Bypassing the Conventional Software Stack Using Adaptable Runtime Systems

Simon Andreas Frimann Lund, Mads R. B. Kristensen,
Brian Vinter, and Dimitrios Katsaros

Niels Bohr Institute, University of Copenhagen, Denmark
{safl,madsbk,vinter}@nbi.dk
Computer Science Department, University of Copenhagen, Denmark
rth738@alumni.ku.dk

Abstract. High-level languages such as Python offer convenient language constructs and abstractions for readability and productivity. Such features and Python's ability to serve as a steering language as well as a self-contained language for scientific computations has made Python a viable choice for high-performance computing. However, the Python interpreter's reliance on shared objects and dynamic loading causes scalability issues that at large-scale consumes hours of wall-clock time just for loading the interpreter.

The work in this paper explores an approach to bypass the conventional software stack, by replacing the Python interpreter on compute nodes with an adaptable runtime system capable of executing the compute intensive portions of a Python program. Allowing for a single instance of the Python interpreter, interpreting the users' program and additionally moving program interpretation off the compute nodes. Thereby avoiding the scalability issue of the interpreter as well as providing a means of running Python programs on restrictive compute notes which are otherwise unable to run Python.

The approach is experimentally evaluated through a prototype implementation of an extension to the Bohrium runtime system. The evaluation shows promising results as well as identifying issues for future work to address.

Keywords: Scalability, Python, import problem, dynamic loading.

1 Introduction

Python is a high-level, general-purpose, interpreted language. Python advocates high-level abstractions and convenient language constructs for readability and productivity. The reference implementation of the Python interpreter, CPython, provides rich means for extending Python with modules implemented in lower-level languages such as C and C++. Lower-level implementations can be written from scratch and conveniently map to Python data-structures through Cython[4], function wrappers to existing libraries through SWIG[3,2], or using the Python ctypes[1] interface.

[1] http://docs.python.org/2/library/ctypes.html

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part II, LNCS 8806, pp. 302–313, 2014.

The features of the language itself and its extensibility make it attractive as a steering language for scientific computing, which the existence of Python at high-performance compute sites confirms. Furthermore, there exists a broad range of Python wrappers to existing scientific libraries and solvers[11,20,13,8,9].

Python transcends its utilization as a steering language. SciPy[2] and its accompanying software stack[17,18,12] provides a powerful environment for developing scientific applications. The fundamental building block of SciPy is the multidimensional arrays provided by NumPy[17]. NumPy expands Python by providing a means of doing array-oriented programming using array-notation with slicing and whole-array operations. The array-abstractions offered by NumPy provides the basis for a wealth of existing[6] and emerging[19,21,14] approaches that increases the applicability of Python in an HPC environment. Even though advances are made within these areas, a problem commonly referred to as the the *import problem*[1,15,22] still persists at large-scale compute sites. The problem evolves around dynamic loading of CPython itself, built-in modules, and third party modules. Recent numbers reported on Hopper[22] state linear scale with the number of cores, which amount to a startup time of 400 seconds on 1024 cores and one hour for 8000 cores.

The approach in this paper explores a simple idea to avoid such expensive startup costs: execute one instance of the Python interpreter regardless of the cluster size. Furthermore, we allow the Python interpreter to run on an external machine that might not be part of the cluster. The machine can be any one of; the user's own laptop/workstation, a frontend/compile node, or a compute node, e.g. any machine that is accessible from the compute-site.

A positive complementary effect, as well as a goal in itself, is that the Python interpreter and the associated software stack need not be available on the compute nodes.

The work in this paper experimentally evaluates the feasibility of bypassing the conventional software stack, by replacing the Python interpreter on the compute nodes with an adaptable runtime system capable of executing the computationally heavy part of the users' program. The approach facilitates the use of Python at restrictive compute-sites and thereby broadens application of Python in HPC.

2 Related Work

The work within this paper describes, to the authors knowledge, a novel approach for handling the Python *import problem*. This section describes other approaches to meeting the same end.

Python itself support a means for doing a user-level override of the import mechanism[3] and work from within the Python community has improved upon the import system from version 2.6 to 2.7 and 3.0. In spite of these efforts, the problem persists.

