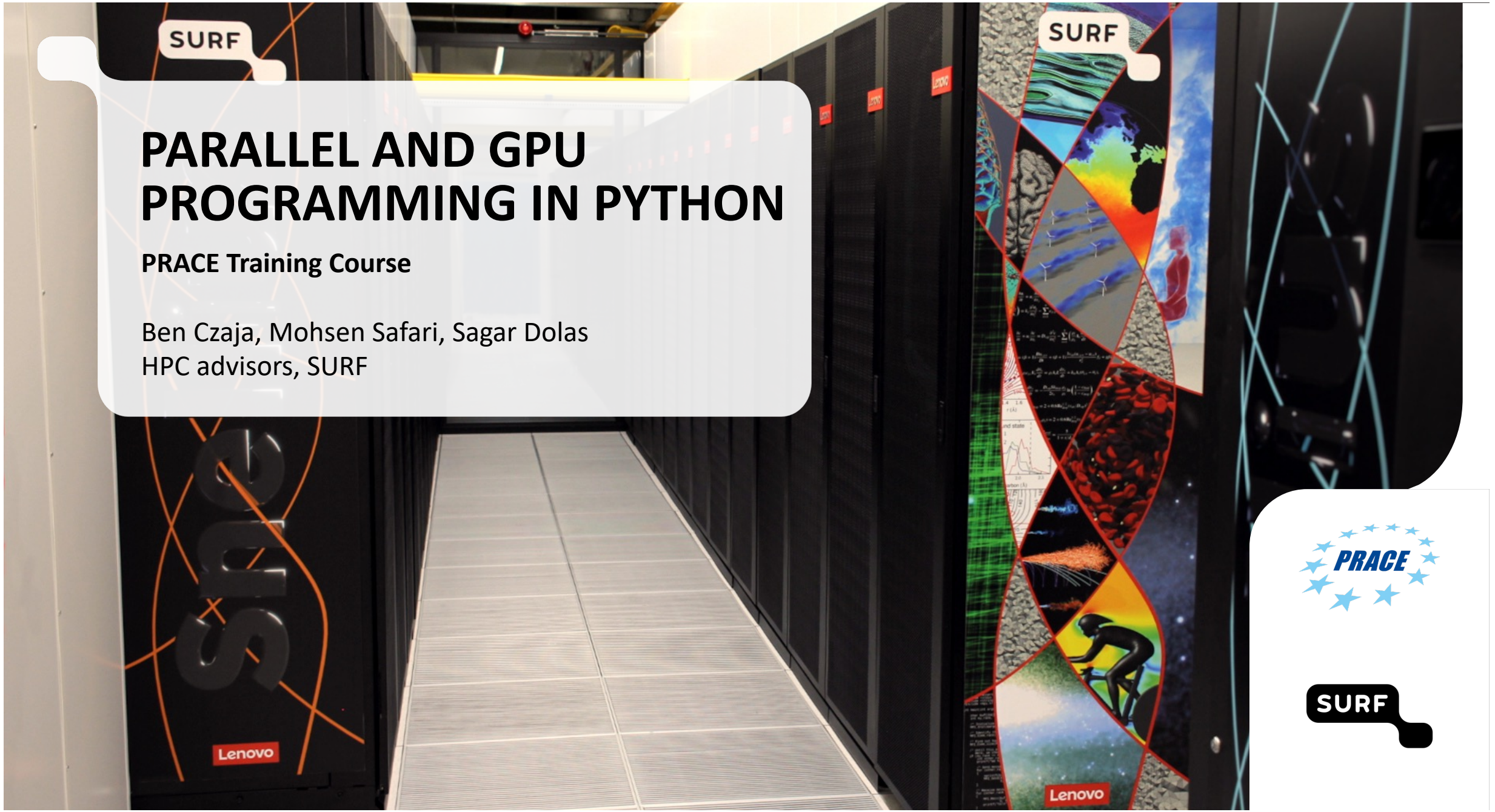


PARALLEL AND GPU PROGRAMMING IN PYTHON

PRACE Training Course

Ben Czaja, Mohsen Safari, Sagar Dolas
HPC advisors, SURF

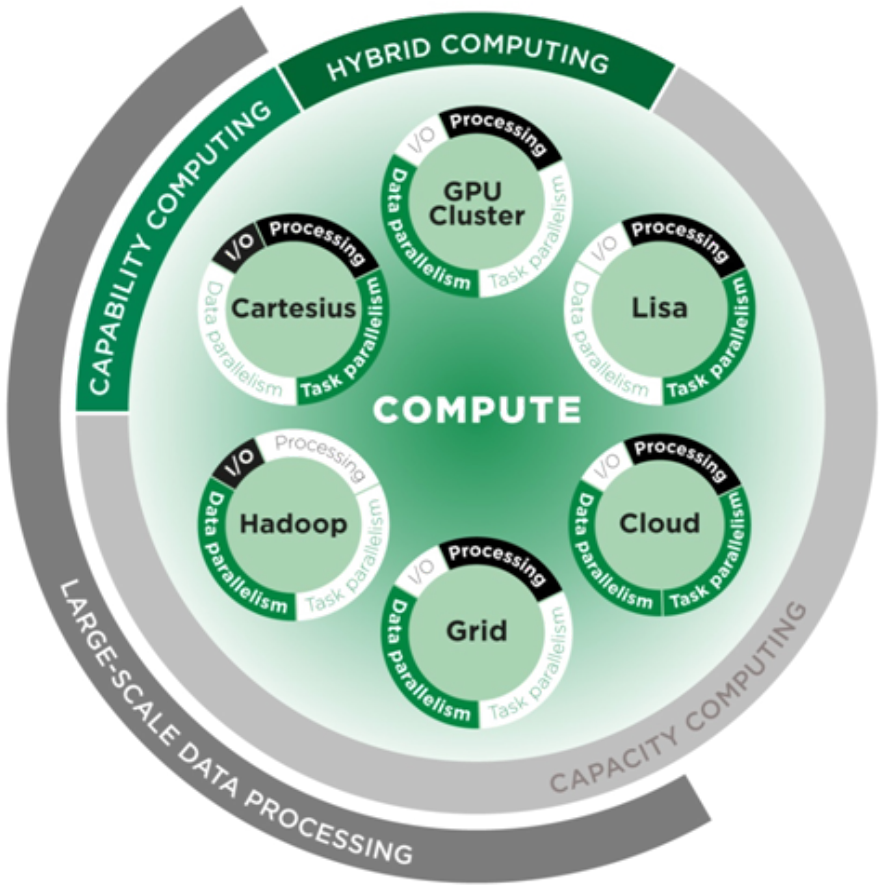
PRACE

SURF

SURF is an ICT cooperative for education and research

At the SURF cooperative, education and research work together to make full use of the opportunities offered by digitalisation,
with the aim: making education and research better and more flexible.

SURF Compute Services



- Low-latency, high-bandwidth national supercomputer
- Capacity compute cluster with capacity computing services
- IAAS, PAAS and SAAS on-demand Cloud services
- Loosely coupled compute and data Grid farms
- Big Data Hadoop cluster and software Services
- Special-purpose services:
  - remote visualization
  - accelerators (GPUs, Xeon Phi)
  - high memory

Towards unifying these platforms into a
Unified Computing Architecture (UCA)
for efficiency, scaling and elasticity

National compute & data infrastructure

High Performance Computing Services

National Supercomputer Snellius (Phase 1)

Lenovo
76,992 cores, 144 NVIDIA A100
202 TB memory
Linpack Performance (Rmax) 2,127.8 TFlop/s
Theoretical Peak (Rpeak) 3,067.08 TFlop/s

National Compute Cluster LISA

Dell cluster
6,876 cores, 92 NVIDIA GTX 1080 Ti,
8 Titan V, 116 Titan RTX, 34 TB memory,
400 TB disk

Data Analysis Services

Visualization

Viz on HPC
Collaboratorium

National Grid

(partners Nikhef & RUG-CIT)
10,000+ cores

Big Data Analytics Cluster

768 cores
1.7 PB disk

HPC Cloud

1,080 cores
500 TB disk

Data Storage Services

Grid mass-storage & Central Archive

9 PB disk
21 PB tape

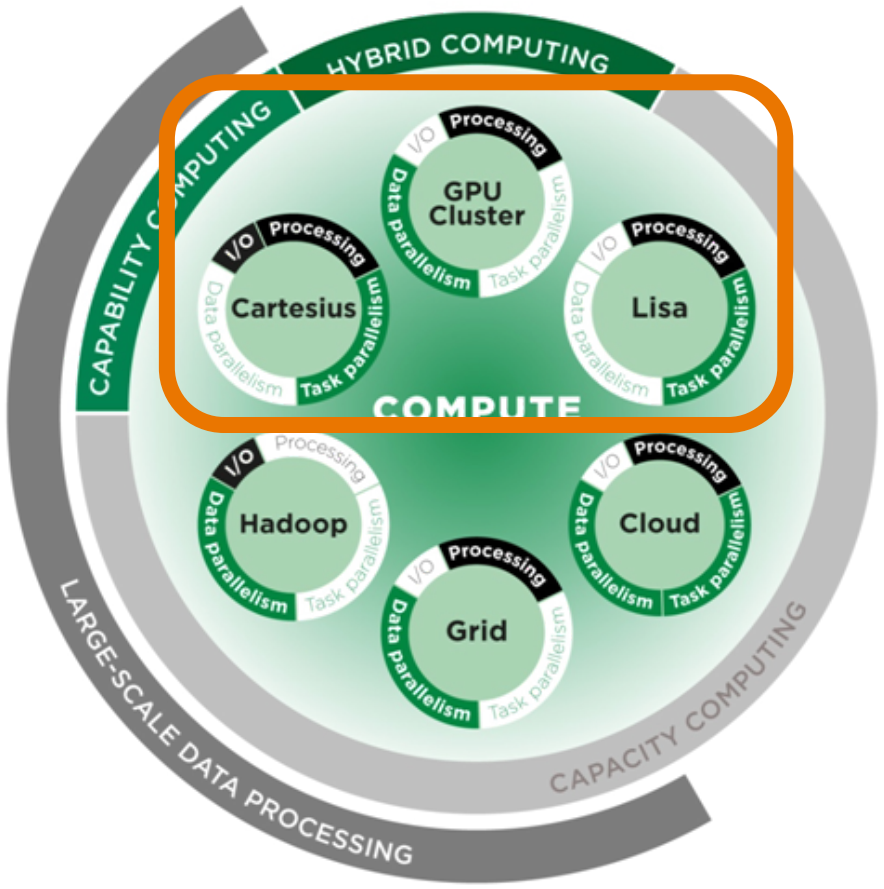
SURFdrive

> 150 TB storage
> 25,000 users

PID service

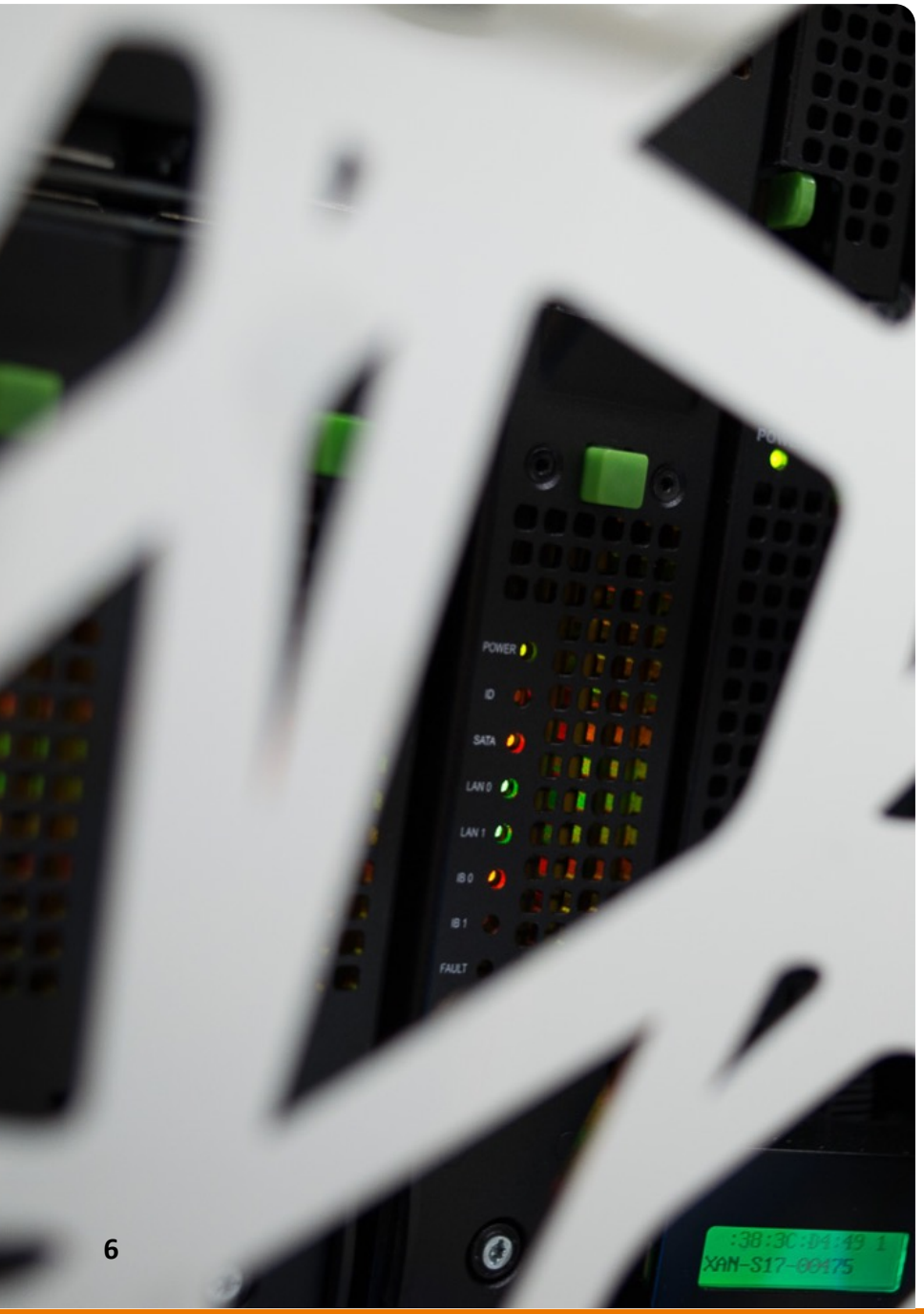
EPIC (handle)

SURF Compute Services



Working in the High Performance and Visualisation team

- Maintain e-infrastructure and services
- User support
- Technical Trainings
- Development and Innovation
- National and International projects



SURF and HPC in Europe

- HPC resources access, and benchmarking
- Best practices in HPC and trainings

www.prace-ri.eu

- Scientific community support
- CompBioMed CoE

www.compbiomed.eu

- A new family of low-power European processors
- https://www.european-processor-initiative.eu

SURF and HPC in Europe

- **Convergence of AI and HPC**

Intel Parallel Compute Centre

www.examode.eu

- **Quantum computing**

Training at SURF

https://www.surf.nl/en/agenda/research-and-ict

SURF Research & ICT

- Introduction to supercomputing
- HPC Cloud and Data management
- ML/DL and Scientific Visualisation
- Domain specific trainings
- Collaborations with national and international organisations

Partnership for Advanced Computing in Europe (PRACE)

- Enable high impact scientific discovery
- Engineering research and development across all disciplines
- Enhance European competitiveness for the benefit of society
- Established as an international not-for-profit association with seat in Brussels
- Collaboration between 26 member countries whose representative organizations create a pan-European supercomputing infrastructure
- Extensive education and training effort: seasonal schools, workshops...

