Karlsruhe Institute of Technology

High Performance Computing with Python

Ivan Kondov

STEINBUCH CENTRE FOR COMPUTING - SCC

Overview

- General aspects
  - Python distributions
  - Virtual environments
- SciPy – the scientific package collection
- Concurrent and parallel programming/computing with Python
  - Python generators
  - Multiprocessing
  - Message passing
  - Workflows
- Heterogeneous programming/computing with Python
  - Linking to Fortran, C and C++
  - Just-in-time compiling (JIT)

Why using Python?

- Increase scientist‘s productivity
- Accelerate prototyping in complex projects
- Reuse existing codes written in any language

General strategies

- Detect performance critical sections using timing and profiling
- Performance irrelevant parts – program rapidly in Python
- Performance critical sections
  - Reuse available high performance libraries
  - Add your high performance codes as extension modules
- You are starting a new project – start it with Python

Disclaimer

- This is only a short introduction to HPC with Python
- No coverage of “basic” HPC and basic Python
- Many relevant aspects not covered – for example performance analysis

Python distributions and venv

- CPython (the standard Python distribution)
  - bwUniCluster, bwForClusters
  - Versions available on bwUniCluster: 2.7, 3.3, 3.4. 3.5

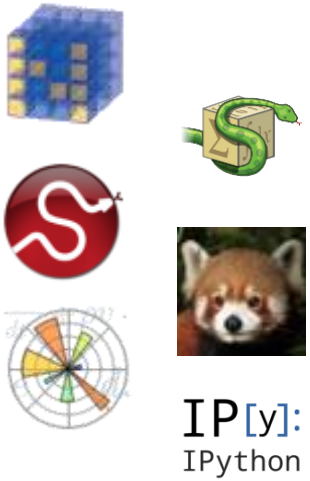
- Anaconda Python – focusing on scientific and engineering applications
- Intel Python
  - Based on Anaconda Python
  - Leverages Intel MKL, Intel TBB and Intel DAAL for high performance
  - Versions available on ForHLR-1 and ForHLR-2: 2.7, 3.5, 3.6

- Recommended Python versions: 3.x
- The most recent version: 3.7
- End of life of Python 2 in 2020

- Virtual environment (venv)
  - Isolated custom installation of python packages
  - Switching with commands “activate” and “deactivate”
  - For working with multiple projects with conflicting requirements

The SciPy collection

- NumPy – numerical computation with arrays
- SymPy – symbolic math computation
- SciPy library – a library for scientific computing
- Pandas – data analysis
- Matplotlib – data visualization
- IPython – interactive Python



Package	Module on bwUniCluster	Module on ForHLR-1 and ForHLR-2
Python	devel/python	devel/python
NumPy	numlib/python_numpy	devel/python
SciPy	numlib/python_scipy	devel/python
pandas	numlib/python_pandas	devel/python
Matplotlib	lib/matplotlib	devel/python
IPython	devel/ipython	devel/python

Exercise 1: Create a venv and install `scipy`

```
# prepare the global environment
module purge
module load devel/python/3.5.2

# create a virtual environment
python -m venv venv-python3

# activate the venv
. venv-python3/bin/activate
# deactivate the venv
deactivate
# activate again
. venv-python3/bin/activate

# pip should be kept up-to-date
pip install --upgrade pip
```

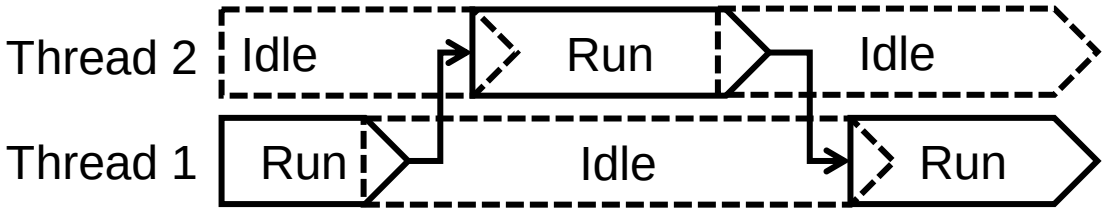
```
# skip this section on FH1/FH2
# https://pypi.org/project/intel-numpy/
# https://pypi.org/project/intel-scipy/
pip install intel-numpy
pip install intel-scipy
pip install sympy
pip install pandas
pip install matplotlib


# unit testing (optional)
pip install nose
pip install pytest

# static code analysis (optional)
pip install pyflakes
pip install pylint
```

