Parallel Programming with Python

Heinrich Widmann

LES Python Seminar 2012

08.11.2012

widmann@dkrz.de

Overview

- Why?
- How?
- Methods and approaches (within Python)
- Some general issues
- Coding of a simple test application
- Some Results
- Discussion

Why parallelisation ?

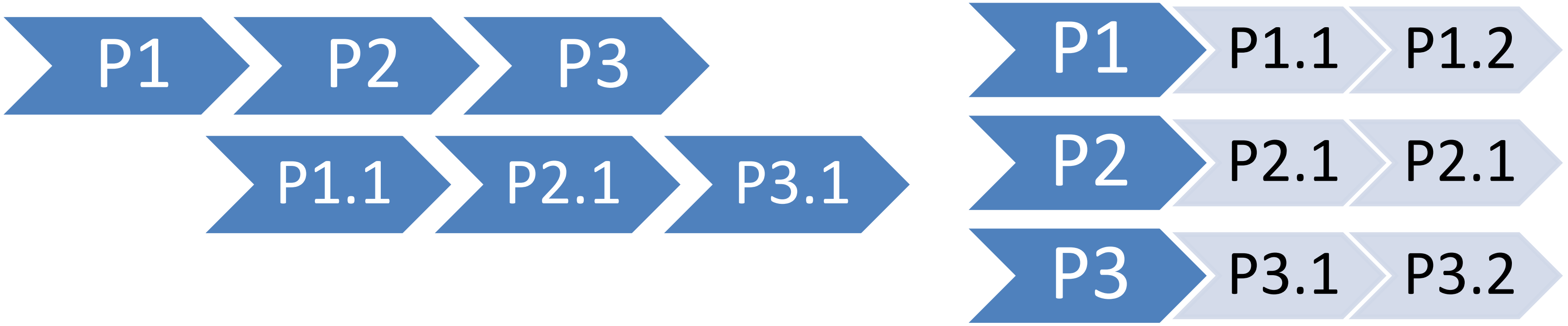
- Improve performance (only speed ?)
- Parallel design, coding and implementation may be more convenient and less complex
- Efficient usage of compute (and memory/storage) resources
- Better scaling over processes/CPUs and nodes
- … but things can as well get worse …

How to parallise

- Examine your application for
  - Ability to paralise
  - Appropriate level/kind of parallelisation
- First analyse and divide your code in ''parallisable'' chunks
- Choose method and ''level'' (OS, Python, cleint/server, …)
  1. sequential code on 1 CPU → parallel code run on n CPUs
  2. Execution on 1-2 CPU on 1 node → multi nodes, clustering
- Avoid interruptions, locks, bottlenecks etc.
- Note process and input/output dependencies
- Scheduling, load balancing, …, usage of memory, …
- Check that results are reproducable

Here : only computing
(forget I/O and data (dependencies))

From sequential to → parallel execution
of processes P1, P2, ...



GIL or no GIL (threading vs. processes)

