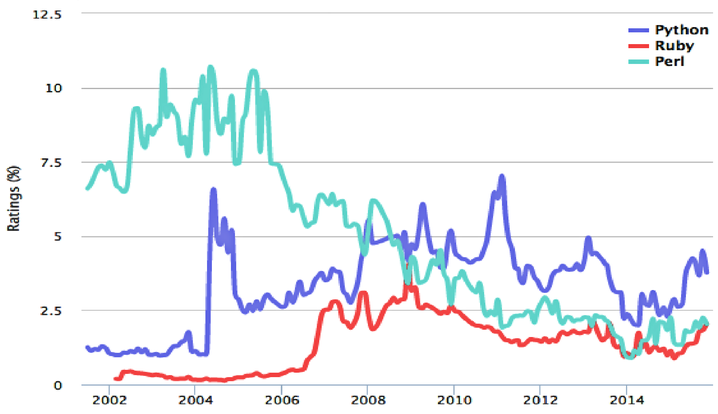
Python code parallelization for ALMA, challenges and alternatives

J. Gonzalez[1], J. Taylor[1], S. Castro[1], J. Kern[2], J. Knudstrup[1], S. Zampieri[1], A. Manning[1], S. Bhatnagar[2], L. Davis[2], K. Golap[2], J. Jacobs[2], T. Nakazato[3], D. Petry[1], M. Pokorny[2], U. Rao[2], M. Rawlings[2], J. Robnett[2], D. Schiebel[2], K. Sugimoto[3], T. Tsutsumi[2], A.Wells[2], S. Williams[4]

The role of Python in the ALMA project

Python is together with C++ and Java one of the standard languages selected for ALMA SW development. At the time when this decision was made (2002) Python was gaining popularity and nowadays is a well stablished scripting language (see picture below):



TIOBE Programming Community Index

The above figure shows the TIOBE Programming Community Index (see [2]) which is built taking into account the number of qualified engineers, courses, and third party vendors using Python.

The fact that Python is one of the most popular scripting languages implies a solid user community, and a wide range of already available packages, fostering rapid and efficient development of high level applications.

As a matter of fact, Python was selected as the scripting language for ALMA due to its suitability to meet the needs of the typically demanding development cycles of high level pipelines, implemented by astronomers [1].

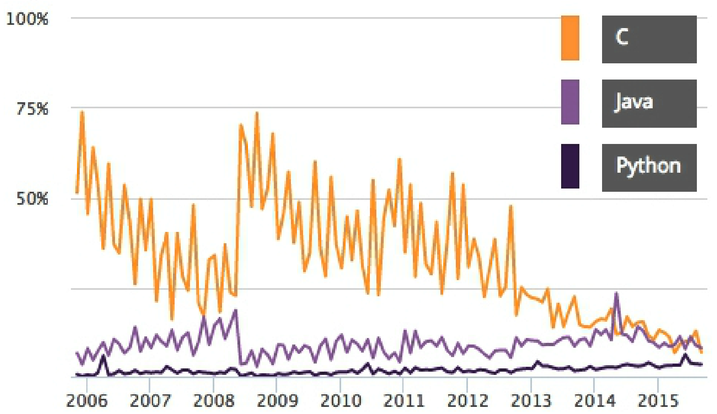
NG/AMS, a Python productivity case

NG/AMS (Next Generation/Archive Management System) is the SW component of a feasibility study started in 2001 at ESO, aiming to provide a disk-based archiving system. NG/AMS was first deployed for ESO's La Silla Paranal Observatory (LPO), established as the central ESO Archive since August 2001, and eventually became the Archive solution for ALMA.

At the time being NG/AMS is deployed at the ALMA site, central offices in Chile, and at the regional centers with more than 300TB of capacity at each site, and expected data rate of 80TB/year. It is also deployed for the LPO , with an ingestion capacity of 10-12 TB/month, and it is serving as well for the JVLA (Jansky Very Large Array) with a total capacity of 1.42PB, and growth rate of 400TB/year. NG/AMS has therefore a high relevance from the operational point of view.

NG/AMS was written completely in Python. The experience of developing and supporting a big system application in Python was quite positive, thanks to the precision and simplicity of the syntax, needing comparably fewer lines of code w.r.t other languages [3].

As a matter of fact, it took only two months, April and June 2001, to develop the first prototype [4], although it was necessary to invest a few years to fully test, stabilize and integrate it with the rest of the data flow systems. This productivity is in-line with indicators such, lines of code changed, which is consistently low in comparison with other languages like C or Java, as shown in the figure below.



