Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2010/08-01

23/August/2010

High-Performance Parallel Computations using Python as High-Level Language

Stefano Masini and Paolo Bientinesi

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

©Stefano Masini and Paolo Bientinesi 2010. All rights reserved

List of AICES technical reports: http://www.aices.rwth-aachen.de/preprints

High-Performance Parallel Computations using Python as High-Level Language

Stefano Masini[*] and Paolo Bientinesi[**]

RWTH Aachen, AICES, Aachen, Germany

Abstract. High-performance and parallel computations have always represented a challenge in terms of code optimization and memory usage, and have typically been tackled with languages that allow a low-level management of resources, like Fortran, C and C++. Nowadays, most of the implementation effort goes into constructing the bookkeeping logic that binds together functionalities taken from standard libraries. Because of the increasing complexity of this kind of codes, it becomes more and more necessary to keep it well organized through proper software engineering practices. Indeed, in the presence of chaotic implementations, reasoning about correctness is difficult, even when limited to specific aspects like concurrency; moreover, due to the lack in flexibility of the code, making substantial changes for experimentation becomes a grand challenge.
Since the bookkeeping logic only accounts for a tiny fraction of the total execution time, we believe that for such a task it can be afforded to introduce an overhead due to a high-level language. We consider Python as a preliminary candidate with the intent of improving code readability, flexibility and, in turn, the level of confidence with respect to correctness. In this study, the bookkeeping logic of SMP-MRRR, a C & Fortran highly optimized multi-core eigensolver, is ported to Python. We report here on the porting process and on the pros and cons of using a high-level language in a high-performance parallel library.

Keywords: Productivity, Code Development, High-Performance Computations, Python, High-Level Languages

1 Introduction

The scientific computing community spends a great deal of effort in developing numerical routines and libraries. The codes are often both large and difficult to manage. As an example, representative codes for 3D Finite Element solvers normally include hundreds of files, thousands of routines, surpass the 100K lines of code, and are entirely written in one or more of the classic languages: C, C++ and Fortran. Even though the situation is considered to be sub-optimal, it is often tacitly accepted in the name of high-performance.

[*] `stefano@stefanomasini.com`
[**] `pauldj@aices.rwth-aachen.de`

Typically, complex numerical solvers and simulations are organized into layers: the key logic is expressed at high level in terms of simpler algorithms that perform most of the number crunching. A large number of separate routines, taken from consolidated libraries and often used as black boxes, needs to be orchestrated through proper data structures, function calls and thread synchronization. The libraries themselves are organized in the same fashion. As an example, LAPACK, the de facto standard linear algebra library, is layered on top of the routines of levels 3, 2 and 1 of the BLAS library. Such a modular approach helps separating the computation—confined to well-defined functions—from the data and thread management.

A considerable challenge arises when concurrency is required: in that situation it is generally hard to be highly confident with respect to correctness and absence of deadlock. The typical approach is to try to keep the logic as simple as possible so that the intricate implementation remains confined to limited sections. Unfortunately, always in the name of high-performance, Fortran and C are often misused to obtain low-level optimizations over instructions, registers and the memory hierarchy. Applying these practices when not strictly necessary makes it unlikely that code rich in semantics is also simple and compact. Furthermore, if achieving correctness is already time consuming, it becomes even more expensive to experiment with algorithmic variants, despite this activity is precisely what scientific research is all about.

The main concerns for numerical code and libraries are correctness and performance. We believe that nowadays other concerns should be considered as equally important: code modularity, flexibility, and development time. In this paper we focus on the development of the logic for the management of data, functions and threads (bookkeeping) in high-performance parallel libraries. For this specific task, high-level languages might be better suited than C or Fortran: we chose Python in our first attempt to investigate the pros and cons of replacing C as the main programming language.

