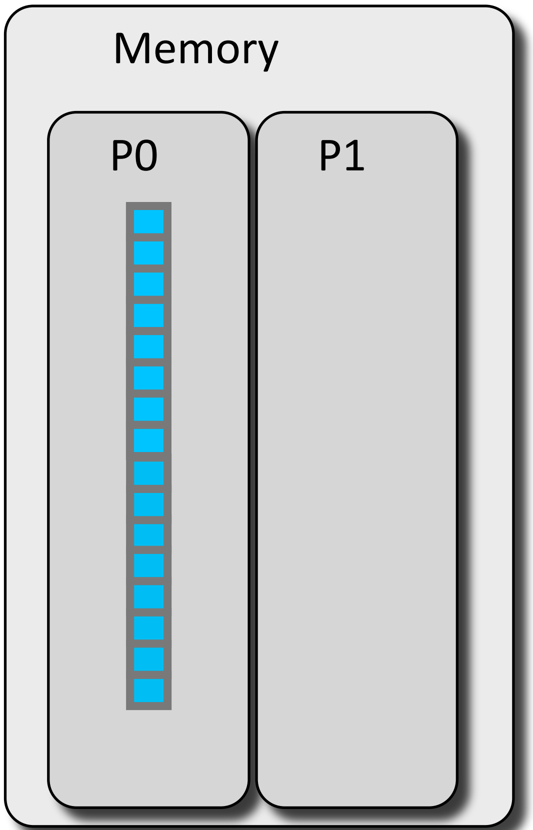
mpi.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD  # communicator object containing all processes
rank = comm.Get_rank()

