HPC Seminar

Parallelization with Python on Multiple Nodes

Sergey Yakubov - Maxwell Team - DESY IT
Hamburg, 19.11.2018

HELMHOLTZ
ASSOCIATION

DESY

Agenda

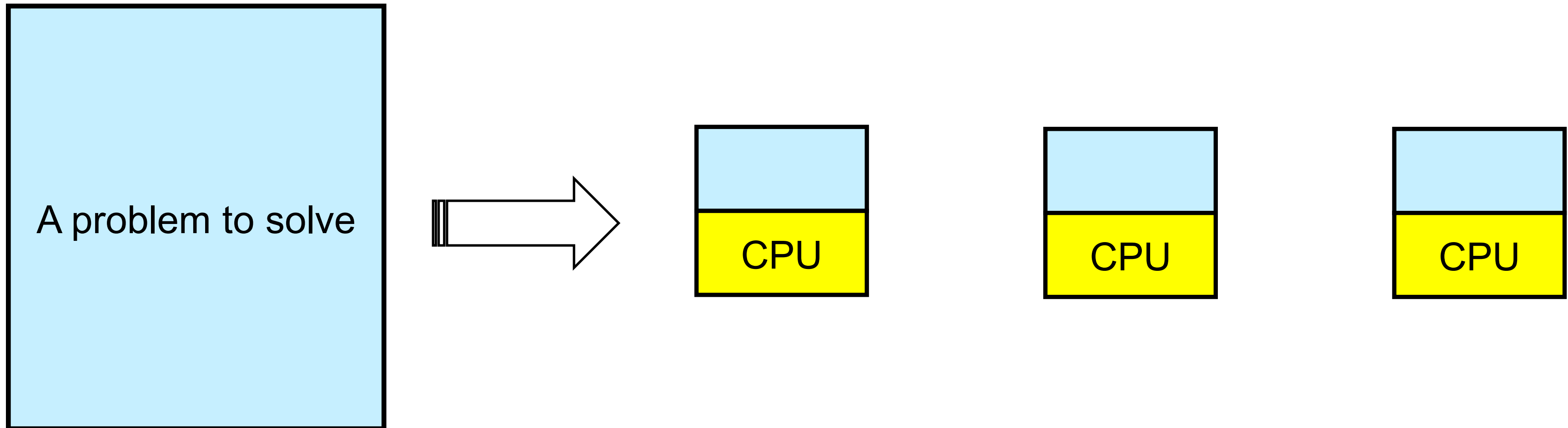
- Recap from the previous seminar
- Parallelization of a computational problem
- Parallelization with Python using Message Passing Interface (MPI)
- Summary

Recap from the previous seminar

- Quite often scientific Python code can be made fast
  - use specific libraries (NumPy, SciPy)
  - detect, isolate and compile pieces of the code that do lot of work
    - numba, ...
- Python code can make use of multicore architecture
  - specific libraries
  - threading package for I/O bound problems
  - multiprocessing package
  - automatic parallelization with numba for NumPy arrays
  - explicit parallelization with numba for loops
  - threading module and numba

Parallelization of a computational problem

How to partition computations to compute in parallel

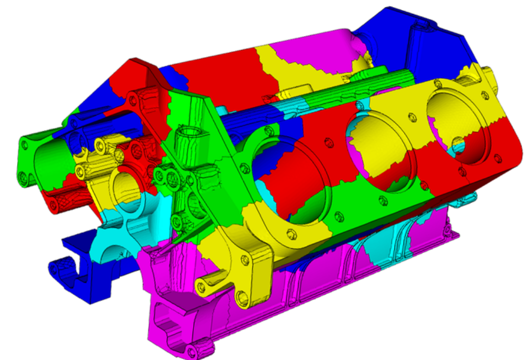


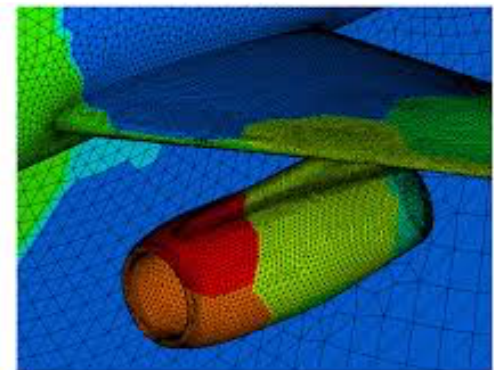
- Two ways to partition a problem
  - Domain decomposition - focus on data
  - Functional decomposition - focus on tasks

Parallelization of a computational problem

Domain Decomposition

- Domain decomposition methods
  - split your geometrical/mathematical domain on multiple subdomains, each CPU take care of a single one
- Challenges
  - partition your domain most efficiently
    - subdomain sizes are equal
    - single process allocates memory only for its own subdomain
    - minimize boundaries
    - keyword – ParMETIS
  - exchange information between subdomains
    - may become a bottleneck
  - I/O should be parallelized as well
  - Result post-processing
    - Paraview, Visit, Tecplot, ...

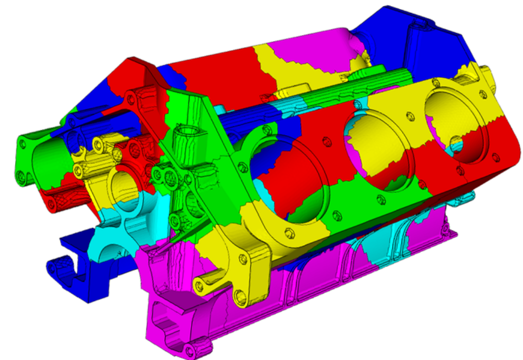


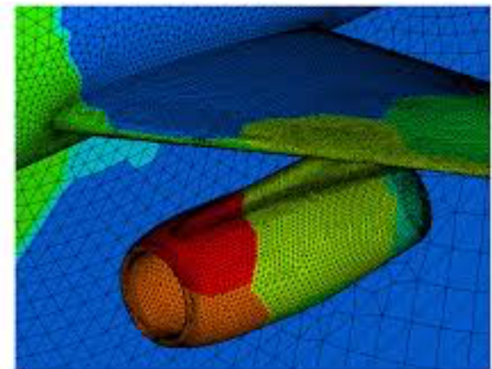


Parallelization of a computational problem

Examples – Domain Decomposition

- Computational Fluid Dynamics
  - heat transfer, mass transfer, phase change, chemical reactions
- Structural mechanics
  - deformations, deflections, forces/stresses
- Climate studies
- Everything where computations with matrices are needed

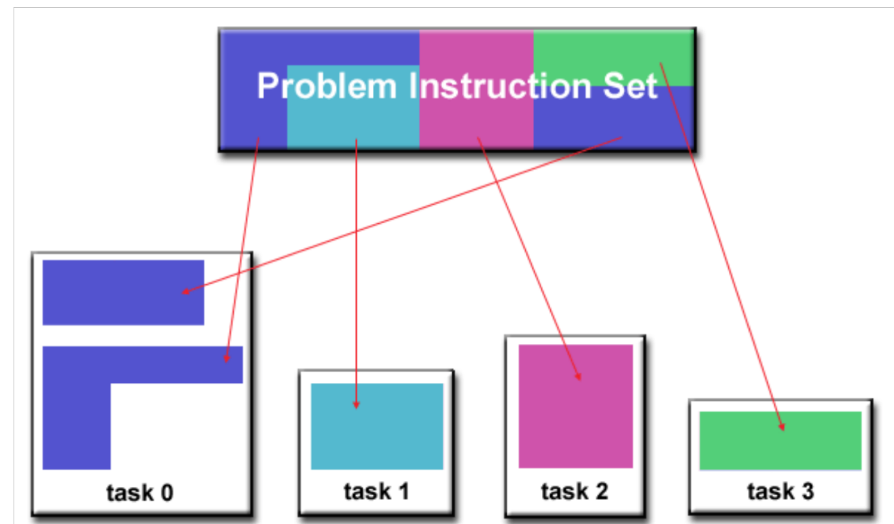




Parallelization of a computational problem

Functional Decomposition

- Function decomposition methods
  - split your computational task on multiple sub-tasks, each CPU take care of a single one
- Challenges – similar to domain decomposition
  - partition your tasks most efficiently
    - efforts to compute a sub-task are equal
  - exchange information between sub-tasks if necessary