The paper is organized as follows. In Section 2 we describe our experimental setup and compare with related approaches. Details on the actual porting of the code are in Section 3. In Section 4 we report on the concrete advantages that we experienced. Section 5 explains the impact of the Global Interpreter Lock in order to understand the performance numbers presented in Section 6, together with future directions. In Section 7 we draw conclusions.

2 Setup

Python is a very appealing language for the scientific computing community at large [6]. Its applicability, even to large projects, has been proven fruitful for a long time already [13]. Most of the investigations and studies targeting parallel computations have only considered the model of distributed memory and message passing [3, 7–12]. In that scenario Python has been used to steer the computation by organizing and synchronizing processes containing number-crunching operations performed by libraries normally written in C, C++ or Fortran. By targeting shared memory parallelism, our investigation departs sharply from previous efforts.

We aim at using Python *within* a high-performance numerical library, i.e., a piece of code that is highly optimized for speed. Our computing model is SMP (Symmetric MultiProcessing), in which parallelism is obtained through multithreading, and synchronization is based on primitives like semaphores. When compared to approaches based on MPI or BSP, multithreading leads to a different type of parallelism: thanks to fast shared memories and the absence of costly message passing operations, algorithms are parallelized at a much finer granularity.

There is a diffuse perception that Python is not mature enough to be used in this context because of its slow interpreted nature and well known limitations like the Global Interpreter Lock. While some of these criticisms are well founded, we are nonetheless interested in exploring the boundaries of applicability of Python to scenarios in which we feel developers would highly benefit from an expressive language. The goal of this study is to shed some light on Python's current actual limits. We envision two favorable consequences: on the one hand scientific developers might come to the realization that some of the drawbacks of the language are not as severe as expected; on the other hand, Python's developers could pinpoint specific weaknesses that is worth improving on.

In the attempt of stressing the limits of the language, we set out for a rather challenging goal: using Python within a highly-optimized parallel numerical library. We selected SMP-MRRR, a multi-core version of the MRRR symmetric eigensolver [1, 2]. Eigensolvers are at the core of innumerable scientific computations and are included in all the standard numerical libraries. SMP-MRRR is currently the fastest eigensolver available for multi-core architectures. It is written in C and Fortran, and makes use of routines from LAPACK and BLAS. It is designed for systems comprising up to 60-80 cores. SMP-MRRR constitutes an especially disadvantageous choice for our goal: not only is it a high-performance library, it also has the lowest algorithmic complexity among all the existing eigensolvers, $O(n^2)$.[1] As a consequence, any overhead introduced by Python will impact the overall execution time much more noticeably than it would had the algorithm had $O(n^3)$ complexity, like most of the dense linear algebra algorithms have. With such a complexity, the overhead would be easily hidden under a much higher amount of computations, quickly becoming negligible.

The execution of SMP-MRRR unfolds by computing an initial approximation of the eigenvalues first, and the eigenvectors together with more accurate eigenvalues later. Depending on the number of available cores, the initial eigenvalue computation is either performed sequentially by the fast *dqds* algorithm, or in parallel by *bisection*. The eigenvectors, together with more accurate eigenvalues,

[1] Given an input matrix of size n, SMP-MRRR computes all the eigenvalues and eigenvectors of the matrix in $O(n^2)$ floating point operations. While such a complexity is an upper bound, the actual completion time depends on the input matrix.

are then computed in parallel by organizing the computation according to a tree of tasks, and utilizing a task queue based approach.

The way that tasks are created, their execution order and the portion of data they manipulate represent the core contribution to the algorithm implementation, and accounts for most of the development time. We will refer to this portion of code as the *bookkeeping logic*, implemented in roughly 5000 lines of C code. The remaining code, mostly written in Fortran, is what we consider actual computation, or informally, *number crunching.* More precisely, we do not include in the bookkeeping logic any loop whose primary purpose is to perform numerical calculations, like $O(n)$ vector manipulations. We count as bookkeeping the boolean logic, the `if` and `switch` statements, and simple (non computational) loops. The code is organized into multiple functions; although the number of function calls depends not only on the matrix size but also on its numerical properties, the time spent on the logic preceding each function call is independent of these factors.