[2] `http://www.scipy.org/stackspec.html`

[3] `http://legacy.python.org/dev/peps/pep-0302/`

One aspect of the *import problem* is the excessive stress on the IO-system caused by the object-loader traversing the filesystem looking for Python modules. Path caching through collective operations is one approach to lowering overhead. The mpi4py[7] project implements multiple techniques to path caching where a single node traverses the file-system and broadcasts the information to the remaining $N-1$ nodes. The results of this approach show significant improvements to startup times from hours to minutes but relies on the *mpi4py* library and requires maintenance of the Python software-stack on the compute-nodes.

Scalable Python[4], first described in[9], addresses the problem at a lower level. Scalable Python, a modification of CPython, seeks to address the *import problem* by implementing a parallel IO layer utilized by all Python import statements. By doing so only a single process, in contrast to N processes, perform IO. The result of the IO operation is broadcast to the remaining $N-1$ nodes via MPI. The results reported in[9] show significant improvements towards the time consumed by Python import statements at the expense of maintaining a custom CPython implementation.

Relying on dynamically loaded shared objects is a general challenge for large-scale compute-sites with a shared filesystem. SPINDLE[10] provides a generic approach to the problem through an extension to the GNU Loader.

The above described approaches apply different techniques for improving performance of dynamic loading. A different strategy which in this respect is thematically closer to the work within this paper is to reduce the use of dynamic loading. The work in[15] investigate such strategy by replacing as much dynamic loading with statically compiled libraries. Such technique in a Python context can by applied through the use of Python freeze[5] and additional tools[6] exists to support it.

3 The Approach

The previous sections describe and identify the CPython import system as the culprit guilty of limiting the use of Python / NumPy at large-scale compute sites. Dynamic loading and excessive path searching are accomplices to the havoc raised. The crime committed is labelled as the Python *import problem*.

Related work let the culprit run free and implement techniques to handling the havoc raised. The work within this paper focuses on restricting the culprit and thereby preventively avoiding the problem.

The idea is to run a single instance of the Python interpreter, thereby keeping the overhead constant and manageable. The remaining instances of the interpreter are replaced with a runtime system capable of efficiently executing the portion of the Python / NumPy program responsible for communication and computation. Leaving the task of interpreting the Python / NumPy program,

[4] `https://gitorious.org/scalable-python`
[5] `https://wiki.python.org/moin/Freeze`
[6] `https://github.com/bfroehle/slither`

conditionals, and general program flow up to the interpreter. The computationally heavy parts are delegated to execution on the compute nodes through the runtime system.

3.1 Runtime System

The runtime used in this work is part of the Bohrium[19] project[7]. The Bohrium runtime system (BRS) provides a backend for mapping array operations onto a number of different hardware targets, from multi-core systems to clusters and GPU enabled systems. It is implemented as a virtual machine capable of making runtime decisions instead of a statically compiled library. Any programming language can use BRS in principle; in this paper though, we will use the Python / NumPy support exclusively.

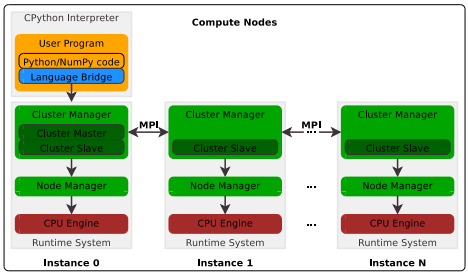

Fig. 1. Illustration of communication between the runtime system components *without* the use of the proxy component

The fundamental building block of BRS is the representation of programs in the form of *vector bytecode*. A vector bytecode is a representation of an operation acting upon an array. This can be one of the standard built-in operations such as element-wise addition of arrays, function promotion of trigonometric functions over all elements of an array, or in functional terms: map, zip, scan and reduction, or an operation defined by third party.

BRS is implemented using a layered architecture featuring a set of interchangeable *components*. Three different types of components exist: *filters*, *managers*, and *engines*. Figure 1 illustrates a configuration of the runtime system configured for execution in a cluster of homogenous nodes. The arrows represent vector bytecode sent through the runtime system in a top-down fashion, possibly altering it on its way.