PRACE

SURF

Partnership for Advanced Computing in Europe (PRACE)

- BSC - Barcelona Supercomputing Center (Spain)
- CSC - IT Center for Science (Finland)
- CINECA - Consorzio Interuniversitario (Italy)
- EPCC at the University of Edinburgh (UK)
- GCS - Gauss Supercomputing Center (Germany)
- GRNET - Greek Research and Technology Network (Greece)
- ICHEC - Irish Centre for High-End Computing (Ireland)
- IT4I - IT4Innovations National Supercomputing Center (Czech Republic)
- MdlS - Maison de la Simulation (France)
- SURF (The Netherlands)

PRACE training centre at SURF

http://www.training.prace-ri.eu/

- Organization of training workshops from 1 to 3 days
- “In person” events in the Netherlands (Amsterdam, Utrecht) and online
- All trainings and materials are provided in English

PTC: Parallel and GPU programming in Python

- Introduction to parallel processing
- Introduction to Python basics
- Parallel codes with Python
- MPI and GPU programming in Python
- Hands-on and Q&A!!!
- Slack workspace to keep the discussion active

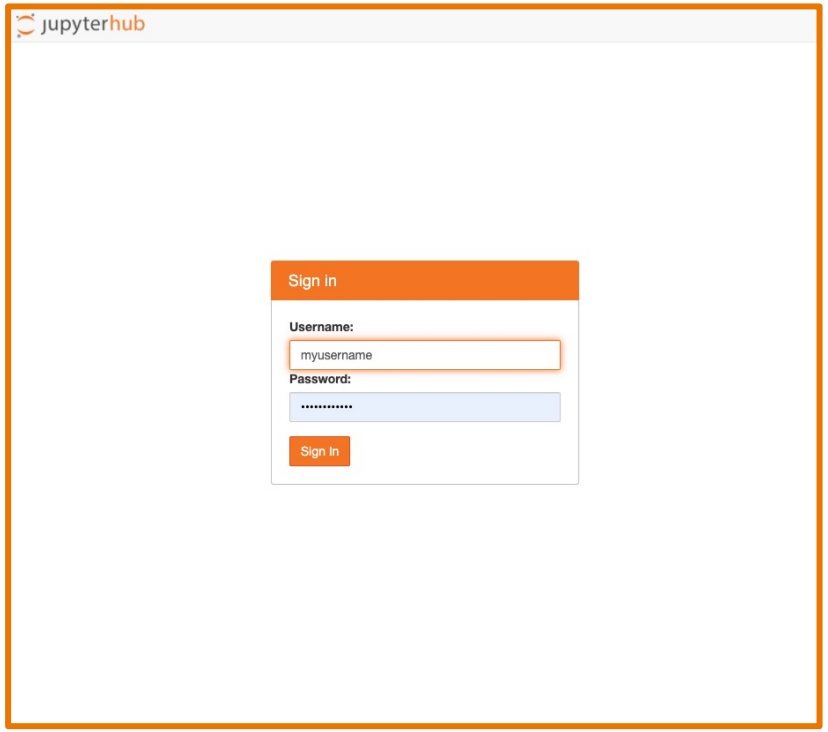
PTC: Parallel and GPU programming in Python

Start Time	End Time	Subject
09:00	10:00	Welcome and Introduction to the Course
10:00	10:45	Introduction to Python and Parallel Computing
10:45	11:00	Coffee Break
11:00	12:30	Hands-on: Introduction to Python and Parallel Programming (threading)
12:30	13:30	Lunch
13:30	15:00	Hands-on Parallel Programming with Python (multiprocess)
15:00	15:15	Coffee Break
15:15	17:00	Hands-on Parallel Programming with Pythin (Mpi4Py)

PTC: Parallel and GPU programming in Python

- **Set up of the training**

  1. Account on the system
  2. Access to the Jupyter Hub
  3. Training material and hands-on (on the cluster!)
  4. https://github.com/sara-nl/Parallel-and-GPU-programming-in-Python



- **Slack workspace**

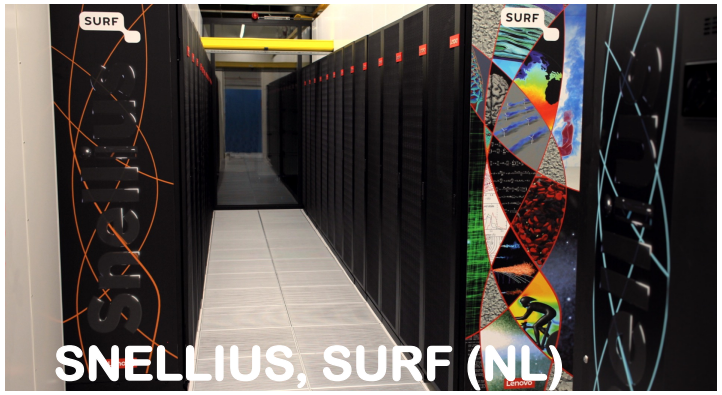
  URL: https://jupyter.lisa.surfsara.nl/jhlsrf014

INTRODUCTION TO PYTHON AND PARALLEL COMPUTING

OUTLINE

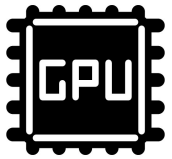
- Working with a Supercomputer
- Parallel vs serial computing
- Basic concepts of parallel computing
- Introduction to Python and Jupyter

Working with a Supercomputer



User Experience

- Multiuser system
- Unix OS
- Optimized software



Compute power

- Many CPUs system
- Specialized Hardware
- Low-latency/High bandwidth Connections

Storage

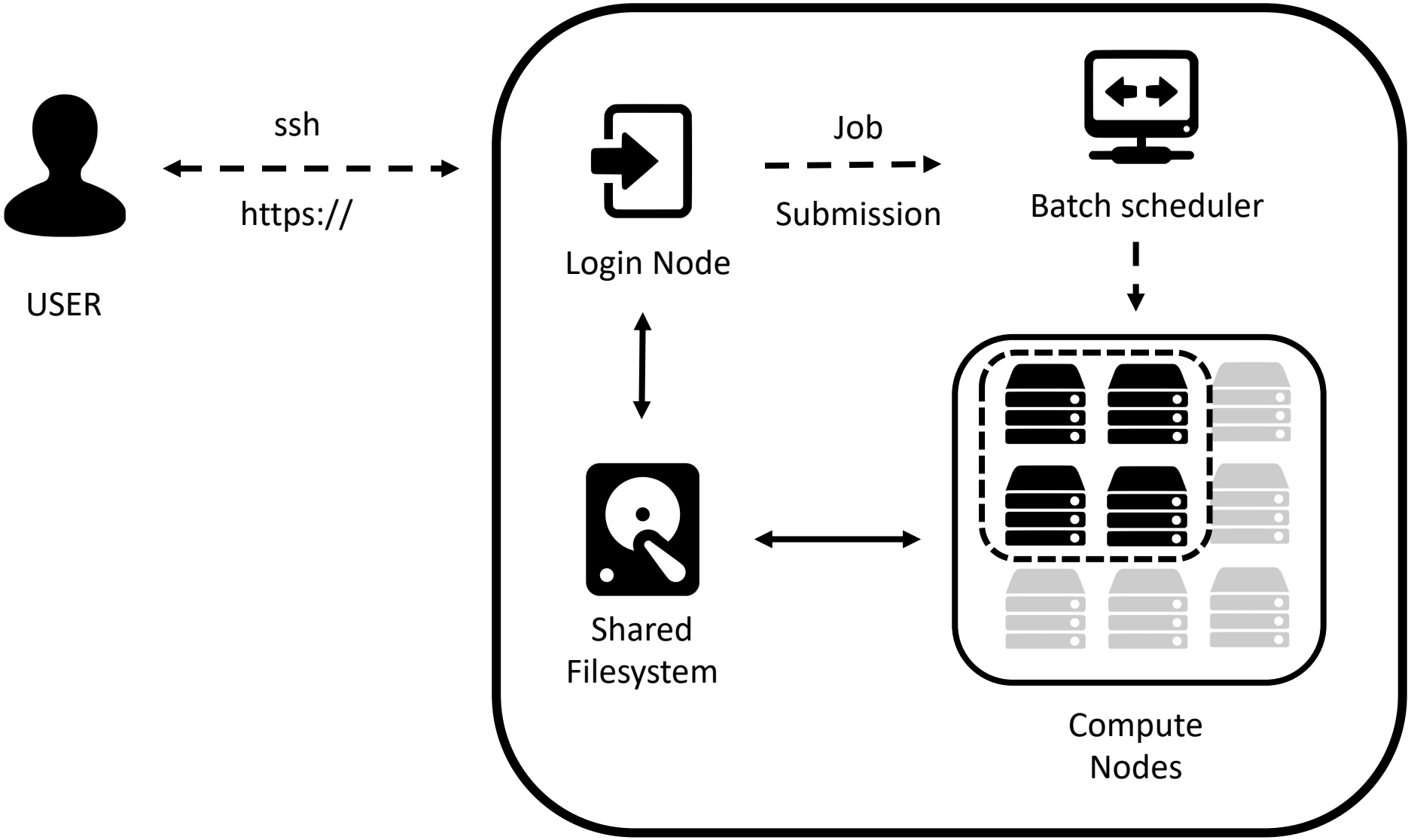
- Efficient I/O
- Large Memories

Working with a Supercomputer

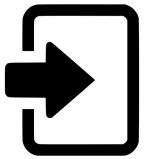
Is NOT like this….

Working with a Supercomputer

Working with a Supercomputer

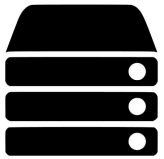


Working with a Supercomputer



Service node(s)

- Editing and transferring files
- Compile programs
- Prepare simulations



Compute nodes

- Multicore architectures
- Large memory
- GPUs and other accellerators

Network

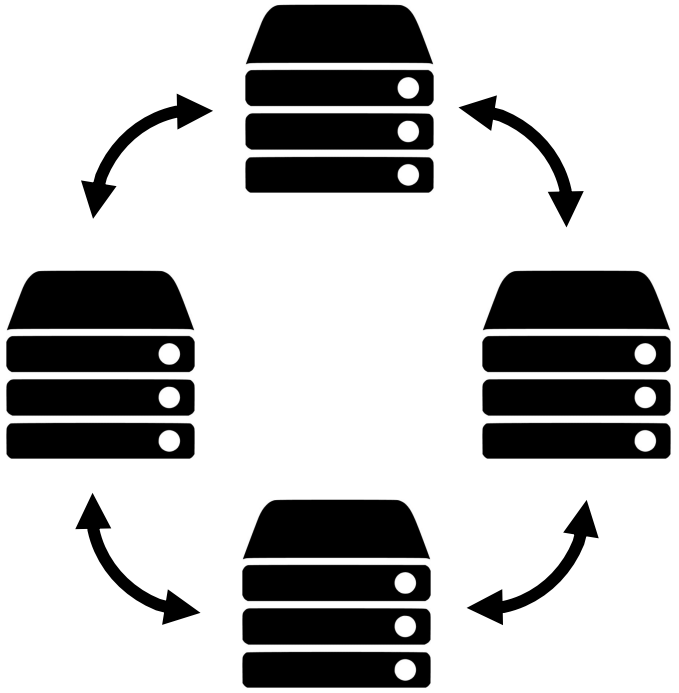
- High-speed interconnections
- Low latency
- High bandwidth

File system

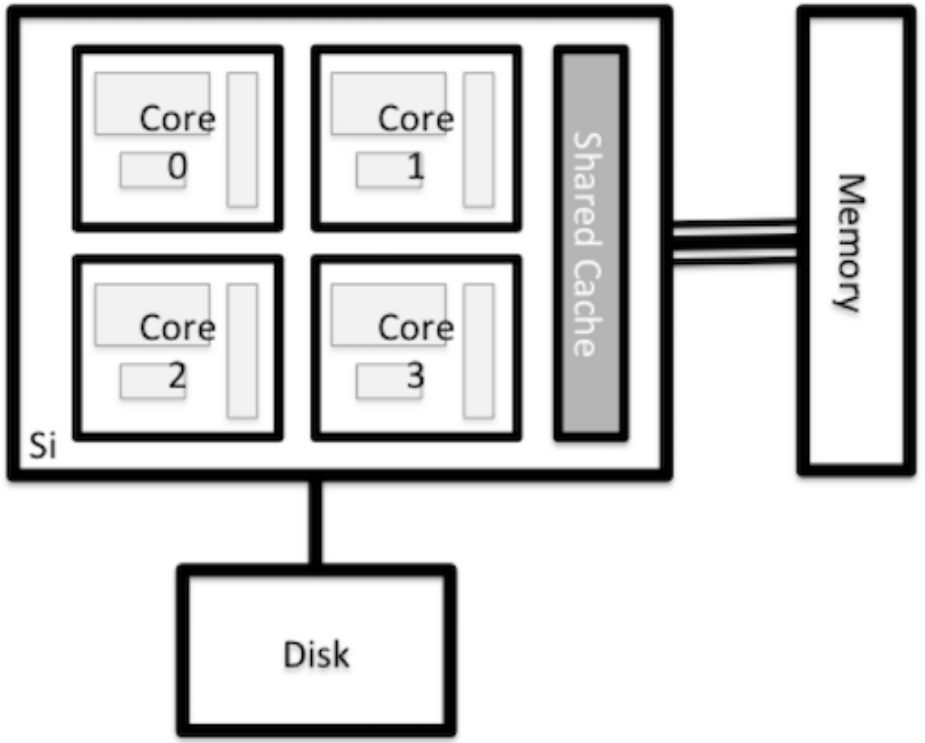
- Parallel read/write
- Long term storage
- Node local disks

Working with a Supercomputer

The batch jobs scheduler



Interconnected
Compute Nodes

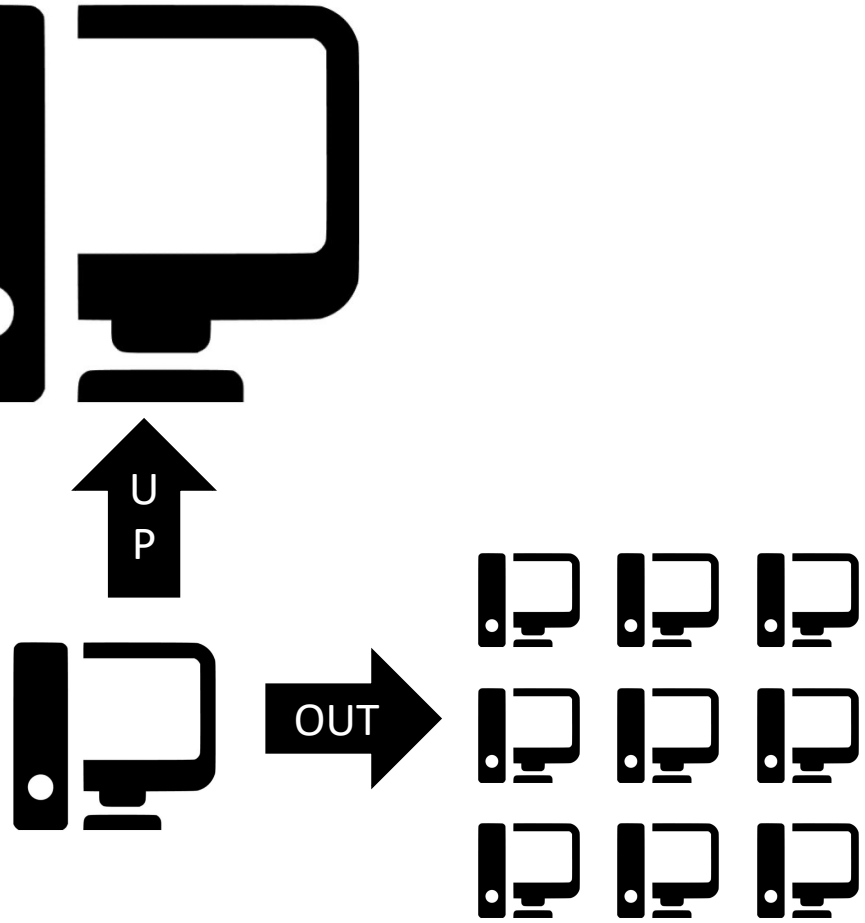


Source: https://epcced.github.io/hpc-intro

Working with a Supercomputer

Why, or more, when you need a Supercomputer?

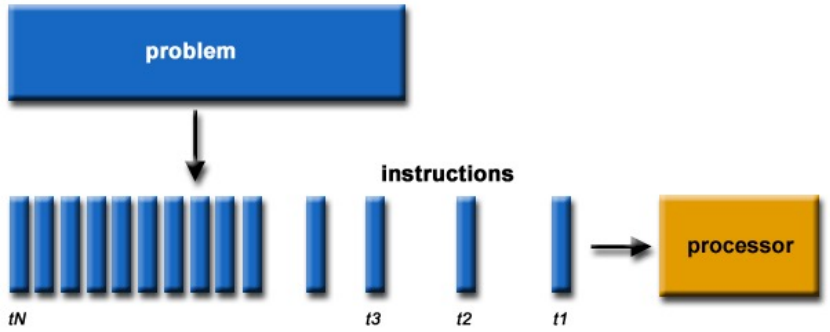
- **Scale up**
  - Faster CPUs
  - Large memories
  - Specialized Hardware/Software
- **Scale out**
  - Large parallel applications
  - Many small- to medium- size jobs



Introduction to parallel computing

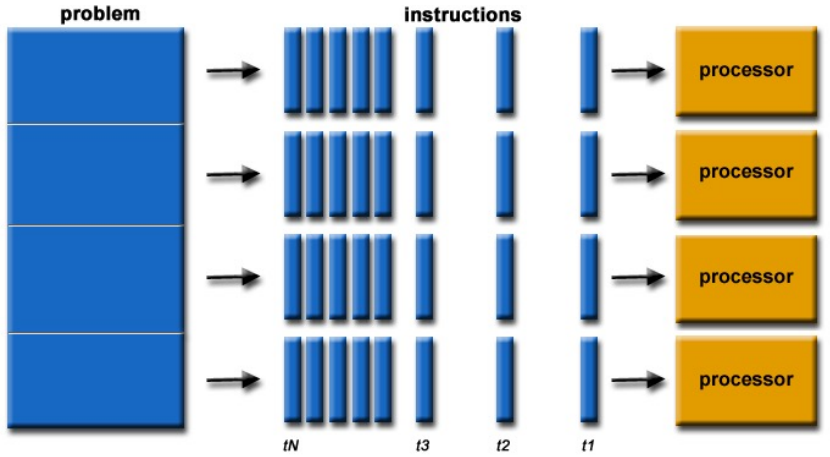
Serial computing

- A problem is broken into a discrete series of instructions, which are executed sequentially on a single processing unit (core).



