Introduction to parallel programming with MPI and Python

Pawel Pomorski, University of Waterloo, SHARCNET
ppomorsk@sharcnet.ca

March 27, 2019

What is parallel computing?

- Using many computers linked together by a communication network to efficiently perform a computation that would not be possible on a single computer
- Single computers have stagnated in performance, computing power advances must be achieved today through parallelism.
- Parallelization cannot be handled for the user by the compiler.
- Various approaches to parallelization since 1990s

Parallelization techniques

- Message passing - most popular method which explicitly passes data from one computer to another as they compute in parallel.
- Assumes that each computer has its own memory not shared with the others, so all data exchange has to occur through explicit procedures.
- Contrast to shared memory processors where processes running on multiple cores can share memory and use threads.
- Can still use message passing in shared memory architectures.

What is MPI?

- **M**essage **P**assing **I**nterface
- Language-independent communications protocol
- Portable, platform independent, de facto standard for parallel computing on distributed memory systems
- Various implementations exist (Open MPI, vendor versions)
- Many popular software libraries have parallel MPI versions
- Principal drawback: it is challenging to design and develop programs using message passing

What is MPI?

- MPI is not a new programming language.
- It is a collection of functions and macros, or a library that can be used in programs written in C, C++, Fortran and Python (via mpy4py)
- Most MPI programs are based on SPMD model - Single Program Multiple Data. This means that the same executable in a number of processes, but the input data makes each copy compute different things.

MPI in Python

- mpi4py (MPI for Python) provides bindings for MPI in Python
- object oriented, more user friendly, will automatically determine many of the needed arguments to MPI calls that have to be explicitly given when using other languages
- `https://bitbucket.org/mpi4py/mpi4py`
- `https://mpi4py.readthedocs.io/en/stable/`
- for best results, try to use latest version
- Execute with: mpirun -np 4 python program.py

Preliminaries

- A process is an instance of a program, can be created or destroyed
- MPI uses a statically allocated group of processes - their number is set at the beginning of program execution, no additional processes created (unlike threads)
- Each process is assigned a unique number or rank, which is from 0 to p-1, where p is the number of processes
- Number of processes is not necessarily number of processors; a processor may execute more than one process
- Generally, to achieve the close-to-ideal parallel speedup each process must have exclusive use of one processor core.
- Running MPI programs with one processor core is fine for testing and debugging, but of course will not give parallel speedup.

Exchanging data between processes

- Processes do not share memory, each has a distict memory space
- Hence, to communicate data between processes, explicit function calls must be made in the program
- In the most basic exchange, the process which has the data needed by another process calls a Send function, and the process which has to receive the data calls a Receive function.
- Both a Send and a Receive must be executed for the communication to be successful

Example parallel MPI program structure with Python

- import mpi4py module
- initialize MPI
- Do computations, using MPI communications between processes
- Shut down MPI

create MPI communicator object

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
```

comm is the communicator object. Its methods will be used to carry out communications between processes.

comm.send

```
send(...) method of mpi4py.MPI.Intracomm instance
    Comm.send(self, obj, int dest, int tag=0)
    Send
```

- obj - object containing the data to be sent
- dest - rank of the process which is the destination for the message
- tag - number, which can be used to distinguish among messages

This method may complete before data is actually transferred if the object being sent is small enough to fit in the buffer used by MPI for communications.

comm.recv

```
recv(...) method of mpi4py.MPI.Intracomm instance
    Comm.recv(self, buf=None, int source=ANY_SOURCE,
    int tag=ANY_TAG, Status status=None)
    Receive
```

- buf - optional buffer for containing the data to be received
- source - rank of the process from which to receive message
- tag - number, which can be used to distinguish among messages
- status - information about the data received, e.g. rank of source, tag, error code.
- Function returns the data received.

This method will not complete until the corresponding Comm.send has been executed on the source process and the data has successfully transferred.

comm.Get_rank

```
Get_rank(...) method of mpi4py.MPI.Intracomm instance
    Comm.Get_rank(self)

    Return the rank of this process in a communicator
```

comm.Get_size

```
Get_size(...) method of mpi4py.MPI.Intracomm instance
    Comm.Get_size(self)

    Return the number of processes in a communicator
```

First program in Python using mpi4py

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
p = comm.Get_size()

if my_rank != 0:
   message = "Hello from "+str(my_rank)
   comm.send(message, dest=0)
else :
    for procid in range(1,p):
        message = comm.recv(source=procid)
        print ("process 0 receives message from process",\
              procid,":",message)
```

Example output

```
$ mpirun -np 4 python greetings.py
process 0 receives message from process 1 : Hello from 1
process 0 receives message from process 2 : Hello from 2
process 0 receives message from process 3 : Hello from 3
```