GIL = Global Interpreter Lock

- enshures that only 1 thread runs in the interpreter at once

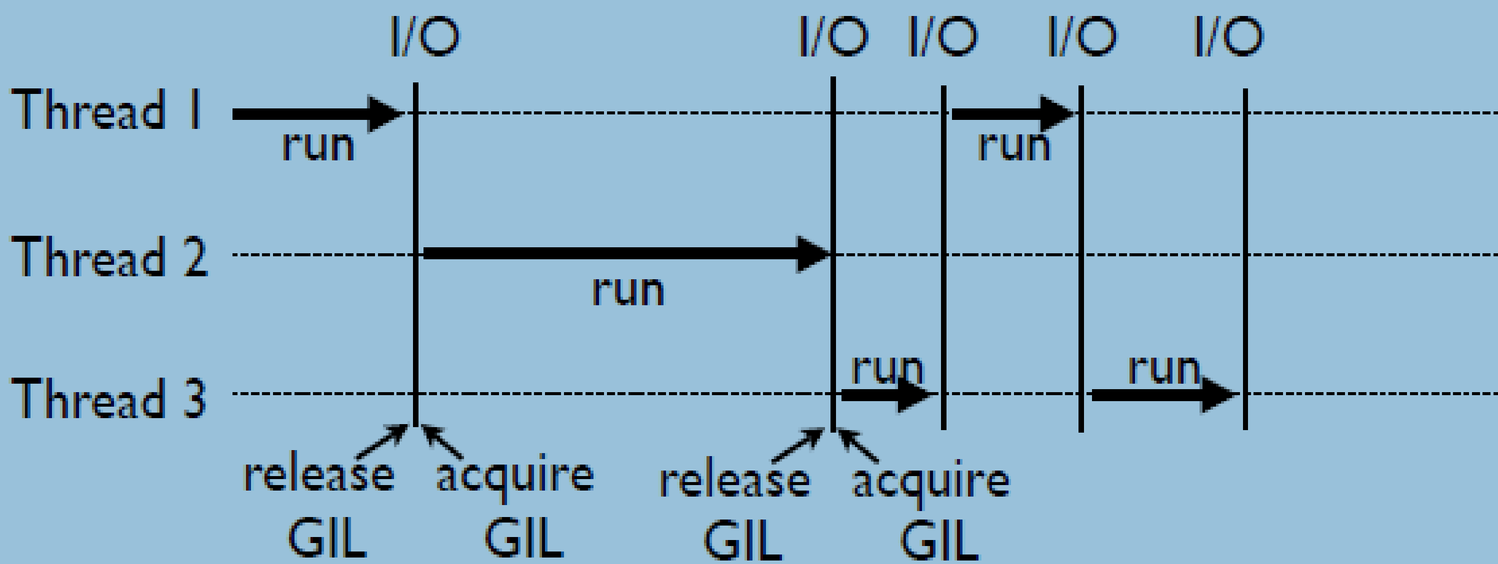
➢ Limits performance by forcing serial execution

  ➢ (lock(T1)-> release(T1)->lock(T2)-> …

- But GIL
  - simplifies maintainance and management (e.g. of memory and process comm.) by the Python Interpreter
  - leads in most cases not to significant difference in performance, because thread management shifted to kernel level

