28th Summer School on Parallel Computing
May 23th, 2019 - Bologna

Introduction to Python for HPC

Fabio Pitari - f.pitari@cineca.it
Massimiliano Guarrasi - m.guarrasi@cineca.it
Nicola Spallanzani - n.spallanzani@cineca.it
Mirko Cestari - m.cestari@cineca.it

High Performance Computing Department at CINECA

Why Python?

Python is a clear and powerful object-oriented programming language:

- **Easy to learn**
  - It is very simple to learn the basis of the language
- **Easy to read**
  - Elegant syntax similar to pseudo-code
- **Easy to use**
  - It is simple to get your program working
  - Ideal for prototype development
- **Large standard library**
  - And easy to extend by adding new modules
- **Open source software**

Python is slow

Python is slow compared with other compiled languages already used in computational science.

So, why Python is becoming so popular in computational science?

- **It is definitely easy to learn and use**
  - Usually scientists are not expert programmers
- **You can start to practice using it like a scripting language**
  - Writing scripts for pre or post-process your data
- **You can accelerate Python**
  - Using scientific modules (e.g. numpy)
  - Using Python combined with other languages

Python for HPC

How you can use Python for HPC?

- Accelerating it
  - Numpy
  - F2py
  - Cython
  - Numba

- Creating ad hoc work-flows
  - High-throughput computing (HTC)
  - Fault tolerance

Python language syntax

Objects and types

Python is strongly typed and dynamically typed

```
>>> a = 4000
>>> b = a
>>> a = 4000.5
>>> type(a) # Everything has a type
```

Operator "=" means a reference to a space in memory that contains an object

```
>>> id(a)
```

Objects are *mutable* (once created can be changed or updated) or *immutable*

strings

Strings can be created using quotes (single, double or triple)

```
>>> a = 'home'
>>> b = “new home”
```

Triple quotes are used for string that span over more than a single line

```
>>> '''This is the first line
... this is the second line'''
```

Escape characters are similar to C (\n \t)

strings

Multiple actions on strings

```
>>> a = 'my new home'
>>> a.upper()
>>> a.split()
```

Single elements of strings can be accessed

```
>>> a[2]
>>> a[0:2] # python index starts from 0
>>> a[-4:] # no values means beginning or end
```

Concatenation of strings

```
>>> a + " is beautiful"
>>> a * 3
```

Containers

List (mutable)

```
>>> a = [1, 1, 2, 'home']
```

Tuple (immutable)

```
>>> a = (1, 4, 'seven', 6)
```

Dict (mutable)

```
>>> a = {'a': 2, 'b':4, 4:5}
```

Set (mutable)

```
>>> a = set([1, 1, 3, 5])
```

lists

Can be not homogeneous

```
>>> a = [1, 1, 2, 'home']
```

Index ranges from 0 to len(list)-1

Slicing

```
>>> a[0:3] # from first to third element [i:j:k]  k = stride
>>> a[-1:]+a[:-1] # ['home', 1, 1, 2]
```

Mutable (in-place)

```
>>> a[0] = 4  # [4, 1, 2, 'home']
```

lists

append

```
>>> a = [1, 1, 2, 'home']
>>> a.append(3) # [1, 1, 2, 'home', 3]
```

pop

```
>>> a.pop()    # remove rightmost element
3
```

Function “range” can be used to create list of integers

```
>>> a = list(range(3)) # [0, 1, 2]
>>> b = list(range(2, 10, 3)) # [2, 5, 8]
              # first, last (excluded), step
```

dictionaries

Map keys to values (mappings)

```
>>> a = {'b':2, 'c': 3}  # 'b', 'c' are keys
                         # 2, 3 are values
>>> a['b']    # returns 2
```

There is no left to right order, only mapping

```
>>> a[-1]    # does not work
```

```
a.keys(), a.values(), a.items()
```

Control-flow statements

Indentation matters

```
>>> if a > 3:   # mind the colon
...     print a
...     print('still in the if statement')
... elif a == 3:
...     print('a is equal to 3')
... else:
...     print('a is less than 3')
...
>>>
```

for loop

Any sequence object is iterable

```
>>> for i in range(5):
...     print(i)    #  prints 0, 1, 2, 3, 4
```

More common in python

```
>>> a = [1, 1, 4, 'home']
>>> for i in a:
...     print(i)    #  prints 1, 1, 4, 'home'
```

```
break      # exit from inner loop
continue   # go to next iteration
```

while loop

If you don’t know the number of the step of the loop

```
>>> while error > tolerance:
...     result, error = compute(result)
...
>>>
```

Sometime you want to perform an infinite loop end exit only after a check

```
>>> while True:
...     result, error = compute(result)
...     if error < tolerance: break
...
>>>
```

while loop

What happens if the computation doesn’t reach the convergence?