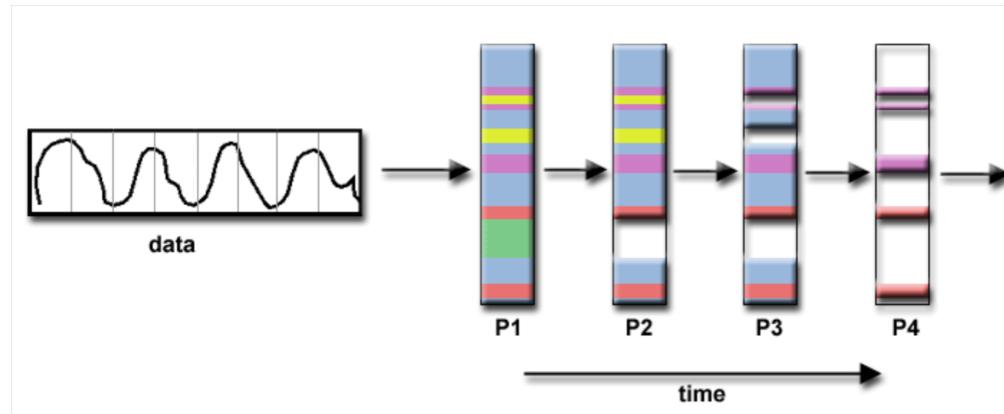


computing.llnl.gov

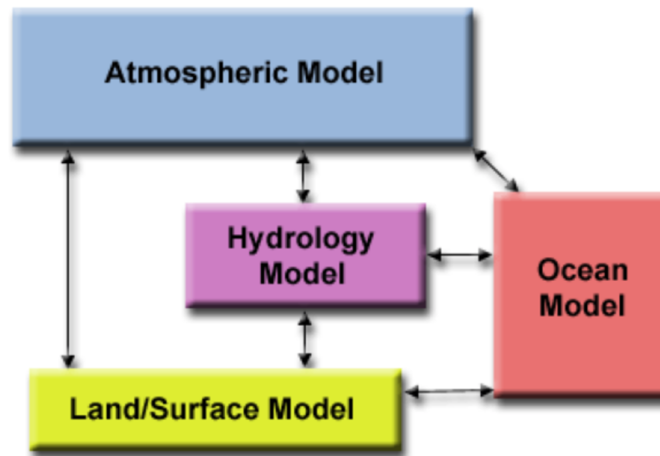
Parallelization of a computational problem

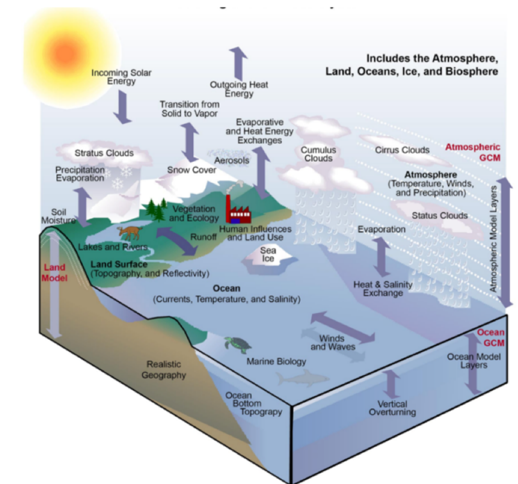
Examples - Functional Decomposition

- Signal processing



- Climate modelling
- ...