Monthly Lines of Code Changed (Percent of Total) of Open Source projects analyzed by Open Hub

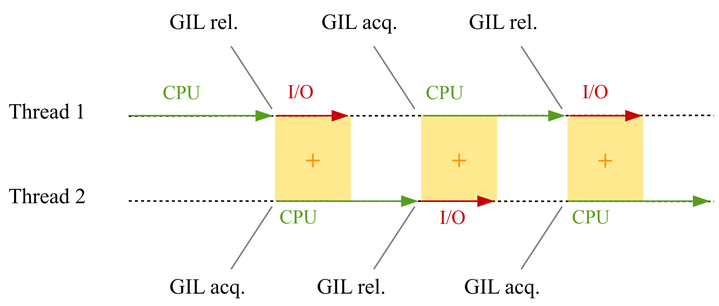
The figure above shows the percentage of lines of code changed for the open source projects analyzed by the Open Hub community (see [5]). More than 680.000 projects are analyzed. In a first approach it indicates a tendency for projects developed in Python to require fewer lines of code changes than lower-level languages like C, thus helping to faster development, more lightweight maintenance, and overall higher productivity.

[1] G.Chiozzi et al., "The ALMA common software: a developer friendly CORBA-based framework", Proc. SPIE Vol.5496-23, Astronomical Telescopes and Instrumentation, Glasgow, June 2004.
[2] Source: www.tiobe.com, TIOBE is specialized in assessing and tracking quality of industrial software
[3] Andreas Wicenec, Jens Knudstrup "ESO's Next Generation Archive System in Full Operation", ESO, The Messenger n129
[4] Jens Knudstrup, "The Next Generation Archive System", www.eso.org/projects/dfs/dfs-shared/web/ngas/ngas-presentation.pps
[5] Data from the Black Duck Open Hub, www.openhub.net

NG/AMS and Python multithreading

NG/AMS is a multithreaded server, with the capability of handling simultaneously various streams of data, whilst maintenance services (like data integrity check) stop/resume in the background during low activity periods.

However, it is a well known-fact that the CPython interpreter which implements the Python language is not thread-safe, and therefore it needs to internally resort to a locking mechanism, named GIL (Global Interpreter Lock) to protect its internal state and objects from concurrent modification. This coarse grained locking effectively serializes execution of code working on Python objects. Threading can therefore only be used to improve multi-core CPU utilization when running non-Python code (I/O or extension-modules written in a different language, e.g. the extension for numerical computing numpy).



CPython GIL in action

As a matter of fact, the performance of NG/AMS is mainly limited by the underlying I/O system and not by Python's GIL. Some computing intensive steps like CRC calculation could benefit from parallelism, but with the current data rates and I/O systems this limit is not hit [6]. Additionally, it is always possible to run several NG/AMS instances on the same server, as long as the I/O system supports it.

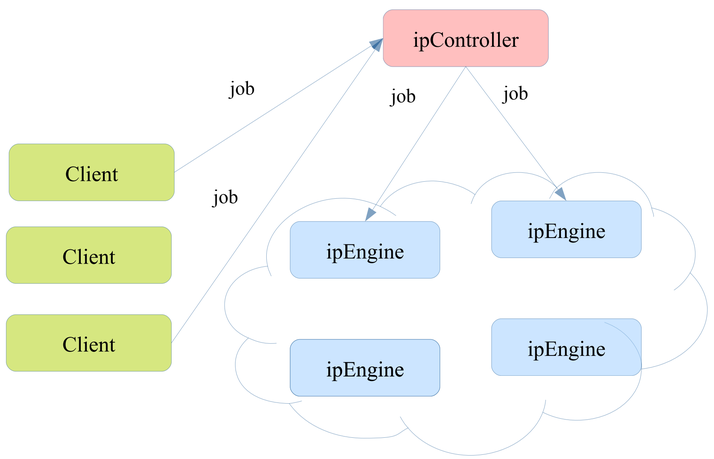
Thus a recommendation for NG/AMS from an operational point of view would be to deploy it on machines with a strong I/O system and a multicore capacity in-line with the number of hosted NG/AMS servers.

Python and IPython multiprocessing

As described in the previous section, using multithreading in a CPU-bound section of Python code cannot yield any improvement, therefore parallelism of Python code can only be achieved by means of multiprocessing.

Starting with version 2.6 Python incorporates a structured multiprocessing module, however it provides only a low-level API, featuring maps and asynchronous actions, therefore it is necessary to resort to external packages/modules for distributed heterogeneous computing.

One example of such a framework is provided by the popular IPython package, which since the very early versions (0.9+) offers a sophisticated and powerful architecture for parallel and distributed computing. Most importantly, IPython enables all types of parallel applications to be developed, executed, debugged and monitored interactively.



IPython MultiEngine Architecture

The IPython architecture consists of three components that live in the IPython.parallel package and are installed with IPython:

- Client: Primary object, for connecting to a cluster
- ipEngine: Python instance that can handle Python commands and incoming/outgoing Python objects over a network connection
- ipController: Collection of processes to which iPython engines and clients can connect, thus providing a single point of contact for users

ALMA, a worldwide collaboration

CASA experience with IPython parallelism

CASA (Common Astronomy Software Applications) is a data reduction package for radio telescopes (both interferometric and single dish data). The CASA infrastructure consists of a set of C++ tools bundled together under an iPython interface as a set of data reduction tasks.