Parallel computing

- A problem is broken into discrete parts that can be solved using simultaneously multiple resources.



credits: https://computing.llnl.gov/tutorials/parallel_com

Introduction to parallel computing

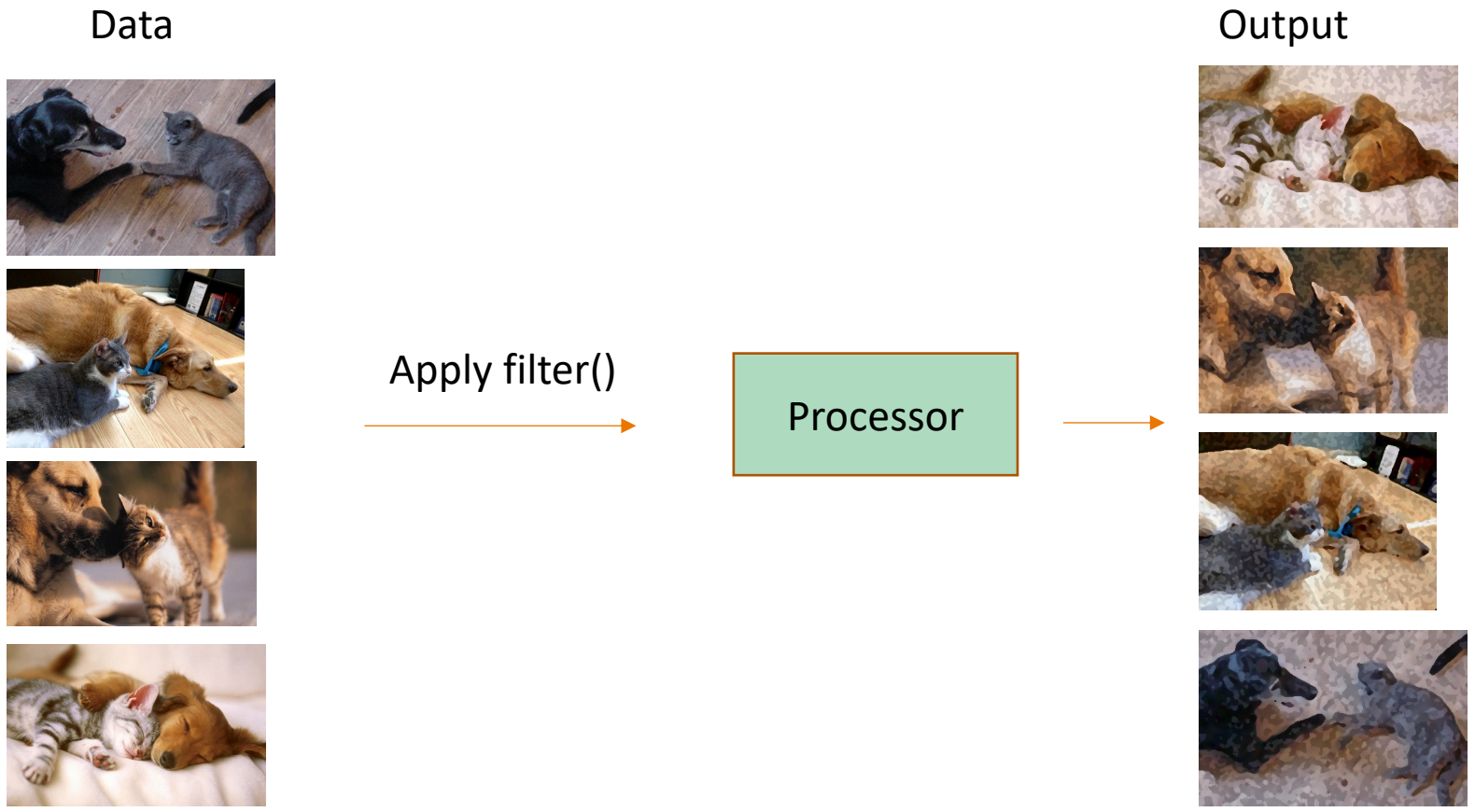
The goal is to understand:

- Merits and limits of parallel computing
- Parallel programming models (task / data parallelism)
- Differences between shared and distributed memory systems

Introduction to parallel computing

Parallel computing

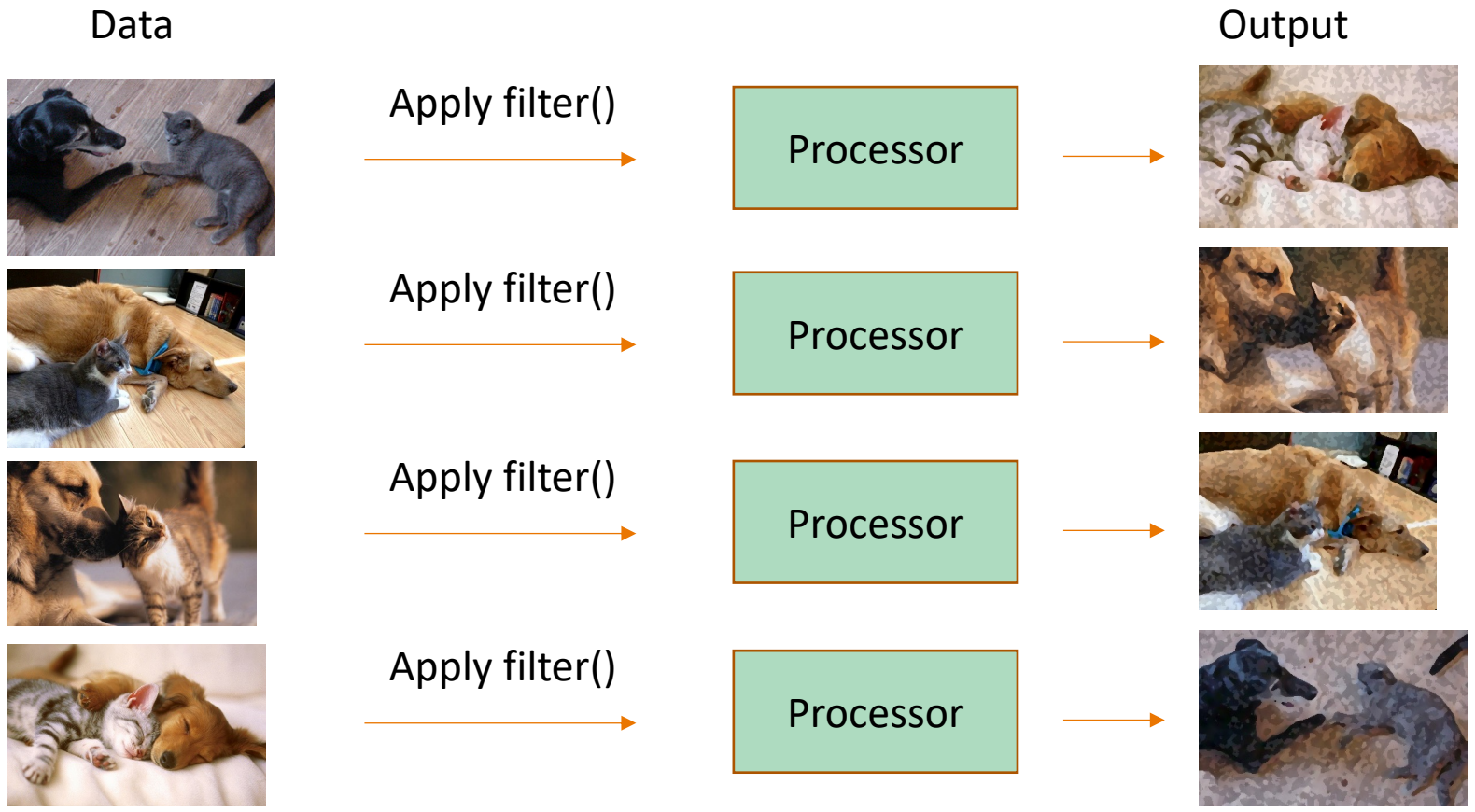
- Multiple processors or computers working on a single computational problem



Introduction to parallel computing

Parallel computing

- Multiple processors or computers working on a single computational problem



Introduction to parallel computing

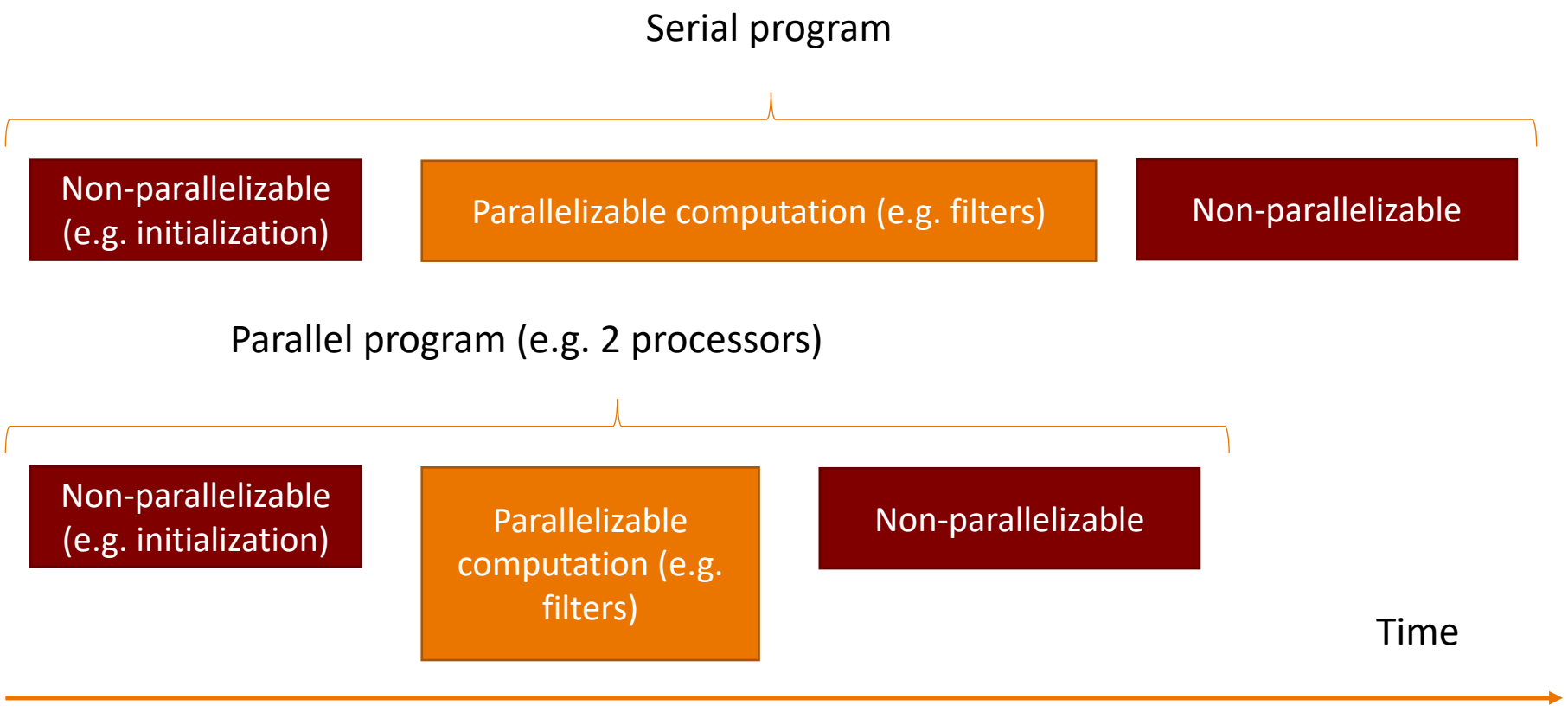
Benefits

- Solve computationally intensive problems (speedup)
- Solve problems that don’t fit a single memory (multiple computers)

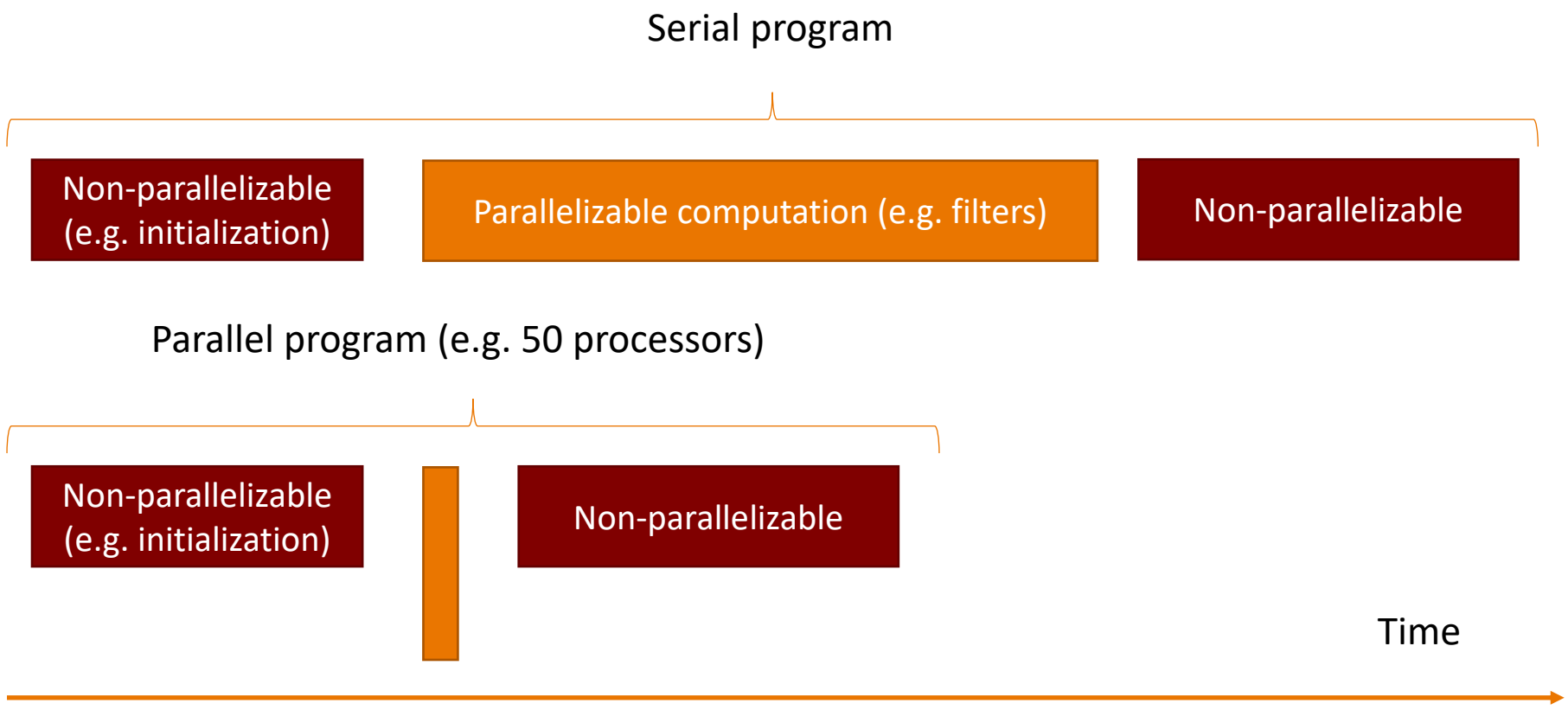
Requirements

- Problem should be divisible in smaller tasks

Introduction to parallel computing



Introduction to parallel computing



Introduction to parallel computing

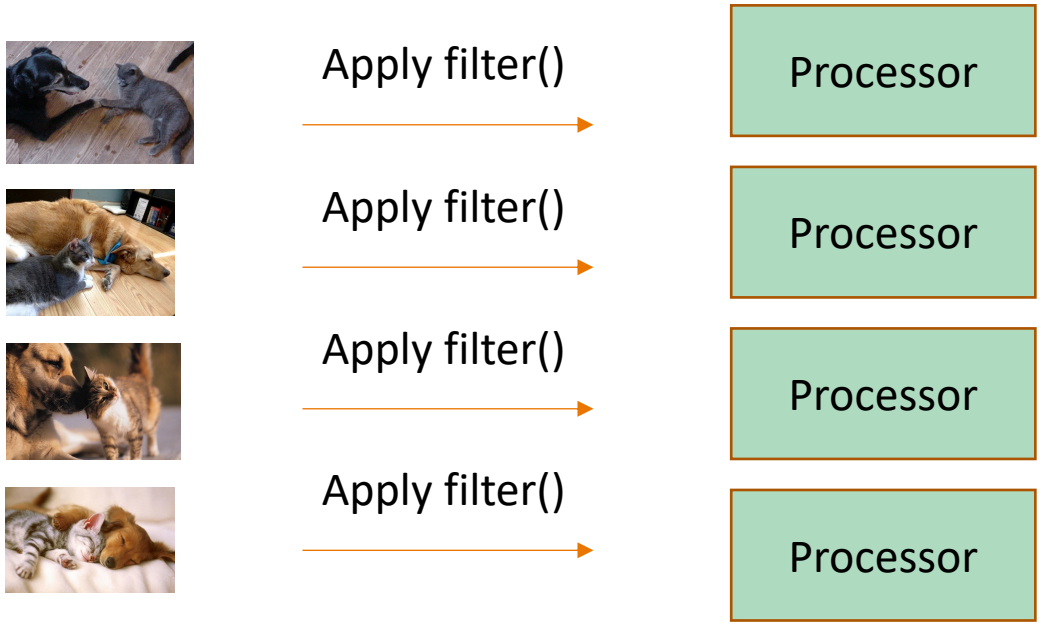
Strong scaling

- Variation of solution time with #processors for fixed *total* problem size
- Possibilities to run the same problem in shorter time