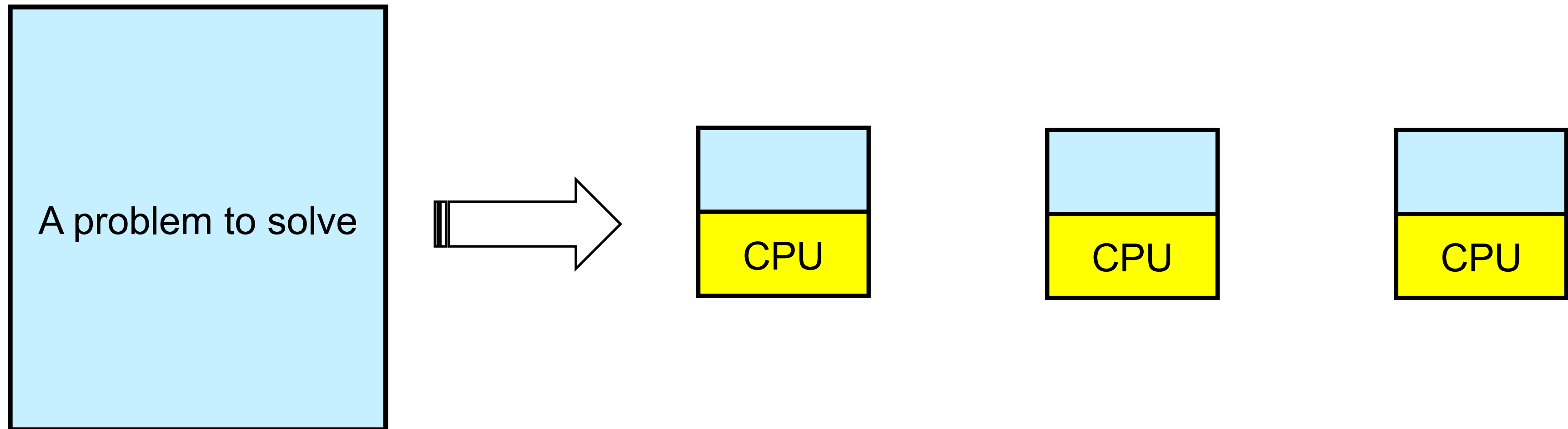




computing.llnl.gov

Parallelization of a computational problem

How to exchange information between parts of a parallel problem

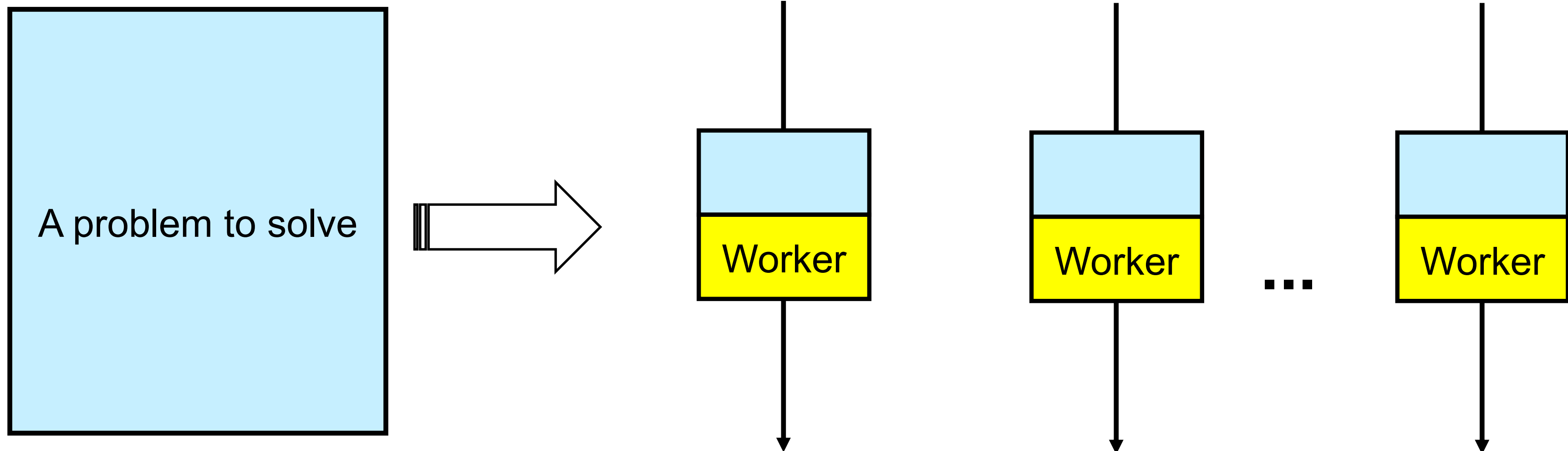


- Simple (if you are lucky)
  - Embarrassingly parallel
  - Nearly embarrassingly parallel (e.g. master-worker)
- Complex communications between parallel processes

Parallelization of a computational problem

Embarrassingly Parallel

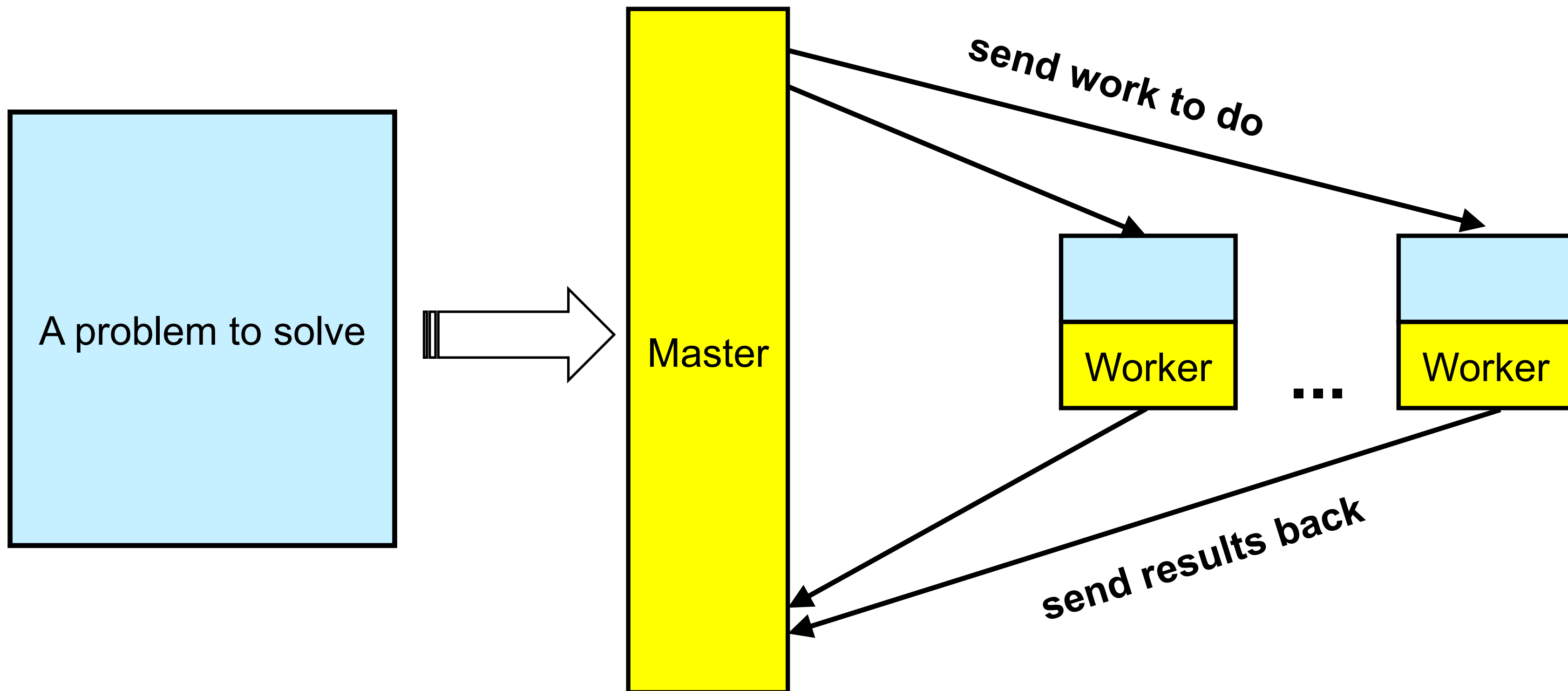
- “Embarrassingly parallel”
  - no communications are needed between parallel tasks



Parallelization of a computational problem

Embarrassingly Parallel

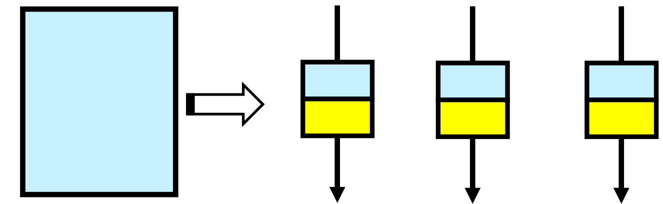
- “Nearly embarrassingly parallel”
  - few trivial data exchange are needed between parallel tasks

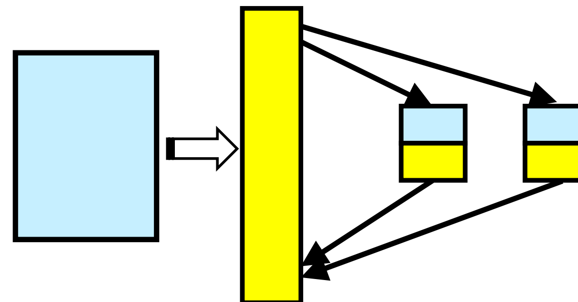


Parallelization of a computational problem

Embarrassingly Parallel - Examples

- HEP – event simulation and reconstruction
- Photon science – image processing (independent)
- Rendering in computer graphics (pixels are independent)
- Face recognition system
- Simulations comparing independent scenarios
  - e.g. climate models
- Discrete Fourier transform
- .....

