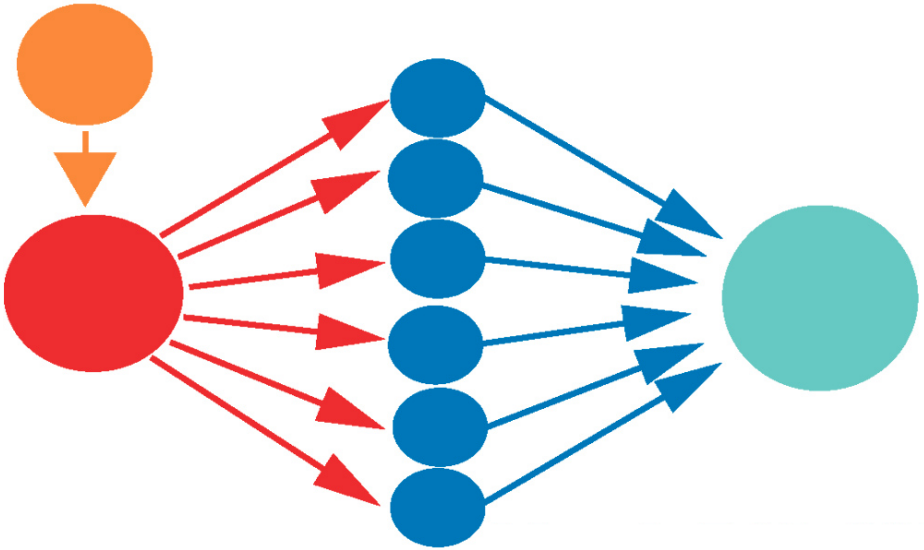
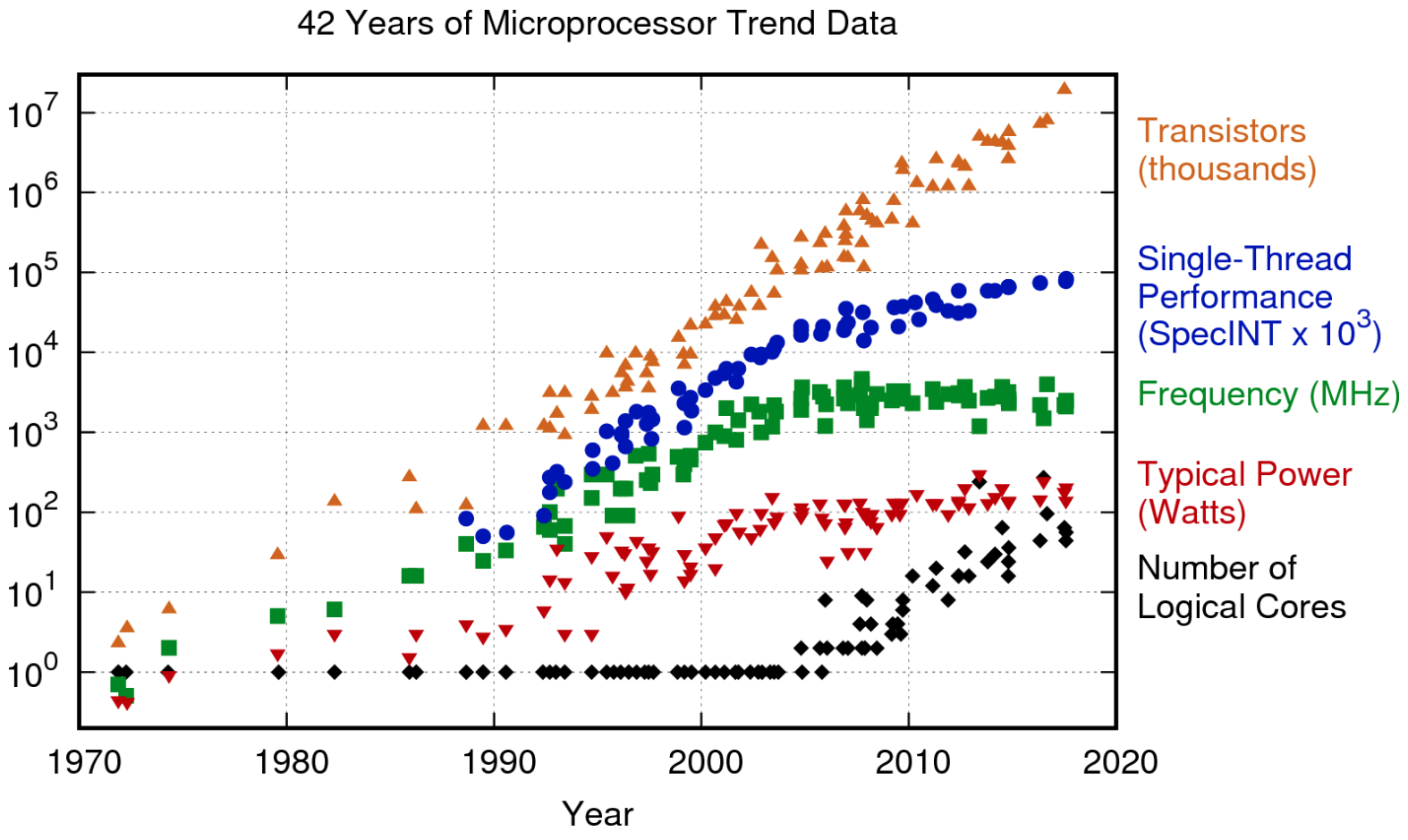