Example: Numerical integration with trapezoid rule

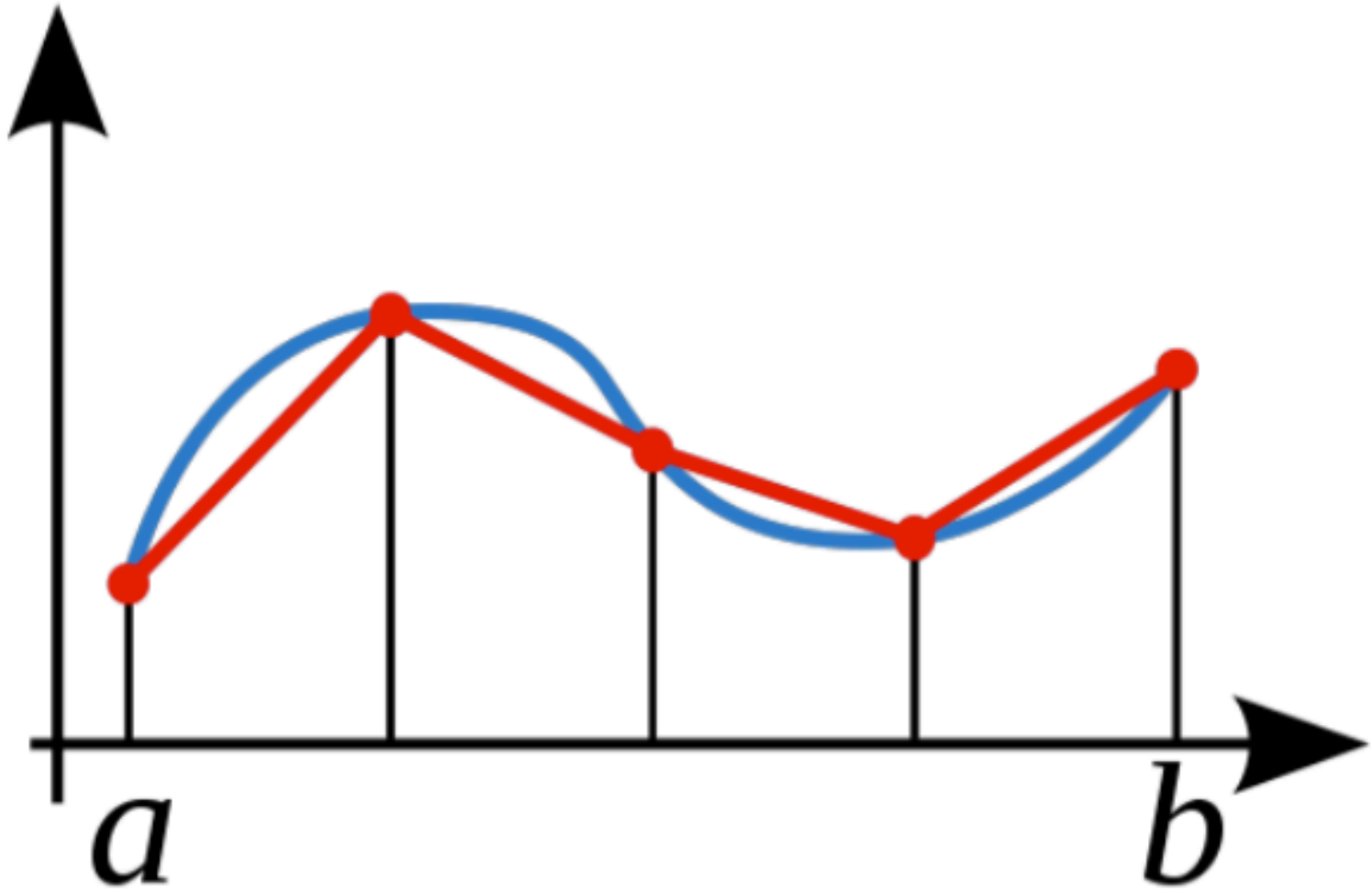


Figure 1: Area under the blue and red curves will match within desired error if number of points sufficiently large

Example: Numerical integration

Trapezoid rule

$$\int_a^b f(x)dx \approx \frac{h}{2}(f(x_0) + f(x_n)) + h\sum_{i=1}^{n-1} f(x_i)$$

where $h = (b-a)/n$, $x_i = a + ih$

Given p processes, each process can work on n/p segments

Note: for simplicity will assume n/p is an integer

process	interval
0	$[a, a + \frac{n}{p}h]$
1	$[a + \frac{n}{p}h, a + 2\frac{n}{p}h]$
...	...
p-1	$[a + (p-1)\frac{n}{p}h, b]$