It’s better to add a safe exit strategy…

```
>>> while c < max_steps:
...       result, error = compute(result)
...       if error < tolerance: break
...       c += 1
...
>>>
```

Bool conversion

Built-in types can be converted in bool, i.e. they can be used as condition expressions

```
int 0               # False
int != 0            # True
float 0.0           # False
float != 0.0        # True
empty string “”     # False
empty sequence      # False
```

stack: lists and while loop

It is possible to use a list as a stack and pop-out objects until the stack is empty.

```
>>> stack = [obj1, obj2, obj3, obj4, obj5]
>>> while stack:
...     obj = stack.pop()
...     do_some_computation(obj)
...
>>>
```

File I/O

Old style:

```
>>> f = open('filename.txt', 'r')
>>> lst = f.readlines()
>>> f.close()
```

DO NOT USE THAT!!!

New style (stronger):

```
>>> with open('filename.txt', 'w') as f:
...       f.write('some string\n')
```

Iterating on file:

```
>>> for line in f:
...       a_list.append(line.strip())
```

Functions

Function definition

```
>>> def mysum(a, b):
...     “A description of the function.”
...     return a + b
...
```

Function call

```
>>> mysum(4, 6)
10
>>> mysum(‘4’, ‘6’)
‘46’
```

modules

A module is a file containing Python definitions and statements.

The file name is the module name with the suffix .py appended.

Within a module, the module’s name (as a string) is available as the value of the global variable __name__, whereas, when you use the .py file as your main script, the value of __name__ is a string containing “__main__”.

How to use a module

```
$ cat fibo.py
def fib(n):
    "Return Fibonacci series up to n."
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

>>> import fibo
>>> fibo.fib(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Command line arguments

```
$ cat myprog.py
import sys
if len (sys.argv) < 2 :
    print("Usage: python {} <args>".format(sys.argv[0]) )
    sys.exit(1)
for x in sys.argv[1:]:
     print("Argument: ", x)


$ python myprog.py arg1 arg2
Argument: arg1
Argument: arg2

$ python myprog.py
Usage: python myprog.py <args>
```

subprocess

```
>>> from subprocess import Popen
```

The Popen constructor execute a child program in a new process.

```
>>> command_line = “executable -i inp.txt -o out.txt”
>>> p = Popen(command_line.split(), stdout=file_obj)
```

p.poll(), p.wait(), p.communicate(), p.kill(), p.pid

mpi4py

Philosophy

Often only a small portion of the code is time-critical

Use python for everything, apart from heavy work calculation

- Memory management
- Input / Output
- User interface
- Error handling

mpi4py

MPI for Python provides bindings of the Message Passing Interface (MPI) standard for the Python programming language.

This package is constructed on top of the **MPI-1/2/3** specifications

And provides an **object oriented interface** which resembles the MPI-2 C++ bindings.

Python Objects and Arrays

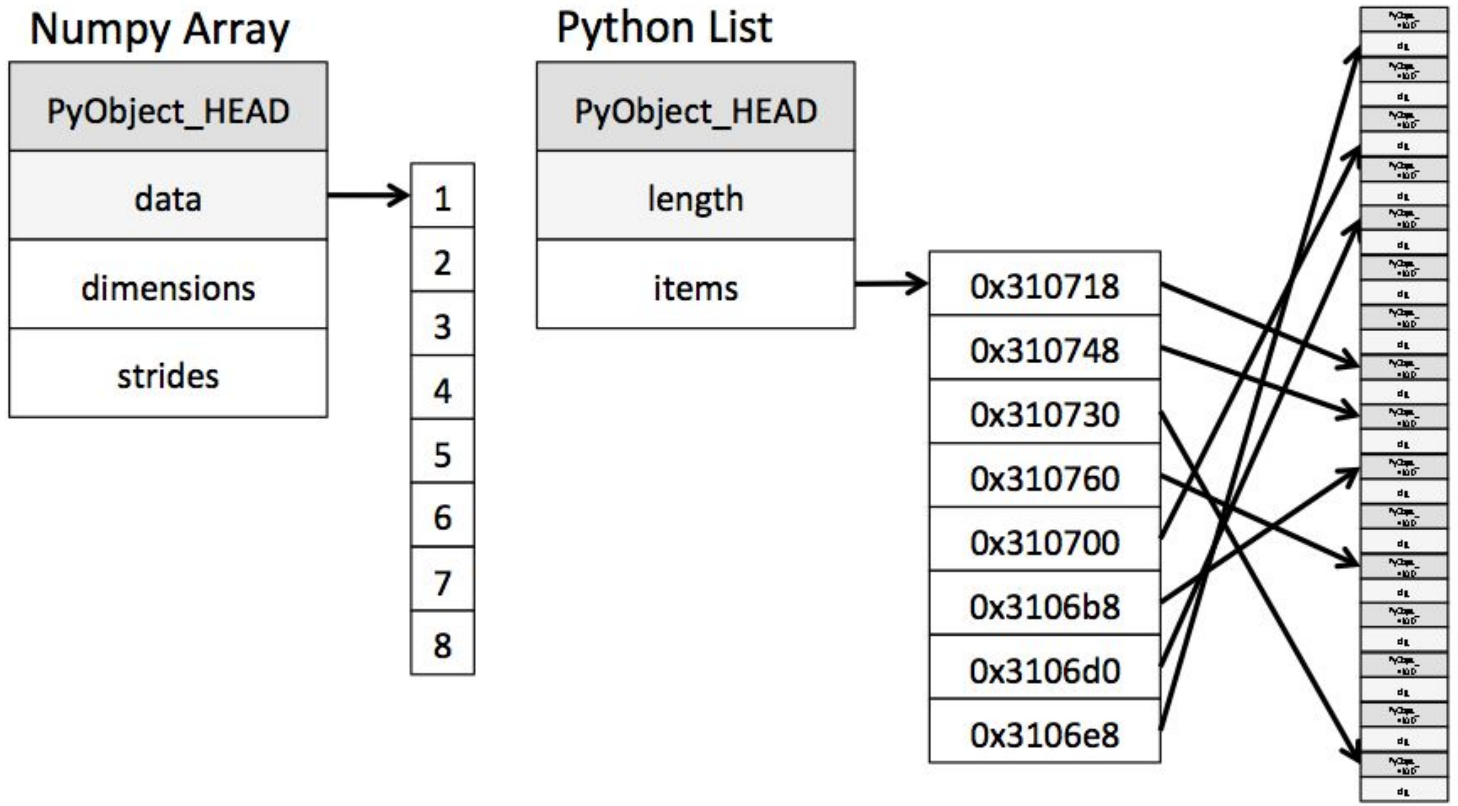
Can communicate any built-in or user-defined Python object

- Under the hood: the pickle module
- Builds binary representations of the objects
- Restores them back (at receiving processes)
- This procedure can be slow

MPI for Python supports also direct communication of any object exporting the single-segment buffer interface.

- e.g. Numpy arrays
- Performance close to C speed

Numpy Array vs. Python List



No need to call MPI_Init() or MPI_Finalize()

- MPI_Init() or MPI_Init_thread() is actually called when you import the MPI module from the mpi4py package, but only if MPI is not already initialized.