Each component exposes the same C-interface for initialization, shutdown, and execution thus basic component interaction consists of regular function calls. The component interface ensures isolation between the language bridge that runs the CPython interpreter and the rest of Bohrium. Thus, BRS only runs a single instance of the CPython interpreter no matter the underlying architecture – distributed or otherwise.

Above the runtime, a language bridge is responsible for mapping language constructs to vector bytecode and passing it to the runtime system via the C-interface.

Managers manage a specific memory address space within the runtime system and decide where to execute the vector bytecode. In figure 1 a node manager manages the local address space (one compute-node) and a cluster-manager

[7] `http://www.bh107.org`

which handles data distribution and inter-node communication through MPI. At the bottom of the runtime system, we have the execution engines, which are responsible for providing efficient mapping of array operations down to a specific processing unit such as a CPU or a GPU.

3.2 Proxy Manager

Currently, all Bohrium components communicate using local function calls, which translates into shared memory communication. Figure 1 illustrates the means of communication within the BRS prior to the addition of the proxy component. As a result, the language bridge, which runs a CPython interpreter, must execute on one of the cluster-nodes. In order to circumvent this problem, we introduce a new *proxy* component.

This new component acts as a network proxy that enables Bohrium components to exchange vector bytecode across a network. Figure 2 illustrates the means for communication which the Proxy component provides. By using this functionality, separation can be achieved between the implementation of any application using Bohrium and the actual hardware on which it runs. This is an important property when considering cases of supercomputers or clusters, which define specific characteristics for the execution of tasks on them.

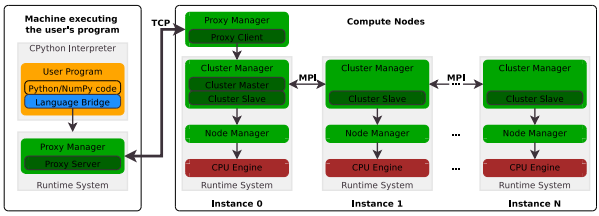

Fig. 2. Illustration of communication between the runtime system components *with* the use of the proxy component

The proxy component is composed of two parts – a server and a client. The server exposes the component interface (init, execute, and shutdown) to its parent component in the hierarchy whereas the client uses its child component interface. When the parent component calls execute with a list of vector bytecodes, the server serialize and sends the vector bytecodes to the client, which in turn uses its child component interface to push the vector bytecodes further down the Bohrium hierarchy. Besides the serialized list of vector bytecodes, the proxy component needs to communicate array-data in two cases.

When the CPython interpreter introduces existing NumPy arrays and Python scalars to a Bohrium execution. Typically, this happens when the user application loads arrays and scalars initially. When the CPython interpreter access the result of a Bohrium execution directly. Typically, this happens when the user application evaluates a loop-condition based on some array and scalar data.

Both the server and the client maintain a record of array-data locations thus avoiding unnecessary array-data transfers. Only when the array-data is involved in a calculation at the client-side will the server send the array-data. Similarly, only when the CPython interpreter request the array-data will the client send the array-data to the server.

In practice, when the client sends array-data to the server it is because the CPython interpreter needs to evaluate a scalar value before continuing. In this case, the performance is very latency sensitive since the CPython interpreter is blocking on the scalar value. Therefore, it is crucial to disable Nagle's TCP/IP algorithm[16] in order achieve good performance. Additionally, the size of the vector bytecode lists is significantly greater than the TCP packet header thus limiting the possible advantage of Nagle's TCP/IP algorithm. Therefore, when the proxy component initiates the TCP connection between server and client it sets the `TCP_NODELAY` socket option.

4 Evaluation

The basic idea of the approach is to have a single instance of CPython interpreting the user's program, such as figure 2 illustrates. With a single isolated instance of the interpreter the *import problem* is solved by design. The second goal of the approach is to facilitate execution of a Python program in a restricted environment where the Python software stack is not available on the compute nodes.