Introduction to parallel computing

Strong scaling

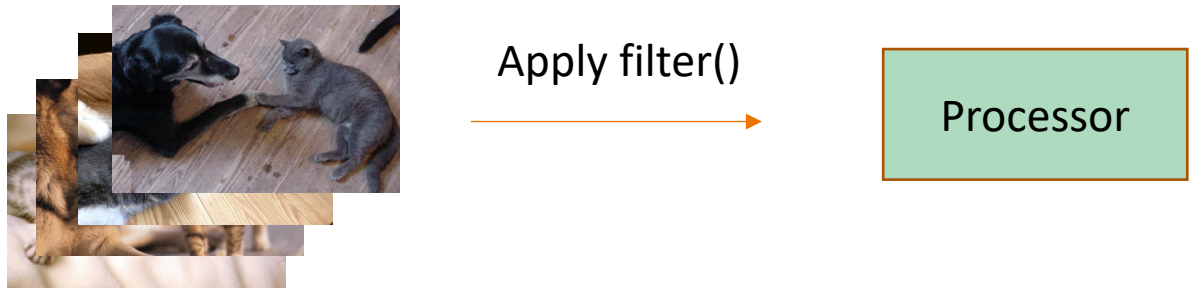
- Variation of solution time with #processors for fixed *total* problem size
- Possibility to run the same problem in shorter time



Introduction to parallel computing

Weak scaling

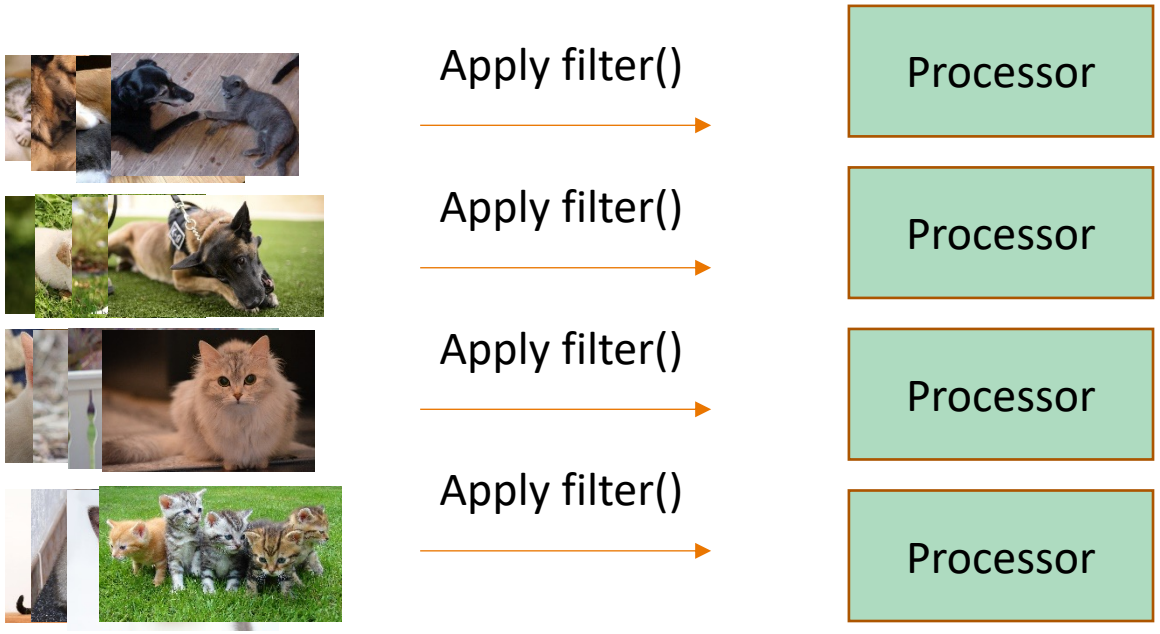
- Variation of solution time with #processors for fixed problem size *per processor*
- Possibilities to run a bigger problem in the same time



Introduction to parallel computing

Weak scaling

- Variation of solution time with #processors for fixed problem size *per processor*
- Possibilities to run a bigger problem in the same time



Introduction to parallel computing

- Weak scaling is often the most relevant to HPC:
  - Physics: ”I can only run my fluid simulation a small domain / low resolution on my local PC”
  - Chemistry: “I can simulate a small molecule on my PC, but I want to simulate a big one”
- Common background
  - A big system / molecule increases the *total* work
  - Distributing larger work over multiple processors keeps the work *per processor* constant
  - … this is weak scaling!

Introduction to parallel computing

Parallel programming models

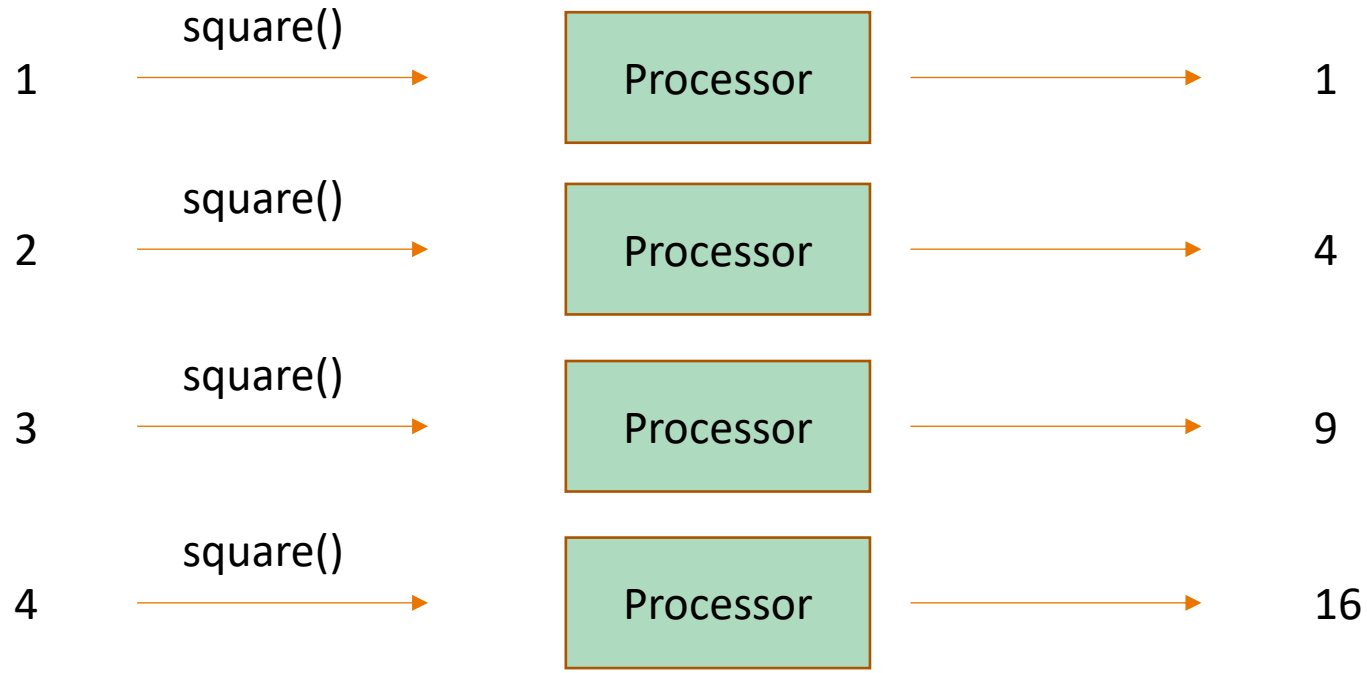
Flynn's Classical Taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

- Two well-known programming models
  - Data parallelism
  - Task parallelism

Introduction to parallel computing

Data parallelism

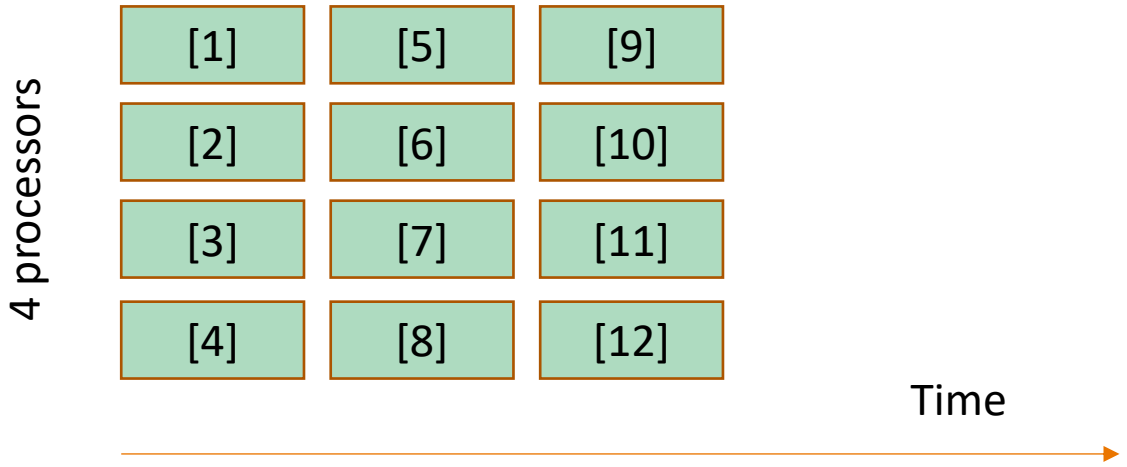
- Each processor performs the same task on different data



Introduction to parallel computing

Data parallelism

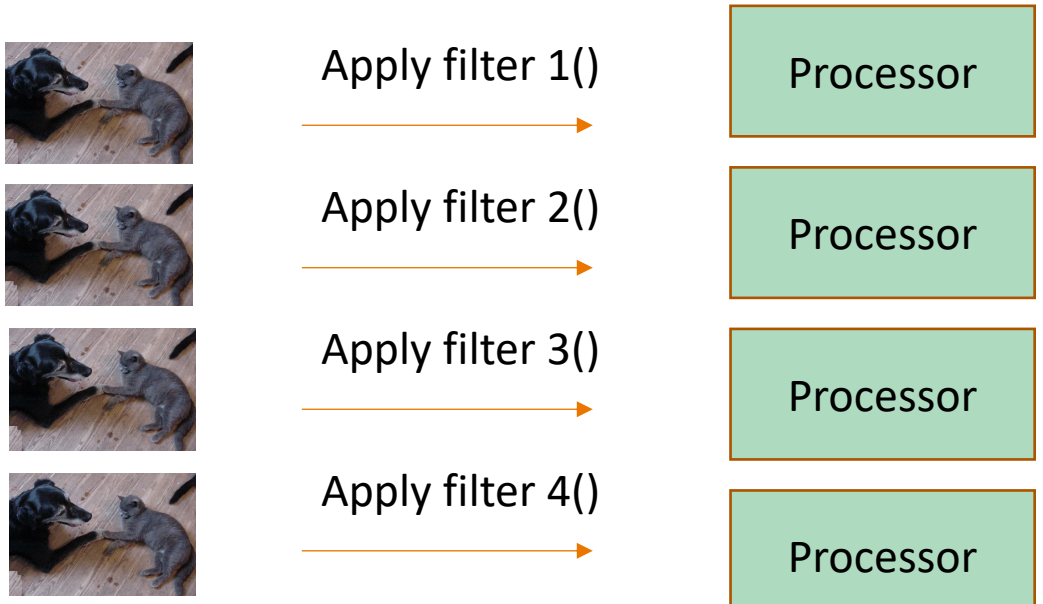
- Amount of parallelization depends on input data size
- Load balancing may be relatively easy
  - Same task on each data element
  - Approximately same time per element



Introduction to parallel computing

Task parallelism

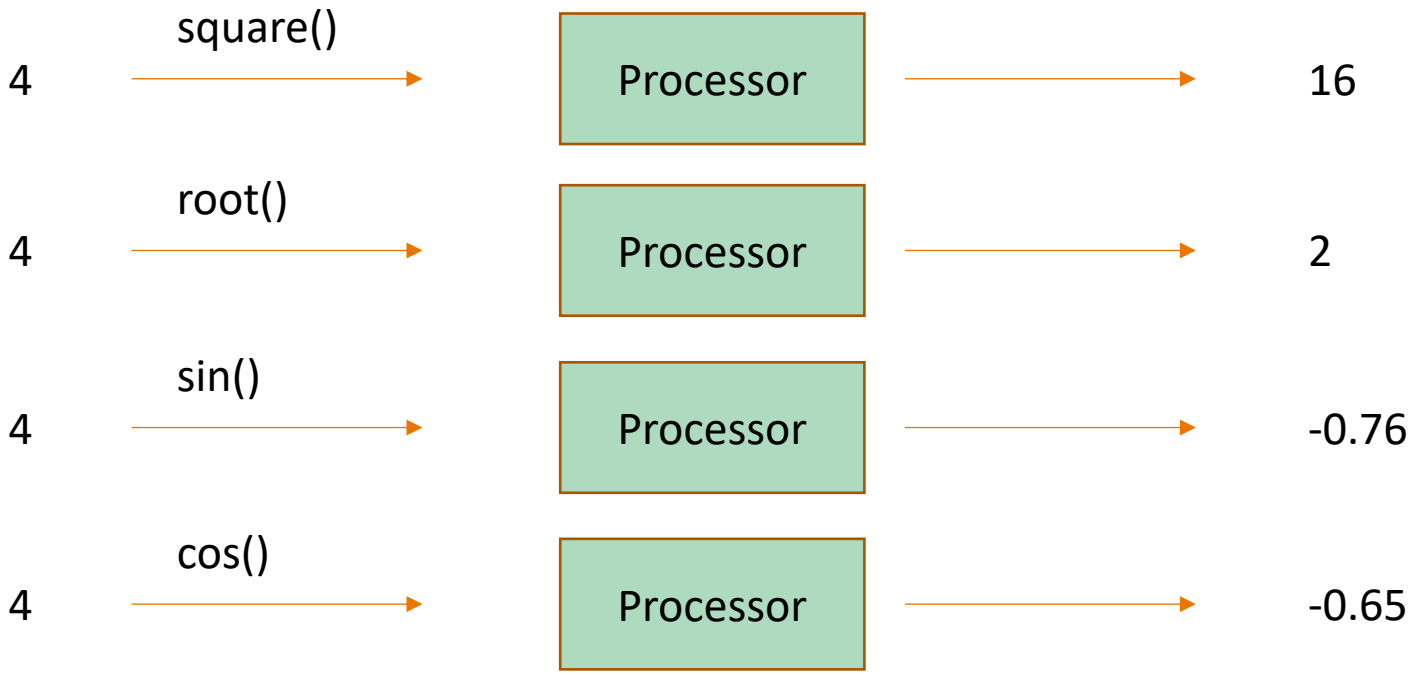
- Each processor performs a different task on the same data