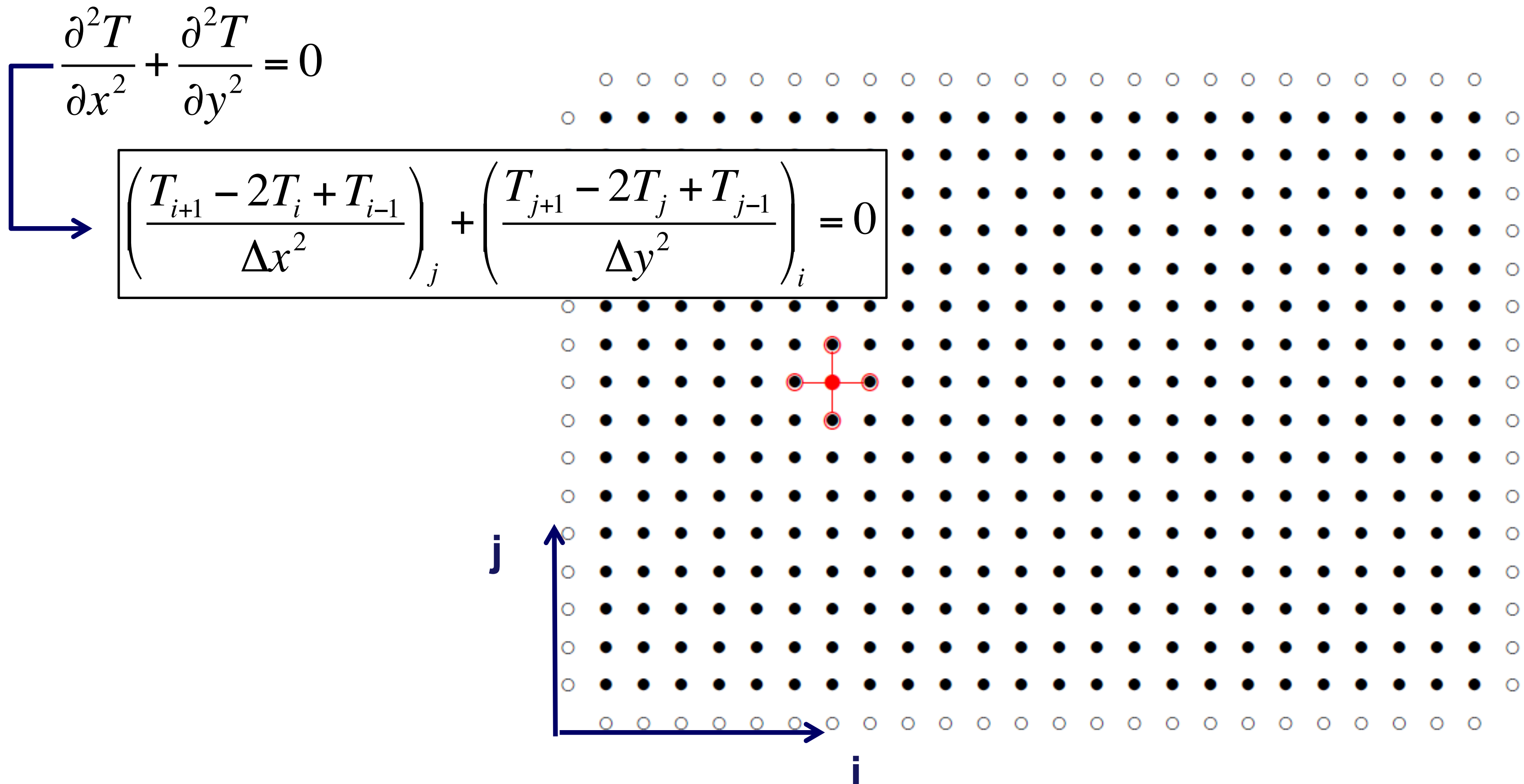




Parallelization of a computational problem

Complex Communications Example – Laplace equation (from introduction seminar)

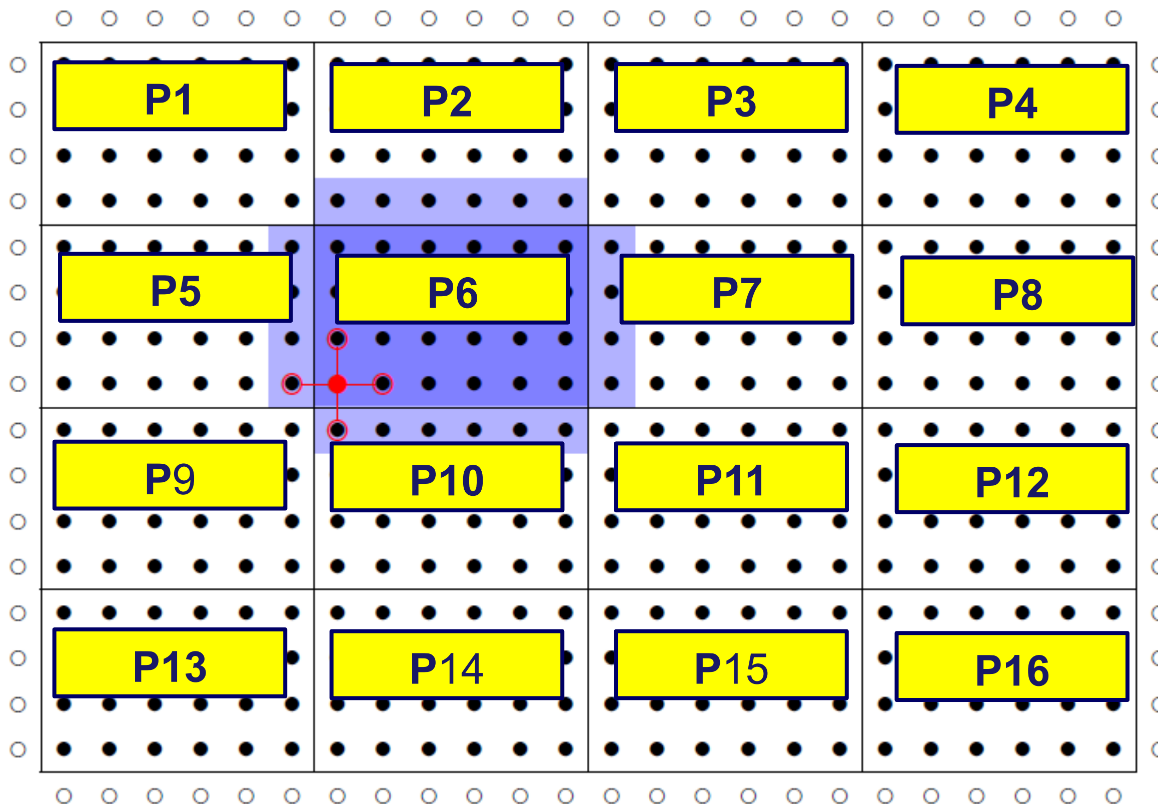
Data values in neighbour points are needed to compute local value