Function to be integrated

Assume $f(x) = x^2$
Of course could have chosen any desired (integrable) function here.

```
def f(x):
    return x*x
```

Serial trapezoid rule

```
# traprule.py
from func import f

def Trap(a, b, n, h):

    integral = (f(a) + f(b))/2.0

    x = a
    for i in range(1,int(n)):
        x = x + h
        integral = integral + f(x)

    return integral*h
```

Parallel trapezoid rule

```
"""
 trap.py -- Parallel Trapezoidal Rule

 Input: None.
 Output:  Estimate of the integral from a to b of f(x)
    using the trapezoidal rule and n trapezoids.

 Algorithm:
    1.  Each process calculates "its" interval of
        integration.
    2.  Each process estimates the integral of f(x)
        over its interval using the trapezoidal rule.
    3a. Each process != 0 sends its integral to 0.
    3b. Process 0 sums the calculations received from
        the individual processes and prints the result.

        The number of processes (p) should evenly divide
        the number of trapezoids (n = 1024)
```

```
"""
  int           my_rank     My process rank
  int           p           The number of processes
  float         a = 0.0     Left endpoint
  float         b = 1.0     Right endpoint
  int           n = 1024    Number of trapezoids
  float         h           Trapezoid base length
  float         local_a     Left endpoint my process
  float         local_b     Right endpoint my process
  int           local_n     Number of trapezoids for
                            my calculation
  float         integral    Integral over my interval
  float         total=-1.0  Total integral
  int           source      Process sending integral
  int           dest = 0    All messages go to 0
"""
```

```
from mpi4py import MPI
from func import f
from traprule import Trap

comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
p = comm.Get_size()

a=0.0
b=1.0
n=1024
dest=0
total=-1.0
```

```
h = (b-a)/n  #   h is the same for all processes
local_n = n/p #  So is the number of trapezoids

#   Length of each process' interval of
#     integration = local_n*h.
local_a = a + my_rank*local_n*h
local_b = local_a + local_n*h
integral = Trap(local_a, local_b, local_n, h)
```

```
#   Add up the integrals calculated by each process
if my_rank == 0:
    total = integral
    for source in range(1,p):
        integral = comm.recv(source=source)
        print("PE ",my_rank,"<-",source,",",integral,"\n")
        total = total + integral

else :
      print("PE ",my_rank,"->",dest,",",integral,"\n")
      comm.send(integral, dest=0)

#  Print the result
if (my_rank == 0):
      print("With n=",n,", trapezoids, \n")
      print("integral from",a,"to",b,"=",total,"\n")

MPI.Finalize
```

Output

```
$ mpirun -np 4 python trap.py
PE  3 -> 0 , 0.19270837306976318
PE  2 -> 0 , 0.09895837306976318
PE  1 -> 0 , 0.036458373069763184
PE  0 <- 1 , 0.036458373069763184
PE  0 <- 2 , 0.09895837306976318
PE  0 <- 3 , 0.19270837306976318
With n= 1024 , trapezoids,
integral from 0.0 to 1.0 = 0.3333334922790527 3
```

Summary of MPI essentials

To write simple MPI parallel programs you only need:

- from mpi4py import MPI
- comm = MPI.COMM_WORLD
- comm.Get_rank()
- comm.Get_size()
- comm.send(...)
- comm.receive(...)
- MPI.Finalize

Beyond the basics

- MPI provides many other methods which can make your programs more efficient, or more convenient to write
- Probably the most useful are collective communications
- Collective communications involve all the processes in a communicator
- Let's look at Broadcast and Reduce