Introduction to parallel computing

Task parallelism

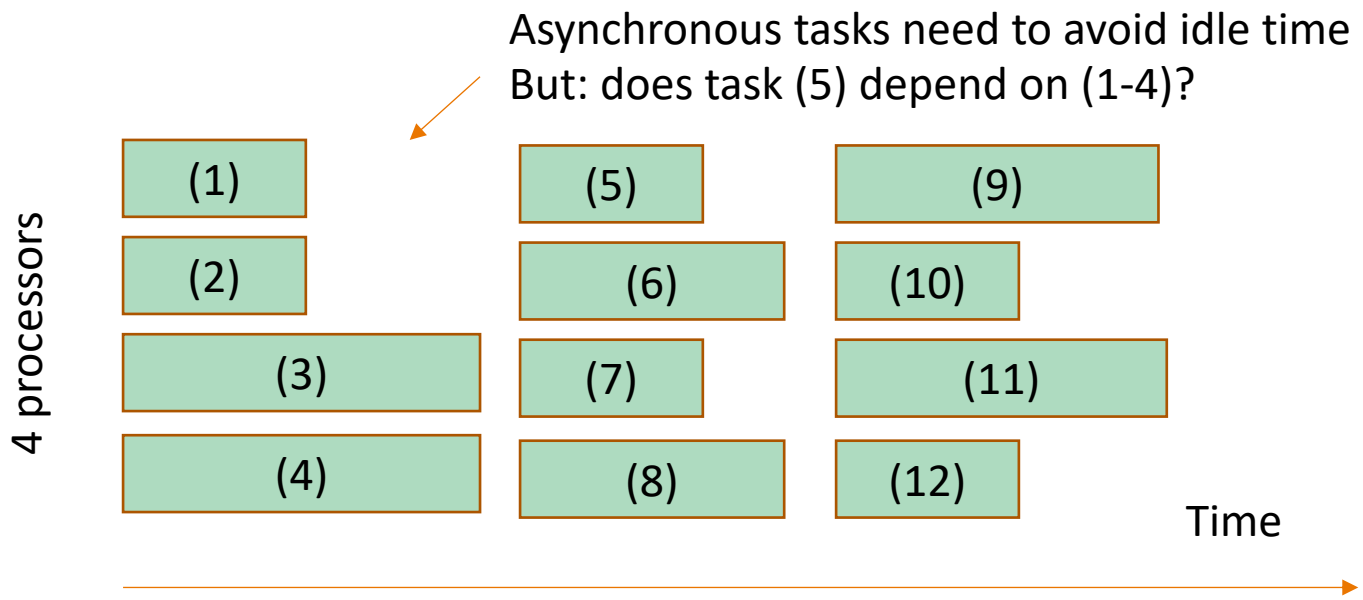
- Each processor performs a different task on the same data



Introduction to parallel computing

Task parallelism

- Amount of parallelization depends on the number of tasks
- Load balancing can be very difficult
  - Heterogeneous tasks may be executed over the same data
  - Each task may take a very different amount of time



Introduction to parallel computing

General computing architectures

- Four main components:
  - Memory
  - Control unit
  - Arithmetic Logic Unit
  - Input/output
- Memory is used to store both
  - Program instructions
  - Data used by the program

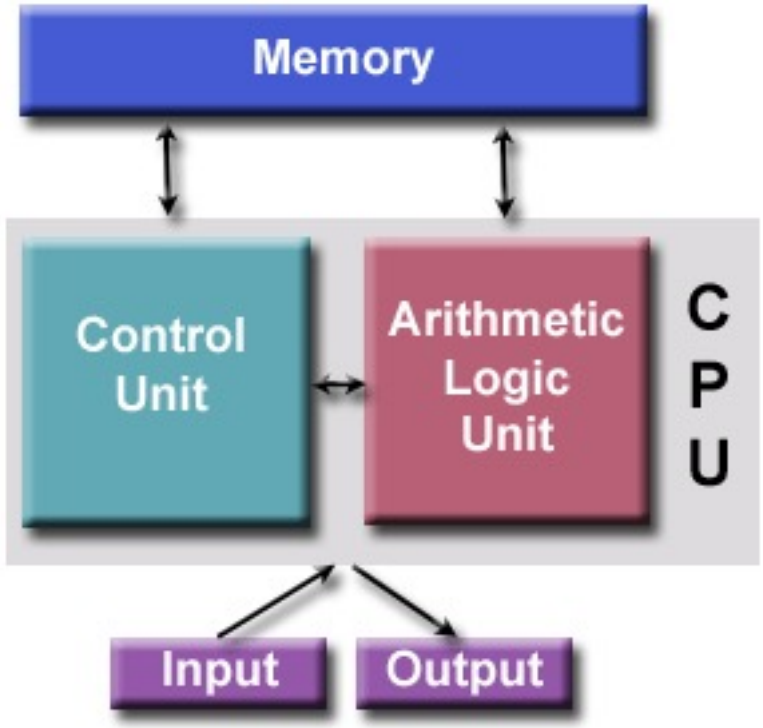


Image source: computing.llnl.gov

Introduction to parallel computing

Parallel computing architectures

- Shared memory
  - All processors access the same memory
  - Different sequences of execution (threads) run on the same process
  - Different memory modules may be used, but only one logical memory space is addressed
  - Communication between processors is done implicitly
- Distributed memory
  - Processes use their own memory
  - Tasks access data from other tasks through communications
  - Multiple tasks can reside on the same physical machine
  - Require libraries. Almost always using MPI

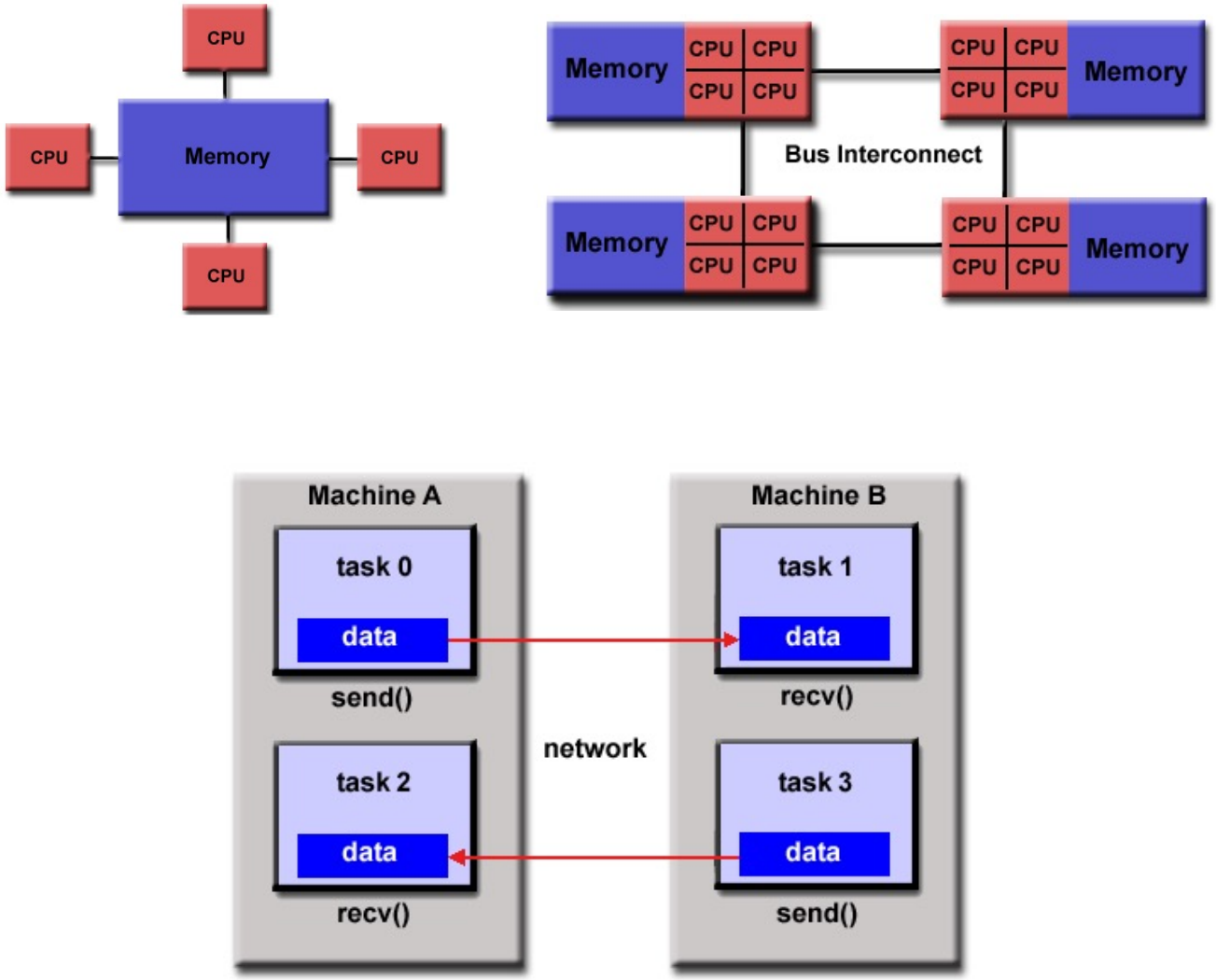
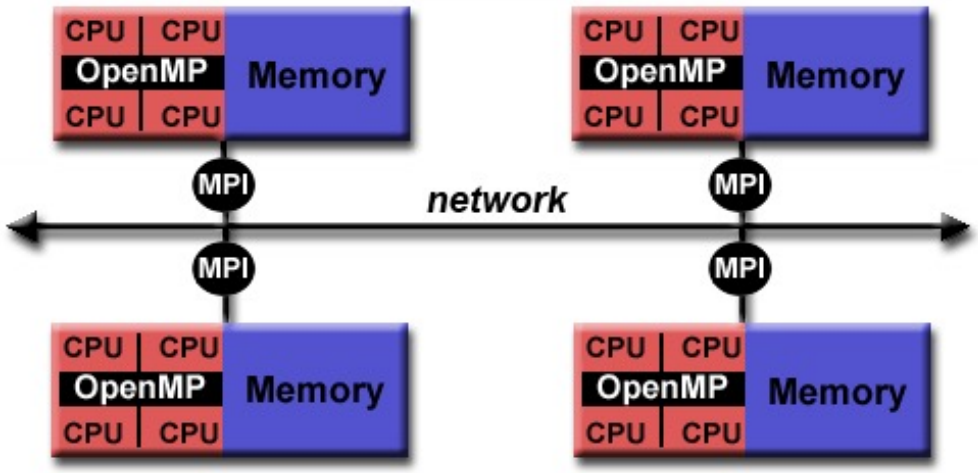


Image source: computing.llnl.gov

Introduction to parallel computing

Parallel computing architectures

- Hybrid model (e.g. Supercomputers)



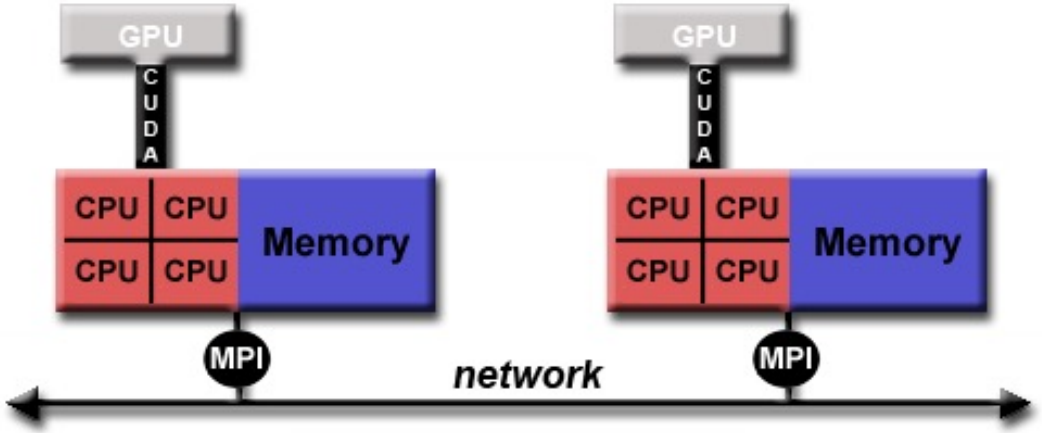


Image source: computing.llnl.gov

Introduction to parallel computing

Parallel computing models

- Task parallel
  - many independent runs
  - needs orchestration
  - for monte-carlo, parameter sweeps
- Shared memory
  - always within one batch node
  - uses threads
  - often implicit
- Distributed memory
  - can use one or more batch nodes
  - uses separate processes
  - almost always using MPI
  - for PDE problems, time stepping

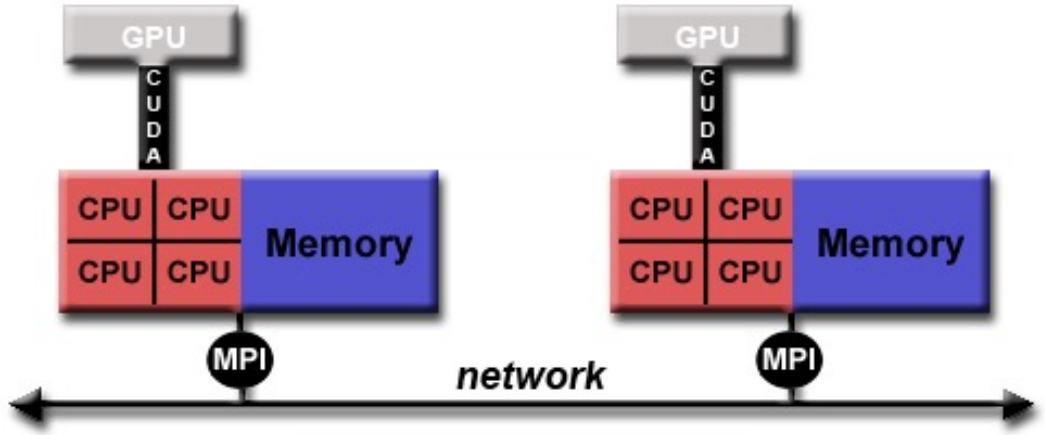


Image source: computing.llnl.gov

HANDS-ON:

INTRODUCTION TO EFFICIENT PYTHON CPU PROGRAMMING

OUTLINE

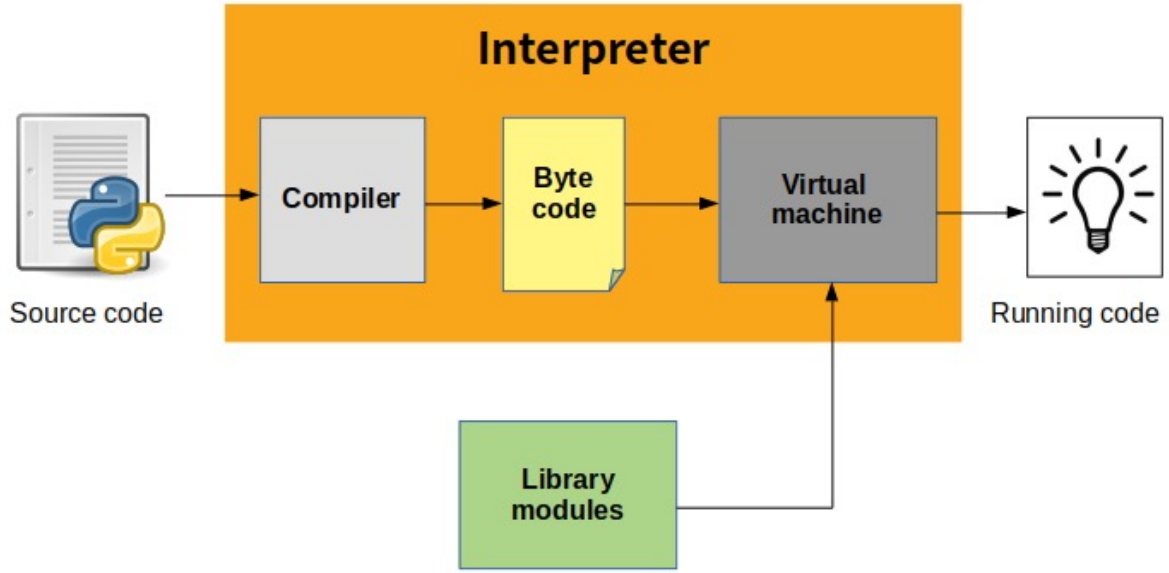
- Getting acquainted with the SURF Jupyter Hub
- Working with the Jupyter environment
- Running and timing a simple Python function

Parallel computing with Python

Programming with Python

https://www.python.org/

- Modern, interpreted programming language
- Object oriented
- Portable (Unix/Linux, Mac OS X, Windows)
- Open source
- Readable and simple syntax
- Many standard and third party libraries
- Can be used interactively



Parallel computing with Python

Python for High-Performance Computing

- How to run Python on an HPC system
  - Using the command line interpreter
    - Interactive
    - *python mypythoncode.py*
  - **Jupyter Notebook App**
    - Server-client application can be installed on a remote server and accessed through the internet

```
Python 3.8.2 (default, Jul 31 2020, 22:06:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import os

In [2]: import numpy

In [3]: m = numpy.array([0,0,1])

In [4]: m.dot()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-79e2982f1c0e> in <module>
----> 1 m.dot()

TypeError: dot() missing required argument 'b' (pos 1)

In [5]: b = numpy.array([0,1,0])

In [6]: m.dot(b)
Out[6]: 0

In [7]:
```

Parallel computing with Python

Python for High-Performance Computing

- Python is considered an easy to use, but “slow” programming language but there are several external libraries (python packages) and tools we can use to improve performances:
  - **Numpy and Scipy packages**
    - Mathematical and scientific libraries collections with optimized algorithms
  - **Subprocess and Multiprocessing packages**
    - Spawn and control processes and threads
  - **mpi4py, pycuda**
    - Interfaces to external MPI and CUDA API
  - **Numba**
    - Translates Python functions to optimized machine code at runtime