At the time being CASA is the official data reduction package for the JVLA and ALMA radio telescopes, and serves a community of more than 1700 users (based on downloads from [7]). It is therefore a relevant package for science operations.

IPython MultiEngine parallel framework was introduced in CASA and presented in the 2010 CASA developers conference. It offered a quick solution to prototype parallelism in CASA with a neat support for interactivity, however starting with IPython v0.11 (July 31, 2011) the MultiEngine API changed, situation that triggered an internal discussion which materialized in a decision taken in the 2013 CASA developers conference to use MPI as a framework to support parallelism in CASA.

Introducing MPI in CASA

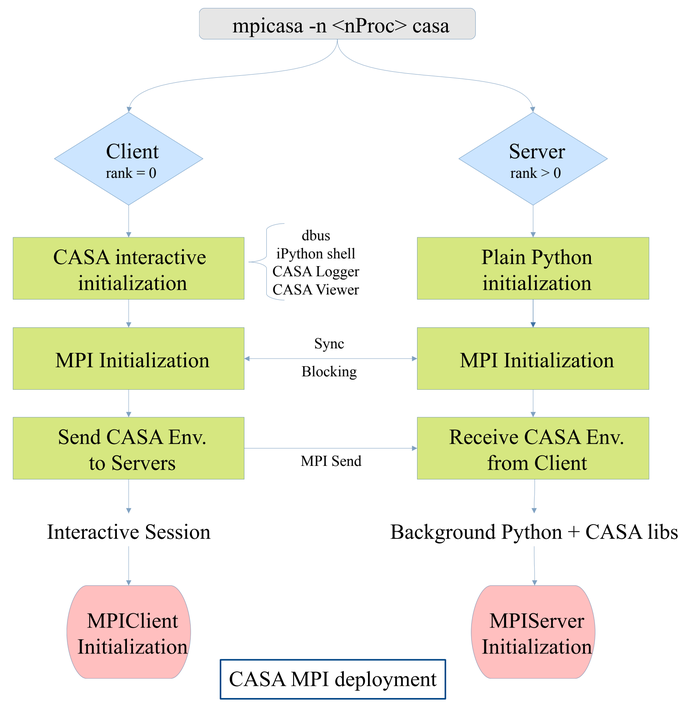
MPI (Message Passing Interface) was introduced in CASA during the 4.3 development cycle (2014). The decision of adopting MPI was based on the facts that it has a stable API since two decades (1994), it has become a standard in cluster computing, and receives sustained founding from US programs (NSF, ARPA) and EU programs (Espirit).

There are several Python bindings for MPI, but two key requirements for CASA are interactivity and capacity to send over MPI an arbitrary Python object. The table below shows what key features are supported by the most popular packages [8].

	pypar	mpi4py	myMPI	pyMPI	Scientific MPI
Pre allocated buffer for send-recv	✓	✓	×	×	✓
Explicit MPI_Initialize	×	✓	✓	×	×
Explicit Communicator	×	✓	✓	×	✓
Interactively parallel run	×	✓	×	✓	✓
Arbitrary Python object	✓	✓	×	✓	×
Latency (micro seconds)	25	14	33	133	23
Bandwidth (Mbytes / seconds)	899	944	364	151	509

As described in the table above mpi4py [9] is not only the most complete package providing MPI for Python, but also the one with smallest latency, therefore it has been adopted as the official Python package to provide MPI in CASA.

However, even though mpi4py is ready for interactive use, it is necessary to add a layer on top of it, so that only one CASA instance interactive interface is exposed to the user, whilst the others stay in the background, awaiting for commands, in a similar way as with the IPython Multiengine interface.



CASA MPI deployment

The diagram above depicts the CASA initialization flow of the processes deployed via MPI:

- Client: A privileged process (rank 0) is established as the client, and serves as main point of interaction with the user, including all the GUIs (logger, image viewer, plotting tool).
- Servers: All the other processes (with rank greater than 0) consist of a plain Python instance which imports the CASA libraries and awaits for commands from the client.

[6] C. Wu , A. Wicenec, D. Pallot and A. Checcucci "Optimising NGAS for the MWA Archive", arXiv:1308.6083, Instrumentation and Methods for Astrophysics (astro-ph.IM)
[7] Official NRAO CASA website http://casa.nrao.edu
[8] Wenjing Lin, "A Comparison of existing Python modules", MSc. Thesis, University of Oslo , August 2010
[9] Lisandro Dalcin, http://pythonhosted.org/mpi4py

1- European Organisation for Astronomical Research in the Southern Hemisphere (ESO)
2- National Radio Astronomy Observatory (NRAO)
3- National Astronomical Observatory of Japan (NAOJ)
4- Royal Observatory Edinburgh (ROE)