Parallelization of a computational problem

Complex Communications Example – Laplace equation

Communications needed when points are distributed over processes



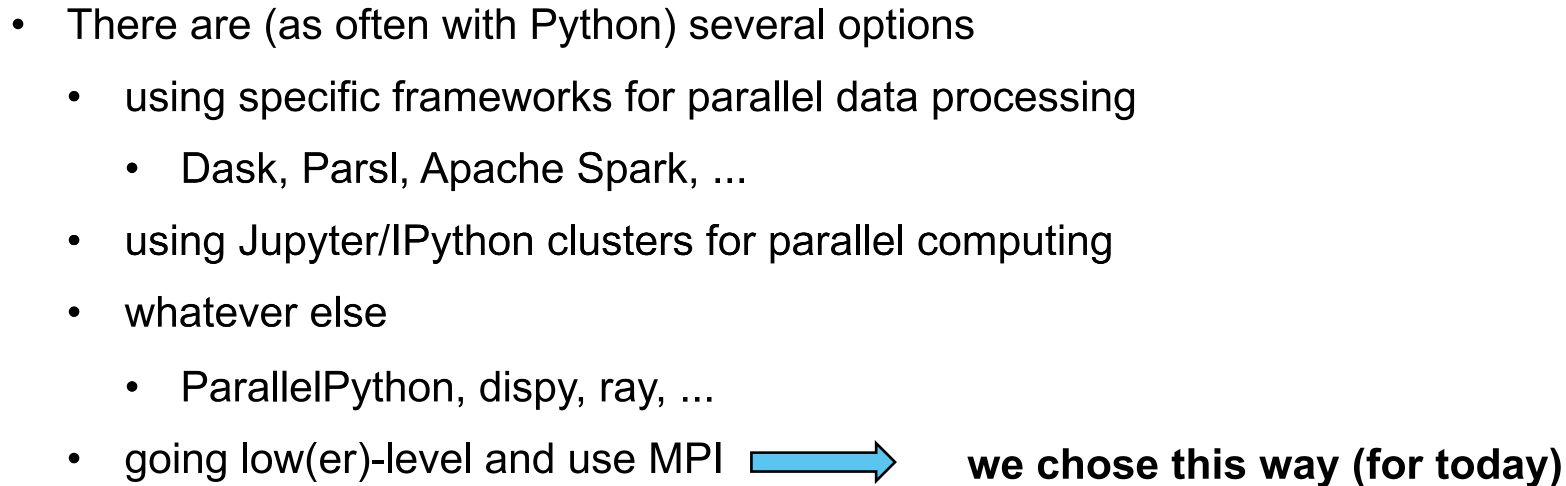
Parallelization with Python

Ways to implement parallelization

- There are (as often with Python) several options
  - using specific frameworks for parallel data processing
    - Dask, Parsl, Apache Spark, ...
  - using Jupyter/IPython clusters for parallel computing
  - whatever else
    - ParallelPython, dispy, ray, ...
  - going low(er)-level and use MPI

Parallelization with Python

Ways to implement parallelization

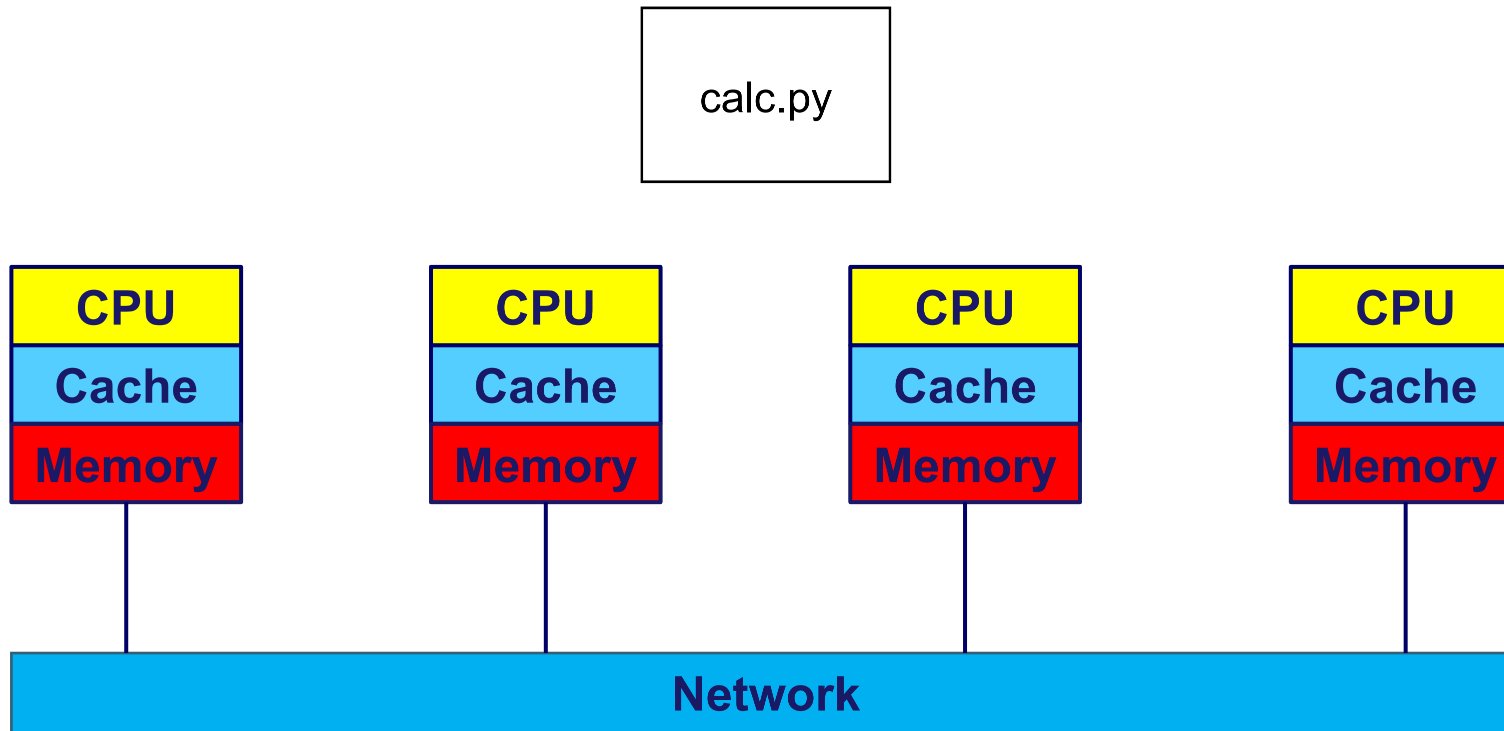


- There are (as often with Python) several options
  - using specific frameworks for parallel data processing
    - Dask, Parsl, Apache Spark, ...
  - using Jupyter/IPython clusters for parallel computing
  - whatever else
    - ParallelPython, dispy, ray, ...
  - going low(er)-level and use MPI ⇒ **we chose this way (for today)**

MPI

Parallelization approach

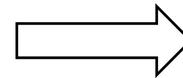
- We have a program calc.py
    - want to complete our simulations faster



MPI

Parallelization approach

- we have to somehow split the job
- each nodes starts its own instance of the program
- instances exchange data