Porting the bookkeeping logic to Python introduces overhead, so it is useful to start from a clear understanding of this portion of the code in relation to the actual number crunching. Table 1 shows the number of calls to the Fortran routines for two different types of matrices that we considered in our experiments: Wilkinson and Hermite.[2] The sections of number crunching are highly fragmented and interspersed with bookkeeping logic; the exact figures depend on the nature of the matrices. With Hermite matrices, the amount of time spent inside the bookkeeping logic decreases with larger matrices, as expected, due to the quadratic complexity of the algorithm. The trend for Wilkinson matrices is not as apparent, because of the numerical properties of these matrices.

Table 1. Analysis of the original C implementation: impact of the bookkeeping code on the overall execution time

Wilkinson matrices			Hermite matrices		
Size	LAPACK calls	Bookkeeping	Size	LAPACK calls	Bookkeeping
3001	15210	0.90%	3001	9303	4.18%
5001	25220	0.92%	5001	15724	3.19%
10001	50319	0.83%	10001	36651	1.55%

All the experiments were run on a Mac Pro with two 2.4Ghz Quad-Core Intel Xeon processors, for a total of 8 available cores. We limited the runs to only 6 cores in order to avoid interference with others applications and collect more stable results.

[2] In order to avoid the possible overhead due to the contention of shared data structures we performed the measurements in single threaded executions.

3 Porting to Python

We chose Python as target high-level language because of its clean syntax, powerful semantics and large standard library featuring thread-safe data structures, like priority queues and shared counters. When compared against more traditional languages, Python presents a much lighter cognitive load: developers do not spend time working out pointer arithmetic or backtracking compiler errors, thus becoming more productive and less likely to introduce subtle bugs.

During the porting phase, our main concern was to preserve correctness. The way we achieved this with high confidence was by constantly comparing the output from the original version with that of the work in progress. Code was always kept in a runnable shape and we performed test runs. The output of SMP-MRRR was also verified through residual and orthogonality of the computed quantities. The translation did not introduce any numerical or computational optimization, but was intentionally limited to keep the logic and the data structures as they were. We also point out that the porting was performed by one of the authors, without specific knowledge of the mathematics involved in the eigensolver.

The initial porting phase aimed at removing all the C code, leaving Python to instantiate the data structures and directly call the Fortran routines. The NumPy [22] package provides facilities to instantiate and manipulate multi-dimensional arrays, and is the de-facto standard for scientific computations in Python. The Fortran routines were made callable by creating simple wrappers using Cython [23] (we discuss Cython in more details in Section 6).

In order to test the code as often as possible, the C functions were ported one at a time, starting from the callers and then moving on to the invoked ones. Thread management and synchronization was very easily rewritten using the Python standard library.

This early translation of the code still presented the typical structure of a procedural programming style: a few huge functions with lots of parameters and local variables. We considered that an Object Oriented design would have been more adequate so we proceeded with a refactoring phase [16, 15]. We first split large blocks of statements into small methods. Then we renamed variables and functions to be more meaningful. Finally we were able to decouple logically independent parts of the algorithm into correspondingly independent classes.

In the end, the entire porting process took an experienced Python developer a total of 7 working days. The Python code is considerably shorter and more readable than the original. Only 1800 lines as opposed to the 5100 of the C code, not including comments. The shorter scope of local variables makes it is easier to understand the logic and the object oriented design makes it possible to experiment with alternative strategies by simply composing different sets of objects.

4 The advantages of Python and refactoring

The refactoring process uncovered some corner cases that, though unlikely, could lead to deadlock. For example, some portions of code taken from the task queue management appeared duplicated inside a task performing function. Duplicated code is generally an indication of poor design. In this case we had a mixture of high-level logic (the task queue management) with lower-level one (the actual execution of a task). The code turned out long and hard to read, so reasoning about correctness was very difficult. Instead, after the refactoring, all of the task queue management logic reacquired integrity by being encapsulated within the same class, while at the same time the task execution code became shorter. Thanks to the improved readability we were able to spot a flaw in the code design that had not been noticed before.