Parallel computing with Python

SURF Jupyter Hub

- Running Jupyter Notebook on the LISA cluster
- Direct access to compute nodes (also with GPUs)
- Dedicated hardware
- Preinstalled packages and optimized libs

PARALLEL PROGRAMMING FOR CPU ARCHITECTURES IN PYTHON

OUTLINE

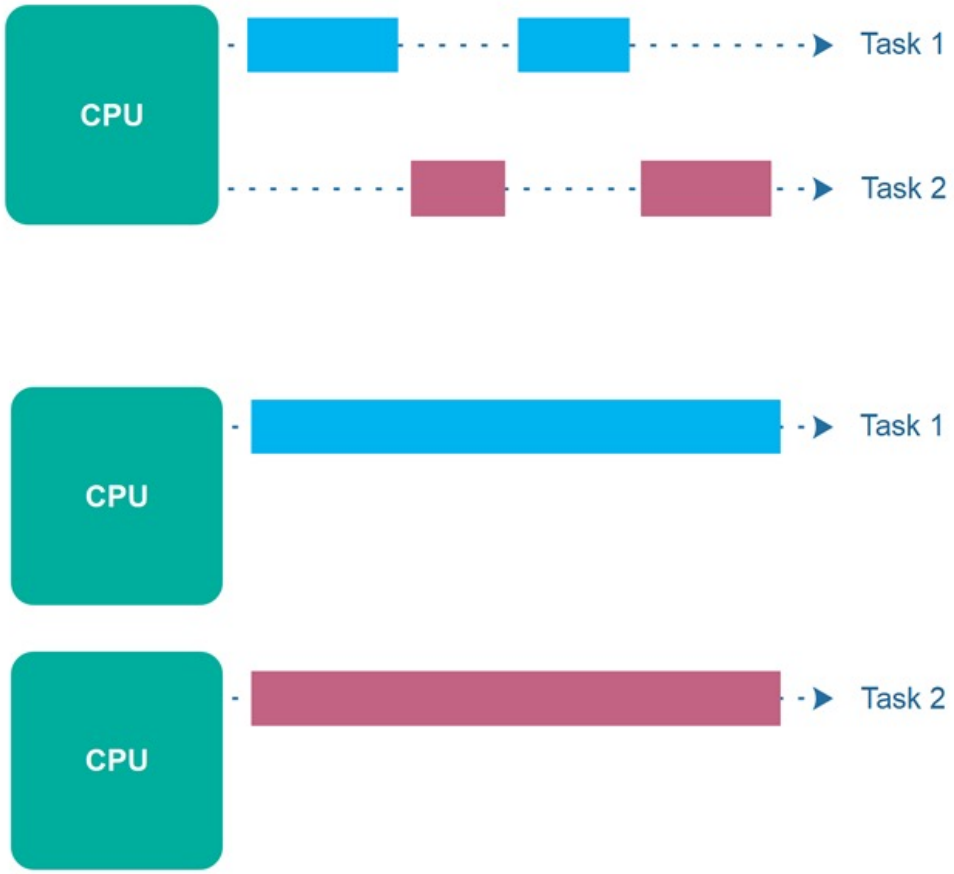
- Processes and threads in Python
- Shared memory parallelism
- Multi-threading and Multi-processing

Parallel programming for CPU architectures in Python

Concurrent vs. parallel execution

Concurrency is the execution of multiple tasks at the same time, regardless of the number of processors. This is the case, for example, when 2 tasks can start, run and complete in overlapping time. On task may not be finished before it begins the next. Concurrency means executing multiple tasks at the same time but not necessarily simultaneously.

Parallelism is the execution on multiple processors of two or more tasks that are running at the same time. Parallel programs use parallel hardware to speed-up computational time by splitting up a task into multiple, simple, and independent sub-task which can be performed simultaneously



Parallel programming for CPU architectures in Python

Concurrent vs. parallel execution

Concurrent, not parallel
This means that it makes progress on more than one task seemingly at the same time (concurrently), but the application switches between making progress on each of the tasks - until the tasks are completed. There is no true parallel execution of tasks going in parallel threads / CPUs.

Parallel, not concurrent
This means that the application only works on one task at a time, and this task is broken down into subtasks which can be processed in parallel. However, each task (+ subtask) is completed before the next task is split up and executed in parallel.

Not parallel, not concurrent
Sequential execution, where the task is too small to be divided into subtasks.

Parallel and Concurrent
This is what happens if an application starts up multiple threads which are then executed on multiple CPUs or the application both works on multiple tasks concurrently, and also breaks each task down into subtasks for parallel execution.

Parallel programming for CPU architectures in Python

Concurrent vs. parallel execution

Advantages

- Concurrent processes can reduce duplication
- The overall runtime of the algorithm can be significantly reduced
- More real-world problems can be solved than sequential algorithm alone.

Disadvantages of concurrency

- Runtime is not always reduced, so careful planning is required.
- Concurrent algorithms can be more complex than sequential algorithms
- Shared data can be corrupted
- Communication between tasks is needed

Parallel programming for CPU architectures in Python

Processes vs. Threads

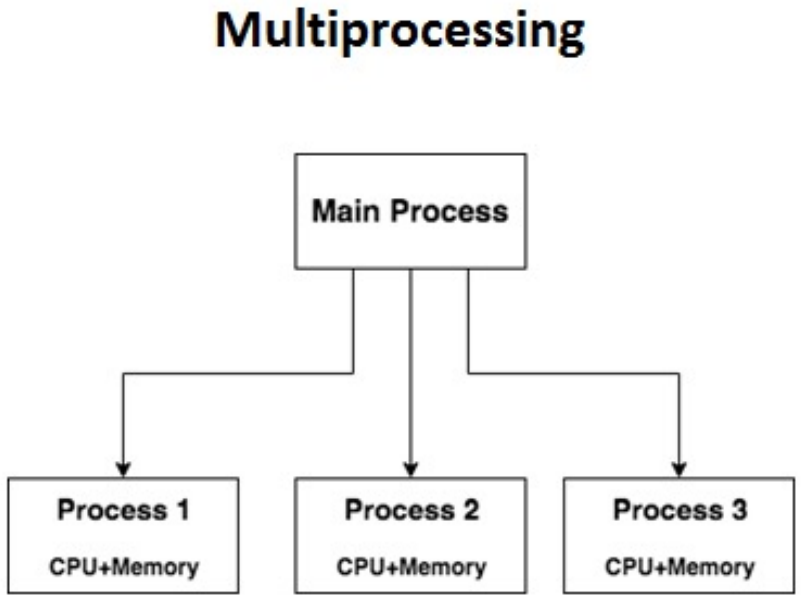
Processes are instances of a program (e.g: python interpreter, browser, etc.). They are independent execution units that have their own memory access space (view of the memory). Each process cannot access directly the shared data in other processes and need explicit communication protocols to exchange data between processes. Also, switching from one process to another requires some time (relatively) for saving and loading registers, memory maps, and other resources.

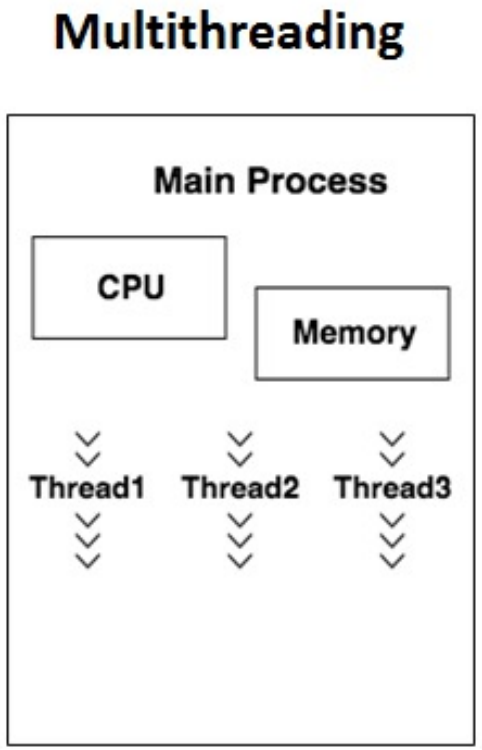
Threads are threads are separate points of execution within a single program, and can be executed either synchronously or asynchronously. A single process can contain multiple threads. Each thread share the same memory space, i.e. the memory space of the parent process. This would mean the code to be executed as well as all the variables declared in the program would be shared by all threads.

SURF

Parallel programming for CPU architectures in Python

Processes vs. Threads





Parallel programming for CPU architectures in Python

Processes vs. Threads

Process	Thread
Processes are heavy-weight operations.	Threads are lighter-weight operations.
Processes can start new processes using e.g. fork() (system call).	A process can start several threads using e.g pthread_create() (system call).
Each process lives in its own memory (address) space and holds a full copy of the program in memory which consume more memory. Processes don't share memory with other processes.	Threads share memory with other threads of the same process. Threads within the same process live within the same address space, and can thus easily access each other's data structures. The shared memory heaps and pools allow for reduced overhead of shared components.
Inter-process communication is slow as processes have different memory addresses.	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to.
Context switching between processes is more expensive.	Context switching between threads of the same process is less expensive.

Parallel programming for CPU architectures in Python

Multi-threading

- Run concurrent tasks within the same process.
- Allow threads to share state and execute from the same memory pool.
- Share single-core CPUs: running multiple threads splitting processing time between different threads. Give the illusion of simultaneous processing through rapid swapping of tasks (interleaving)..
- Concurrent execution is possible on a single-core CPU (multiple threads, managed by scheduler or thread-pool) and archived by context switching. There may be a time-shared algorithm or a priority algorithm for determining which task to run next.
- Data consistency may be complex to ensure given the nature of the shared memory and resources (use of lock and synchronizers).

Parallel programming for CPU architectures in Python

Multi-processing

- Multiple tasks are running at the same time.
- Uses multiple processors to accomplish the task.
- Each processor may also timeshare among different tasks.

- Shared memory multi-processing:
  Two or more cores share same memory access. Threads can run in parallel on different cores and work together on a single copy of a dataset in memory.
- Distributed memory multi-processing:
  Each task runs in a different process and do not share memory and state. Communications between processes (may be also not needed, task farming!) need to be handle explicitly and overhead can be critical.

SURF

Parallel programming for CPU architectures in Python

Where to use what?

I/O bounded applications

Program execution is limited by access to disk or interactions with the external processes (e.g. GUI, web-scrapers). The “wasted” waiting CPU time can be used to run additional threads without loosing performances.

CPU bounded applications

CPU intensive programs that do not have I/O or interactions, will not benefit by having multiple threads as the limiting factor is the ability of the CPU in performing enough operations per second. In this case having the possibility to divide the process in sub-tasks and execute them on different cores will improve drastically the execution

Parallel programming for CPU architectures in Python

Parallel Python in action!

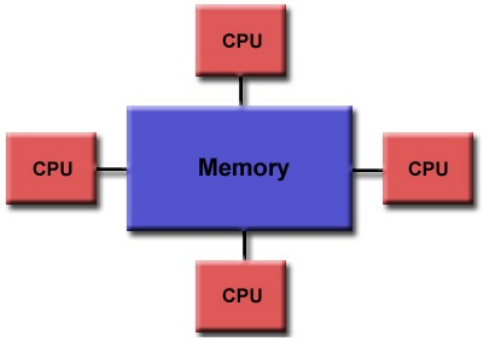
DISTRIBUTED MEMORY PARALLELISM WITH MPI4PY

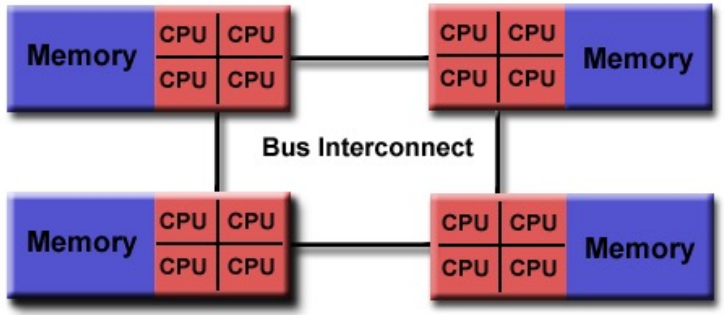
OUTLINE

- Basic concept of MPI
- MPI communications and processes management
- Run an MPI application with Python

Distributed memory parallelism with mpi4py

Parallel computing architectures





- Shared memory
  - All processors access the same memory
  - Different sequences of execution (threads) run on the same process
  - Different memory modules may be used, but only one logical memory space is addressed
  - Communication between processors is done implicitly
- Distributed memory
  - Processes use their own memory
  - Tasks access data from other tasks through communications
  - Multiple tasks can reside on the same physical machine
  - Require libraries. Almost always using MPI

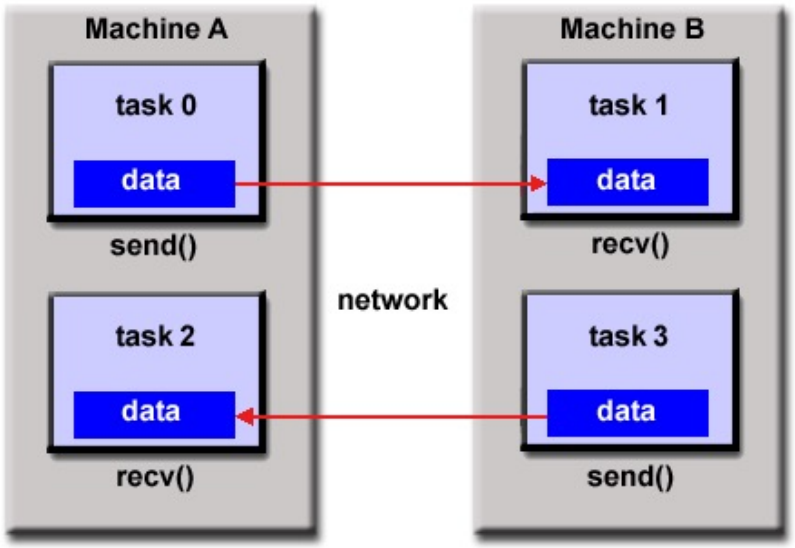


Image source: computing.llnl.gov

Distributed memory parallelism with mpi4py

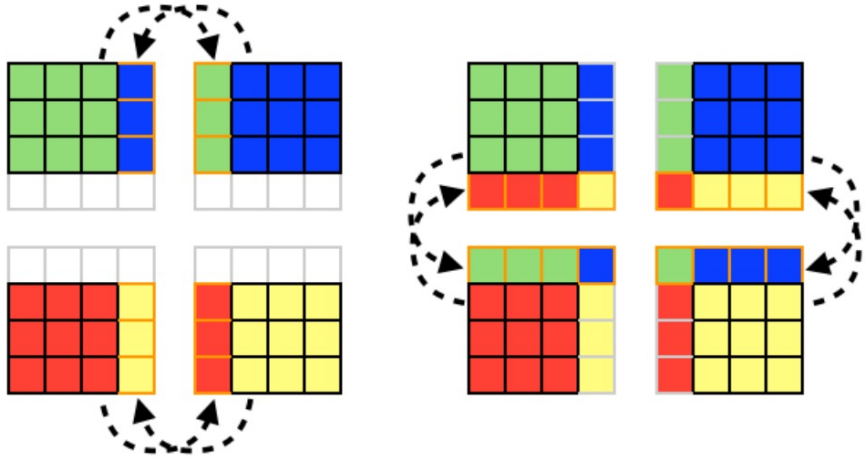
What is MPI?

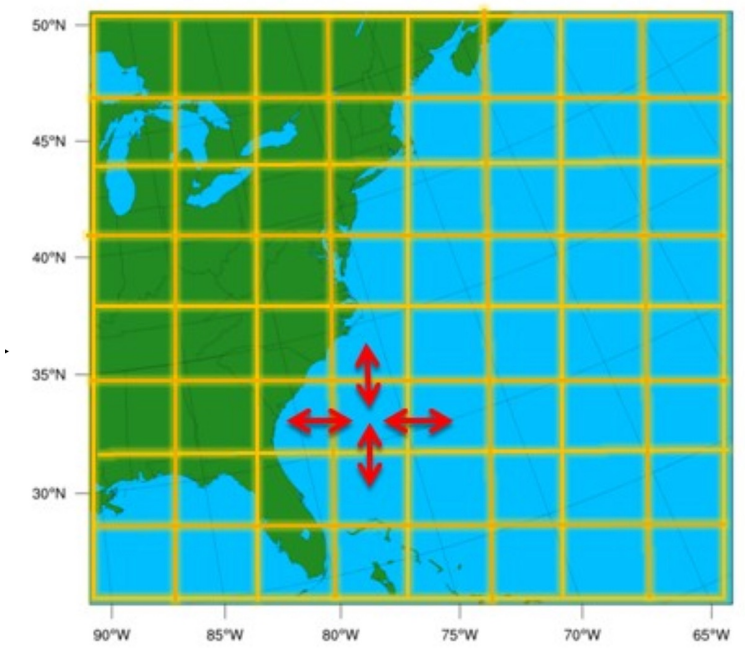
- MPI stands for *Message Passing Interface*
- MPI calls are for sending messages between processes
- Initiative started in 1991; first release 1994
- MPI is a **Standard**
- Implementations include OpenMPI, MPICH, Intel MPI
- Language support in standard for C, FORTRAN
- **…** but we will use Python

Distributed memory parallelism with mpi4py

What is MPI?