if rank == 0:
   data = {'a': 7, 'b': 3.14}
   comm.send(data, dest=1, tag=11)
elif rank == 1:
   data = comm.recv(source=0, tag=11)
```

Sending and receiving data

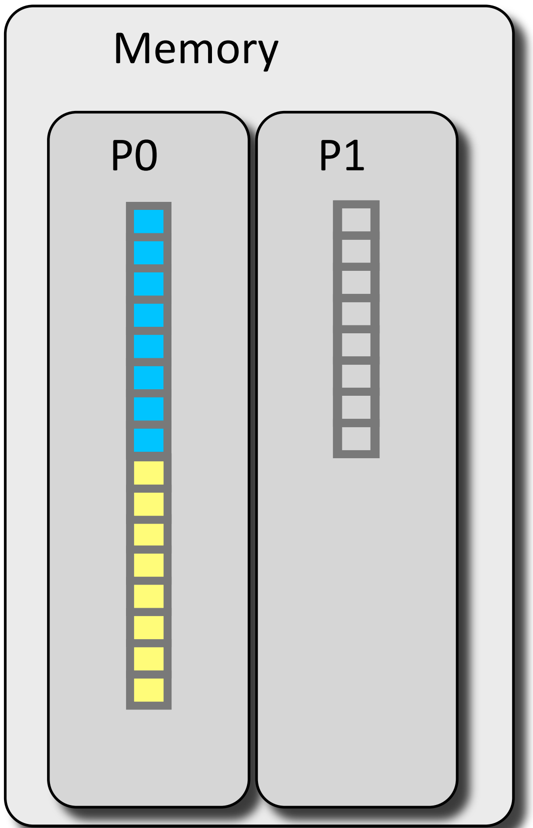
- Arbitrary Python objects can be communicated with the **send** and **receive** methods of a communicator
- **send(data, dest, tag)**
  - **data** Python object to send
  - **dest** destination rank
  - **tag** ID given to the message
- **recv(source, tag)**
  - **source** source rank
  - **tag** ID given to the message
  - data is provided as return value
- Destination and source ranks as well as tags have to match

Case study: parallel sum



- Array originally on process #0 (P0)
- Parallel algorithm