```
from mpi4py import MPI

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

- MPI_Finalize() is automatically called when Python processes exit, but only if mpi4py actually initialized MPI.

Point to point

- Functions with the first upper case letter can communicate memory buffers:

  Send(), Recv(), Sendrecv()

- Functions with lower case letters can communicate generic Python objects:

  send(), recv(), sendrecv()

- Non-blocking communications are also available.

Point to point

```
from mpi4py import MPI

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

if rank == 0:
    comm.send([rank, 10+rank, 100+rank], dest=1, tag=10)
    buf = comm.recv(source=1, tag=20)
else:
    buf = comm.recv(source=0, tag=10)
    comm.send([rank, 10+rank, 100+rank] , dest=0, tag=20)

print("my rank is %d, I received %s from %d" % (rank, buf, buf[0]))
```

Point to point

```
import numpy as np

# passing MPI datatypes explicitly
if rank == 0:
    data = np.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = np.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)
```

Point to point

```
import numpy as np

# automatic MPI datatype discovery
if rank == 0:
    data = np.arange(100, dtype=np.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = np.empty(100, dtype=np.float64)
    comm.Recv(data, source=0, tag=13)
```

MPI.ANY_SOURCE

```
...
if rank == 0:
    buf = comm.recv(source=MPI.ANY_SOURCE, tag=20)
else:
    comm.send([rank, 10+rank, 100+rank] , dest=0, tag=20)
...
```

MPI.ANY_TAG

```
...
if rank == 0:
    comm.send([rank, 10+rank, 100+rank], dest=1, tag=10)
else:
    buf = comm.recv(source=0, tag=MPI.ANY_TAG)
...
```

MPI.Status()

```
...
infos = MPI.Status()

buf = comm.recv(source=0, tag=MPI.ANY_TAG, status=infos)

print(infos.Get_tag() )
print(infos.Get_count() )
print(infos.Get_elements() )
print(infos.Get_error() )
print(infos.Get_source() )
...
```

Collective communications

```
comm.Barrier()     # synchronization
```

Global communications:

- Broadcast
- Gather
- Scatter

Global reduction operations

Broadcast

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'key1' : [7, 2.72, 2+3j],  'key2' : ( 'abc',
'xyz')}
else:
    data = None

data = comm.bcast(data, root=0)   # broadcast of a dict
print(rank, data)
```

- Documentation:

  https://mpi4py.readthedocs.io/en/stable/index.html

- Tutorial:

  https://mpi4py.readthedocs.io/en/stable/tutorial.html

- API Reference:

  http://inf2.uniri.hr:8080/docs/mpi4py/apiref/

Hands on: starting material

You can start from this three files:

- serial_program.c
- get_inputs.py
- single_queue.py

The idea of this framework is to have a toy model replicating a general situation in which we have a serial program which takes some time to elaborate an input and print an output.

The idea of the exercises is to develop a tool which allows to take advantage of the many-core infrastructure in order to execute many serial instances in parallel.

A toy serial program

serial_program.c is a C code who sleeps for some seconds, prints how long it slept and then exit. The number of seconds it sleeps are read from a text file given as input.

e.g. if input.txt contains “0.631”

```
./serial_program.x input.txt > output.txt
```

print an output file “output.txt” which contains “Waiting for 0.631 seconds…”, it waits for 0.631 seconds and then exit.

You can inspect the source code if you wish.

An inputs generator for the serial program

The purpose of **get_inputs.py** is to generate a huge number of input for the serial program above. As argument, it takes the number of inputs to be generated, and the content of each input is a random number between 0 and 1 inside a directory called “inputs”.

e.g.