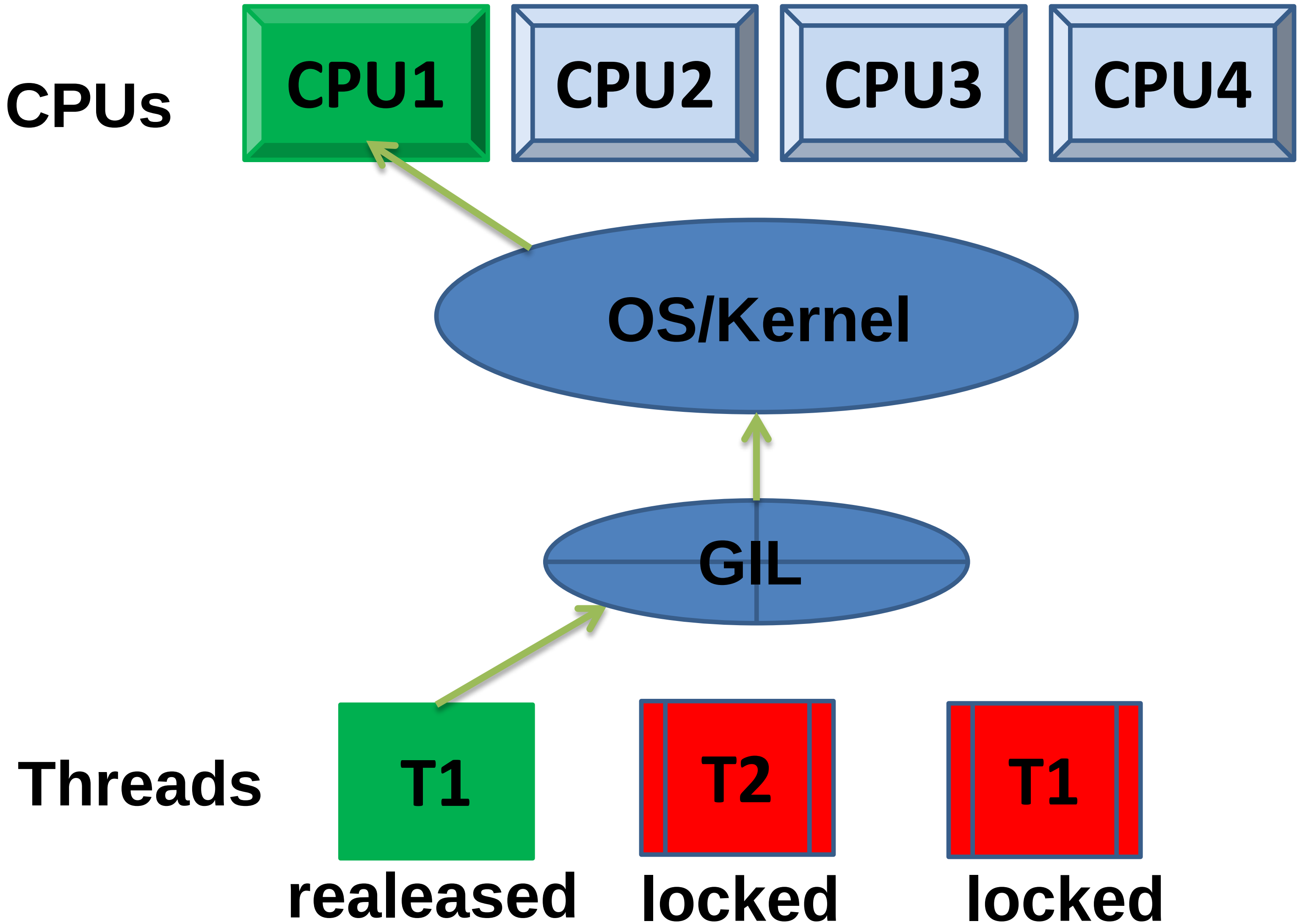
Thread Execution Model

- With the GIL, you get cooperative multitasking

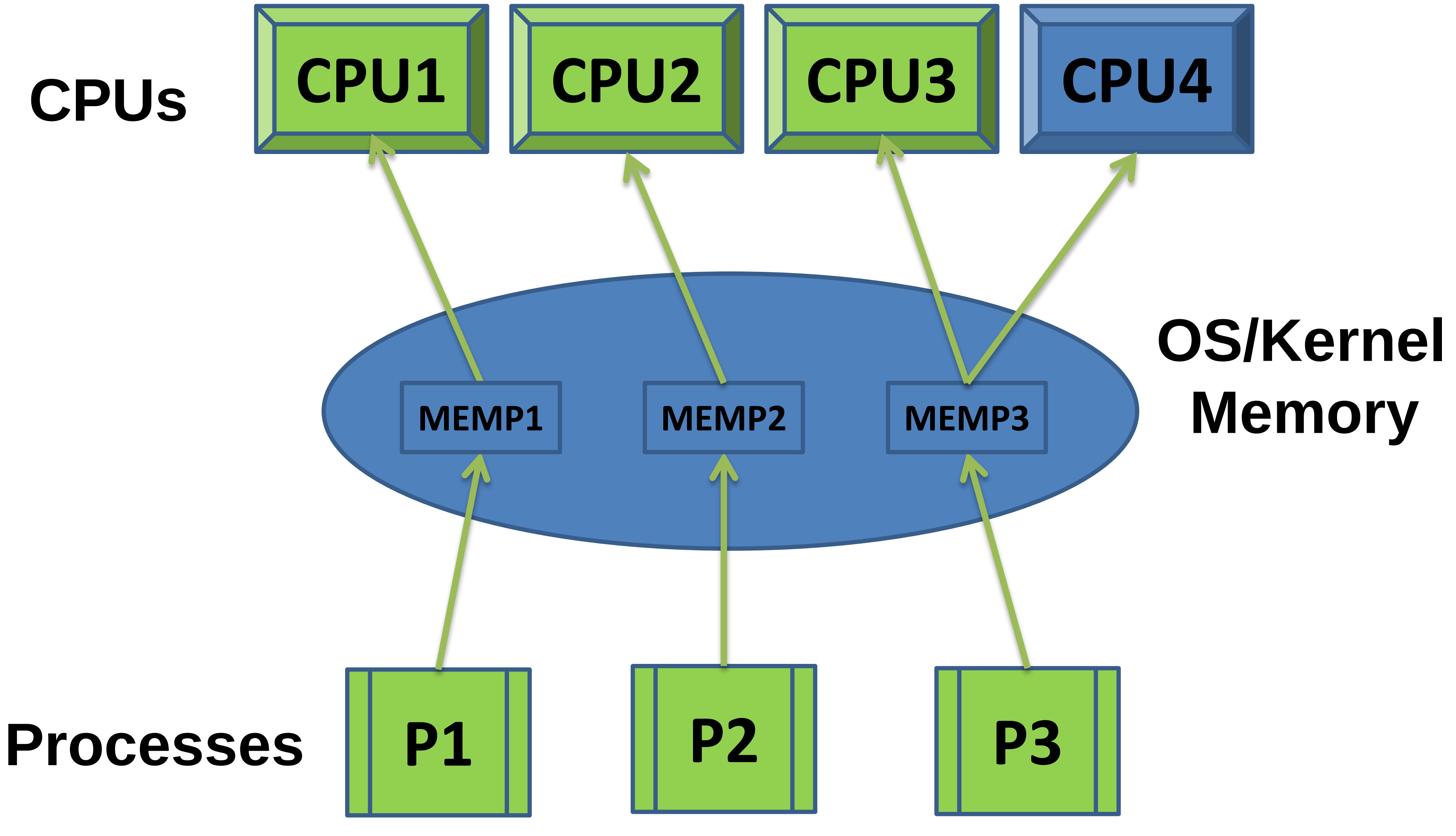


- When a thread is running, it holds the GIL
- GIL released on I/O (read,write,send,recv,etc.)

Threading scheme



Processing scheme



Methods

- C (sequential reference code)
- Python
  - Sequential
  - Multiprocessing
  - Threading (joined/not joined)
  - ParallelPython (ncpus)
  - Cython

Further methods
(not addressed here)

- (vectorisung in) Numpy
- Jython
- Gearman
- PyCloud
- PyPy
- Ipython cluster
- … and a lots more …

C [and/or … ?]

- Reference code
- Often C is already a good choice (in terms of performance)
- Code : sum_primes.c
- Compile :
  – $ gcc [-O ??] -o sum_primes.o sum_primes.c
- [what‘s about other lang.‘s as fortran, C++, etc.]

Threading

- comes with GIL
- All threads work on the same memory space
- N>1 threads slower than 1 thread implementation
- Explicit lock/release management must be „self done“
- But some advantages (at least in Python) :
  - allows fast developement
  - results in nicer, structured code, which is easier to maintain

Multiprocessing

- Similar to threading API, but
- Only process based
  - no GIL
  - transparent for developers
- Because processes are copied
  - starting of processes is slower
  - more memory is required
  - process synchronisation is more complex and tedious

multiprocessing Pool

- `# multiproc.py`
- `p = multiprocessing.Pool()`
- `po = p.map_async(fn, args)`
- `result = po.get() # for all po objects`
- join the result items to make full result