  - Scatter
    - Half of the array is sent to process 1
  - Compute
    - P0 & P1 sum independently their segments
  - Reduction
    - Partial sum on P1 sent to P0
    - P0 sums the partial sums

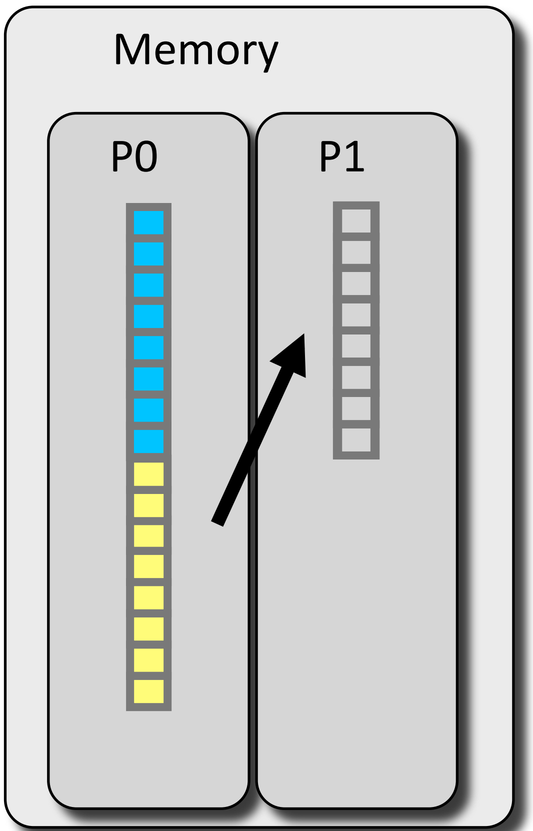
Case study: parallel sum



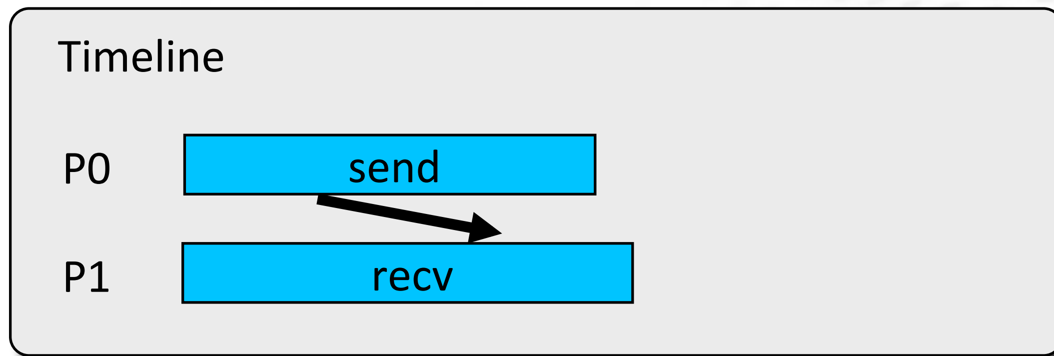
Step 1.1: Receive operation in scatter

P1 posts a receive to receive half of the array from P0

Case study: parallel sum

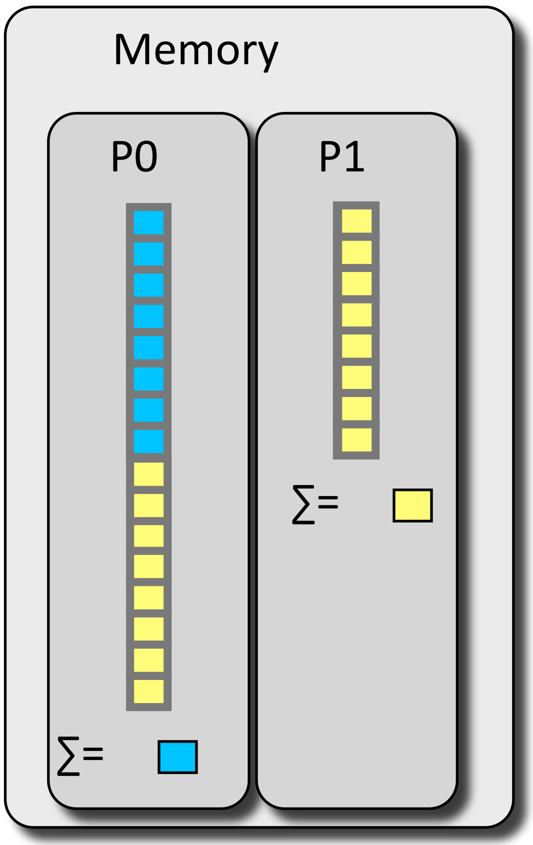


Step 1.2: Send operation in scatter

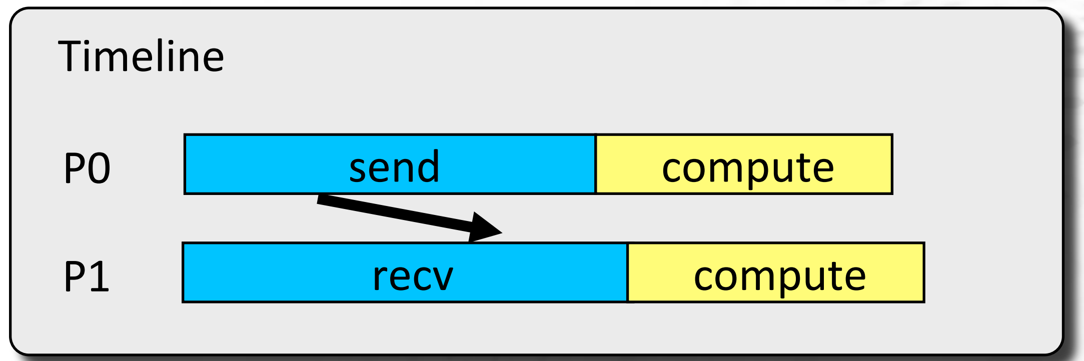


P0 posts a send to send the lower part of the array to P1

Case study: parallel sum

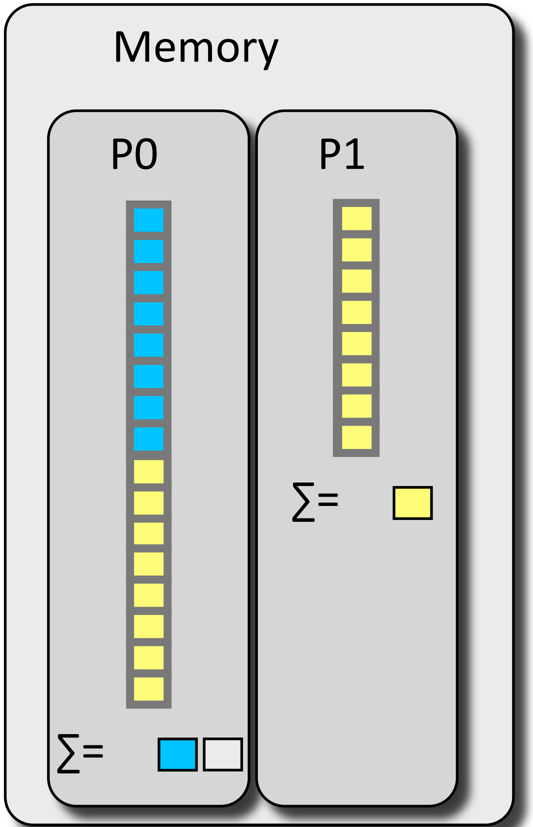


Step 2: Compute the sum in parallel