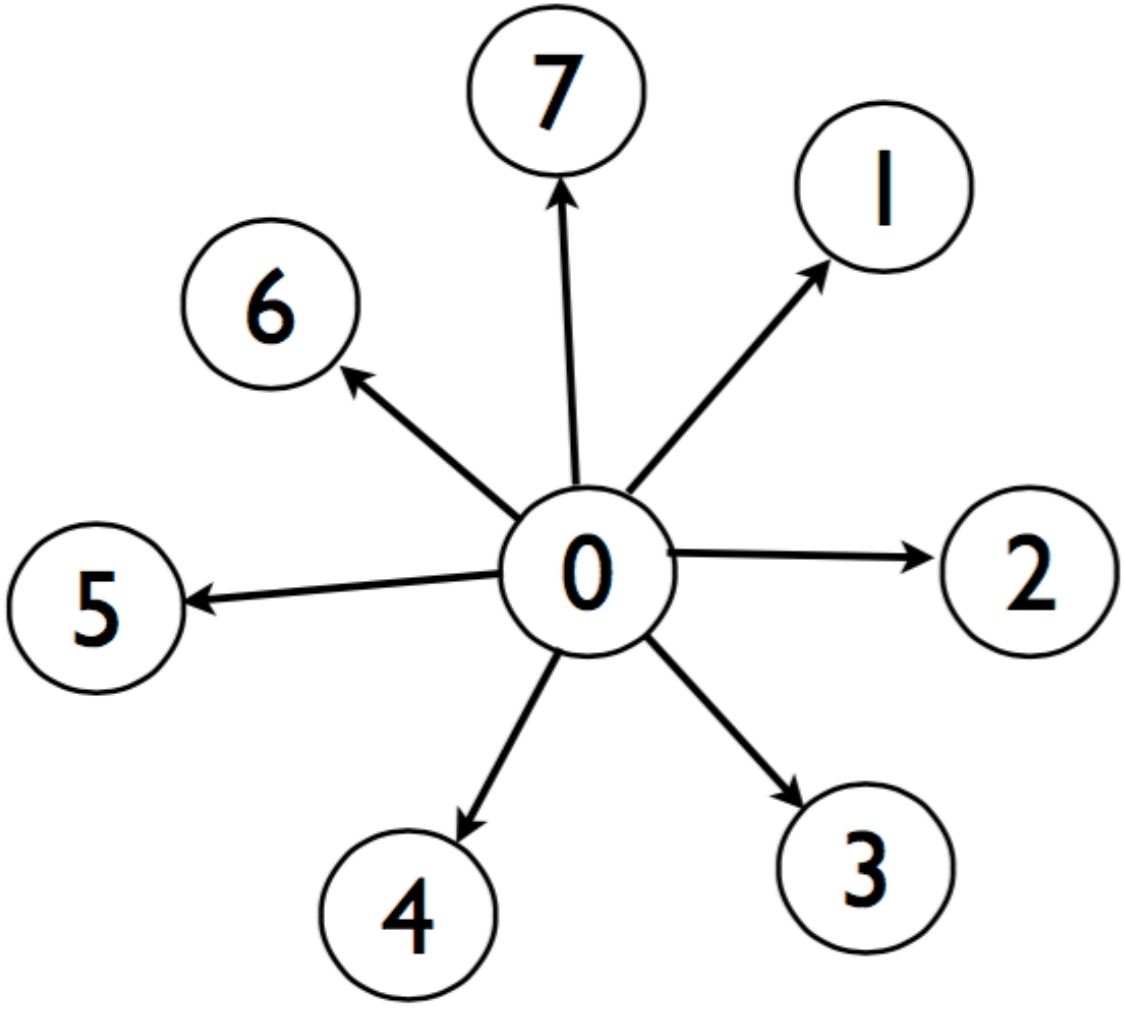
comm.bcast

```
bcast(...) method of mpi4py.MPI.Intracomm instance
    Comm.bcast(self, obj, int root=0)
    Broadcast
```

- obj - object being broadcasted, must be defined on root process before bcast called.
- root - rank of broadcast root proceses
- must be called on each process