- Work distribution for *Monte Carlo* methods
- Solutions of PDE problems in a spatial domain, using *Halo regions* or *Ghost cells*
- Now: Machine Learning





Distributed memory parallelism with mpi4py

MPI - principles

- Communicators
  - rank, size
- Collective communication
- Point-to-point communication
  - blocking or synchronous
  - non-blocking or asynchronous

Distributed memory parallelism with mpi4py

MPI - principles

- All communication is done through communicators
- There is always a global communicator MPI_COMM_WORLD
- A communicator always has a size
- Each process has a unique rank in a communicator
- New communicators can be derived, with, for example:
  - only a subset of processes
  - a cartesian topology
  - with optional cyclic connections

Distributed memory parallelism with mpi4py

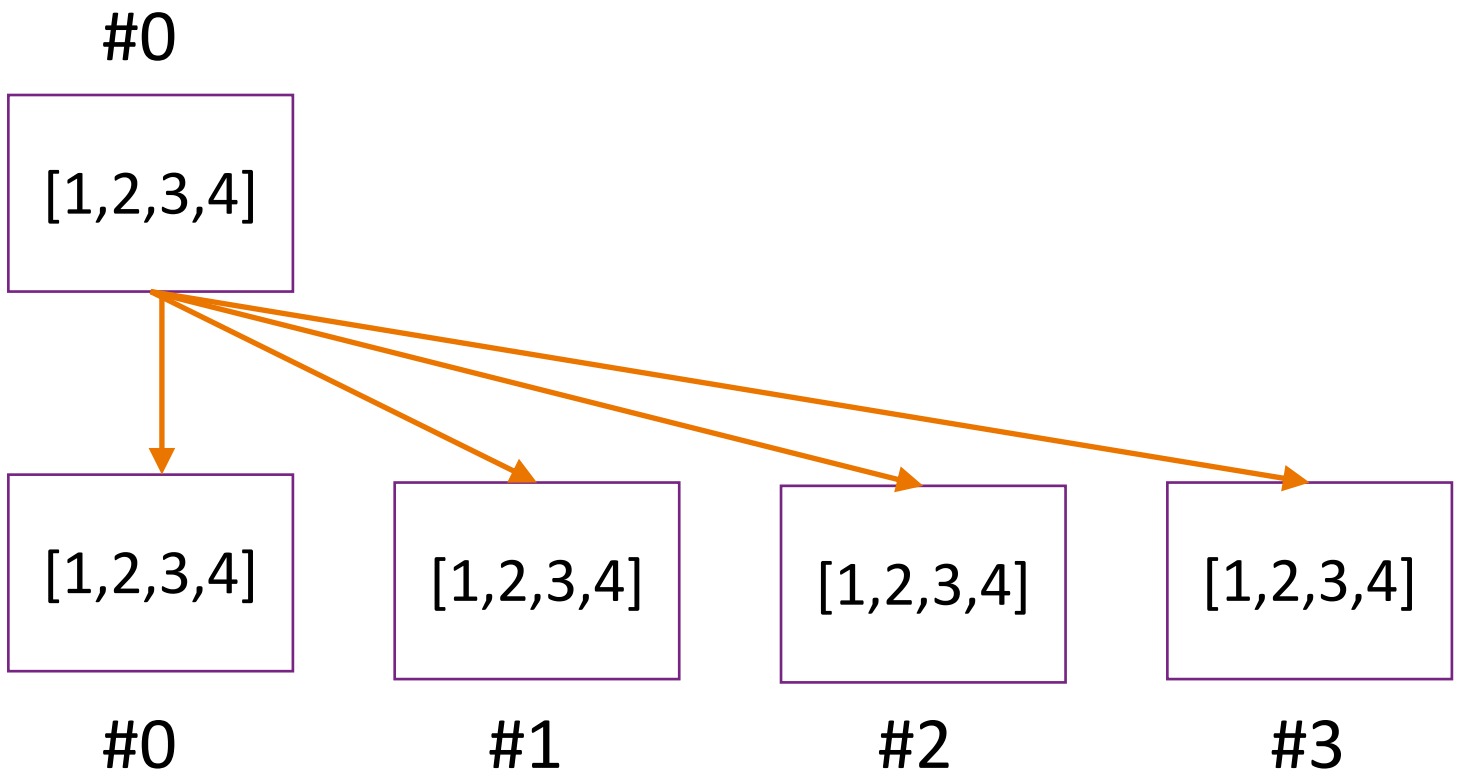
Communicator example

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

print("rank:", comm.rank, "size:",
 comm.size)
```


SURF

Distributed memory parallelism with mpi4py

Collective communication: Broadcast



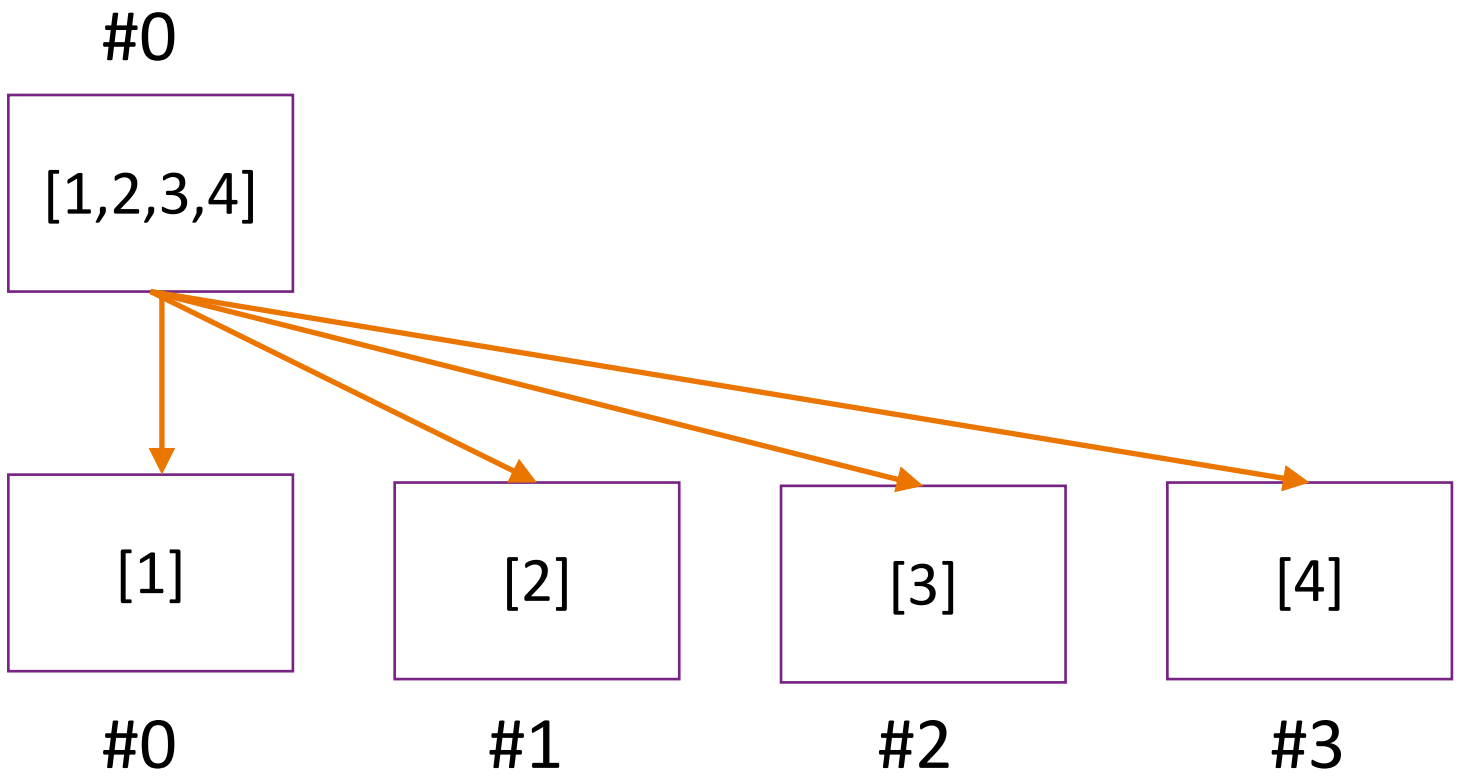
Distributed memory parallelism with mpi4py

Broadcast example

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

if comm.rank == 0:
    data = [1,2,3,4]
else:
    data = None

data = comm.bcast(data)
print("rank:", comm.rank, "data:", data)
```

Distributed memory parallelism with mpi4py

Collective communication: Scatter



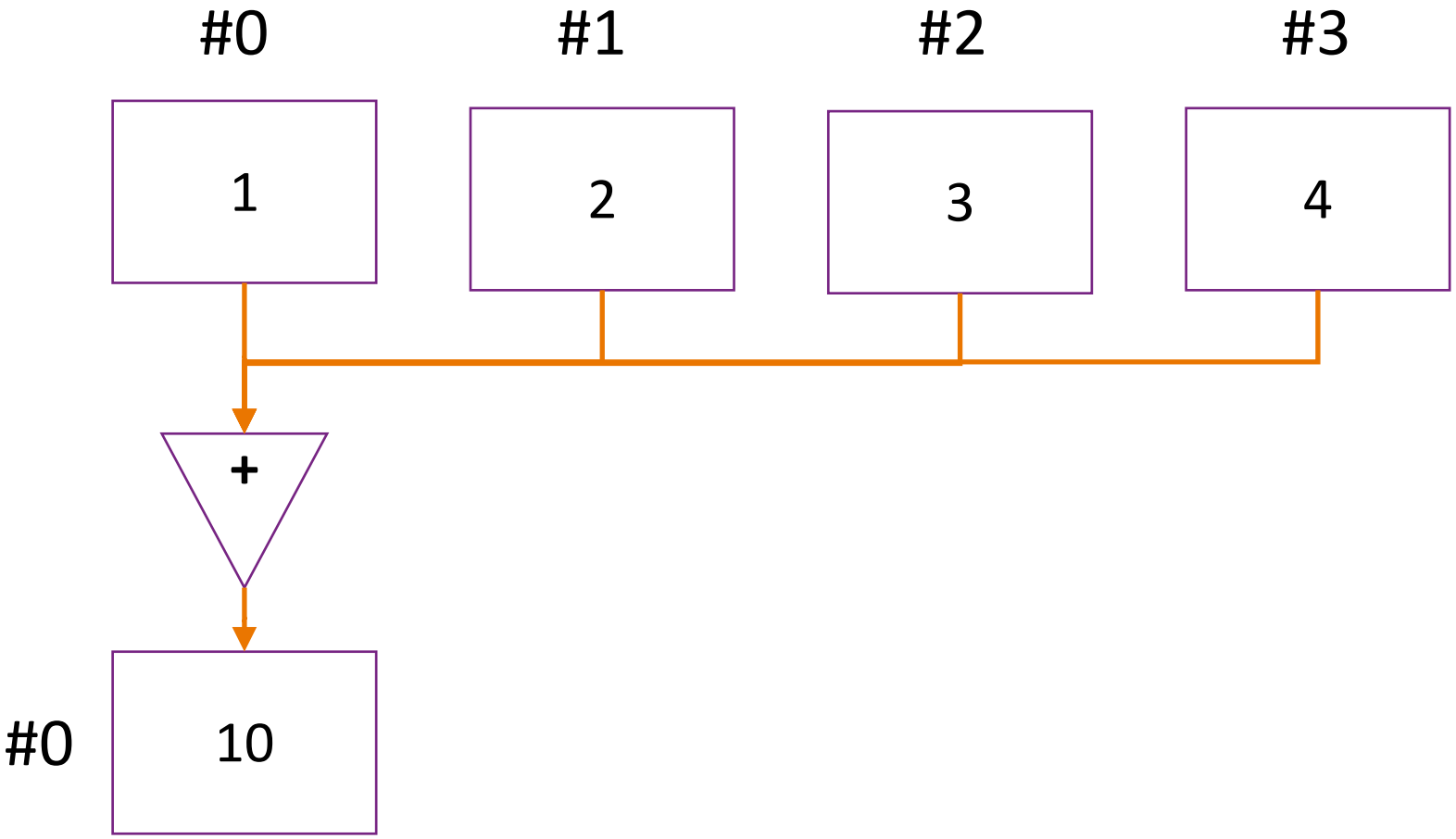
Distributed memory parallelism with mpi4py