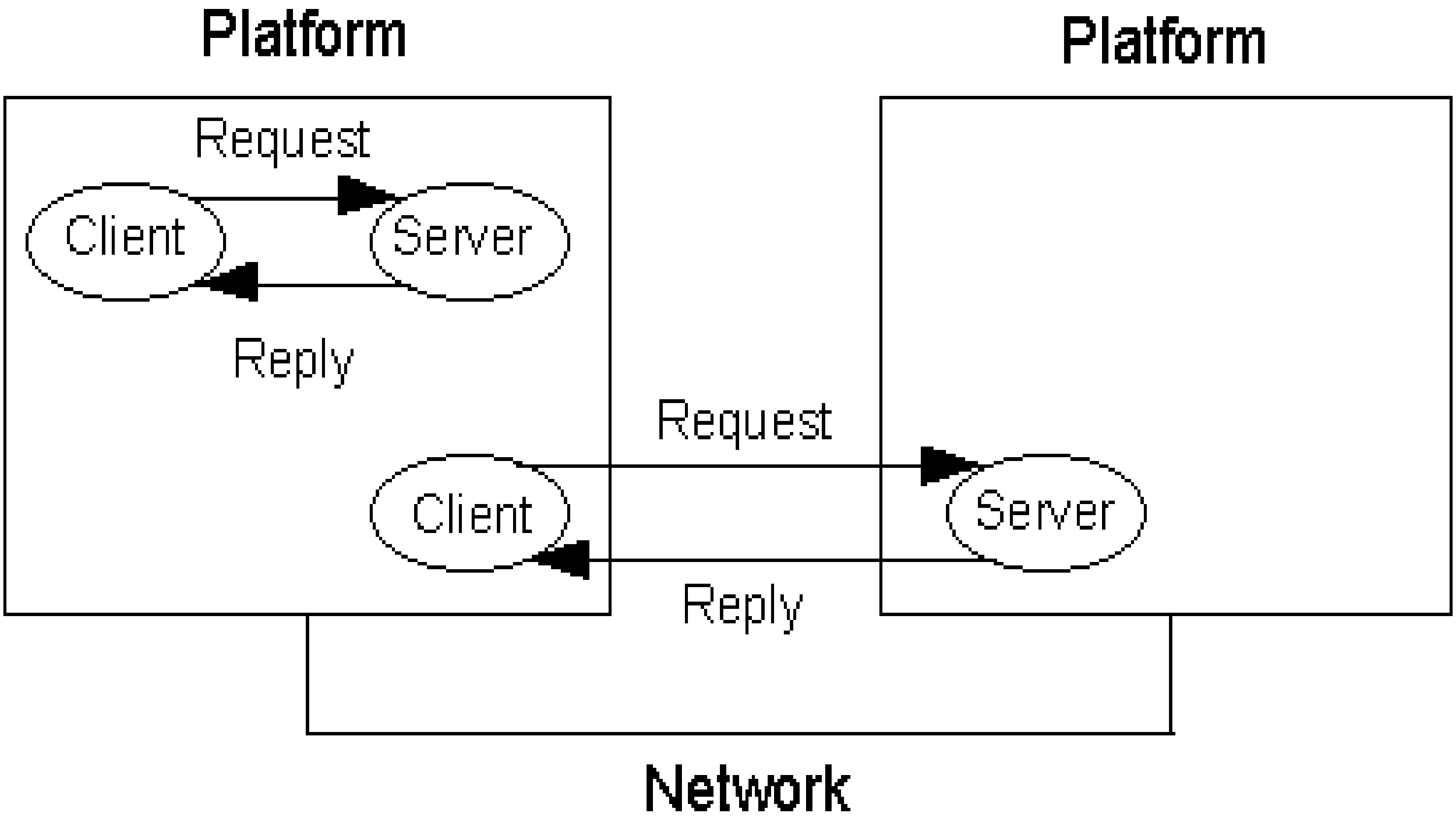
How much memory moves?

- `sys.getsizeof(0+0j) # bytes`
- 250,000 complex numbers by default
- How much RAM used in `q`?
- With 8 chunks - how much memory per chunk?
- `multiprocessing` uses `pickle`, max 32MB `pickles`
- Process forked, data pickled

ParallelPython pp

- client/server based approach
- pp starts own server processes
- This works on single nodes, SMPs and on clusters
  - if on all nodes a pp server is running
- parallel excecution with discrete processes

Client/server model



Cython

- Own (but python similar) programming language
- Easy and fast developement (compared to C)
- Easy parallelisation (OpenMP used)
- Very performant
- Meanwhile one of the favorite choices, used in more and more new projects …

Cython work flow

- Code :
  - sum_primes_cython.pyx (your Cython code)
  - ➢ sum_primes_cython.c (generated C-code)
- Convert to and compile C-code:
  - $ python setup_prim_cython.py build_ext –inplace
- Run :
  - $ python run_sum_primes_cython.py

Test application „sum_primes“

- Sum of all prime numbers < N,
  - Parallise over n in N ={100000,100100, … }
- Implementation as nested loop (recursive function call)
  - Loop1 : n in N (parallize !!)
    - sum_primes(n) {

      Loop2 : x in (2,…,sqrt(n))