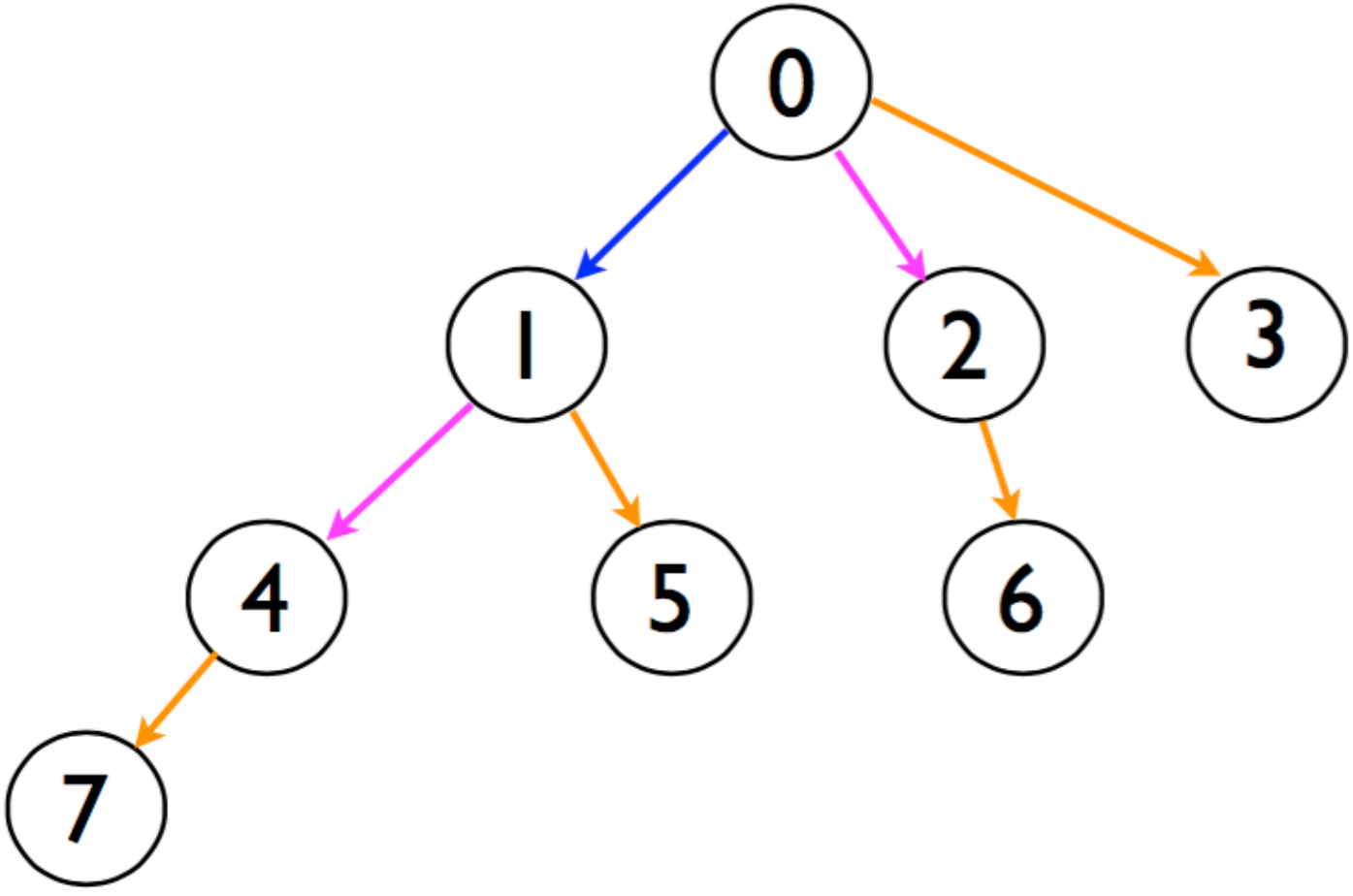
Broadcast - poor implementation

- Serial, 7 time steps needed



Broadcast - actual, parallel implementation

- Parallel, 3 time steps needed



Example: reading and broadcasting data

```
def Get_data(my_rank, p, comm):

    a=None
    b=None
    n=None
    if my_rank == 0:

        print("Rank ",my_rank,": Enter a, b, and n\n")
        a=float(input("enter a \n"))
        b=float(input("enter b \n"))
        n=int(input("enter n \n"))
        print( "ready for broadcast \n")

    a=comm.bcast(a)
    b=comm.bcast(b)
    n=comm.bcast(n)

    return a,b,n
```

comm.reduce

Data from all processes are combined using a binary operation

```
reduce(...) method of mpi4py.MPI.Intracomm instance
    Comm.reduce(self, sendobj, op=SUM, int root=0)
    Reduce
```

- sendobj - object to send
- op - reduce operation; predefined, e.g. MPI.SUM, MPI.MIN
- root - rank of root process
- returns the result of the reduce operation
- must be called in all processes in a communicator, BUT result only available in root process

Example - trapezoid with reduce

```
from mpi4py import MPI
from func import f
from traprule import Trap
from getdata2 import Get_data

comm = MPI.COMM_WORLD
my_rank = comm.Get_rank()
p = comm.Get_size()

a,b,n=Get_data(my_rank, p, comm) # process 0 will read dat

dest=0
total=-1.0

h = (b-a)/n  #   h is the same for all processes
local_n = n/p #  So is the number of trapezoids
```

```
local_a = a + my_rank*local_n*h
local_b = local_a + local_n*h
integral = Trap(local_a, local_b, local_n, h)

#   Add up the integrals calculated by each process

total=comm.reduce(integral)

#  /* Print the result */
if (my_rank == 0):
      print("With n=",n,", trapezoids, ")
      print("integral from",a,"to",b,"=",total)

MPI.Finalize
```

Further MPI features to explore

- Non-blocking communications
- Communicators
- Topologies
- User defined datatypes
- Parallel input/output operations
- Parallel algorithms
- Parallel libraries (eg. Scalapack)