Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 4 (2011) 17–25

Procedia Computer Science

International Conference on Computational Science 2011

GPAW - massively parallel electronic structure calculations with Python-based software

Jussi Enkovaara[a], Nichols A. Romero[b], Sameer Shende[c], Jens J. Mortensen[d]

[a] *CSC - IT Center for Science Ltd. P.O. Box 405 FI-02101 Espoo, Finland*
[b] *Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA*
[c] *Performance Research Laboratory, University of Oregon, Eugene, OR, 97403, USA*
[d] *Center for Atomic-scale Materials Design, Department of Physics, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark*

Abstract

Electronic structure calculations are a widely used tool in materials science and large consumer of supercomputing resources. Traditionally, the software packages for these kind of simulations have been implemented in compiled languages, where Fortran in its different versions has been the most popular choice. While dynamic, interpreted languages, such as Python, can increase the efficiency of programmer, they cannot compete directly with the raw performance of compiled languages. However, by using an interpreted language together with a compiled language, it is possible to have most of the productivity enhancing features together with a good numerical performance. We have used this approach in implementing an electronic structure simulation software GPAW using the combination of Python and C programming languages. While the chosen approach works well in standard workstations and Unix environments, massively parallel supercomputing systems can present some challenges in porting, debugging and profiling the software. In this paper we describe some details of the implementation and discuss the advantages and challenges of the combined Python/C approach. We show that despite the challenges it is possible to obtain good numerical performance and good parallel scalability with Python based software.

Keywords: Python, Numpy, MPI, Density-functional theory

1. Introduction

Electronic structure calculations are widely used theoretical tool for investigating atomic-level properties for various research problems in materials science, chemistry and physics. These kind of simulations are also large consumer of computing resources in several supercomputing centers. There exists several software packages, both open source and proprietary, which have been implemented in typical high performance computing way using compiled languages, mostly Fortran. We have implemented an open source software package GPAW [1, 2, 3] using the Python programming language combined with extensions written in C. In this work, we present some Python related implementation details of GPAW and discuss some challenges one must address when using Python in massively parallel high performance computing.

Python is alleged to enhance programming productivity compared to traditional compiled languages like C and Fortran. This is due to, for example, the language's dynamic nature, very high level of abstraction, support for object oriented programming and large set of tools and utilities included in the standard library. However, due to interpreted nature of Python, Python applications run typically several orders of magnitude slower than corresponding applications written in compiled language and as such Python is not suitable for high performance computing. However, by

1877–0509 © 2011 Published by Elsevier Ltd. Selection and/or peer-review under responsibility of Prof. Mitsuhisa Sato and Prof. Satoshi Matsuoka
doi:10.1016/j.procs.2011.04.003

using the NumPy package [4, 5] it is possible to perform different numerical operations on multidimensional arrays with a speed close to the compiled language. If a higher performance is required, it is relatively easy to extend Python with functions written in C (or Fortran), giving the performance of compiled language to selected parts of the application. The so called 90/10 rule of performance analysis states that typically 90 % of time is spent in 10 % of program code. Thus, it is often enough the write only few routines in C and use Python for the rest of the program, obtaining still a performance on par with a pure C program. We have used this kind of approach successfully with GPAW. Even though majority of the code is written in Python, by using few C-extension functions and efficient numerical libraries the program can reach ∼25 % of peak floating performance of massively parallel supercomputer systems.

While in our experience Python makes programming of high level algorithms and addition of new features into software efficient, using Python in massively parallel computing introduces some challenges. First of all, many parallel programming tools such as debuggers and profilers support only C or Fortran. Also, in the most widely used Python interpreter, the CPython the innocent looking module import mechanism can become severe bottleneck in large scale parallel computing.

2. Overview of GPAW

GPAW is based on the density-functional theory (DFT) [6, 7] which is one of the most widely used theoretical frameworks for electronic structure calculations. Also the time-dependent density-functional theory (TD-DFT) [8] is implemented. For numerical approximations we use the projector augmented wave (PAW) method [9, 10], which is implemented using uniform real-space grids and finite differences. An optional atomic orbital basis set is also available. As we focus here on the Python related implementation issues, we use only the real-space grid based standard ground state DFT to illustrate the important aspects.