```
./generate_inputs.py 3000
```

create an “inputs” subdir in the current directory containing text files named from “inputs0000.txt” to “inputs2999.txt”, each one containing a random number between 0 and 1. You can inspect the source code if you wish.

A wrapper for the serial program

In order to process such huge number of inputs, the compiled executable can be wrapped for instance by a python script which cycle every input “feeding it” to the executable, and redirect the output to a different file each time.

A possible implementation is shown in **single_queue.py**. It’s worth exploring its content in the next slides.

single_queue.py: imported modules

```
#!/usr/bin/env python

from __future__ import print_function

import os

import sys

import glob

import time

from subprocess import Popen
```

(...)

single_queue.py: main

(here there are the functions definitions, shown in the next slide)

```
if __name__ == '__main__':

    executable = os.path.abspath(sys.argv[1])

    input_dir = os.path.abspath(sys.argv[2])

    inputs = input_files(input_dir)

    print("Number of elaborations: {:d}".format(len(inputs)))

    t1 = time.time()

    run(executable, inputs)

    t2 = time.time()

    print("Elapsed time {:5.2f}".format(t2-t1))
```

single_queue.py: functions

```
def input_files(my_path):

      return glob.glob(my_path+'/input*.txt')



def run(executable, inputs):

      for inp in inputs:

      with open(inp+'.out', 'w') as out_file:

            pid = Popen([executable, inp], stdout=out_file)

            pid.wait()
```

Run single_queue.py

- Generate the input files

```
module load python/2.7.12

python get_inputs.py 3000
```

- Load the module of the compiler and compute the serial code:

```
module load profile/archive

module load intel/pe-xe-2017--binary

icc -o serial_program.x serial_program.c
```

- Submit the batch script

```
chmod 755 single_queue.py

sbatch job_launcher-single_queue.sh
```

Exercise 1: Multiple queues

The simplest approach is to replicate what is done by **serial_queue.py** for each task, by assigning a list of inputs to each of them.

Try to do it writing a **multiple_queues.py** python script, based on **single_queue.py.**

When the script is ready, you can run it by typing

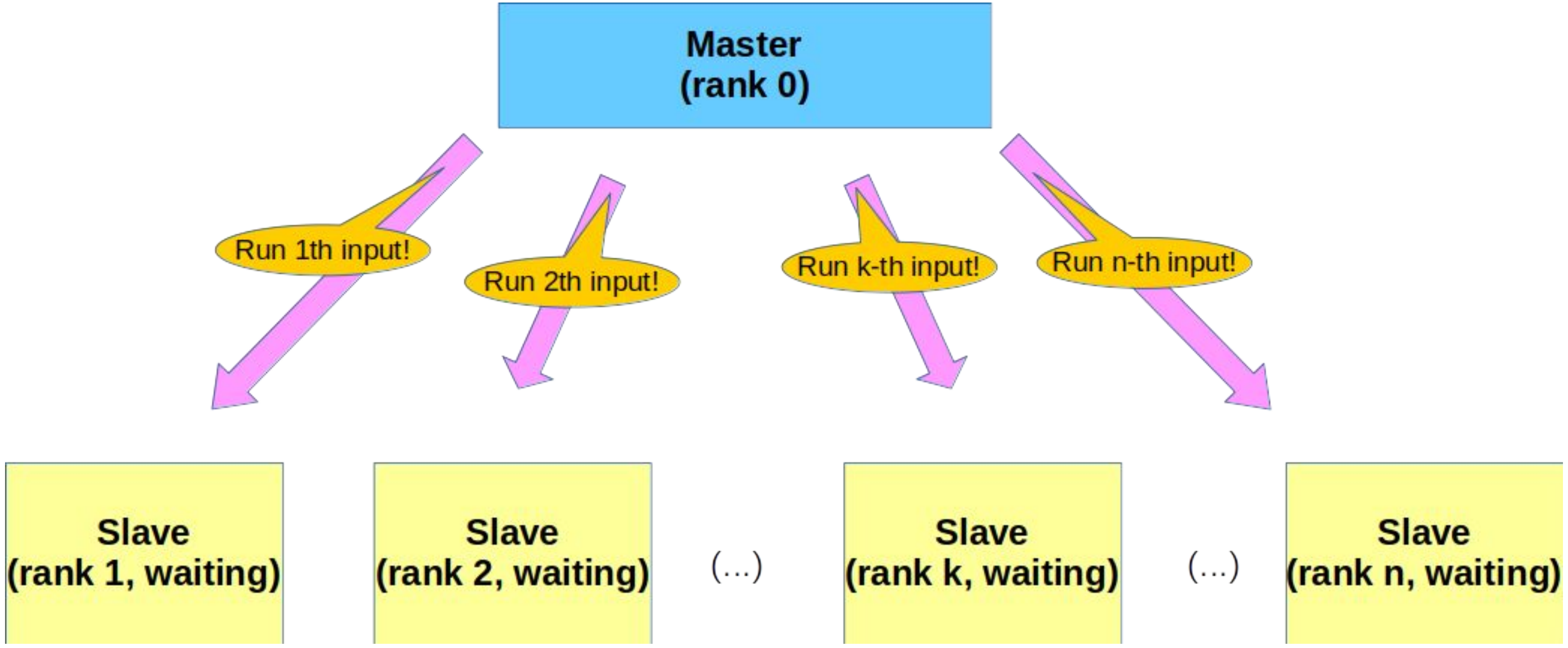
```
chmod 755 multiple_queues.py