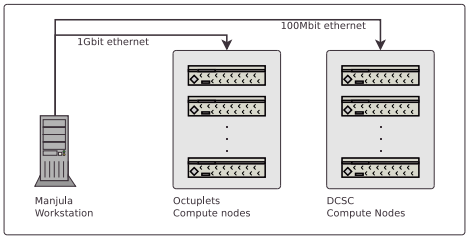

Fig. 3. Octuplets and DCSC two physically and administratively disjoint clusters of eight and sixteen nodes. Octuplets is a small-scale research-cluster managed by the eScience group at the Niels Bohr Institute. DCSC is a larger compute-site for scientific computation in Denmark. Gbit ethernet facilitate the connection between Manjula and the octuplet cluster and 100Mbit ethernet between Manjula and DCSC.

The potential *Achilles heel* of the approach is in its singularity, with a single *remote* instance of the interpreter network latency and bandwidth limitations potentially limit application of the approach.

Network latency can stall execution of programs when the round-trip-time of transmitting vector bytecode from the interpreter-machine to the compute node exceeds the time spent computing on previously received vector bytecode. Bandwidth becomes a limiting factor when the interpreted program needs large amounts of data for evaluation to proceed interpretation and transmission of vector bytecode. The listing below contains descriptions of the applications used as well as their need for communication between interpreter and runtime. The sourcecode is available for closer inspection in the Bohrium repository[8].

Black Scholes implements a financial pricing model using a partial differential equation, calculating price variations over time[5]. At each time-step the interpreter reads out a scalar value from the runtime representing the computed price at that time.

[8] `http://bitbucket.org/bohrium/bohrium`

Heat Equation simulates the heat transfer on a surface represented by a two-dimensional grid, implemented using jacobi-iteration with numerical convergence. The interpreter requires a scalar value from the runtime at each time-step to evaluate whether or not simulation should continue. Additionally when executed with visualization the entire grid is required.

N-Body simulates interaction of bodies according to the laws of Newtonian physics. We use a straightforward algorithm that computes all body-body interactions, $O(n^2)$, with collisions detection. The interpreter only needs data from the runtime at the end of the simulation to retrieve the final position of the bodies. However, the interpreter will at each time-step, when executed for visualization purposes, request coordinates of the bodies.

Shallow Water simulates a system governed by the Shallow Water equations. The simulation initates by placing a drop of water in a still container. The simulation then proceeds, in discrete time-steps, simulating the water movement. The implementation is a port of the MATLAB application by Burkardt [9]. The interpreter needs no data from the runtime to progress the simulation at each time-step. However, the interpreter will at each time-step, when executed for visualization purposes, request the current state of the simulated water.

We benchmark the above applications on two Linux-based clusters (Fig. 3). The following subsections describe the experiments performed and report the performance numbers.

4.1 Proxy Overhead

We begin with figure 4 which show the results of running the four benchmark applications on the octuplet cluster using eight compute nodes and two different configurations:

With Proxy The BRS configured with the proxy component and the interpreter is running on Manjula. This configuration is equivalent to the one illustrated in figure 2.

Without Proxy The BRS configured without the proxy component. The interpreter is running on the first of the eight compute nodes. This setup is equivalent to the one illustrated in figure 1.

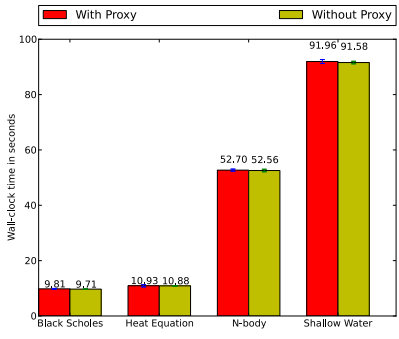

Fig. 4. Elapsed wall-clock time in seconds of the four applications on the octuplet compute nodes with and without the proxy component

We cannot run Python on the DCSC cluster for the simple reason that the software stack is too old to compile Python 2.6 on the DCSC compute nodes.

[9] `http://people.sc.fsu.edu/~jburkardt/m_src/shallow_water_2d/`

Thus, it is not possible to provide comparable results of running with and without the Proxy component.

The purpose of this experiment is to evaluate the overhead of introducing the proxy component in a well-behaved environment. There were no other users of the network, filesystem, or machines. Round-trip-time between Manjula and the first compute node was average at $0.07ms$ during the experiment. The error bars show two standard deviations from the mean. The overhead of adding the proxy component is within the margin of error and thereby unmeasurable.