The basic equations of ground state DFT are (atomic units are used):

$$\begin{aligned}
H\psi_n(r) &= e_n\psi_n(r), && (1)\\
H &= -\frac{1}{2}\nabla^2 + v_{eff}(r) = -\frac{1}{2}\nabla^2 + v_H(r) + v_{ext}(r) + v_{xc}(r), && (2)\\
\nabla^2 v_H(r) &= -4\pi\rho(r), && (3)\\
\rho(r) &= \sum_i |\psi_i(r)|^2, && (4)\\
& && (5)
\end{aligned}$$

where e_n and $\psi_n(r)$ are the eigenenergy and eigenvector of electron n, respectively, v_{eff} is the effective potential consisting of the electrostatic Hartree potential v_H, the external potential due to atomic nuclei v_{ext}, and the so called exchange-correlation part v_{xc} (the main physical approximation in DFT). As the electronic charge density $\rho(r)$ is determined by the eigenvectors, the system of equations is non-linear, and the equations are solved self-consistently starting from initial guess for charge density and iterating the equations until the density does not change any longer. From the eigenvectors and eigenvalues one can determine other physical quantities such as total energies and forces acting on the atoms.

When the equations are discretized, the eigenvectors, potentials and charge density are represented by their values at the grid points, ψ_{nG}, ρ_G, $v_{H,G}$, etc., with the three indices designating a point in the real-space grid condensed into single G index. The PAW method adds some non-local contributions to the Hamiltonian in terms of atomic Hamiltonian matrix elements $H^a_{i_1 i_2}$ and projector functions $p^a_i(r)$, which are defined within so called augmentation sphere centered on atom a. In discretized form the Hamiltonian operator H in PAW approximation is

$$H_{GG'} = -\frac{1}{2}L_{GG'} + v_{eff,G}\delta_{GG'} + \sum_{i_1 i_2} p^a_{i_1 G} H^a_{i_1 i_2} p^a_{i_2 G'}. \qquad (6)$$

where the $L_{GG'}$ is the finite-difference stencil for the Laplacian. The Hamiltonian is sparse, and the full matrix is never stored explicitly, one only needs to able to apply the Hamiltonian to a function, i.e. evaluate a matrix free matrix-vector product. A more detailed description about the theoretical and numerical framework used in GPAW can be found in the recent review [2].

When using Python together with NumPy, the natural data structure for the quantities defined on the real-space grid of the whole simulation cell is NumPy array (three dimensional for density and potential, four dimensional for wave functions). The projector functions require more logic, and a special data structure containing elements defined both on C and on Python level is used. The advantage of NumPy is two-fold, first it provides a more efficient numerical evaluations for vectorizable operations, second NumPy arrays provide convenient container for contiguous data which can be passed easily to C-functions.

In principle, one could first implement the whole program in Python, and convert the most time consuming parts to C later on. However, the algorithms used in real-space DFT are well known, and the most time consuming parts are known *a priori*. The most numerically intensive parts in DFT algorithm are:

1. The Poisson equation
$$\nabla^2 v_H = -4\pi\rho \tag{7}$$
is solved using a multigrid algorithm, where the basic operations are evaluations of second order derivatives with finite-differences, restriction of functions from a fine grid to a coarser grid, and interpolation from a coarser grid to finer grid.
2. Subspace diagonalization involves application of Hamiltonian operator to wave functions and diagonalizing a Hamiltonian matrix
$$H_{nn'} = \sum_G \psi_{nG} \sum_{G'} H_{GG'}\psi_{n'G'} = \sum_G \psi_{nG}(H\psi)_{n'G}. \tag{8}$$
When applying the Hamiltonian operator, one needs finite-difference derivatives of wave functions, integrals of projector functions times wave function
$$P^a_{ni} = \sum_{G^a} p^a_{iG^a}\psi_{nG^a} \tag{9}$$
and addition of projector function times matrix to the wave function
$$\psi'_{nG^a} = \psi_{nG^a} + \sum_i O^a_{ni} p^a_{iG^a}. \tag{10}$$
The grid points G^a are defined inside the augmentation sphere of atom a. Finally, the wave functions are multiplied by the diagonalized matrix $H^D_{nn'}$,
$$\psi'_{nG} = \sum_{n'} H^D_{nn'}\psi_{n'G}. \tag{11}$$
3. Iterative refinement of wave functions requires applications of Hamiltonian to wave functions as well as multigrid preconditioning which uses restriction and interpolation operations similar to the Poisson problem.
4. Orthonormalization of wave functions requires construction of overlap matrix
$$S_{nn'} = \sum_G \psi_{nG} \sum_{G'} S_{GG'}\psi_{n'G'}, \tag{12}$$
where the overlap operator involves the same operations with the projector functions as the subspace diagonalization. The overlap matrix is Cholesky decomposed and inverted to get a matrix $S^D_{nn'}$, after which orthonormal wave functions are obtained as
$$\psi'_{nG} = \sum_{n'} S^D_{nn'}\psi_{n'G}. \tag{13}$$