Scatter example

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

if comm.rank == 0:
    data = [1,2,3,4]
else:
    data = None

data = comm.scatter(data)
print("rank:", comm.rank, "data:", data)
```

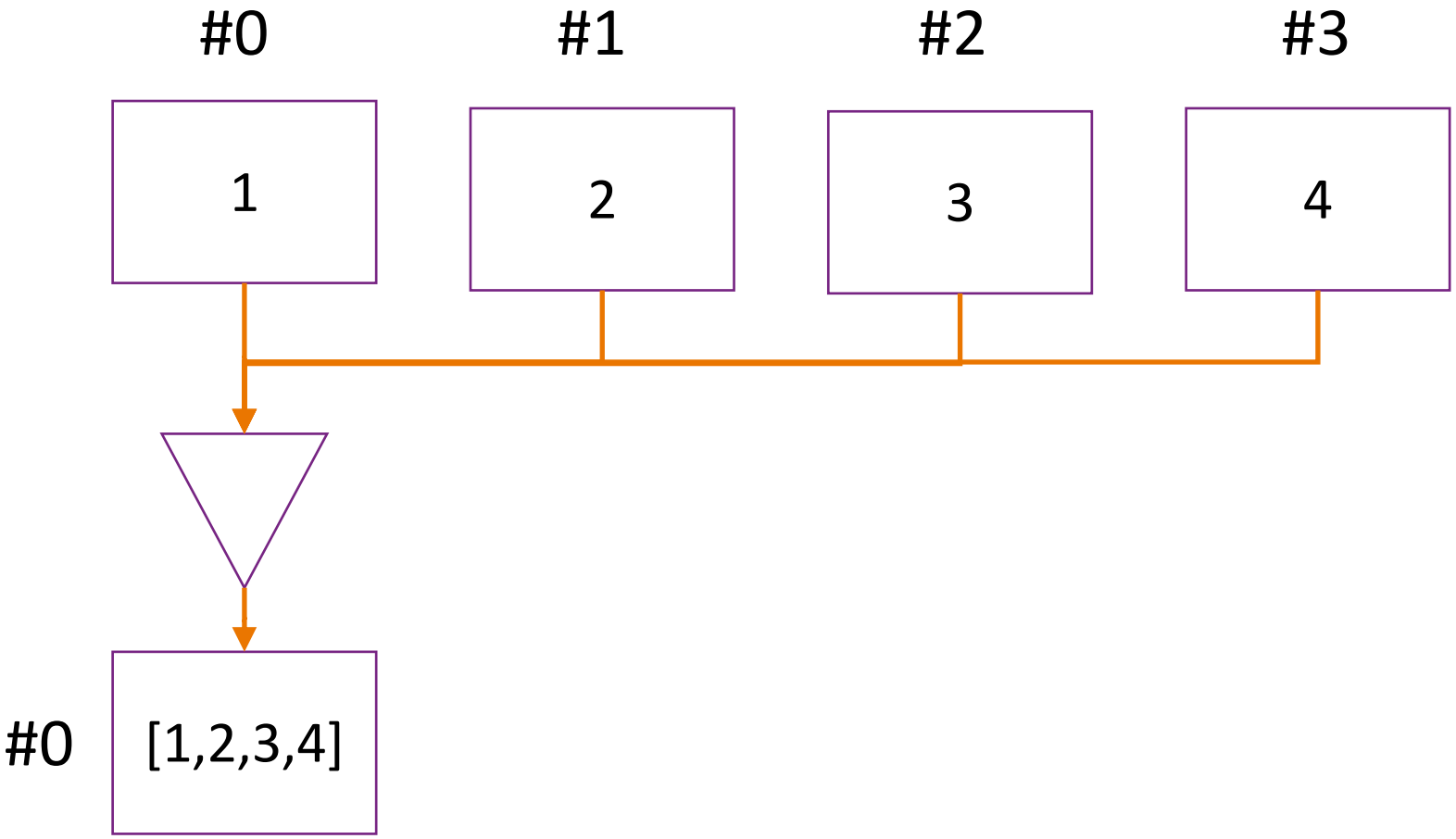
Distributed memory parallelism with mpi4py

Collective communication: reduce



Distributed memory parallelism with mpi4py

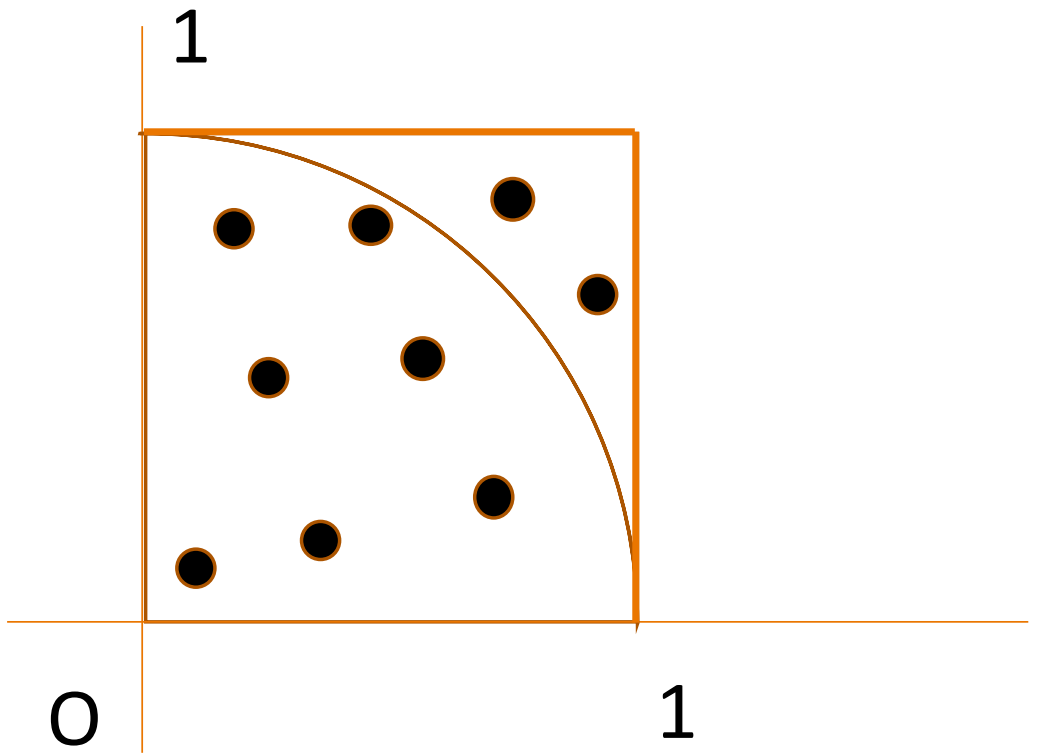
Collective communication: gather



Distributed memory parallelism with mpi4py

Real case problem: PI with Monte Carlo methods

- pick *n* random points
- **split the work**
- count the number of points *inside* the unit circle
- **take the sum of all workers**
- divide by *n*
- for infinite number of points, approaches ¼ π



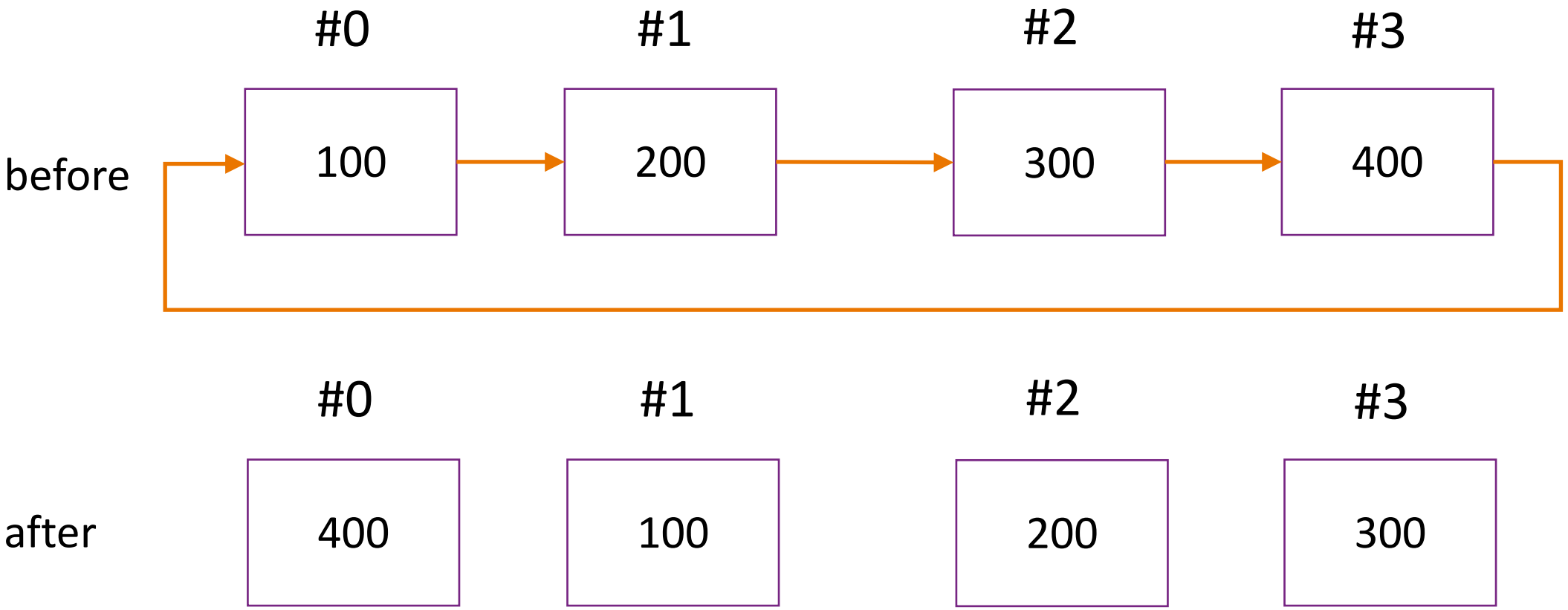
Distributed memory parallelism with mpi4py

Computing Pi with MC method and MPI

```
count = 0
for i in range(int(10000 / comm.size)):
    (x,y) = (random(), random())
    if x * x + y * y < 1.0:
        count += 1

sum_count = comm.reduce(count, MPI.SUM)

if comm.rank == 0:
    print("pi = %9.3f" %
       (4.0 * sum_count / 10000) )
```

Distributed memory parallelism with mpi4py

Point-to-point communication



Distributed memory parallelism with mpi4py

Point-to-point communication

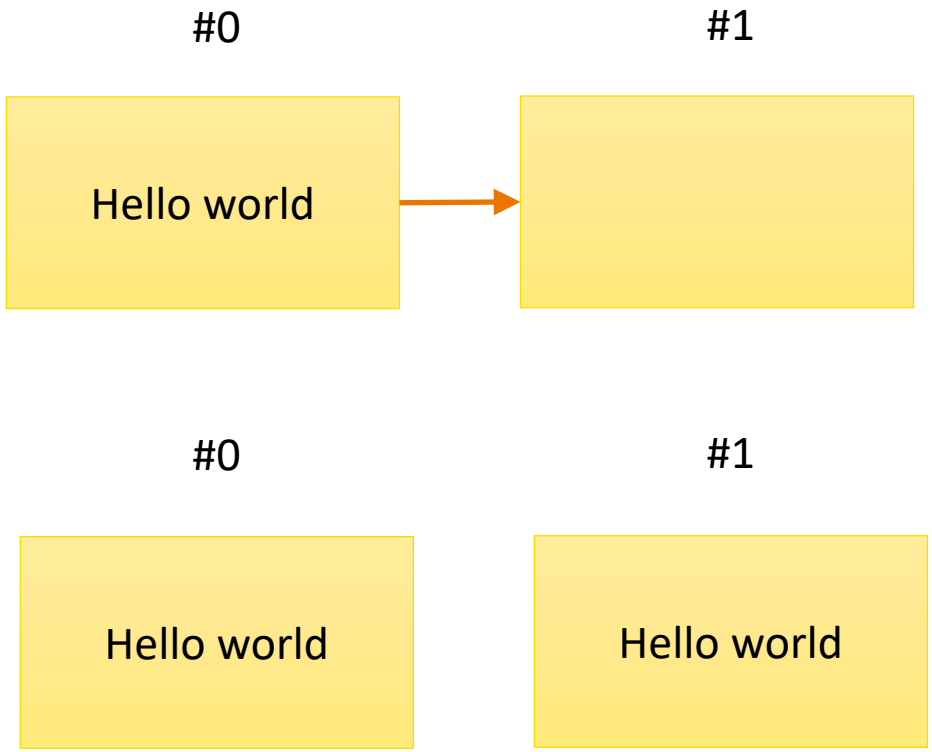
- Sending requires an explicit destination
- Receiving may specify a source
- Both can use *tags* for verifying content consistency
- send() and recv() are **blocking**

```
comm.send(obj, int dest, int tag=0)

comm.recv(int source=ANY_SOURCE, int tag=ANY_TAG)
```

Distributed memory parallelism with mpi4py

Point-to-point communication

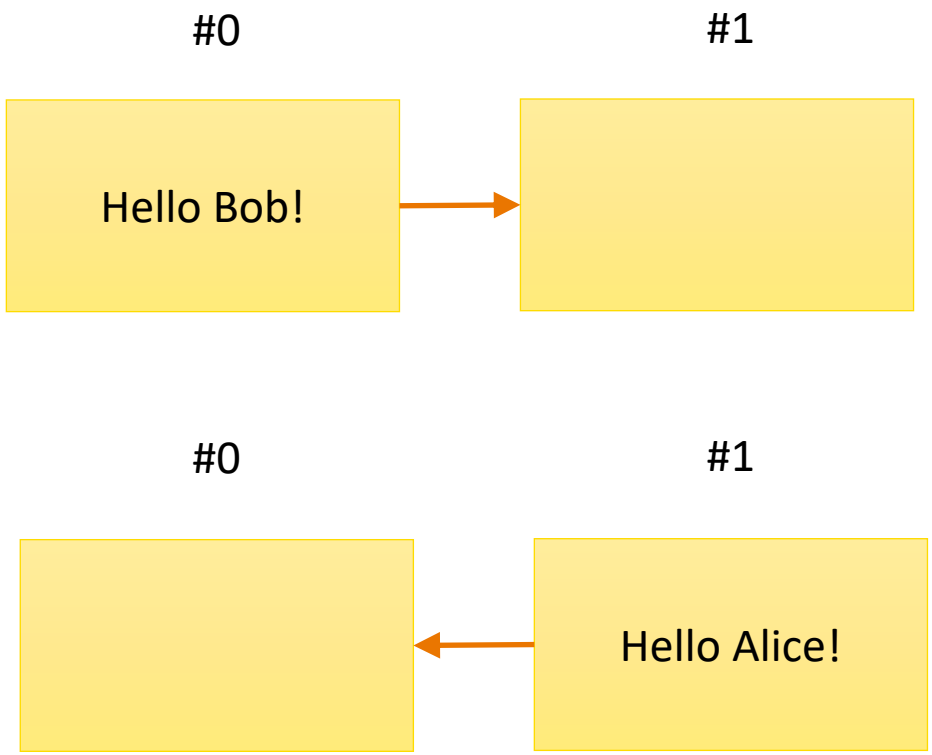
```
from mpi4py import MPI
comm = MPI.COMM_WORLD

if comm.rank == 0:
    comm.send("Hello world", 1)

if comm.rank == 1:
    message = comm.recv()
    print("Rank 1 received '%s'" %
          message)
```



Distributed memory parallelism with mpi4py

Point-to-point communication

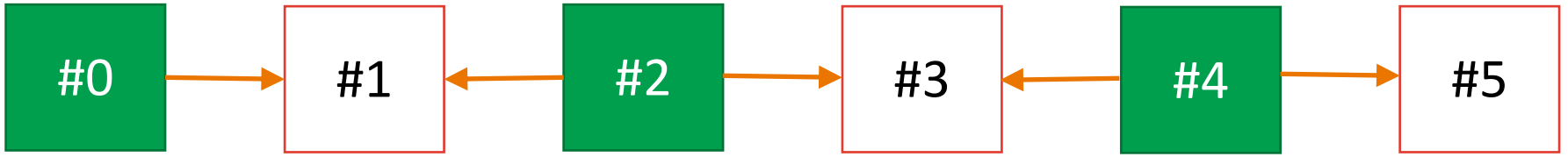
```
# Alice; say Hello to Bob
if comm.rank == 0:
    comm.send("Hello Bob!”, 1)
    mesg = comm.recv()
    print("Alice: Bob said ‘%s'" % mesg)

# Bob; say Hello to Alice
if comm.rank == 1:
    comm.send("Hello Alice!”, 0)
    mesg = comm.recv()
    print("Bob: Alice said ‘%s'" % mesg)
```



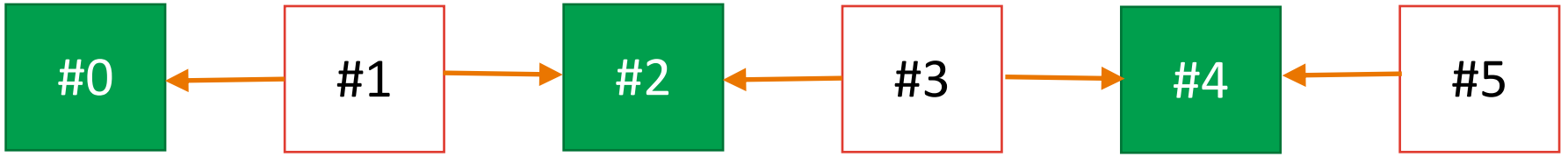
Distributed memory parallelism with mpi4py

Avoiding deadlocks alternating communications

Step 1:



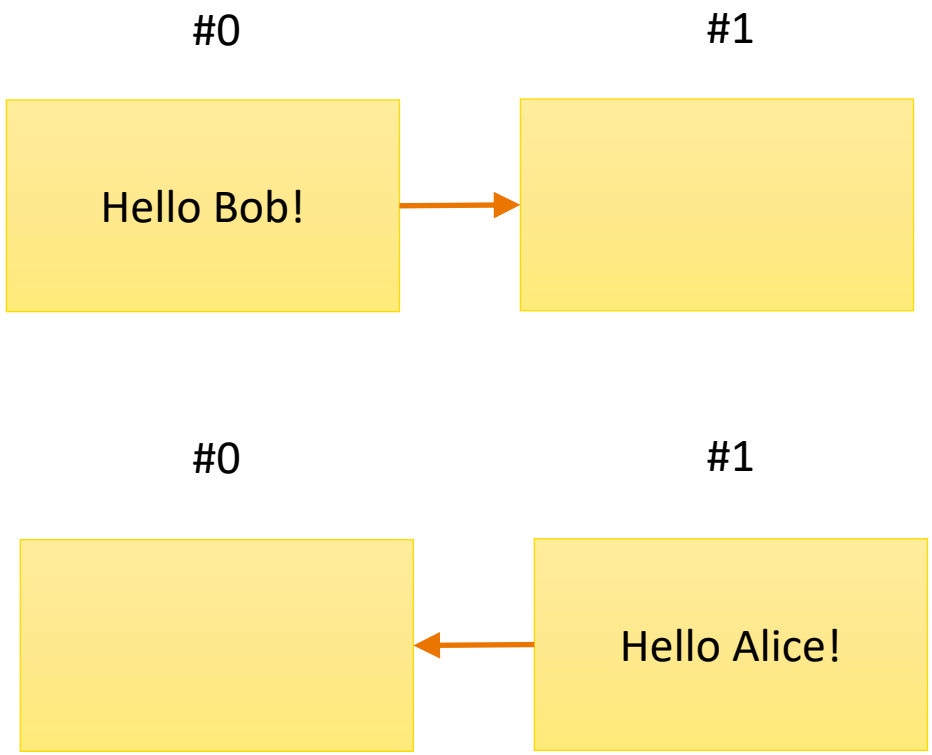
Step 2:



Distributed memory parallelism with mpi4py

Point-to-point communication

```
# Alice; say Hello to Bob
if comm.rank == 0:
    comm.send("Hello Bob!”, 1)
    mesg = comm.recv()
    print("Alice: Bob said ‘%s'" % mesg)

# Bob; say Hello to Alice
if comm.rank == 1:
    mesg = comm.recv()
    comm.send("Hello Alice!”, 0)
    print("Bob: Alice said ‘%s'" % mesg)
```



Distributed memory parallelism with mpi4py

Non-blocking communication

- All communication thus far has been *blocking*, or *synchronous*
- Many routines have non-blocking, or *asynchronous* versions

Principle:

- instead of “send”, do “isend”, and capture the status
- do something else
- “wait” for the send to finish, by calling “status.wait()”

```
comm.isend(obj, int dest, int tag=0)
```

Parallel programming for CPU architectures in Python

Parallel Python in action!