11: multiprocessing in python

https://github.com/matthiaskoenig/itbtechtalks

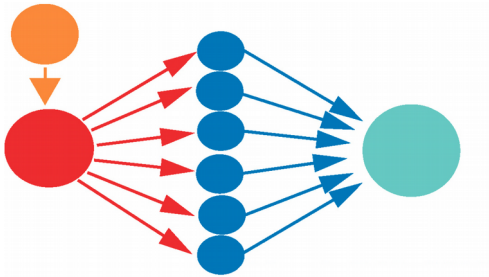
Dr Matthias König
Humboldt University Berlin,
Institute for Theoretical Biology

The future is parallel



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

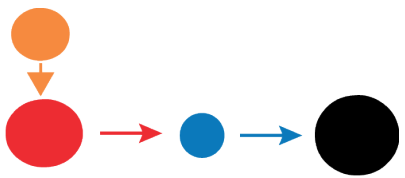
Amdahls law

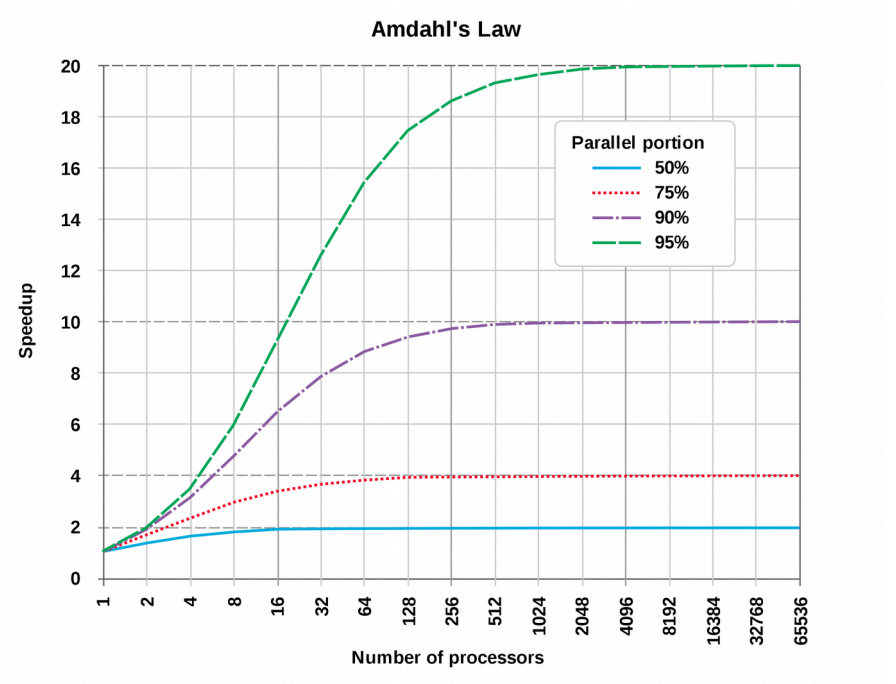


- Formula which gives the theoretical speedup of the execution of a task at fixed workload with increasing resources
- Speedup is limited by serial part of program

  max_speedup = 1/(1-p)

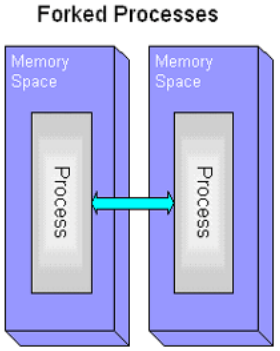
- **Most programs and programming languages don't take advantage of multiple cores**
  - R, Python, Stata, SAS (by default only use 1 core)





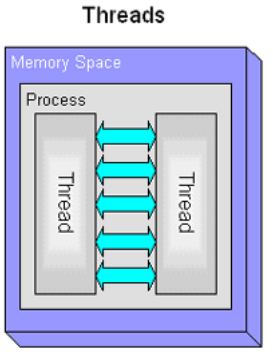
For example, if a program needs 20 hours using a single processor core, and a particular part of the program which takes one hour to execute cannot be parallelized, while the remaining 19 hours (p = 0.95) of execution time can be parallelized, then regardless of how many processors are devoted to a parallelized execution of this program, the minimum execution time cannot be less than that critical one hour.

