Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Computational performance for Python applications

Victor Correal, Ivan Alsina, Xavier Yepes-Arbós

victor.correal@bsc.es

November 14, 2024

Session overview

- Process to request a profiling and performance evaluation study of a data consumer application
- Basics concepts of parallel computation
- Computational aspects of Python applications
- Introduction to profiling analysis
- Hands-on: flat profiling of a Python application
- Concluding remarks

A3 performance assessment services

- New to phase II: HPC services upon request
- What kind of services? Profiling and performance assessment of data consumer applications
  - Suggestion of optimizations for potential bottlenecks
  - To be implemented by developers
- Services provided by BSC and CSC in A3
- Resources are limited
  - Developers should elaborate a prioritised list based on the bottlenecks they have previously identified

A3 performance assessment services

- Formally submit a request by opening an issue in the BSC DE_340 GitLab and tag the A3 leaders (Mario Acosta and Tuomas Lunttila)
- Essential information should be included to make a good estimation of the effort needed to address the request:
  - Purpose of the use case
  - Why do you think there is a bottleneck and an initial estimation of its impact
  - Computational cost (time and resources used)
  - Documentation and diagram of the code
  - Repository, use case, etc
  - Programming languages used, e.g. MPI
  - ...

A3 performance assessment services

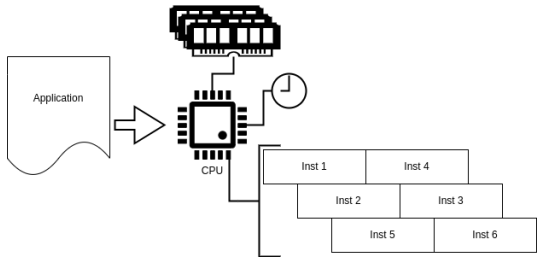
- Based on the information provided in the request, A3 leaders will determine:
  - Feasibility
  - Assign responsibilities
  - Priority level
- Task coordination will be via Gitlab
- If the request is rejected, feedback will be provided to reconsider or adapt the request
- More details can be found in deliverable D340.15.1.1.

Programming paradigms

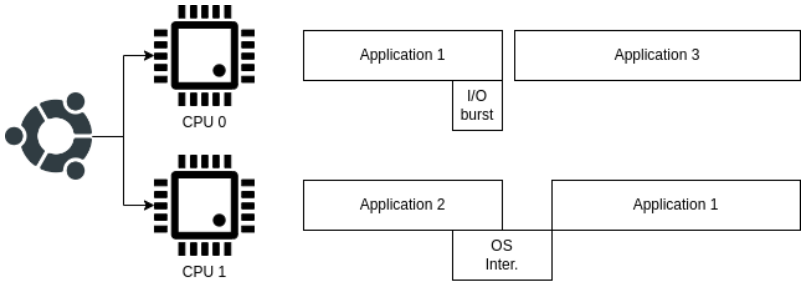
Introduction to computational performance

Three factors define the computational performance:

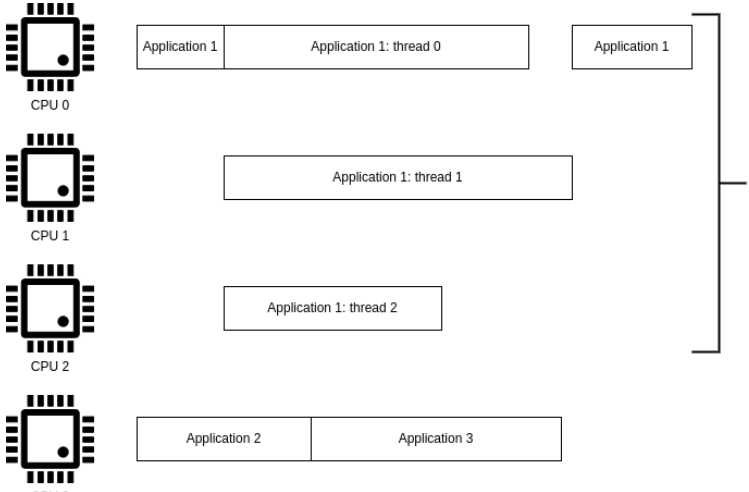
1. Number of instructions
2. Instructions executed per cycle
3. Clock frequency