The finite-difference, restriction and interpolation operations are implemented in C, as well as as integrals of projector function times wave function products and addition of projector functions to wave functions. In subspace diagonalization and orthonormalization a large fraction of numerical operations is spent in matrix-matrix products $H_{nn'} = \sum_G \psi_{nG}(H\psi)_{n'G}$ and $\psi'_{nG} = \sum'_n H^D_{nn'}\psi_{n'G}$. Normally, highly optimized BLAS libraries exist for these kind of linear algebra operations, and GPAW contains its own Python interface to a few BLAS routines. In principle, the matrix-matrix products could be evaluated also with NumPy, however, large temporary arrays would be needed in some cases (e.g. when transposes of matrices are needed). Also, even though NumPy can be built against optimized

BLAS, we have found that the build process is non-trivial for other libraries than ATLAS, especially in cross-compile environments.

After the core numerical kernels are implemented in C (or in libraries), the programmer can use all productivity enhancing features of Python when focusing on the higher level algorithms. For example, object oriented programming is used heavily in GPAW. Depending on the type of calculation the wave functions can be either real or complex valued, and the dynamic typing of Python allows the high-level algorithm, which is written in Python, to be independent of the data type; only a few compute kernels which are Python-C extensions are explicitly coded in real or complex data. This is in sharp contrast to compiled languages like C, Fortran and even C++ which would require the whole algorithm to be coded for different data types. Figure 1 shows the evolution of the GPAW's code base, where one sees clearly that once the basic C-kernels were implemented, the improvement of algorithms and addition of new features has been done largely on Python level.

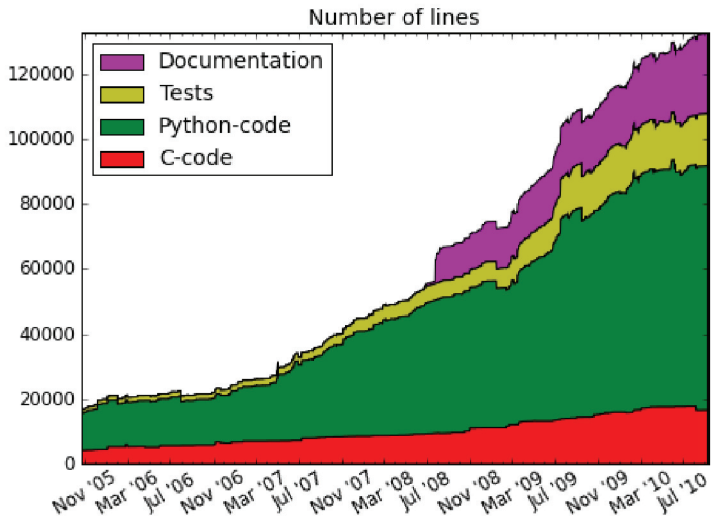


Figure 1: The development of the volume of GPAW's code.