Processes



vs.

Threads



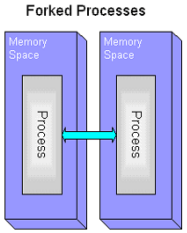
A **process** is an **instance of program**

- **Created by the operating system** to run programs
- Processes **can have multiple threads**; spawning threads (sub-processes) to handle subtasks
- **Sharing information** between processes is **slower** than sharing between threads as processes do not share memory space. In python they share information by pickling data structures like arrays which requires IO time.

Threads are like **mini-processes that live inside a process**

- **Share memory space** and efficiently read and write to the same variables
- **Two threads cannot execute code simultaneously** in one python program

Process



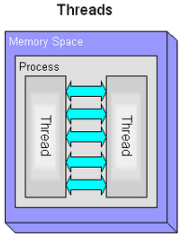
Pros

- **Separate memory space**
- **Takes advantage of multiple CPUs & cores** (Two processes can execute code simultaneously in the same python program)
- Eliminates most needs for synchronization primitives unless if you use shared memory (instead, it's more of a communication model for IPC)
- Code is usually straightforward
- Child processes are interruptible/killable

Cons

- **Processes have more overhead than threads** (opening and closing processes takes more time)
- **Inter-process communication more complicated** (communication model vs. shared memory/objects);
- **Larger memory footprint**

Threads



Pros

- **Lightweight** (low memory footprint)
- **Shared memory** (makes access to state from another context easier)
- Great option for I/O-bound applications
- Allows you to easily make responsive UIs

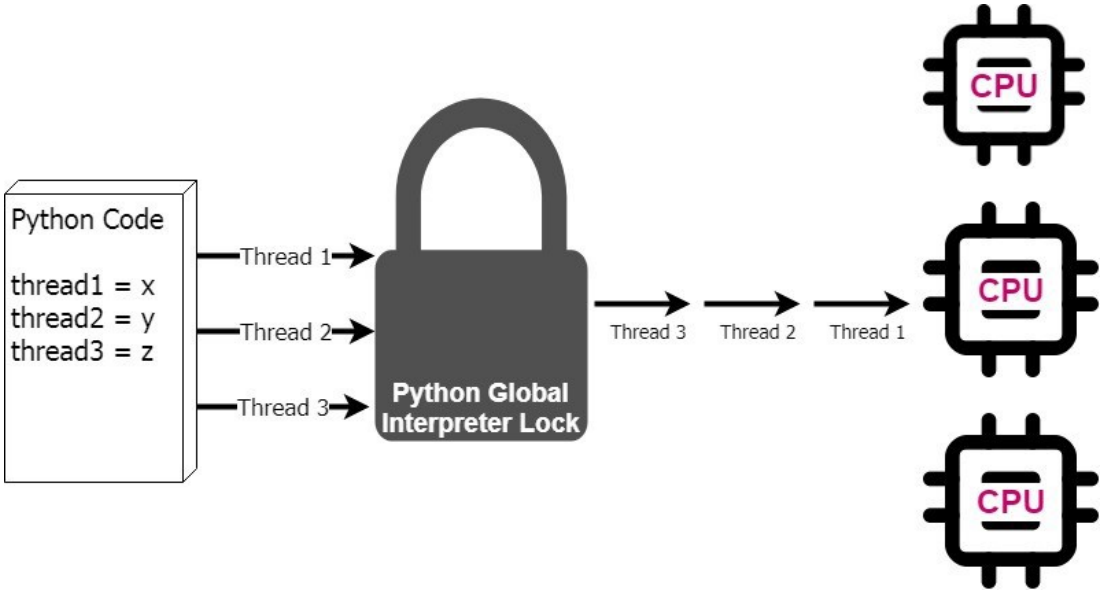
Cons

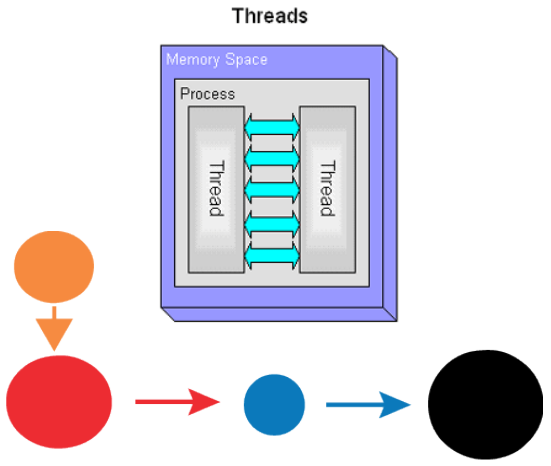
- cPython - subject to the GIL
- Not interruptible/killable
- If not following a command queue/message pump model (using the Queue module), then **manual use of synchronization primitives** become a necessity (decisions are needed for the granularity of locking)
- Code is usually harder to understand and to get right - the potential for **race conditions** increases dramatically