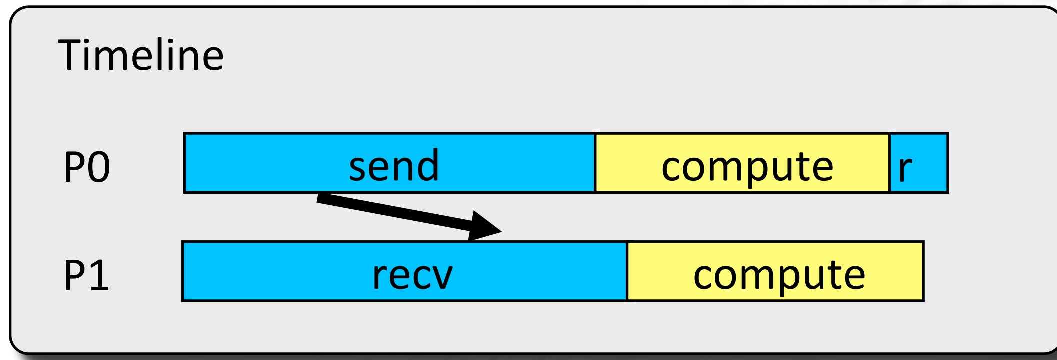


P0 & P1 computes their parallel sums and store them locally

Case study: parallel sum

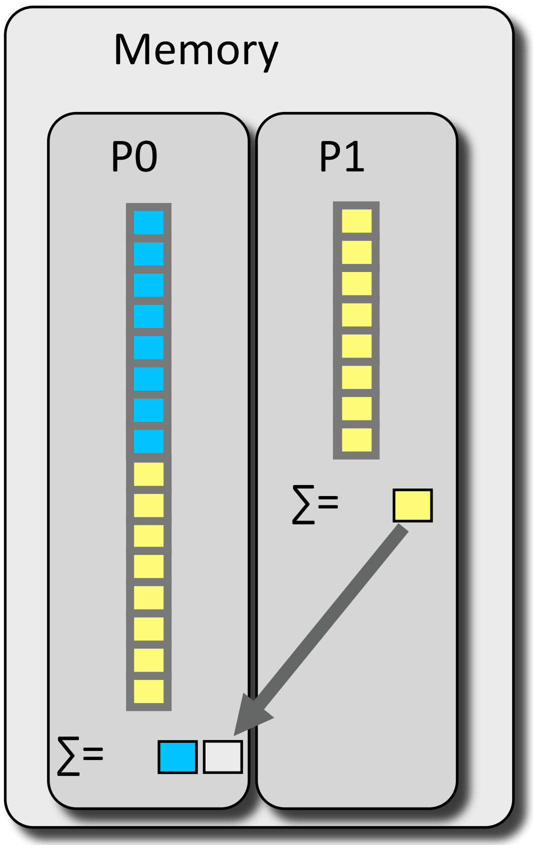


Step 3.1: Receive operation in reduction

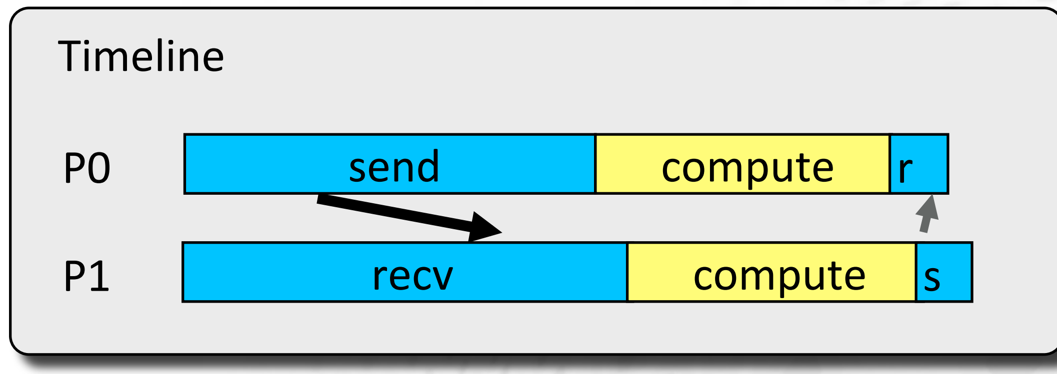


P0 posts a receive to receive partial sum

Case study: parallel sum

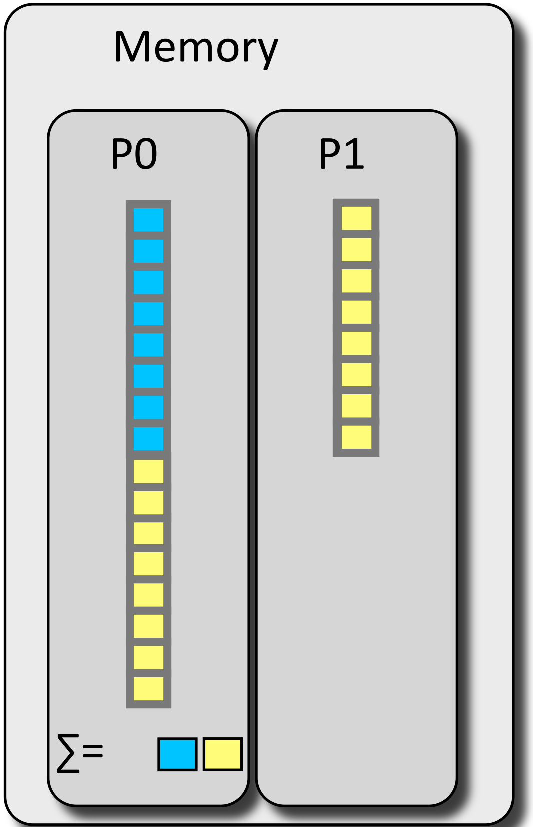


Step 3.2: send operation in reduction

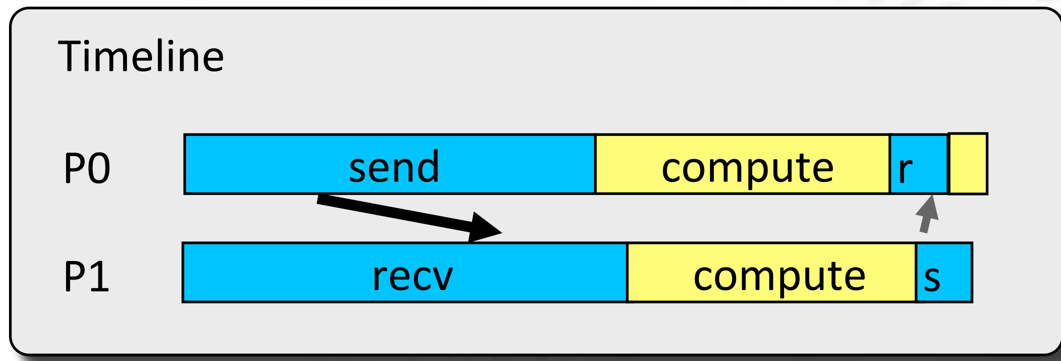


P1 posts a send with partial sum

Case study: parallel sum



Step 4: compute final answer

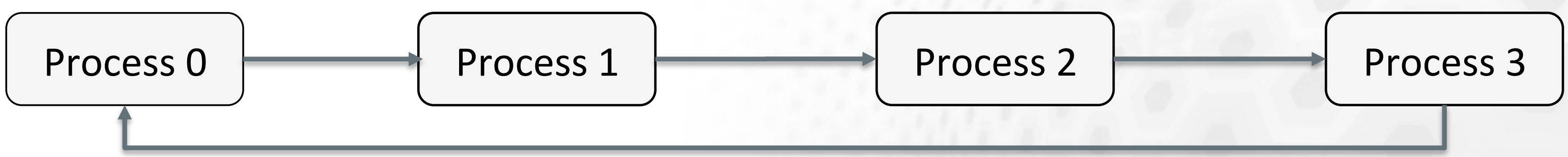


P0 sums the partial sums

Point-to-point communication patterns

Pairwise exchange

Pipe, a ring of processes exchanging data



Communicating NumPy arrays

- Arbitrary Python objects are converted to byte streams when sending
- Byte stream is converted back to Python object when receiving
- Conversions give overhead to communication
- (Contiguous) NumPy arrays can be communicated with very little overhead with upper case methods:
- **S**end(data, dest, tag)
- **R**ecv(data, source, tag)
  - Note the difference in receiving: the data array has to exist in the time of call

Communicating NumPy arrays

- Sending and receiving a NumPy array

mpi.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = numpy.arange(100, dtype=numpy.float)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float)
    comm.Recv(data, source=0, tag=13)
```