MPI

calc.py | calc.py | calc.py | calc.py

CPU | CPU | CPU | CPU

Cache | Cache | Cache | Cache

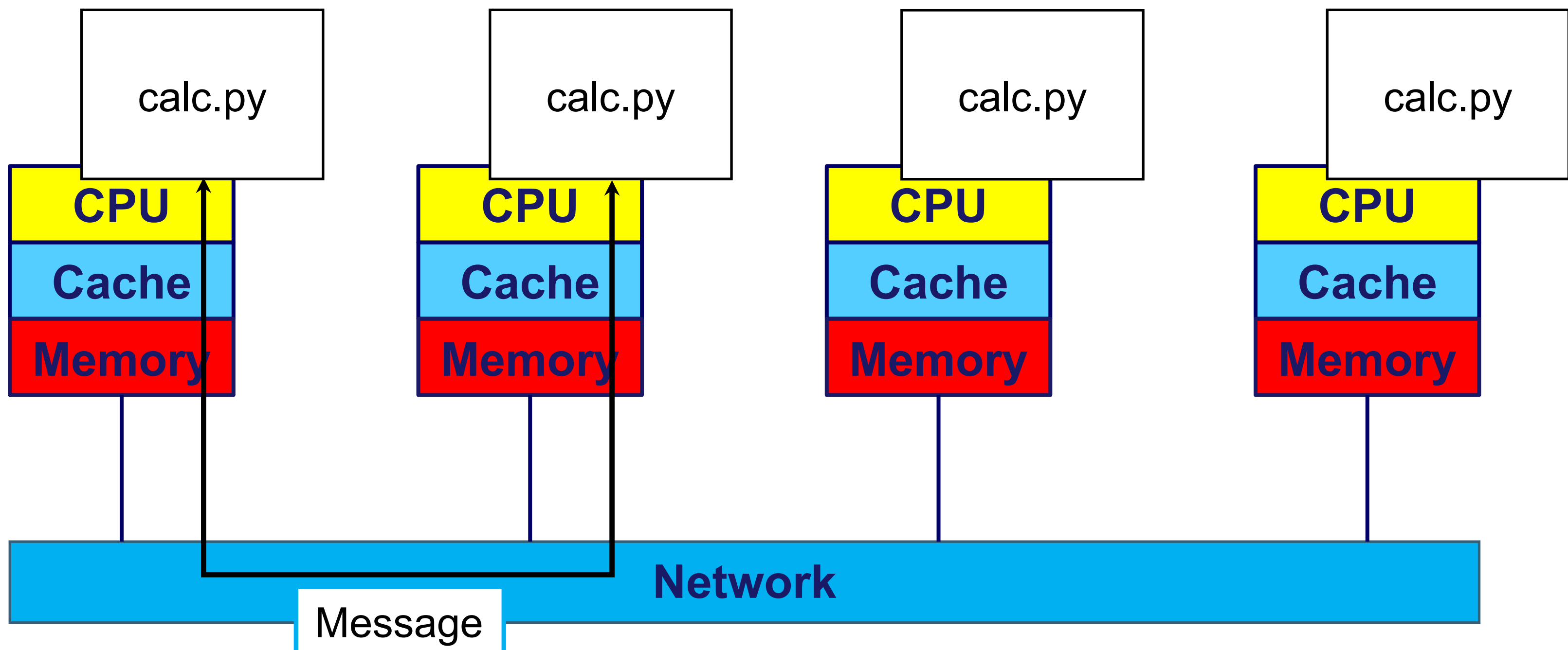
Memory | Memory | Memory | Memory

Network

MPI

Overview

- Standardized interface to a communication library
- Hides hardware/software communication mechanisms from a user
- Implements internode communications via messages
- SPMD – each process starts same program. All variables are local for each process.



MPI

Example

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

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

print('Hello, World! I\'m process #%d of %d' % (rank, size))
```

MPI

Configuring and execution

- You don‘t have to compile with Python, but you need to use same MPI version which was used for mpi4py - `module load python3.4/openmpi` will do it on Maxwell.
- Execute python script via mpirun/mpiexec
  - `mpirun –np 4 python3 script.py`
  - see docs for more info
- On Maxwell we use SLURM
  - one can omit number of processes and configure required resources via SLURM parameters
- Btw, numba will not compile in `@njit` mode for functions using mpi4py calls.

MPI

First Program

- Let's have a look at the first example:

/data/netapp/hpc-seminars/PythonMultiNodeParallelization/1_hello_world

  - copy to your folder
  - run it via SLURM – `sbatch run.sh`
  - check output
  - play with SLURM parameters and see what changes
    - --nodes, --tasks, --cpus-per-task, --ntasks-per-node
    - set parameters so that one instance of the program was run on two nodes
  - add measuring time and print it at the end (after `comm.Barrier()`)
    - should be printed only once

Finished? https://goo.gl/forms/79AIFWZXwCSqtT962

MPI

First Program - Solution

```
from mpi4py import MPI
import socket
import time
start = time.time()

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
node = socket.gethostname()

print('Hello, World! I\'m process #%d of %d, running on node %s' %
(rank, size,node))

comm.Barrier()

end = time.time()
if rank == 0:
    print("Elapsed %.6f sec" % (end - start))
```

MPI

First Program - Solution

```
#!/bin/bash
#SBATCH --ntasks=2
#SBATCH --nodes=2
#SBATCH --partition=all
#SBATCH -t 00:01:00

module load python3.4/openmpi
#python->python3
mpirun python ./hello_world.py
```

MPI

Communications

- mpi4py makes communications between processes easy
  - can communicate generic Python objects (using pickle)
  - can communicate buffer-like objects (NumPy arrays)

- MPI (mpi4py) supports different kinds of communications
  - communication categorized by locality
    - local (point-to-point)
    - global (collective, all-to-all)
  - communication categorized by operating mode
    - blocking
    - non-blocking

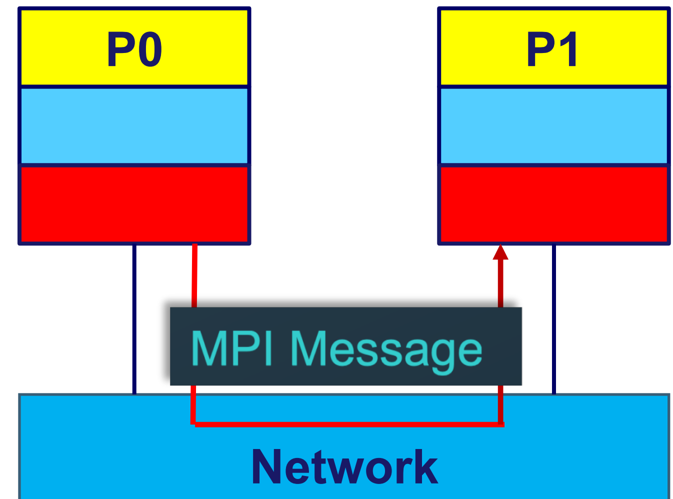
MPI

Point-to-point communications

- MPI implements interprocess communications via messages
  - Object/data to send
  - Who recieves the message
  - (Data type of the message)
  - (Message size)

  - Message Tag

  - Who sent the message
  - Where to store the received message



MPI

Point-to-point communications – Python Object

```
comm.send( obj, \ # Python object (number, dictionary, etc.)
           int_dest, \ # rank of destination process
           int_tag = 0) # message tag
result  =  comm.recv( int_source = none, \ # rank of source process
            int_tag = None, # message tag
            Status_status=None) # request status
```

- these are blocking calls – execution will stop until communication is finished

MPI

Point-to-point communications – NumPy arrays

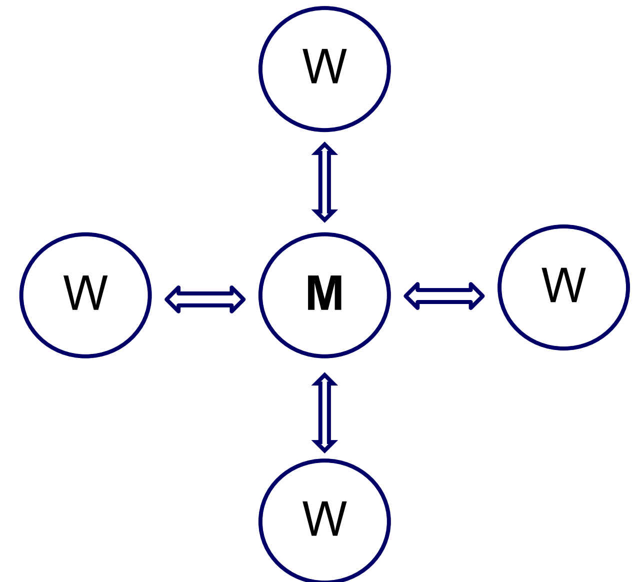
```
comm.Send( buf, \ # buffer
           int_dest, \ # rank of destination process
           int_tag = 0) # message tag
comm.Recv( buf, \ # buffer
           int_source = none, \ # rank of source process
            int_tag = None, # message tag
            Status_status=None) # request status
```

- buf - tuple like `[data, MPI.DOUBLE]`, or `[data, count, MPI.DOUBLE]` or just `data` for basic C types
- buf must be preallocated with enough space
- these are blocking calls – execution will stop until communication is finished

MPI

Master-Worker Approach

- Classical approach to parallel programming
  - One process is a master
  - The other processes are workers
  - Master collects results from workers
- Uses only Send and Recv
- Point-to-point communication pattern



MPI

Integration in parallel

$$\int_a^b f(x)\,dx$$

/data/netapp/hpc-seminars/PythonMultiNodeParallelization/2_integration

```
...