sbatch job_launcher-multiple_queues.sh
```

Exercise 2: Master-Slave

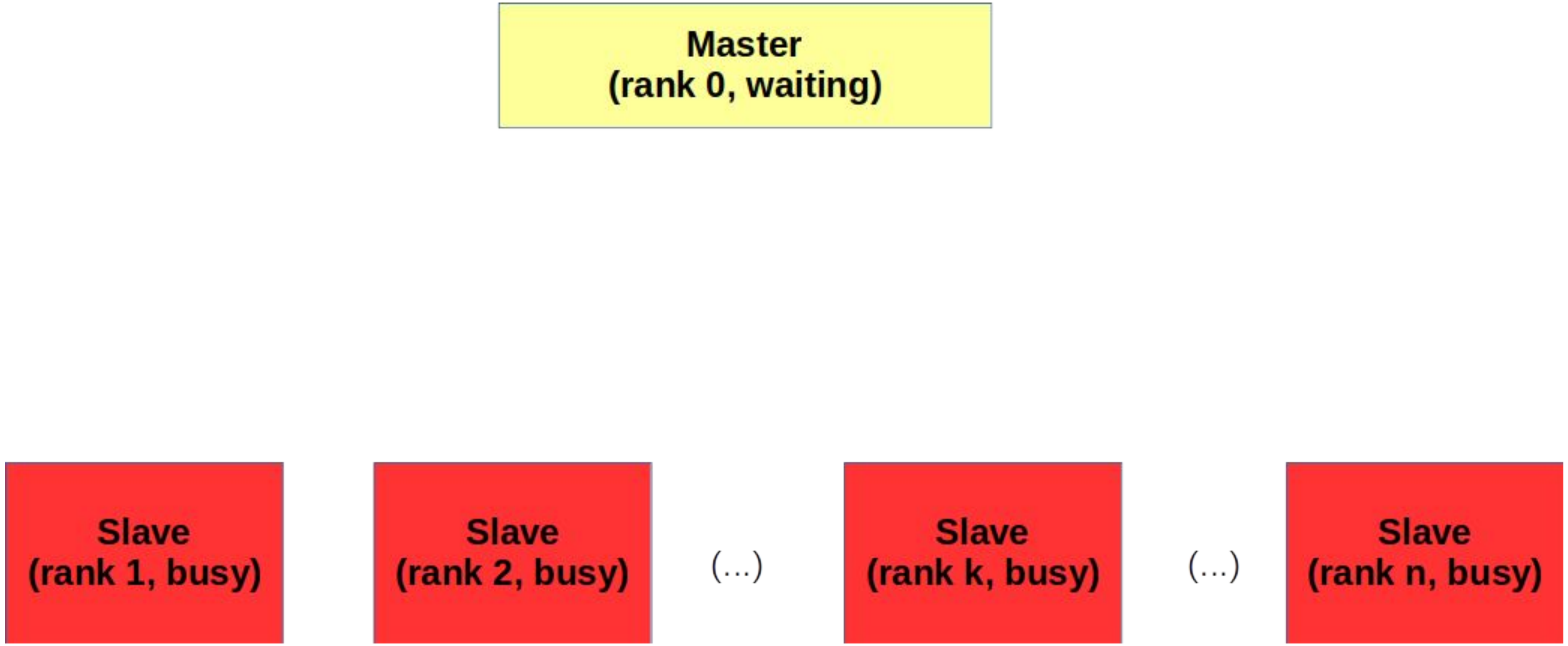
The approach seen in Exercise 1 is not the most efficient when you have a huge number of inputs, because when one task completes its queue, it becomes idle despite the fact that there is still work to do (carried on by the other tasks).

This can be improved with a **master-slave approach**: the inputs for a