3. Parallel implementation

The parallellization of GPAW is based on the message passing interface MPI [11]. Calls to MPI routines in finite-difference derivatives, interpolation and restriction operations are coded in C, while in the other parts of the algorithm MPI calls are done from Python. The MPI standard defines interfaces to Fortran and C (and C++), but there are unofficial Python interfaces, such as MPI4Py [12]. In GPAW, only limited number of MPI functions are needed and the code uses its own custom Python interface to MPI. The decision is partly historical, as GPAW pre-dates the availability of MPI4Py, also by relying on custom interface, the additional software requirements can be kept minimal which improves the portability of the software especially in more exotic architectures.

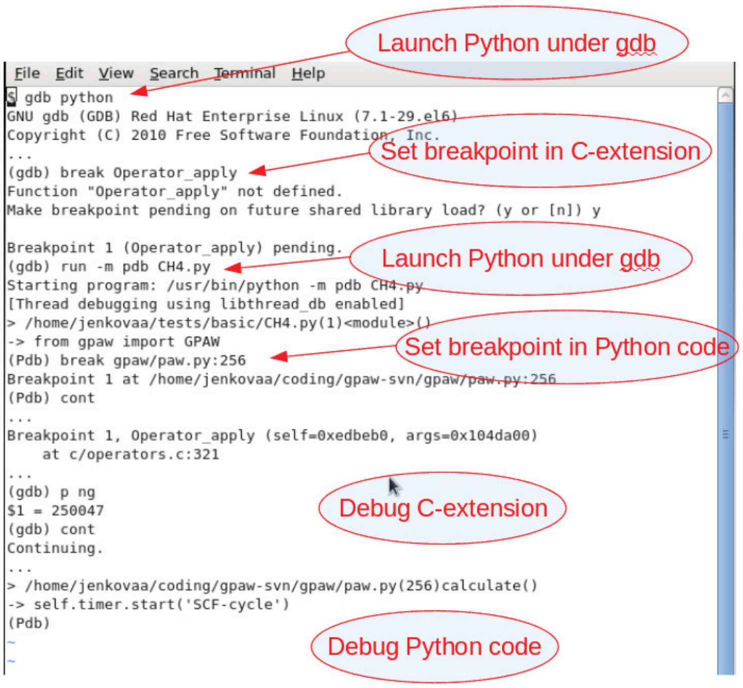
In large scale calculations parallel dense matrix diagonalization is needed for good performance, and we have implemented a custom Python interface to the parallel dense linear algebra library ScaLAPACK [13].

As a special feature which has been useful in several supercomputer usage related issues, GPAW uses a custom MPI-enabled Python interpreter. In practice the custom interpreter is largely just a C main() routine, where the MPI initialization with MPI_Init() is performed before calling Python's main routine PyMain(), linked to the standard Python library. Also, all the GPAW's C-extensions are included in the custom interpreter binary.

4. Python challenges

4.1. Debugging

No programmer writes non-trivial bug free code, thus debugging is always needed in the development process. A standard Python installation includes a command line based debugger *pdb*, which allows one to set breakpoints, step through the code and investigate values of variables. Also, several graphical front-ends can use *pdb*, enabling Python debugging with a GUI. While several conventional debuggers, such as *gdb*, can be used for debugging C-extensions, we are not aware of any tool that enables simultaneous debugging of Python and C-code. One solution we have found is to first launch the Python interpreter under *gdb*, and set break points to C-extension before actually starting the Python interpreter. Now, the Python script can be started under *pdb*, and Python code and variables investigated. When execution reaches the break point at C-extension, *gdb* takes over and one can investigate the behaviour of C-code. Figure 2 shows a screen shot of combined *gdb*-*pdb* session.


Launch Python under gdb
File Edit View Search Terminal Help
Set breakpoint in C-extension
Launch Python under gdb
Set breakpoint in Python code
Debug C-extension
Debug Python code