- Note the difference between upper/lower case!
  - send/recv: general Python objects, slow
  - Send/Recv: continuous arrays, fast

Non-blocking communication

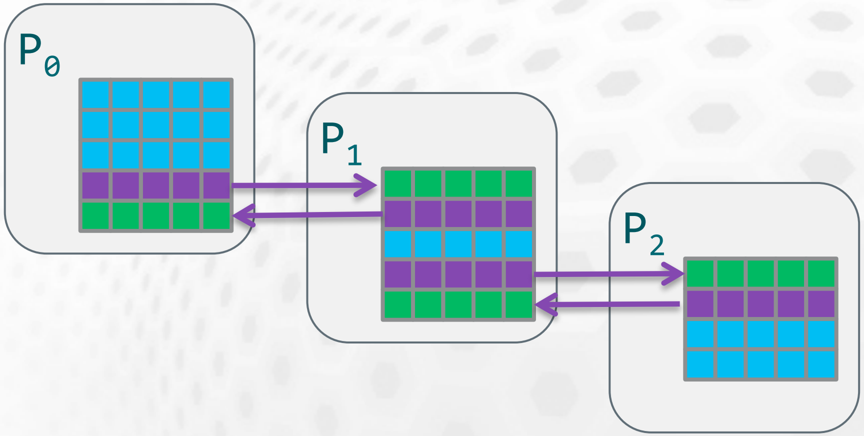
- Non-blocking sends and receives
  - `isend` & `irecv`
  - returns immediately and sends/receives in background
  - return value is a `Request` object
- Enables some computing concurrently with communication
- Avoids many common dead-lock situations

Non-blocking communication

- Have to finalize send/receive operations
  - `wait()`
    - Waits for the communication started with `isend` or `irecv` to finish (blocking)
  - `test()`
    - Tests if the communication has finished (non-blocking)
- You <u>can</u> mix non-blocking and blocking p2p routines
  - e.g., receive `isend` with `recv`

Typical usage pattern

```
request = comm.Irecv(ghost_data)
comm.Isend(border_data)
compute(ghost_independent_data)
request.Wait()
compute(border_data)
```



Collective communication

- Collective communication transmits data among all processes in a process group (communicator)
  - These routines must be called by all the processes in the group
- Collective communication includes
  - data movement
  - collective computation
  - synchronization

<u>Example</u>
`comm.barrier`()
makes every task hold until all tasks in the communicator **`comm`** have called it

Collective communication

- Collective communication typically outperforms point-to-point communication
- Code becomes more compact (and efficient) as well as easier to read:

```
if rank is 0:
  for i in range(1, ntasks):
    comm.Send(data, i, tag)
else
  comm.Recv(data, 0, tag)
```

```
comm.Bcast(data, 0)
```

Communicating a Numpy array of 1M elements from the task 0 to all other tasks

Collective communication

- Amount of sent and received data must match
- No tag arguments
  - Order of execution must coincide across processes

Collective communication

- Broadcast sends same data to all processes

bcast.py

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    n = 100
    data = numpy.arange(n, dtype=float)
    comm.bcast(n, root=0)
    comm.Bcast(data, root=0)
else:
    n = comm.bcast(None, root=0)   # returns the value
    data = numpy.zeros(n, float)   # prepare a receive buffer
    comm.Bcast(data, root=0)       # in-place modification
```

Collective communication

- Scatter distributes data to processes

scatter.py