4.2 Latency Sensitivity

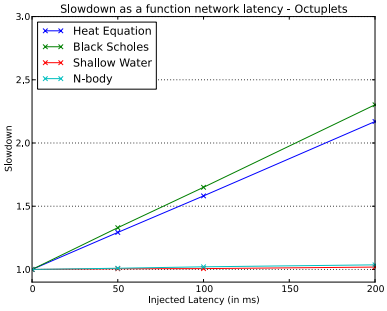

Fig. 5. Slowdown of the four applications as a function of injected latency between Manjula and octuplet compute node

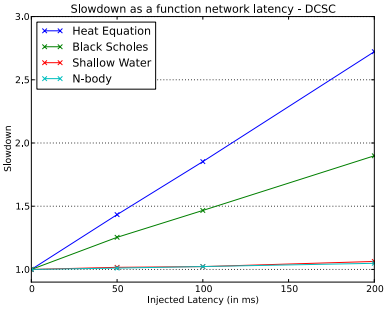

Fig. 6. Slowdown of the four applications as a function of injected latency between Manjula and DCSC compute node.

We continue with figures 5 and 6. The BRS configured with the proxy component, running the interpreter on Manjula. Figure 2 illustrates the setup. The purpose of the experiment is to evaluate the approach' sensitivity to network latency. Latencies of 50, 100, 150, and 200ms are injected between Manjula and the compute node running the proxy client. The figures show slowdown of the applications as a function of the injected latency.

The applications Shallow Water and N-body are nearly unmeasurably affected by the injected latency. The observed behavior is as expected since the interpreter does not need any data to progress interpretation. It is thereby possible to overlap transmission of vector bytecode from the interpreter-machine with computation on the compute nodes.

The injected latency does, however, affect the applications Heat Equation and Black Scholes. The observed behavior is as expected since the interpreter requires a scalar value for determining convergence criteria for Heat Equation and sampling the pricing value for Black Scholes. Network latency affects the results from the DCSC cluster the most, with a worst-case of a 2.8 slowdown. This is due to the elapsed time being lower when using the sixteen DCSC compute nodes. Since less time is spent computing more time is spent waiting and thereby a relatively larger sensitivity to network latency.

4.3 Bandwidth Sensitivity

The last experiment sought to evaluate the sensitivity to high network bandwidth utilization. Figures 7 and 8 show the results of an experiment where the four applications were running with visualization updated at each time-step. The BRS configured with the proxy component; Manjula is running the Python interpreter. Figure 2 illustrates the setup.

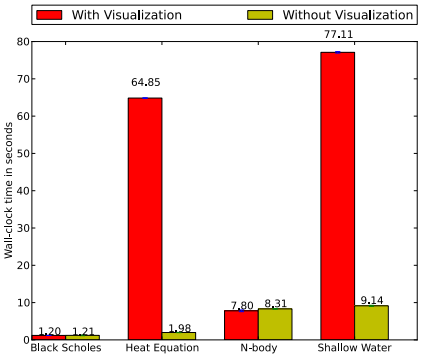

Fig. 7. Elapsed wall-clock time of the four applications with and without visualization on the octuplet compute nodes.

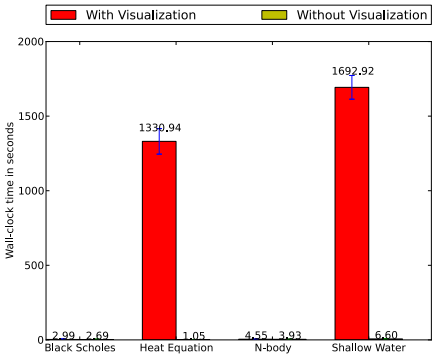

Fig. 8. Elapsed wall-clock time of the four applications with and without visualization on the DCSC compute nodes.

When executing with visualization, the interpreter requires a varying (depending on the application) amount of data to be transmitted from the compute nodes to the interpreter-machine at each time step. Thereby straining the available bandwidth between the interpreter-machine and the compute node running the proxy-client.

Black-Scholes although sensitive to latency due to the need of transmitting the computed price at each time-step, does not require any significant amount of data to be transferred for visualization, neither does the N-Body simulation. However, the two other applications Heat Equation and Shallow Water require transmission of the entire state to visualize dissipation of heat on the plane and the current movement of water. These two applications are sufficient to observe a key concern of the approach.