      If isprime(x) => sum+=x

      return sum

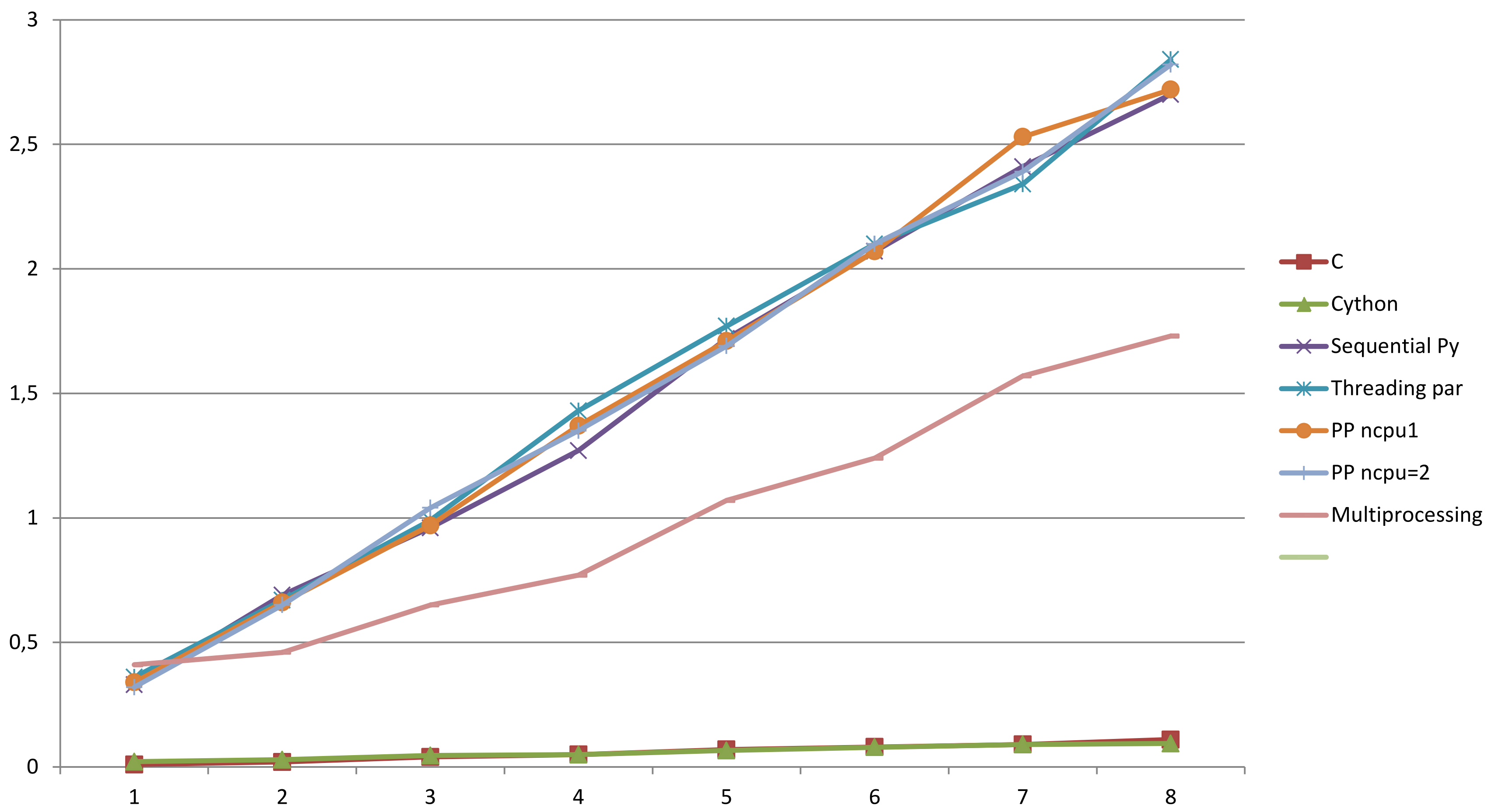
      }
- Already on this level good coding is essential !