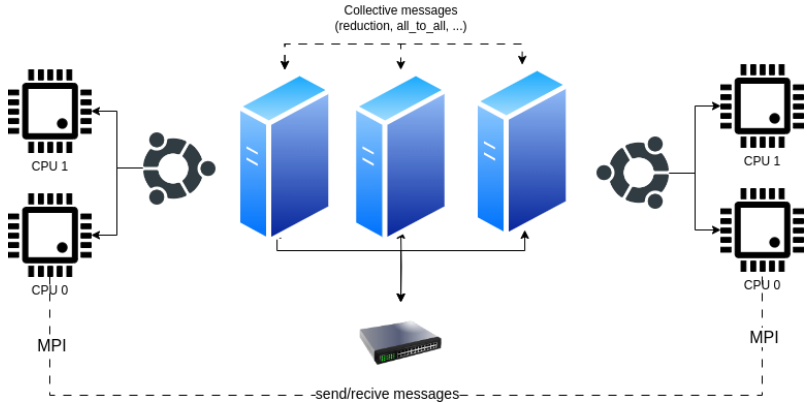
Concurrency versus Parallelism



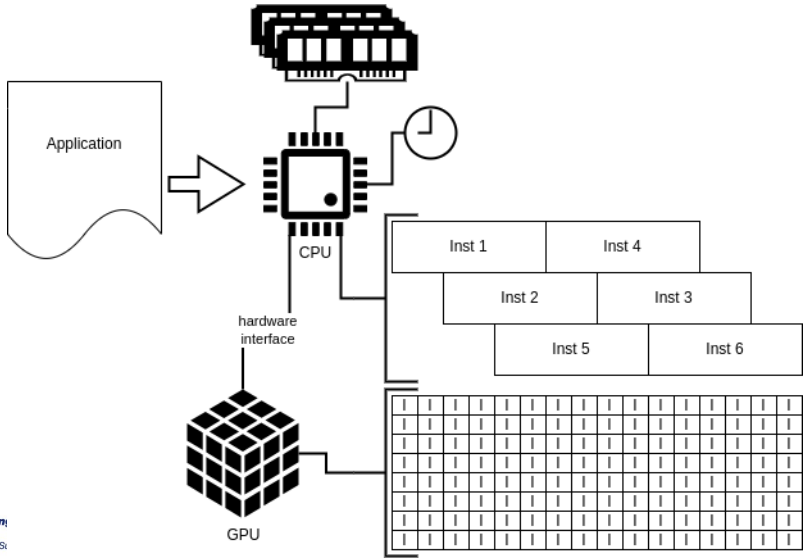
Concurrency versus Parallelism



Multi-node parallelism: Message-passing interface (MPI)



Computing accelerators



In summary

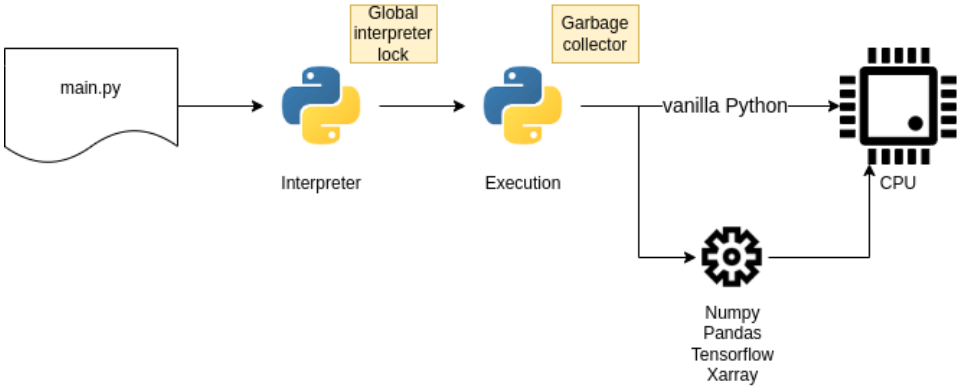
- Concurrent execution: Several programs share hardware resources.
- Parallel execution: 1 program splits the workload among the available hardware resources, sharing resources (i.e. memory, files, disk).
- Multi-node: 1 program, split the workload among several machines sharing resources (memory, files, disk) at each machine.

Always…

All the executions of the same program need to ensure the same results.