start_x = a + rank * (b - a) / size
int_local = trap(start_x, h, npoints_local)

if rank == 0:
    res = int_local
    for i in range(1, size):
        tmp = comm.recv(source = i)
        res = res + tmp
    print("Result: %.5f "%res)
else:
    comm.send(int_local, 0)
```

Finished? https://goo.gl/forms/79AIFWZXwCSqtT962

MPI

Deadlocks

- communication between processes can result in deadlock

```
other = 1 if rank == 0 else 0;
comm.Send(data_send,other);
comm.Recv(data_recv,other);
```

MPI

Deadlock

- /data/netapp/hpc-seminars/PythonMultiNodeParallelization/3_deadlock
  - copy to your folder, compile, run with two MPI processes (without SLURM or via an interactive job)
  - do you get deadlock?
  - correct the program

Finished? https://goo.gl/forms/79AIFWZXwCSqtT962

MPI

Deadlock - Solution

```
from mpi4py import MPI

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

neighbour = 1 if rank == 0 else 0

if rank == 0:
    res = comm.recv(source = neighbour)
    comm.send(1-rank,neighbour)
else:
    comm.send(1-rank,neighbour)
    res = comm.recv(source = neighbour)

print("received %d from %d"%(res,neighbour))
```

MPI

Non-blocking communications

- Motivation
  - Avoids deadlocks
  - Simplifies programming
  - Reduces syncronization
  - May allow to overlap communication and computation

- Requires additional request handle
  - Created for each nonblocking communication call
  - Used to check the communication state (wait/test operation)

MPI

Non-blocking communications

- Non-blocking send
  - r1 = comm.**isend**(obj, int_dest, int_tag = 0)
  - r1 = comm.**Isend**(buf, int_dest, int_tag = 0)
- Non-blocking receive
  - r2 = comm.**irecv(**int_source = none, int_tag = None)
  - r2 = comm.**Irecv**(buf, int_source = none, int_tag = None)
- Wait operations to finish
  - r1.**wait**()
  - r1.**Wait**()
  - r2.**Wait**()
  - result = r2.**wait**()

  or

  - MPI.Request.**Waitall**([r1, r2])

MPI – Non-blocking Communications

Deadlock

- /data/netapp/hpc-seminars/PythonMultiNodeParallelization/4_nonblocking
  - rewrite the program using non-blocking calls

Finished? https://goo.gl/forms/79AIFWZXwCSqtT962

MPI – Non-blocking Communications

Deadlock - solution

```
from mpi4py import MPI

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

neighbour = 1 if rank == 0 else 0

req_wait = comm.irecv(source = neighbour)
req_send = comm.isend(1-rank,neighbour)

res = req_wait.wait()
req_send.wait()

print("received %d from %d"%(res,neighbour))
```

MPI - Collective Communications

- Three types:
  - Synchronization (Barrier)
  - Data exchange (Scatter, Gather, Alltoall, Allgather)
  - Reductions (Reduce, Allreduce, Reduce_scatter)
- Usually blocking (non-blocking also possible)
- Can be implemented with point-to-point calls
- Collective implementation is usually better optimized (tree-based algorithms, etc.)

Barrier

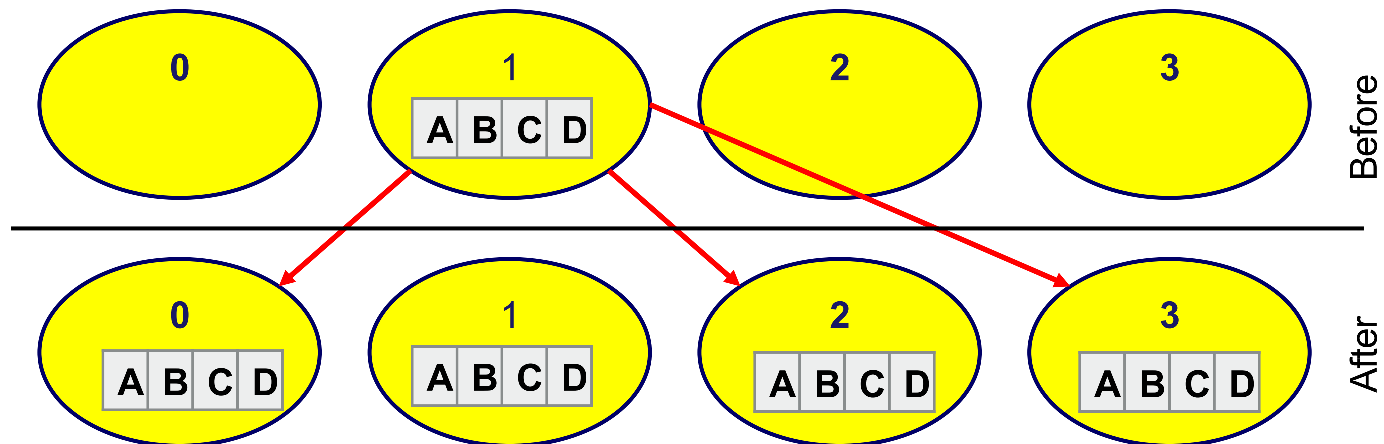
comm.**Barrier**()

- Explicit syncronization
- Block the process until all processes in the communicator called it
- Usually not needed
  - Syncronization is done implicitly by other communication calls
  - Can be used for debugging, profiling, etc.

Broadcast

result = comm.**bcast**(obj, int_root = 0)
comm.**Bcast**(buf, int_root = 0)

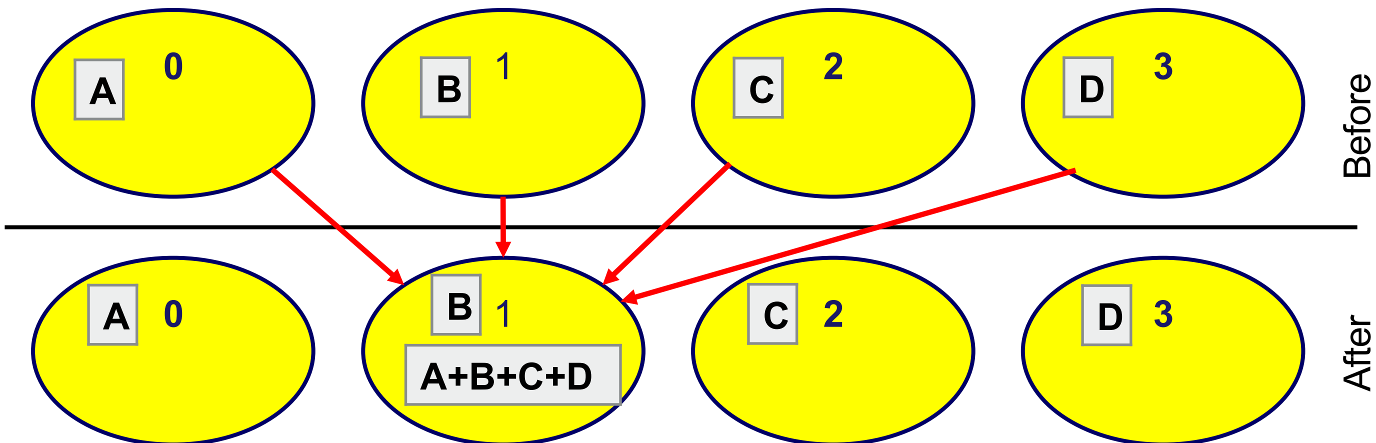
- One process (root) sends data to all others



Reduce

result = comm.**reduce**(sendObj,Op_op = None, int_root = 0)
comm.**Reduce**(sendbuf, recvbuf,Op_op = None, int_root = 0)

- Op_Op (MPI.SUM (default) , MPI.MAX, MPI.MIN, MPI.PROD, or user defined)



MPI

Integration in parallel

- /data/netapp/hpc-seminars/PythonMultiNodeParallelization/5_integration_collective
  - rewrite the program using comm.reduce

Finished? https://goo.gl/forms/79AIFWZXwCSqtT962

MPI

Integration in parallel - solution

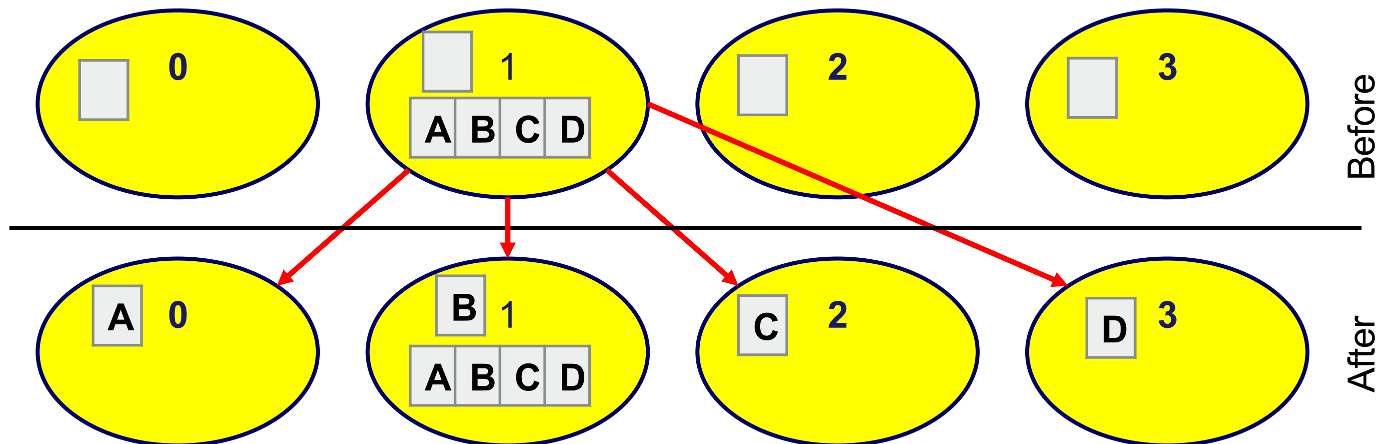
```
...
int_local = trap(start_x, h, npoints_local)
res = comm.reduce(int_local, root=0)