Coding

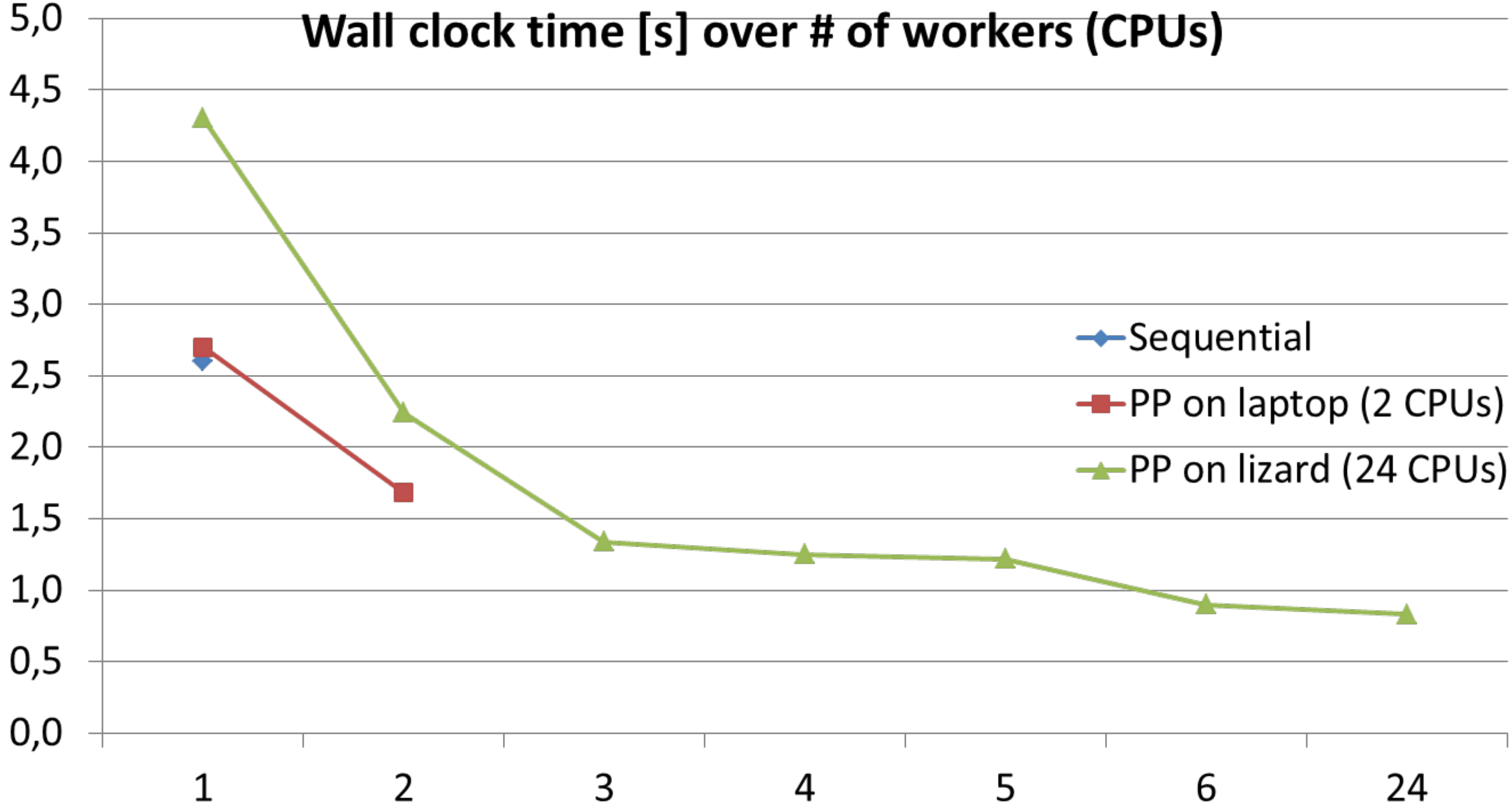
- → https://wiki.zmaw.de/lehre/PythonCourse/PythonLES/Parallel_Programming

- Tests done
  - on VM of my Windows laptop
    - 2 CPUs (hyperthreaded)
  - On lizard cluster
    - 24 CPUs available
    - Tests in progress …

Performance of different methods on 2 CPU laptop (8 jobs)



Performance (8 processes) on multi-CPU systems … to continue …



Future trends

- Very-multi-core is obvious
- Cloud based systems getting easier
- CUDA-like APU systems are inevitable
- disco looks interesting, also blaze
- Celery, R3 are alternatives
- numpush for local & remote numpy
- Auto parallelise numpy code?

Discussion

- Parallelisation
  - needs a lot of analysis, design and evaluation work
  - difficult to find the appropriate approach for your specific application and available resources
  - Leads sometimes not to really better performance
  - But sometimes to other benefits and insights