We observe a slowdown of about $\times 1260$ (Heat Equation) and $\times 257$ (Shallow Water) when running on the DCSC nodes. We observe a slowdown of about $\times 32.8$ (Heat Equation) and $\times 8.5$ (Shallow Water) when running in the octuplet nodes. These results clearly show that network bandwidth becomes a bottleneck, with disastrous consequences in terms of execution time and thus a limiting factor for applying the approach for such use.

The slowdown is much worse when running on the DCSC compute nodes compare to the slowdown on the octuplet nodes. This is due to the interconnect

being 100Mbit ethernet to the DCSC in relation to the 1Gbit ethernet connection to the octuplet nodes.

5 Future Work

The evaluation revealed bandwidth bottlenecks when the machine running the interpreter requests data for purposes such as visualization. The setup in the evaluation was synthetic and forced requests of the entire data-set at each time-step without any transformation of the data, it can, therefore, be regarded as a worst-case scenario.

One could argue that the setup is not representative for user behaviour and instead assume that the user would only need a snapshot of data at every $timestep/K$ iteration and with lowered resolution such as every I'th datapoint and thus drastically lowering the bottleneck. However, to address the issue future work will involve compressed encoding of data transmitted as well as suitable downsampling for the visualization purpose.

The primary focus point for future work is now in progress and relates to the effective throughput at each compute-node. The current implementation of the execution engine uses a virtual-machine approach for executing array operations. In this approach the virtual machine delegate execution of each vector bytecode to statically compiled routine. Within this area, a wealth of optimizations are applicable by composing multiple operations on the same data and hereby *fusing* array operations together.

Random-number generators, linear spaces of data, and iotas, when combined with reductions are another common source for optimization of memory utilization and locality. Obtaining such optimizations within the runtime require the use of JIT compilation techniques and potentially increase the use dynamic loading of optimized codes. The challenge for this part of future work involves exploration of how to get such optimization without losing the performance gained to runtime and JIT compilation overhead.

6 Conclusions

The work in this paper explores the feasibility of replacing the Python interpreter with an adaptable runtime system, with the purpose of avoiding the CPython scalability issues and providing a means of executing Python programs on restrictive compute nodes which are otherwise unable to run the Python interpreter.

The proxy component, implemented as an extension to the Bohrium runtime system (BRS), provides the means for the BRS to communicate with a single *remote* instance of the Python interpreter. The prototype implementation enabled evaluation of the proposed approach of the paper.

Allowing the interpreter to execute on any machine, possibly users' own workstations/laptops, leverages a Python user to utilize a cluster of compute nodes or a supercomputer with direct realtime interaction. However, it also introduces concerns with regards to the effect of network latency and available bandwidth,

between the machine running the interpreter and the compute node running the proxy client, on program execution. These concerns were the themes for the conducted evaluation.

Results showed that the overhead of adding the proxy component and thereby the ability for the BRS to use a remote interpreter was not measurable in terms of elapsed wall-clock time, as results were within two standard deviations of the measured elapsed wall-clock. The results additionally showed a reasonable tolerance to high network latency, at $50ms$ round-trip-time, slowdown ranged from not being measurable to $\times 1.3 - \times 1.4$. In the extreme case of $200ms$ latency ranged from not being measurable to a slowdown of $\times 1.9 - \times 2.8$.

The primary concern, and focus for future work, presented itself during evaluation of bandwidth requirements. If the Python program requests large amounts of data then the network throughput capability becomes a bottleneck, severely impacting elapsed wall-clock as well as saturating the network link, potentially disrupting other users.

The results show that the approach explored within this paper does provide a possible means to avoid the scalability issues of CPython, allowing direct user interaction and enabling execution of Python programs in restricted environments that are otherwise unable to run interpreted Python programs. The approach is, however, restricted to transmission of data such as vector bytecode, scalars for evaluation of convergence criteria, boolean values, and low-volume data-sets between the interpreter-machine and runtime. This does, however, not restrict processing of large-volume datasets within the runtime on and between the compute nodes.