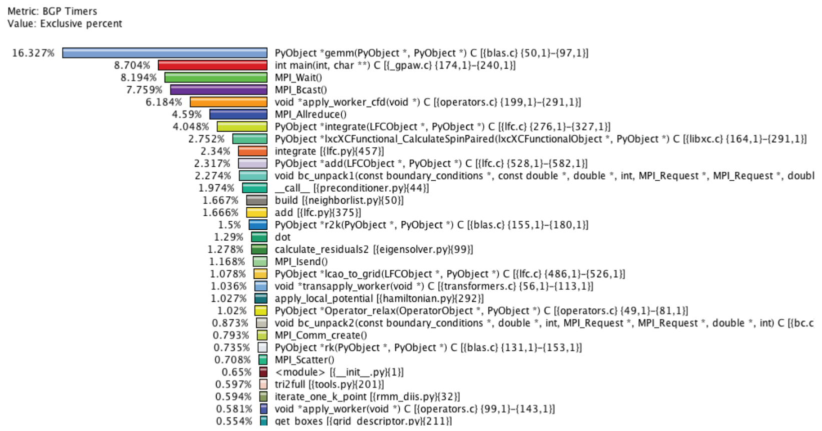
```
$ gdb python
GNU gdb (GDB) Red Hat Enterprise Linux (7.1-29.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
...
(gdb) break Operator_apply
Function "Operator_apply" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (Operator_apply) pending.
(gdb) run -m pdb CH4.py
Starting program: /usr/bin/python -m pdb CH4.py
[Thread debugging using libthread_db enabled]
> /home/jenkovaa/tests/basic/CH4.py(1)<module>()
-> from gpaw import GPAW
(Pdb) break gpaw/paw.py:256
Breakpoint 1 at /home/jenkovaa/coding/gpaw-svn/gpaw/paw.py:256
(Pdb) cont
...
Breakpoint 1, Operator_apply (self=0xedbeb0, args=0x104da00)
    at c/operators.c:321
...
(gdb) p ng
$1 = 250047
(gdb) cont
Continuing.
...
> /home/jenkovaa/coding/gpaw-svn/gpaw/paw.py(256)calculate()
-> self.timer.start('SCF-cycle')
(Pdb)
~
~
```

Figure 2: Debugging both C-extensions and Python code with gdb and pdb.

As (large scale) parallel Python programs are still a sort of esotery, there are no parallel Python debuggers. Parallel debuggers supporting C, such as Totalview and Allinea DDT can be used for debugging the C-extensions, but for the Python code one is largely left with print statements. For debugging the C-extensions, GPAW's custom Python interpreter provides a convenient initial breakpoint.

4.2. Profiling

When working with computationally intensive applications where a single run can take hours or days, performance analysis is crucial aspect of the development process. The Python standard library includes profiling modules (cProfile, profile, hotshot) for serial applications. These profilers show also the time spent in a C-extension function, however, they cannot break down the contributions within the C-extension. Standard profiling tools, such as gprof, can be used to profile the extensions without the Python contributions, and as most of the time is actually spent in C-extensions, this can be often sufficient. In contrast to debugging there is however a tool that can be used for profiling simultaneously Python and C-code. The TAU performance system [14] supports source-based instrumentation in the Python-C mixed language environment. The TAU profiler supports even parallel performance analysis, thus also communication bottlenecks can be investigated transparently. Figure 3 shows a screenshot of TAU profile, where both Python and C-functions as well as MPI routines are seen.



```
Metric: BGP Timers
Value: Exclusive percent

16.327%  PyObject *gemm(PyObject *, PyObject *) C [{blas.c} {50,1}-{97,1}]
 8.704%  int main(int, char **) C [{_gpaw.c} {174,1}-{240,1}]
 8.194%  MPI_Wait()
 7.759%  MPI_Bcast()
 6.184%  void *apply_worker_cfd(void *) C [{operators.c} {199,1}-{291,1}]
  4.59%  MPI_Allreduce()
 4.048%  PyObject *integrate(LFCObject *, PyObject *) C [{lfc.c} {276,1}-{327,1}]
 2.752%  PyObject *lxcXCFunctional_CalculateSpinPaired(lxcXCFunctionalObject *, PyObject *) C [{libxc.c} {164,1}-{291,1}]
  2.34%  integrate [{lfc.py}{457}]
 2.317%  PyObject *add(LFCObject *, PyObject *) C [{lfc.c} {528,1}-{582,1}]
 2.274%  void bc_unpack1(const boundary_conditions *, const double *, double *, int, MPI_Request *, MPI_Request *, doubl
 1.974%  __call__ [{preconditioner.py}{44}]
 1.667%  build [{neighborlist.py}{50}]
 1.666%  add [{lfc.py}{375}]
   1.5%  PyObject *r2k(PyObject *, PyObject *) C [{blas.c} {155,1}-{180,1}]
  1.29%  dot
 1.278%  calculate_residuals2 [{eigensolver.py}{99}]
 1.168%  MPI_Isend()
 1.078%  PyObject *lcao_to_grid(LFCObject *, PyObject *) C [{lfc.c} {486,1}-{526,1}]
 1.036%  void *transapply_worker(void *) C [{transformers.c} {56,1}-{113,1}]
 1.027%  apply_local_potential [{hamiltonian.py}{292}]
  1.02%  PyObject *Operator_relax(OperatorObject *, PyObject *) C [{operators.c} {49,1}-{81,1}]
 0.873%  void bc_unpack2(const boundary_conditions *, double *, int, MPI_Request *, MPI_Request *, double *, int) C [{bc.c
 0.793%  MPI_Comm_create()
 0.735%  PyObject *rk(PyObject *, PyObject *) C [{blas.c} {131,1}-{153,1}]
 0.708%  MPI_Scatter()
  0.65%  <module> [{__init__.py}{1}]
 0.597%  tri2full [{tools.py}{201}]
 0.594%  iterate_one_k_point [{rmm_diis.py}{32}]
 0.581%  void *apply_worker(void *) C [{operators.c} {99,1}-{143,1}]
 0.554%  get_boxes [{grid_descriptor.py}{211}]
```