In the context of parallel computing, such defects are especially dangerous as they both add unnecessary complications to the logic—which is already difficult to keep in sync with the mathematical model—and might lead to idling processors. An effective practice is then to keep the code in the best possible shape in order to maintain a high level of confidence in the correct behaviour of the system.

The competencies required to develop high quality software are largely independent of domain specific knowledge. For example, while porting the library to Python, the author became intimately familiar with the data flow and the parallelism of the program, despite his lack of knowledge in eigensolvers. Indeed, the bookkeeping logic in numerical code, once stripped of the number-crunching sections, is no different than any other type of general purpose program.

In general, sane software engineering practices can and should be applied in order to keep the complexity under control, regardless of the programming language used. Good practices include, but are not limited to, the use of meaningful names for variables, small functions that do just one thing, possibly with few parameters and no logic repeated in more than one place [14].

Code refactoring is a necessary activity but it requires knowledge and discipline. Learning and applying it though seems to be easier on modern dynamic languages like Python because of their simplicity and flexibility. We believe that by using such languages developers have a better chance of becoming more conscious about software engineering issues thus writing better code and becoming more productive.

5 Simulating the Global Interpreter Lock

Python supports multithreading but the internal implementation of its most commonly used interpreter (CPython, written in C) limits the effectiveness of this model in the case of multi-core CPUs. In order to simplify the implementation details of the basic data types, like lists and dictionaries, CPython makes use of a Global Interpreter Lock (GIL). A thread can progress the execution only after it internally acquires the GIL with exclusive access. Therefore it is not uncommon that only one core is used while the others remain idle. Fortunately, in our case this is far from the truth as we observed that all the available cores were effectively used.

Indeed there are situations when the GIL is actively released so that other threads have a chance to run. For example when an I/O operation is performed, or when a call is made to C or Fortran extensions that are known to be thread-safe. This latter case is especially relevant for our application, as this means that in SMP-MRRR the threads will compete for the GIL only during the execution of bookkeeping sections.

It is important to realize that due to the presence of shared data structures, e.g., the task queue, the use of locks and synchronization devices is standard in the SMP model. Therefore, as the number of threads grows, increasing levels of contention are expected even in the original C algorithm. However, there is a great difference in granularity between simple data structure locks and a lock like the GIL that encapsulates all the bookkeeping logic. We are interested in measuring how locks of different granularity affect the overall performance in our scenario because this will be of primary importance to understanding the overhead introduced by Python, as described in Section 6.

In order to reproduce the contention caused by the GIL in Python, we created a modified C version of SMP-MRRR in which we artificially introduced a global lock. Since the modified and the original versions have no other differences, the resulting measured overhead is a direct indication of the amount of extra contention introduced. Our artificial global lock is implemented by means of a simple mutex from the *pthread* standard library, whereas the internal implementation of the Python GIL is much more elaborate. Thus, our measurements represent a lower bound on the amount of overhead generated by the real GIL: its current implementation is known to be inefficient because it originates many system calls and can induce the threads into an unproductive condition known as the *GIL battle* [19]. The latest Python 3.2—still under active development—will include a better version of the GIL [18, 17]. We expect that with every new release that improves the GIL implementation, the Python interpreter will perform closer to the lower bound that we are observing in our simulation.