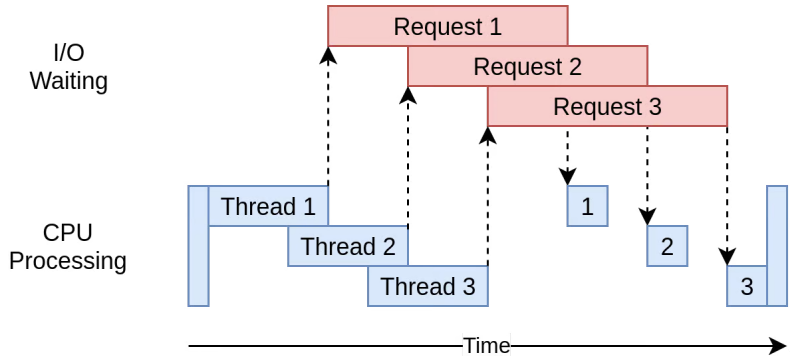
Computational aspects of Python

Python environment



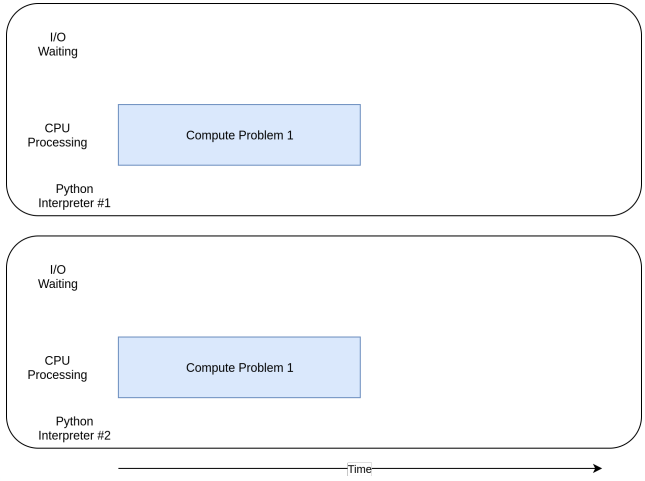
I/O-bound applications

Using `threading` or `asyncio` Python modules you can optimize a reading or writing process



CPU-bound applications

The module `multiprocessing` can help with speed-up in computing operations



Example

Example at `https://earth.bsc.es/gitlab/ialsina/python-benchmarks/`

1. Read data vectors A_i, B_i from binary files
2. Compute $\alpha \cdot A_i + B_i$
3. Write results to binary files

```
python3 main.py <mode> --nums-per-file 1000000 --files 50
```

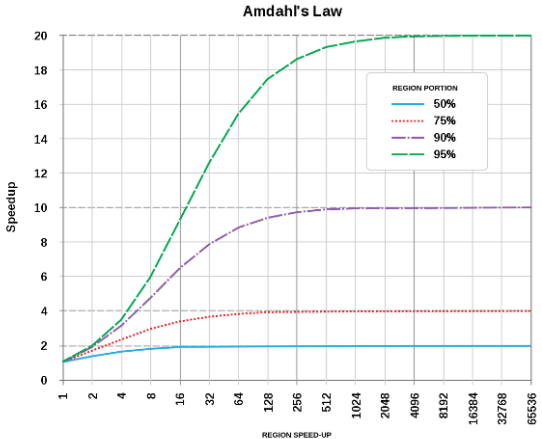
<mode>	**Processes**	**Time**
sequential	1	10.48 s
concurrent	4	4.59 s

Table: Executing a toy Python Application concurrently and sequentially

Profiling basics

Profiling state-of-the-art: Amdahl's law

"The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used"



Profiling state-of-the-art: Best practices

- Profile real-world scenarios.
- Premature optimization can be useless.
- Validate optimizations with test.
- Stick to Amdahl's law
- Use appropriate tools (and make the correct measures).

Python generic optimizations

Unfortunately, almost all optimization problems require an ad-hoc solution, but…

- Always use already-optimized libraries that matches your requirements. Don’t re-invent the wheel.
- Split your application workload using concurrency or parallelism.
- Overlap I/O and computation.
- Hide latency of heavy tasks with other tasks.

Python profiling utils

```
      828758 function calls (795087 primitive calls) in 0.529 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   7/1    0.000    0.000    0.527    0.527 .../one_pass/opa.py:466(compute)
   7/1    0.000    0.000    0.447    0.447 .../one_pass/opa.py:342(_finished_no_append)
   6/1    0.000    0.000    0.446    0.446 .../one_pass/opa.py:280(_call_recursive)
     7    0.000    0.000    0.412    0.059 .../one_pass/statistics/update_statistics.py:755(update_statistics)
     7    0.000    0.000    0.399    0.057 .../one_pass/statistics/update_statistics.py:705(update)
     7    0.001    0.000    0.399    0.057 .../one_pass/statistics/update_statistics.py:90(update_mean)

                                          ...
```

Hands-on


BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Python profiling: cProfile

Python is equipped with the standard library modules `profile` and `cProfile`.

1. Command line interface

```
$ python -m cProfile my_script.py
```

2. Programmatically

```
import cProfile
cProfile.run("my_function()")
```

3. Programmatically, to file

```
import cProfile
cProfile.run("my_function()", "my_benchmark.prof")
```

Python profiling: cProfile

4. Via a profiler instance

```
import cProfile

profiler = cProfile.Profile()
profiler.enable()

my_function()

profiler.disable()
profiler.dump_stats("my_benchmark.prof")
```

Python profiling: cProfile

5. Via a wrapper

```
import cProfile
import functools

def profile(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        profiler = cProfile.Profile()
        profiler.enable()
        result = func(*args, **kwargs)
        profiler.disable()
        profiler.dump_stats(f"benchmark_func.__name__.prof")
        return result
    return wrapper

@profile
def my_function():
    ...
```