Exercise 2: Install and test mpi4py

```
# deactivate/activate the virtual environment
module purge
module load mpi/impi
module load devel/python/3.5.2
. venv-python3/bin/activate

# install mpi4py
pip install mpi4py

# test the mpi4py installation
mpiexec -n 3 python -m mpi4py.bench helloworld
mpiexec -n 5 python -m mpi4py.bench ringtest -n 1024 -l 1000

# optional: run the unit tests (thousands of tests taking long time!)
wget https://bitbucket.org/mpi4py/mpi4py/downloads/mpi4py-3.0.1.tar.gz
tar xzf mpi4py-3.0.1.tar.gz
cd mpi4py-3.0.1
mpiexec -n 4 nosetests -w test
```

Python threads

- Concurrency versus parallelism
- Global Interpreter Lock (GIL)
- The `multithreading` package
- Preemptive multitasking



- Not relevant for HPC parallelization
- Relevant for processing I/O asynchronously (`asyncio` package)

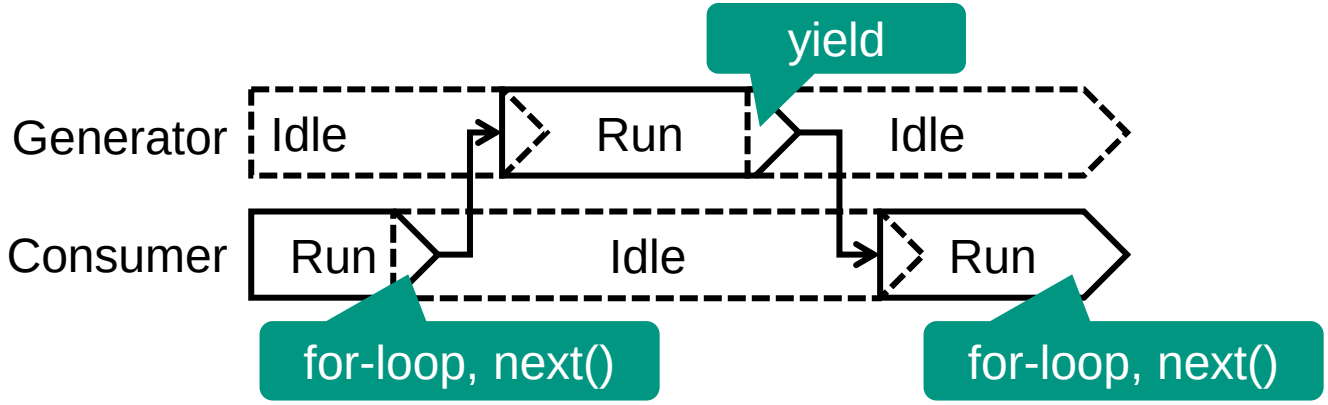
Python generator

- Implemented concepts
  - Lazy evaluation / evaluation on demand
  - Cooperative multitasking
  - Iterator
- In data intensive computing
  - Save memory for IO operations
  - Save memory for intermediately stored objects
- Defined in different ways
  - Generator function: **yield** keyword
  - Generator expression using “()”
- Chains of generators - pipelines

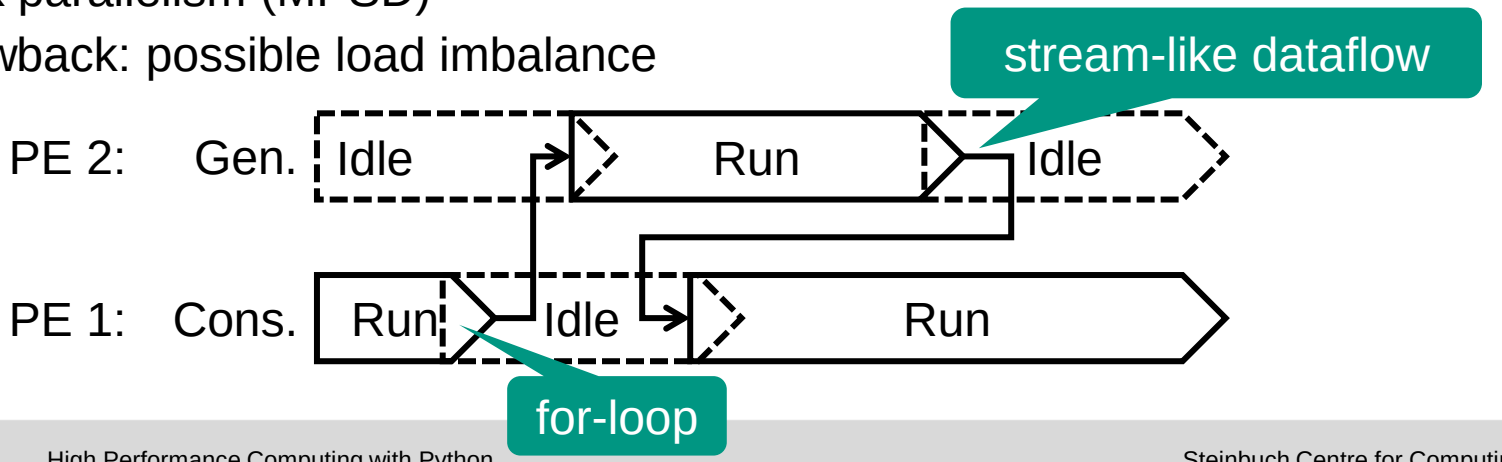
```
def data_producer():
    # do work
    return lots_of_data

def data_consumer(data):
    for element in data:
        # do work

bulky = data_producer()
result = data_consumer(bulky)


def smart_data_producer():
    while is_working:
        yield piece_of_data

gen = smart_data_producer()
result = data_consumer(gen)
```