single task aren’t communicated all at once to each of them, but they’re distributed one by one as soon as one task becomes free.

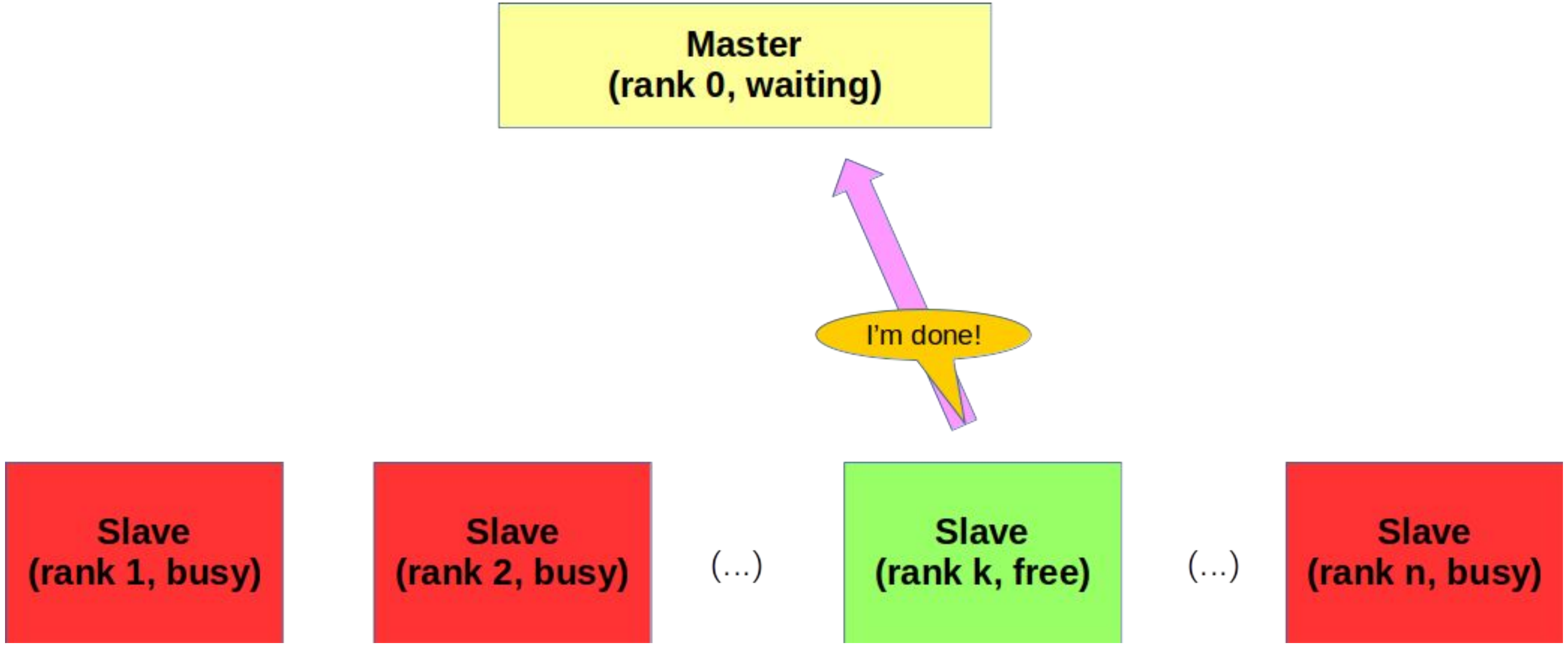
Master sends to the slaves



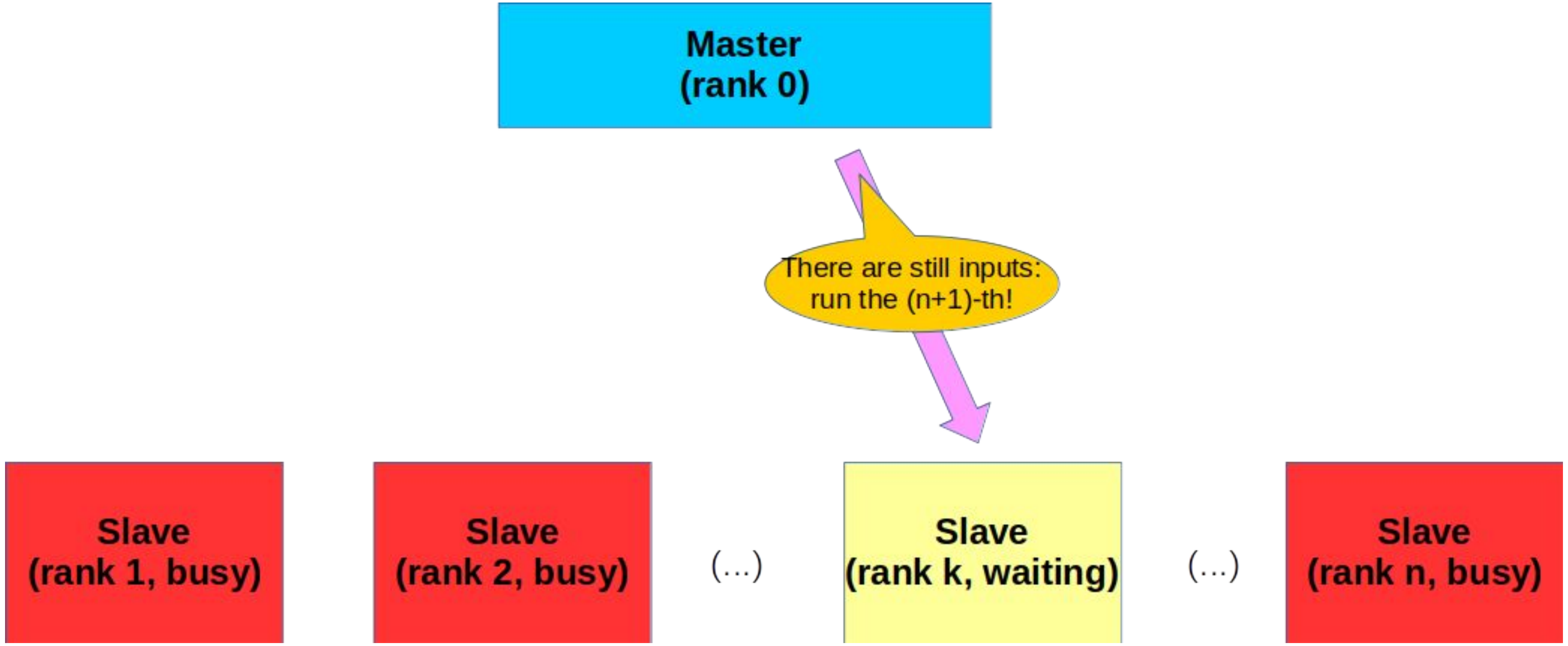
Slaves perform the first n computations



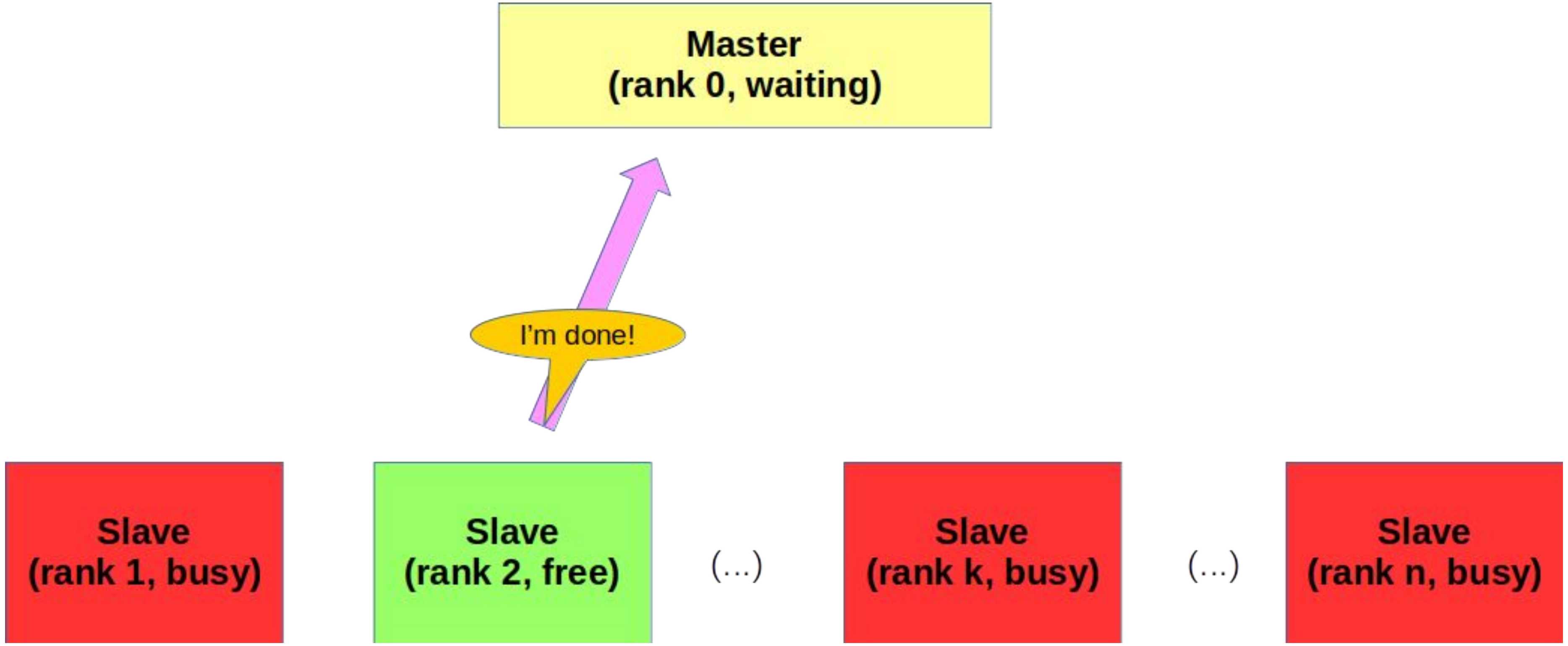
The slaves inform the masters when they finish



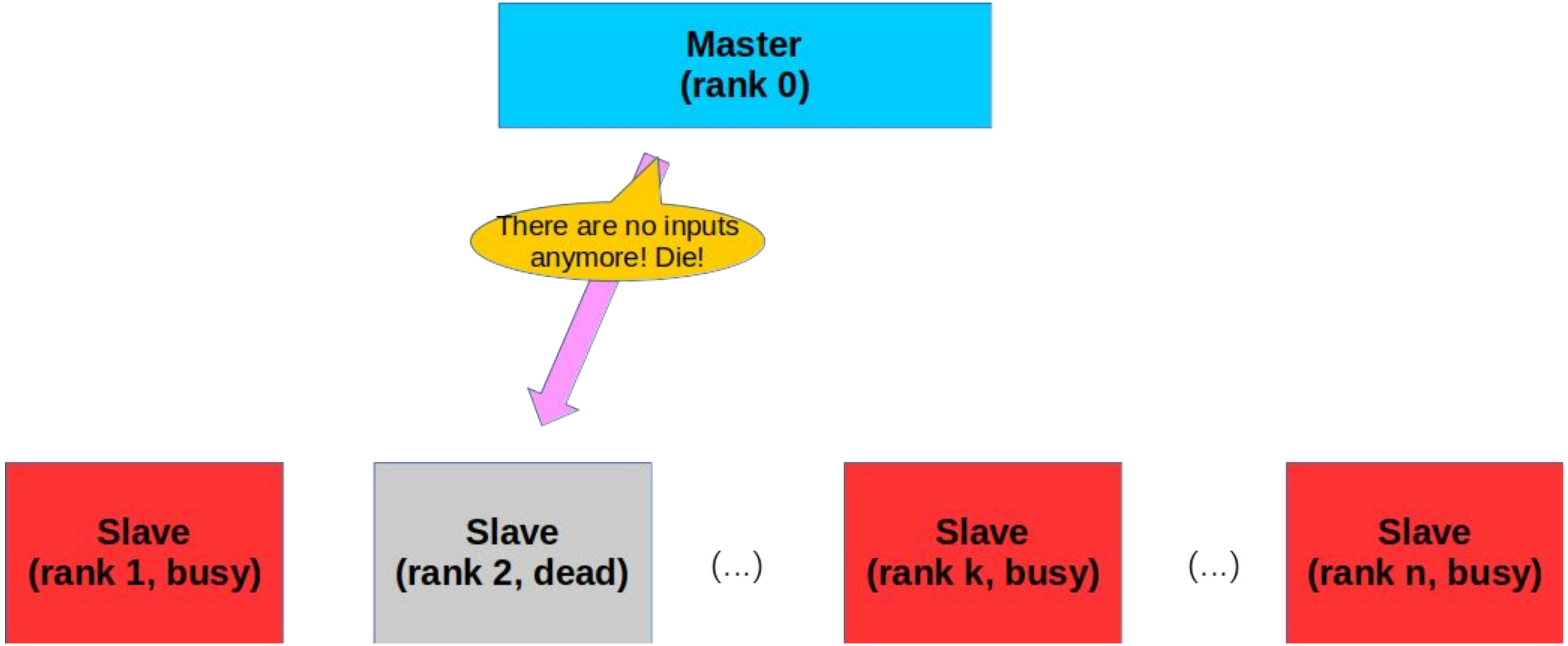
Master distributes inputs as long as there are



When the master run out of inputs, it kills the slaves



When the master run out of inputs, it kills the slaves



Additional hints

- How does the master receive without knowing who sent?
- How does the slave receive different informations from the master?
- How does the slave extract which informations has been received?

When the script is ready, you can run it with