We report on the execution of the modified version of SMP-MRRR for two types of input matrices, Wilkinson and Hermite. The eigenspectrum of Wilkinson matrices is such that the computation of eigenvalues and eigenvectors is especially involved, which translates to number-crunching sections that take longer to complete. The Hermite matrices are instead quite favorable, meaning that the outputs can be computed with shorter number-crunching sections. The impact of the bookkeeping logic will therefore be significantly more evident in the latter case. In the case of Hermite matrices, a larger problem size makes the bookkeeping become less noticeable, as more and more time is spent within number-crunching sections. For Wilkinson such a trend is more subtle, because the complexity of the eigenspectrum increases together with the problem size. Finally a note about multithreading: as the number of threads increases, there is a higher chance of threads competing for the lock to start a bookkeeping section, thus increasing the overhead.

Table 2 summarizes the results of our experiments on the simulated global lock in the C code. The top two tables show the likelihood that a thread, upon

Table 2. Contention due to the use of a global lock.

Lock busy count

Wilkinson				Hermite			
Size	2 cores	4 cores	6 cores	Size	2 cores	4 cores	6 cores
3001	0.48%	1.66%	4.15%	3001	4.16%	12.86%	27.09%
5001	0.30%	1.06%	2.57%	5001	4.48%	11.74%	27.06%
10001	0.17%	0.62%	2.16%	10001	1.73%	6.12%	12.25%

Execution time penalty

Wilkinson				Hermite			
Size	2 cores	4 cores	6 cores	Size	2 cores	4 cores	6 cores
3001	0.60%	1.07%	2.62%	3001	0.85%	1.42%	4.76%
5001	0.66%	1.82%	3.28%	5001	0.65%	1.16%	5.81%
10001	0.60%	2.17%	3.53%	10001	0.20%	0.43%	0.56%

completion of a Fortran call, finds the global lock unavailable (because taken by another thread). A value of 0% signifies that the lock is always available, while 100% means that it is never available, and the thread is always forced to stay idle, waiting for the release. The percentage has been calculated as the average across 30 runs. The standard deviation increases slightly with the number of cores, as one may expect, but remains within 10% of the average value on most cases.

The bottom two tables show the impact of introducing the global lock on the overall running time. The two C versions of the algorithm were tested 30 times for every combination of matrix type, size and number of cores. The percentage represents the amount of overhead computed using the average values across the runs. There is an obvious correlation between the frequency of hitting a busy lock and the performance penalty. The standard deviation for the original algorithm always remains below 10% of the minimum value, indicating that the behaviour of the system is quite stable and predictable. In the case of the global lock instead we measured an increase also of the standard deviation, of up to 2 to 3 times in some cases, indicating that the contention not only decreased the overall performance, but also made it somewhat less predictable.

6 Performance hit and future work

In Table 3 we compare the execution time of the original SMP-MRRR and the Python version. The overhead should be considered in light of the observations made in Section 5 about the Global Interpreter Lock. There is no clear way to identify precisely the amount of overhead due to the GIL, but it certainly has an impact and explains why the overhead increases with the number of cores.

Table 3. Overall performance overhead: Python over C

Wilkinson matrix				Hermite matrix			
Size	2 cores	4 cores	6 cores	Size	2 cores	4 cores	6 cores
3001	11.80%	28.20%	57.80%	3001	95.40%	231.30%	386.90%
5001	15.00%	22.20%	40.10%	5001	85.20%	121.20%	197.80%
10001	13.20%	15.00%	19.20%	10001	43.50%	54.60%	53.00%

The interpreted nature of Python is the other obvious source of overhead. Previous studies [4, 5] have shown that for some simple calculations there can be a difference of up to 2 orders of magnitude between the speed of a Python implementation and the equivalent C. The bookkeeping logic cannot be considered proper algorithmic code because it does not include loops and numerical calculations but, still, we expect it to be much slower than C.

There are well known approaches to circumvent the speed limit of the Python interpreter. Ultimately they are all based on the generation of efficient machine code, or C source code that can in turn be compiled. Some of these solutions are enough low-level to also give the opportunity to work around the GIL limitation by allowing the developer to release it when possible.

We will review some of the available tools—of which only one we were able to test—and consider how much it is necessary to alter the starting Python code in order to achieve the speedup.