Acknowledgments. This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

References

1. Ahmadia, A.: Solving the import problem: Scalable Dynamic Loading Network File Systems. Technical report, Talk at SciPy conference, Austin, Texas (July 2012), `http://pyvideo.org/video/1201/solving-the-import-problem-scalable-dynamic-load`
2. Beazley, D.M.: Automated scientific software scripting with SWIG, vol. 19, pp. 599–609. Elsevier (2003)
3. Beazley, D.M., et al.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the 4th USENIX Tcl/Tk workshop, pp. 129–139 (1996)
4. Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D.S., Smith, K.: Cython: The best of both worlds. Computing in Science & Engineering 13(2), 31–39 (2011)
5. Black, F., Scholes, M.: The pricing of options and corporate liabilities. The Journal of Political Economy, 637–654 (1973)
6. Daily, J., Lewis, R.R.: Using the global arrays toolkit to reimplement numpy for distributed computation. In: Proceedings of the 10th Python in Science Conference (2011)

7. Dalcin, L., Paz, R., Storti, M., Elia, J.D.: MPI for Python: Performance improvements and MPI-2 extensions. Journal of Parallel and Distributed Computing 68(5), 655–662 (2008)
8. Drummond, L.A., Galiano, V., Migallón, V., Penadés, J.: High-level user interfaces for the DOE ACTS collection. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 251–259. Springer, Heidelberg (2007)
9. Enkovaaraa, J., Louhivuoria, M., Jovanovicb, P., Slavnicb, V., Rännarc, M.: Optimizing GPAW. Partnership for Advanced Computing in Europe (September 2012), http://www.prace-ri.eu/IMG/pdf/Optimizing_GPAW.pdf
10. Frings, W., Ahn, D.H., LeGendre, M., Gamblin, T., de Supinski, B.R., Wolf, F.: Massively Parallel Loading. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013, pp. 389–398. ACM, New York (2013)
11. Gawande, K., Webers, C.: PyPETSc User Manual (Revision 1.0). Technical report, NICTA (2009), http://elefant.developer.nicta.com.au/documentation/userguide/PyPetscManual.pdf
12. Hunter, J.D.: Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering 9(3), 90–95 (2007)
13. Ketcheson, D.I., Mandli, K.T., Ahmadia, A.J., Alghamdi, A., Quezada de Luna, M., Parsani, M., Knepley, M.G., Emmett, M.: PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems. SIAM Journal on Scientific Computing 34(4), C210–C231 (2012)
14. Kristensen, M.R.B., Vinter, B.: Numerical Python for scalable architectures. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, PGAS 2010, pp. 15:1–15:9. ACM, New York (2010)
15. Marion, P., Ahmadia, A., Froehle, B.M.: Import without a filesystem: scientific Python built-in with static linking and frozen modules. Technical report, Talk at SciPy conference, Austin, Texas, July 2012 (2013), https://www.youtube.com/watch?v=EOiEIWMYkwE
16. Nagle, J.: Congestion Control in IP/TCP Internetworks. RFC 896 (January 1984)
17. Oliphant, T.E.: Python for Scientific Computing. Computing in Science & Engineering 9(3), 10–20 (2007)
18. Pérez, F., Granger, B.E.: IPython: A System for Interactive Scientific Computing. Computing in Science & Engineering 9(3), 21–29 (2007)
19. Kristensen, M.R.B., Lund, S.A.F., Blum, T., Skovhede, K., Vinter, B.: Bohrium: a Virtual Machine Approach to Portable Parallelism. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). IEEE (2014)
20. Sala, M., Spotz, W., Heroux, M.: PyTrilinos: High-Performance Distributed-Memory Solvers for Python. ACM Transactions on Mathematical Software (TOMS) (March 34, 2008)
21. Smith, K., Spotz, W.F., Ross-Ross, S.: A Python HPC Framework: PyTrilinos, ODIN, and Seamless. In: 2012 SC Companion High Performance Computing, Networking, Storage and Analysis (SCC), pp. 593–599. IEEE (2012)
22. Zhao, Z., Davis, M., Antypas, K., Yao, Y., Lee, R., Butler, T.: Shared Library Performance on Hopper. In: CUG 2012, Greengineering the Future, Stuttgart, Germany (2012)