How does a generator work?

- The generator and consumer share the same processing element (PE)



- The generator and consumer can be distributed on PEs
  - Task parallelism (MPSD)
  - Drawback: possible load imbalance



Exercise 3: Generators

- Calculate the first eight Fibonacci numbers
- Calculate the primes in a given range of numbers
- Find the first ten primes from the Fibonacci sequence

```
cd generator
# traditional method
python fib1.py

# infinite generator function
python fib2.py

# infinite iterator class
python fib3.py

# primes
python primes.py

# Fibonacci primes
# python fibprimes.py
```

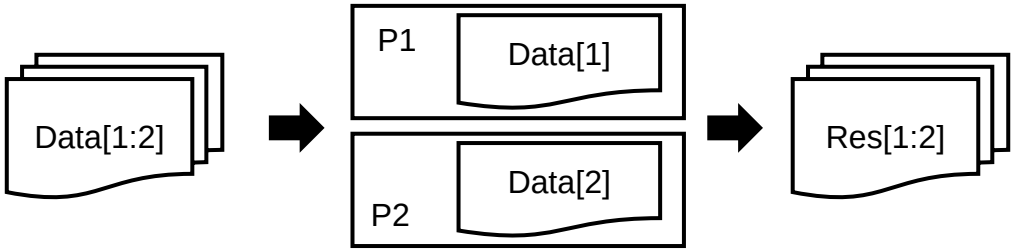
Data parallelism

- SPMD (Single Process Multiple Data) and MPMD
- Fine-grained vs. coarse grained parallelism

Package[.module[.class]]	Methods	Parallel	Many nodes
built-in/functools	map, filter, reduce	No	No
itertools	starmap	No	No
multiprocessing.Pool	map, map_async, imap, imap_unordered, starmap, starmap_async, apply, apply_async	Yes	No
concurrent.futures. ProcessPoolExecutor	map, submit, shutdown	Yes	No
mpi4py.futures. MPICommExecutor	map, starmap, submit, bootup, shutdown	Yes	Yes
mpi4py.futures. MPIPoolExecutor	map, starmap, submit, bootup, shutdown	Yes	Yes

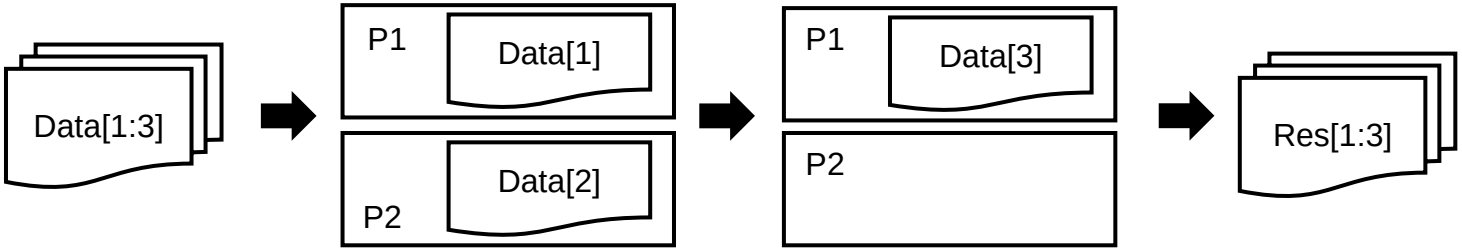
The map method

Len(Data) == N(P)