Threads in python

`threading` module

- **global interpreter lock (GIL)**: In Python, single-CPU use is caused by the GIL, which allows only one thread to carry the Python interpreter at any given time. As a result, Python is limited to using a single processor.
- **Threads are best for IO tasks** or tasks involving external systems because threads can combine their work more efficiently.
- Threads provide **no benefit** in python **for CPU intensive tasks** because of the GIL.

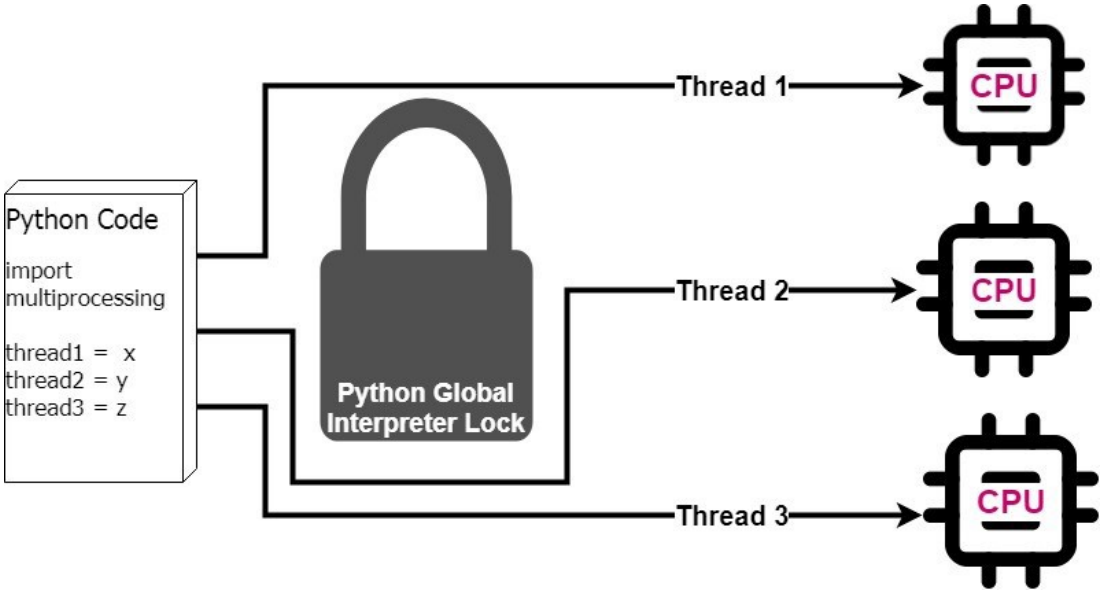


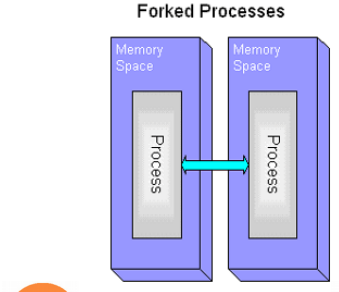


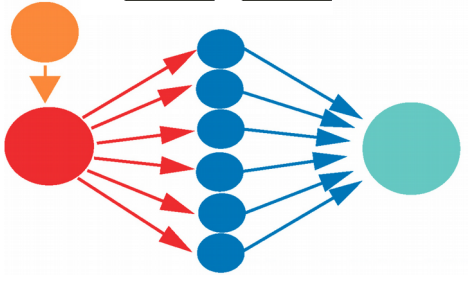
Processes in python

`multiprocessing` module

- package that supports **spawning processes**, thereby sending code to multiple processors for simultaneous execution
- **offers both local and remote concurrency**, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads
- allows to **fully leverage multiple cores** on a given machine
- bypassing the GIL when executing Python code allows the code to run faster







References

- https://medium.com/@bfortuner/python-multithreading-vs-multiprocessing-73072ce5600b
- https://stackoverflow.com/questions/3044580/multiprocessing-vs-threading-python
- https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing
- https://en.wikipedia.org/wiki/Amdahl%27s_law#/media/File:AmdahlsLaw.svg
- https://keyboardinterrupt.org/multiprocessing-using-python-3-7/
- https://medium.com/@urban_institute/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba
- https://docs.python.org/3.5/library/multiprocessing.html
- https://pymotw.com/3/multiprocessing/basics.html
- https://www.blog.pythonlibrary.org/2016/08/02/python-201-a-multiprocessing-tutorial/