Python profiling: pstats

1. Write to `io.StringIO`

```
from pstats import Stats, SortKey
from io import StringIO

sortby = SortKey.CUMULATIVE
s = StringIO()
stats = Stats("my_benchmark.prof", stream=s).sort_stats(sortby)
# "my_benchmark.prof" can be a profiler instance instead.
stats.print_stats()
print(s.getvalue())
```

2. Write to file stream: Pass a file object as a *stream* argument.

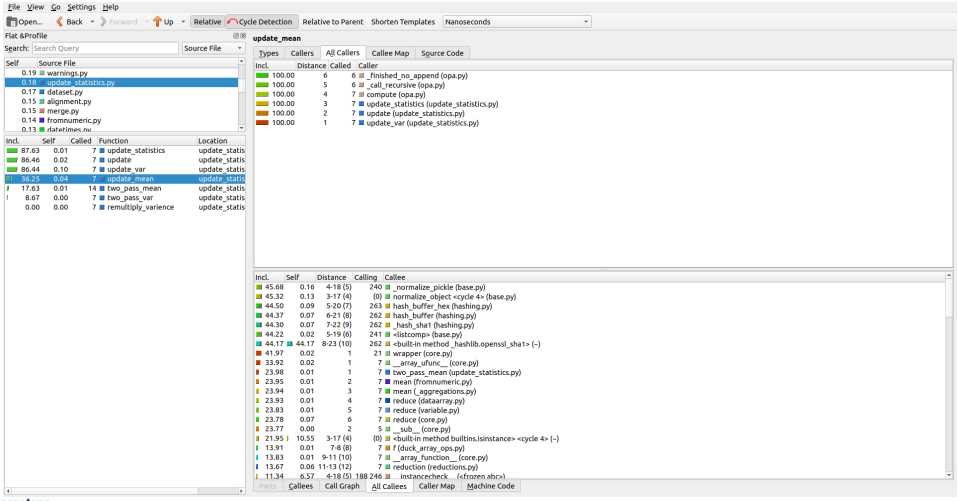
KCacheGrind

Tool that allows graphical visualization of the profiler.

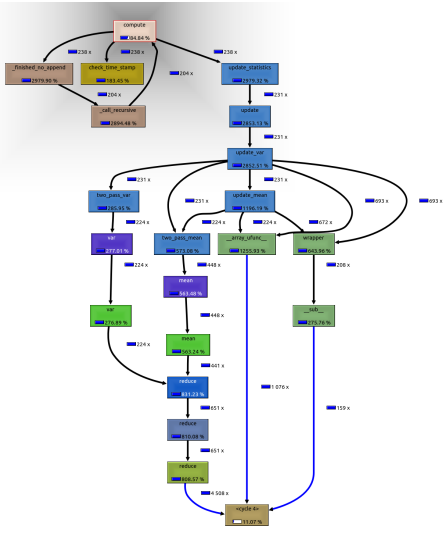
- Absolute or relative times
- Cycle detection
- Sortable caller and callees lists and maps
- Graph call
- Need external Python package: `pyprof2calltree`

```
# Install
sudo apt install kcachegrind
pip install pyprof2calltree
# Launch
pyprof2calltree -i my_benchmark.prof -k
```

KCacheGrind



KCacheGrid


compute
184.84 %
238 x
238 x
238 x
_finished_no_append
2979.90 %
check_time_stamp
183.45 %
204 x
update_statistics
2979.32 %
206 x
231 x
_call_recursive
2694.48 %
update
1863.13 %
231 x
update_var
1852.51 %
231 x
231 x
two_pass_var
305.95 %
231 x
update_mean
1156.19 %
693 x
693 x
224 x
224 x
224 x
672 x
var
277.01 %
two_pass_mean
573.08 %
array_ufunc__
1255.93 %
wrapper
643.96 %
224 x
448 x
206 x
mean
363.48 %
var
276.89 %
448 x
_sum
275.76 %
224 x
axes
141.24 %
441 x
reduce
331.23 %
1 076 x
451 x
159 x
reduce
277.08 %
451 x
reduce
206.17 %
4 508 x
<cycle 6>
11.07 %

To take away

- You are developing for an HPC computer, use it correctly
- Consider basic profiling to develop your application
- Don’t re-invent the wheel!
- Contact A3 if you have any doubt

Concluding remarks

- Do not hesitate to make use of the A3 peformance assessment service!
  - Make a preliminary basic profiling of the application
  - Request support by opening a Gitlab issue and tag A3 leaders
  - Attach to your request all necessary infomration to help us to process it properly.
- Any question?

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you for your attention

Victor Correal, Ivan Alsina, Xavier Yepes-Arbós

victor.correal@bsc.es