```
chmod 755 master_slave.py

sbatch job_launcher-master_slave.sh
```

Exercise 3: fault tolerance

Python can be employed in HPC in many different ways, but often it can be useful to wrap a compiled code in order to enhance its performances (as seen in the previous exercises).

A further application can be thought in order to prevent possible errors on the nodes for some compiled MPI application. The idea is to run an mpi program avoiding some possible non-working node.

This can be achieved by wrapping an MPI executable with a python script which check the status of the nodes (assigned by the workload manager), and run the executable (with mpirun) only on the working ones.

Additional hints

You might for instance implement a function which check the nodes status: in order to simulate a condition to check, you might use a fake condition generated by the following:

```
def some_condition():
        rn = randint(1,20)
        if rn < 20:
            return True
        else:
            return False
```

Additional hints

As mpi executable to submit, you can use **mpi_program.c**, which takes a string as argument (from **fault_tolerance-input.txt**), print how many cores are reading that string and then sleeps for 2 seconds. You can inspect the code if you wish.

Compile it with

```
module load autoload profile/archive intelmpi/2017--binary

mpiicc -o mpi_program.x mpi_program.c
```

A “nodfile.txt” with the list of nodes will be created by the slurm script. When the script **fault_tolerance.py** is ready, you can submit it it with

```
chmod 755 fault_tolerance.py

sbatch job_launcher-fault_tolerance.sh
```