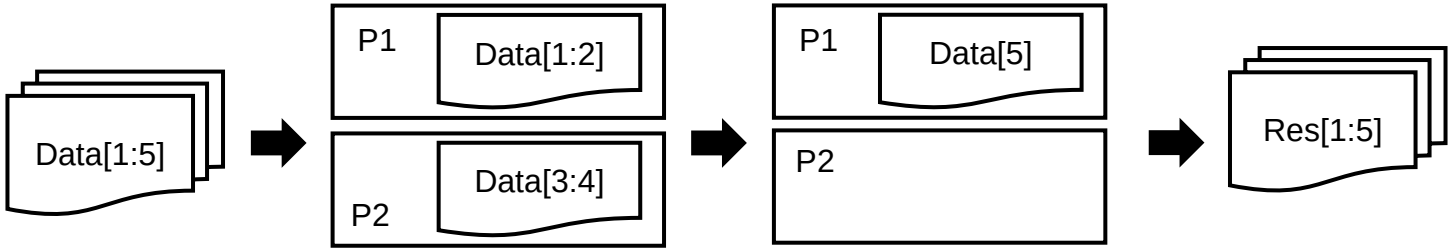


default chunk size == 1

Len(Data) > N(P)



Len(Data) > N(P) and chunk size == 2



Exercise 4: Parallelizing the map method

- Find the primes from the first 5000 Fibonacci numbers
- Measure the run time with different parallelization packages, number of processes and chunk sizes

```
cd spmd

# serial
python map.py

# multiprocessing package
python multiprocessing_pool.py

# concurrent.futures package
python process_pool_executor.py

# mpi4py package, static launch, 1 master, 4 workers
mpiexec -n 5 python -m mpi4py mpi_comm_executor.py

# mpi4py package, dynamic launch (spawn), 1 master, up to 4 workers
mpiexec -n 1 -usize 5 python -m mpi4py mpi_pool_executor.py
```

The mpi4py package

- Message passing: a paradigm for parallel computing
- Message Passing Interface (MPI) – the standard
- Allows distributed computing on many cluster nodes
- The mpi4py package – a python interface to MPI

- Point-to-point communication
  - Blocking send/recv
  - Non-blocking isend/irecv
- Collective communication: broadcast, scatter, gather, reduce, …

- The item count and MPI datatype discovered automatically for
  - generic Python objects (lower-case methods)
  - `numpy` arrays with “standard datatypes” (upper-case methods)

Note on mpi4py runtime environment

- Usually the application is launched with

```
mpiexec -n <nprocs> python pyfile.py
```

- An unhandled exception (e.g. division by zero) may cause a deadlock
- Instead, there is a mechanism catching and handling such exceptions

```
mpiexec -n numprocs python -m mpi4py pyfile [arg] ...
mpiexec -n numprocs python -m mpi4py -m mod [arg] ...
mpiexec -n numprocs python -m mpi4py -c cmd [arg] ...
mpiexec -n numprocs python -m mpi4py - [arg] ...
```

- Example

Exercise 5: Launching mpi4py programs

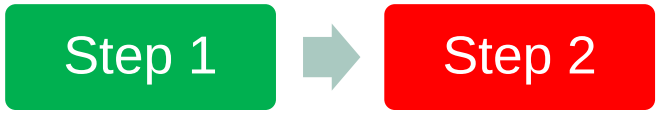
```
cd mpi4py
# the “standard” way
mpiexec -n 2 python deadlock.py
```

Why is there no deadlock if the exception occurs in rank 1?

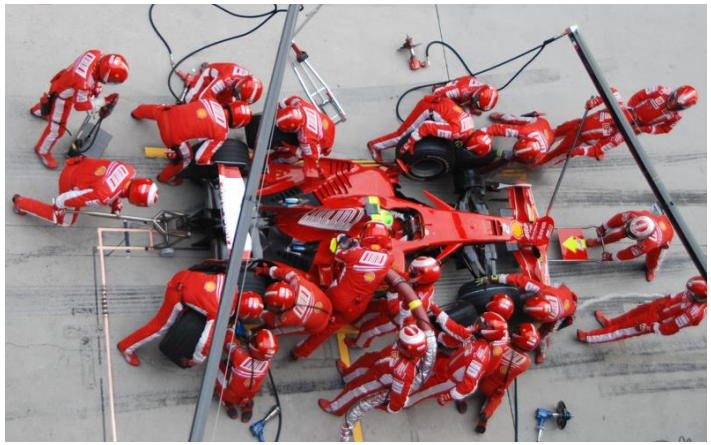
```
# now the better way to launch
mpiexec -n 2 python -m mpi4py deadlock.py
```