Psyco [20] is an easy to use Just In Time (JIT) compiler. It requires no modification to the original Python code and runs in the standard Python interpreter. It works by generating efficient machine code blocks in a lazy fashion at run-time. When the interpreter hits for the first time a given Python function it compiles it for the current argument types. The next time that the function is called with the same argument types, the previously generated machine code is used instead of interpreting the source code.

Previous studies [4, 5] have shown that this approach can increase the performance even by 70-80% in the case of simple numerical computations with nested loops. Conversely, in our case we have observed a consistent performance loss of up to 15%. We believe the reason lies in the very nature of the bookkeeping code that, by definition, does not include computation intensive loops. It is possible that in our case the startup cost of just-in-time compilation is not compensated by enough repetitions of the same code.

Psyco development has ceased because the author decided to contribute to the PyPy [21] project instead. This alternative implementation of the Python interpreter—written in Python itself—is bundled with a more advanced JIT compiler that performs better than Psyco. We plan on trying PyPy in the future, even though we do not expect the JIT to bring substantially different results in our case.

A radically different approach than JIT is to try to statically compile a Python program. In the general case this is not possible because the variable types are known only at run-time. Therefore all the available tools try to overcome this limitation, each adopting a different strategy.

Cython [23] is a language almost identical to Python except that variable types can be statically declared. A dedicated compiler translates the Cython code to C, which in turn is further compiled to build a conventional extension module. If the original Python code is properly modified to add type declarations for variables and function parameters, it is possible to obtain very efficient compiled code. We have used Cython as a way to build simple wrappers around the Fortran routines, but we could extend its usage to incorporate a portion or all of the bookkeeping logic. With Cython it is also possible to declare which parts of the code do not need the GIL thus reducing the granularity of global locking, at the expense of not using high-level data structures like lists or dictionaries. The downside is that Cython is technically a different language than Python and once the starting code has been altered, it is not possible anymore to run it inside the Python interpreter.

ShedSkin [24] is another Python to C translator: again, the ability to generate efficient C code depends on the knowledge of the variable types. In contrast to Cython, Shedskin attempts to guess the types by static analysis and does not require annotations in the Python code. In order for the translator to perform its guesswork, the code must be written according to appropriate restrictions (some constructs are not allowed); the advantage is that the code remains valid Python which can therefore still be run inside the interpreter. Unfortunately ShedSkin is still at an early development stage. It can only be used with short modules and does not directly support NumPy types.

Finally, another promising approach to static compilation is RPython, a subset of the Python language (the "R" stands for "Reduced") that can efficiently be translated to machine code. It is used within the PyPy project to actually define and implement the real Python interpreter. Just like ShedSkin it offers the advantage of leaving the code runnable even inside a regular Python interpreter.

We foresee two possible directions for future development of our experiments. On the one hand we could optimize the current Python implementation by adopting one or more of the tools described above. It would be interesting to see how close we can get to the original C performance by still maintaining a high-level Python development environment. On the other hand, we could look at different languages that perform better than Python when used in our context. We could see how they relate to Python and if they can be considered as equally attractive and productive from the point of view of the developer.

7 Conclusions

Porting the bookkeeping logic from C to Python proved to be an incredibly valuable exercise: it yielded a code readable and easy to reason about. Thanks to this, we were able to thoroughly investigate the correctness of the implementation. As a result, we spotted possible deadlocks and hidden constraints that could affect performance. The new code also lends itself for experimentation and testing of new design and algorithm strategies. We believe that proper software engineering practices should discipline the development process even for scientific codes. High-level languages like Python can greatly enhance this opportunity and so they deserve full attention by the scientific computing community.

On the downside, Python added overheads, as expected, and is still far from being a concrete alternative to traditional languages like C or C++ in performance critical environments. Nonetheless, considering the extremely disadvantageous situation represented by an $O(n^2)$ SMP-parallel algorithm, we observed an interesting performance in spite of the Global Interpreter Lock and the slow interpreted nature of Python. We believe that in many cases these limitations are outweighed by the enhanced flexibility of the language.