if rank == 0:
    print("Result: %.5f "%res)
```

Scatter

result = comm.**scatter**(sendObj, int_root = 0)
comm.**Scatter**(sendbuf, recvbuf, int_root = 0)

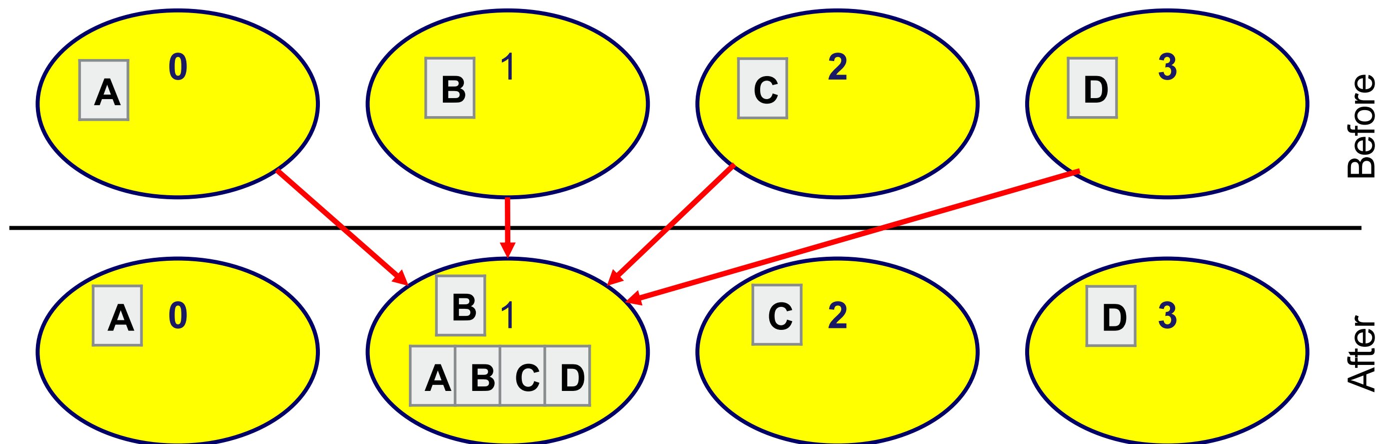
- One process scatters data to all others (including itself)



Gather

result = comm.**gather**(sendObj, int_root = 0)
comm.**Gather**(sendbuf, recvbuf, int_root = 0)

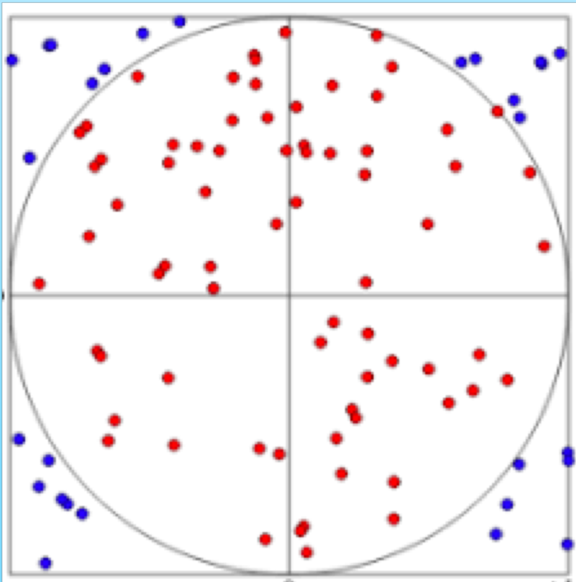
- One process gathers data from all others (including itself)



MPI – Collective Communications

Computation of Pi

- /data/netapp/hpc-seminars/PythonMultiNodeParallelization/6_pi
    - Create random points in process 0
    - send them to other processes (use comm.Scatter)
    - compute part of Pi on every process, use comm.reduce to get Pi.



$$\pi = 4\frac{A_{circle}}{A_{square}} \approx 4\frac{N_{in_circle}}{N_{in_square}}$$

Finished? https://goo.gl/forms/79AlFWZXwCSqtT962

MPI – Collective Communications

Computation of PI - Solution

```
...
npoints = int(sys.argv[1])
npoints_local = npoints//size

random_points = None
if rank == 0:
    random_points = np.random.rand(npoints_local*size*2)

random_points_local = np.empty(npoints_local*2)
comm.Scatter(random_points, random_points_local, root=0)

count_local = get_count(random_points_local,npoints_local)

count = comm.reduce(count_local, root=0)

if rank == 0:
    pi = 4.0 * (count / npoints)
    print("PI: %f"%pi)
```

Summary

- Splitting a problem into multiple parts is most challenging part of the parallelization process
  - the rest is just applying appropriate library calls
  - embarrassingly parallel is easy, but it is not “real HPC”
  - domain decomposition is a most commonly used approach
- Several options exist in Python to write a parallel program
  - mpi4py implements MPI library – general and widely used approach
- MPI starts specified amount of processes and runs an independent program instance on each of them
  - A set of send/receive routines provides point-to-point data exchange
    - Non-blocking calls may help to avoid deadlocks and be more efficient
  - Collective communications efficiently distribute/collect data within many processes