```
from mpi4py import MPI
from numpy import array, zeros

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

buffer = zeros(10, float)  # prepare a receive buffer
if rank == 0:
    n = range(size)
    data = array(range(10*size), float)
    comm.scatter(n, root=0)
    comm.Scatter(data, buffer, root=0)
else:
    n = comm.scatter(None, root=0)        # returns the value
    comm.Scatter(None, buffer, root=0)  # in-place modification
```

Collective communication

- Gather pulls data from all processes

gather.py

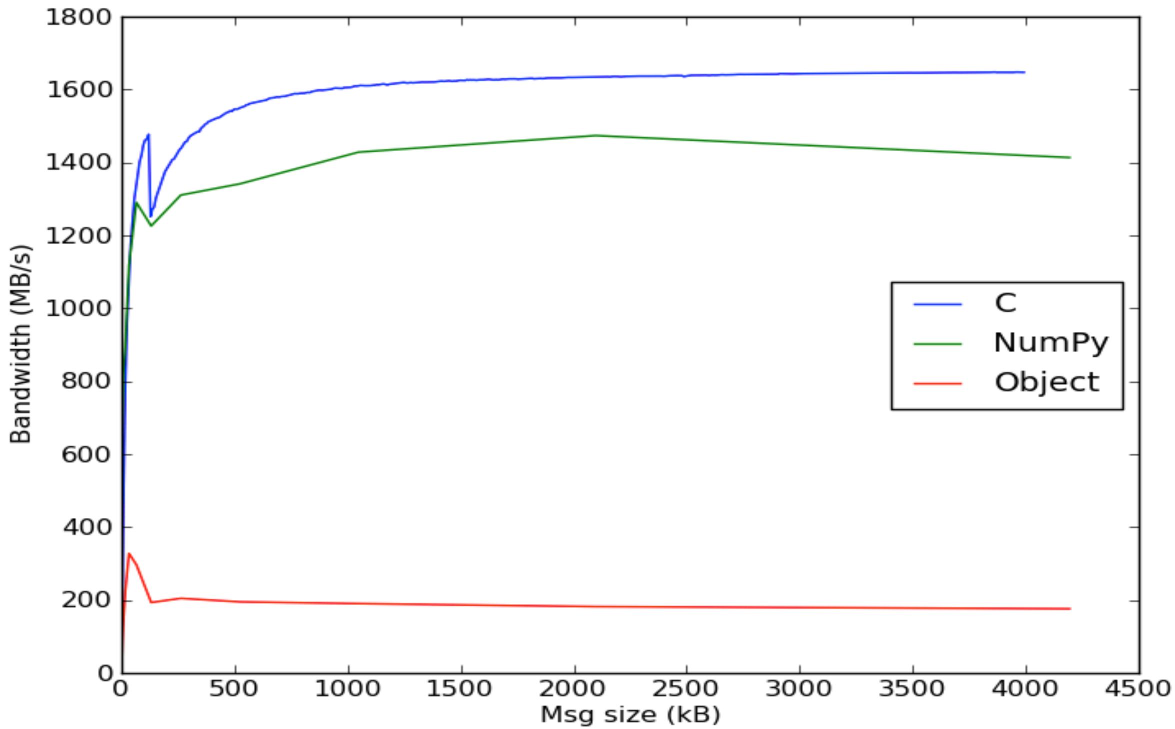
```
from mpi4py import MPI
from numpy import array, zeros

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

data = array(range(10), float) * rank
buffer = zeros(size * 10, float)
if rank == 0:
    n = comm.gather(rank, root=0)
    comm.Gather(data, buffer, root=0)
else:
    comm.gather(rank, root=0)
    comm.Gather(data, buffer, root=0)
```

On-line resources

- Documentation for mpi4py is quite poor
  - short on-line manual and API reference available at
    http://pythonhosted.org/mpi4py/
- Some good references:
  - "A Python Introduction to Parallel Programming with MPI" *by Jeremy Bejarano*
    http://materials.jeremybejarano.com/MPIwithPython/
  - "mpi4py examples" *by Jörg Bornschein*
    https://github.com/jbornschein/mpi4py-examples

mpi4py performance

- Ping-pong test



Summary

- mpi4py provides Python interface to MPI
- MPI calls via communicator object
- Possible to communicate arbitrary Python objects
- NumPy arrays can be communicated with nearly same speed as from C/Fortran

Martti Louhivuori // CSC – IT Center for Science Ltd.

Python in High-Performance Computing

April 21-22, 2016 @ University of Oslo