The high-level dynamic languages scene is rapidly evolving, so it will be interesting to see how these performance issues will be addressed in the coming years.

References

1. I. Dhillon. *A new $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* Ph.D. thesis, University of California, Berkeley, 1997.
2. M. Petschow and P. Bientinesi. *The Algorithm of Multiple Relatively Robust Representations for Multi-Core Processors.* PARA 2010: State of the Art in Scientific and Parallel Computing, (submitted).
3. J. K. Nilsen, X. Cai, B. Høyland, H. P. Langtangen. *Simplifying parallelization of scientific codes by a function-centric approach in Python.* Submitted to Computational Science & Discovery for publication, 2010.
4. I. Wilbers, H. P. Langtangen and Å. Ødegard. *Using Cython to Speed up Numerical Python Programs.* Proceedings of MekIT'09, ed. by B. Skallerud and H. I. Andersson, pp. 495-512, NTNU, Tapir.
5. P. Ramachandran et al. *A beginners guide to using Python for performance computing.* http://www.scipy.org/PerformancePython.
6. Hans Petter Langtangen. *Python Scripting for Computational Science, 3rd edition.* Springer Publishing Company, Incorporated, 2009.
7. Hans Petter Langtangen and Xing Cai. *On the Efficiency of Python for High-Performance Computing: A Case Study Involving Stencil Updates for Partial Differential Equations.* Proceedings of the Third International Conference on High Performance Scientific Computing, Hanoi, Vietnam, 2008, pp. 337-357.
8. Konrad Hinsen. *Parallel Scripting with Python.* Computing in Science and Engineering, Vol. 9, Issue 6, pp 82-89, 2007.
9. K. Hinsen, H. P. Langtangen, O. Skavhaug and Å. Ødegard. *Using BSP and Python to simplify parallel programming.* Future Generation Computer Systems 22(1-2):123-157, 2006.

10. X. Cai, H. P. Langtangen. *Parallelizing PDE solvers using the Python programming language.* In A. M. Bruaset and A. Tveito, editors, Numerical Solution of Partial Differential Equations on Parallel Computers. Volume 51 of Springer Lecture Notes in Computational Science and Engineering. pp. 295-325, Springer, 2006.
11. X. Cai, H. P. Langtangen, H. Moe. *On the performance of the Python programming language for serial and parallel scientific computations.* Scientific Programming, Vol. 13, Issue 1, pp. 31-56, 2005.
12. Konrad Hinsen and Rue Charles Sadron. *High-Level Parallel Software Development with Python and BSP.* Parallel Processing Letters, Vol. 13, 2003.
13. K. Hinsen. *The Molecular Modeling Toolkit: a case study of a large scientific application in Python.* Proceedings of the 6th International Python Conference, San Jose, California, 1997.
14. Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, 2009.
15. Joshua Kerievsky. *Refactoring to patterns.* Addison-Wesley, 2004.
16. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
17. David Beazley *Understanding the Python GIL.* [Presentation] PyCON 2010 in Atlanta, February 20.
18. David Beazley *Inside the New Python GIL.* [Presentation] Chipy, January 14, 2010.
19. David Beazley *Inside the Python GIL.* [Presentation] Chipy, June 11, 2009.
20. Armin Rigo *Psyco: Python Specializing Compiler.* http://psyco.sourceforge.net/
21. The PyPy Group. *PyPy: a Python implementation written in Python.* http://codespeak.net/pypy
22. T. Oliphant et al. *NumPy software package.* http://numpy.org/
23. G. Ewing, R. Bradshaw, S. Behnel, D. S. Seljebotn et al. *Cython: C-extensions for Python.* http://cython.org/
24. M. Dufour, J. Coughlan. *ShedSkin.* http://code.google.com/p/shedskin/