Workflows versus pipelines

Workflow



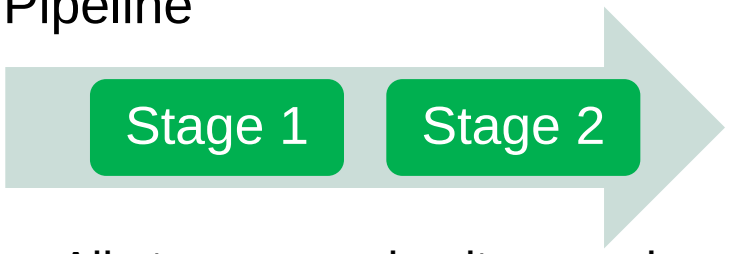
- Step 2 begins after Step 1 is ready
- Data is passed discretely
- Directed acyclic graph: no loops



Source: Bert van Dijk, https://www.flickr.com/photos/zilpho/2964165616

FireWorks

Pipeline



- All stages run simultaneously
- Data is passed continuously
- Directed acyclic graph: no loops

Source: Ford Europe, https://www.flickr.com/photos/fordeu/5709826282

scikit-learn, python-pipes

Heterogeneous computing with Python

- Frameworks that compile optimized kernels
  - Theano – focus on multidimensional arrays and expressions
  - TensorFlow – focus on multidimensional arrays and dataflow

- Just-in-time (JIT) compilers
  - PyCUDA (NVidia GPU) https://documen.tician.de/pycuda
  - PyOpenCL (CPU/GPU) https://documen.tician.de/pyopencl
  - Numba (CPU/GPU) http://numba.pydata.org
  - Pythran (CPU) https://pythran.readthedocs.io

- Linking to C, C++ and Fortran

Linking to C, C++ and Fortran

- External codes can be linked as extension modules
- Main approach: write or generate a wrapper

Name	Language	Approach	Web site
Python/C API	C	API (Python.h)	docs.python.org
ctypes	C, C++	Link to DLLs	docs.python.org
Boost.Python	C++	API	www.boost.org
SWIG	C, C++	compiler	www.swig.org
F2PY	C, Fortran	compiler wrapper	docs.scipy.org
Pyrex/Cython	C	language / compiler	cython.org

Exercise 6: Linking to C++ and Fortran

- Link to C++ using `ctypes`
  - given the class definition in `fib.cpp`
  - write an extension interface `fib_ext.cpp`
  - write an extension module `fib.py`

```
cd ctypes
make fibc++.so
python fibtest.py
```

- Link to Fortran using `f2py`
  - automatic building an extension module using module `numpy.f2py`
  - see makefile for detailed commands

```
cd f2py
# compile and run the Fortran 77 program
make fibprog && ./fibprog
```

- Case 1: Link to `libfib1` with no explicit interface

```
make libfib1
python fib1.py
```

Creating explicit interfaces to Fortran

- Case 2: Adapting the interface using a pyf-file

```
make fib1pyf
# open fib1.pyf and change
    integer dimension(n) :: a
    integer, optional,check(len(a)>=n),depend(a) :: n=len(a)
# with
    integer dimension(n), intent(out), depend(n) :: a
    integer, intent(in) :: n
# and save as fib2.pyf
make libfib2
python fib2.py
```

- Case 3: Instrumenting the Fortran source

```
# add to the end of the header of the fib subroutine
cf2py intent(out) a
cf2py depend(n) a
cf2py intent(in) n
make libfib3
python fib3.py
```

Using explicit interfaces to Fortran

- Case 4: Linking to Fortran 90
  - No changes necessary – just use the Fortran 90 modules and interfaces
  - Result the same as in Cases 2 and 3
  - Example in file `fib4.f90`

```
make libfib4
python fib4.py
```

Note on the Intel Python distribution (FH1/FH2)

- If bootstrapping pip does not work:

```
module load devel/python/3.5
python -m venv --without-pip venv-python3
. venv-python3/bin/activate
pip install -t venv-python3/lib/python3.5/site-packages pip
# pip upgrade not necessary because most recent installed
python -m pip <pip command and options>
```