Figure 3: The execution profile of GPAW obtained with the TAU performance system.

4.3. Module import mechanism

Python programs and libraries are typically separated into modules, which are taken into use via **import** statements. Large majority of imports are often performed when starting up the program, and just launching the Python interpreter without running any Python code involves importing of few standard modules. During the import statement, the content of the module (pure Python code or more often bytecode) is read from the disk. In normal parallel Python application, each MPI task launches its own Python interpreter and each task performs the same import statements. When hundreds or thousands of MPI tasks perform are trying to read the same files the filesystem will be overwhelmed and the time to perform the imports starts to increase drastically. Especially file metadata related operations (file open/close, time stamps, directory operations) can become a bottleneck, as the details of CPython's import implementation generate heavy metadata load. In Python 2.X series, a statement **import foo** triggers the following operations:

- In the current working directory, one tries to open file named **foo**. If the opening succeeds, it is checked if the file is actually a directory, and if that is the case, a file called **__init__.py** within the directory opened if it exists.
- If none of the above operations succeed, next opening of files **foo.so**, **foomodule.so**, **foo.py**, **foo.pyc** is tried. If both a **.py** and a **.pyc** exist, their modification times are compared.
- If none of the above operations succeed, the process is repeated for every directory in the PYTHONPATH and finally in the systems default module path until a proper file is found which is then read from the disk.

Thus, depending on the actual location of the modules and the length of PYTHONPATH the number of metadata operations per import can reach easily 20-30. GPAW together with NumPy contains several hundred Python modules, and when thousands of processes perform the same imports, it is clear that filesystems cannot handle the huge metadata load. As an example, on Blue Gene P just starting up Python and importing NumPy and GPAW with 32768 MPI tasks can take 45 minutes!

We have tried to address the import problem in two different ways. In Blue Gene P it is possible to create a ramdisk containing all the required Python modules, so that the I/O operations will be actually done from the memory instead of disk. However, the ramdisk complicates the installation of software considerably. In Cray XT ramdisk cannot be used, and we have experimented with the following approach, similar to the one presented in Ref. [15]: the CPython is modified in such a way that during import operations only a single process performs the actual I/O, and MPI is used for broadcasting the data to other MPI tasks. In practice this is achieved by creating a special header file which redefines C stdio routines. By including the special header in proper source files of CPython and linking in the wrapper routines containing the MPI aware stdio functions, only a minor modifications to the actual Python implementation is needed. Figure 4 shows how the "parallel" import mechanism keeps the import time modest compared to vastly increasing time with the standard interpreter. One limitation of the approach is that all the MPI tasks have to perform the same import statements.

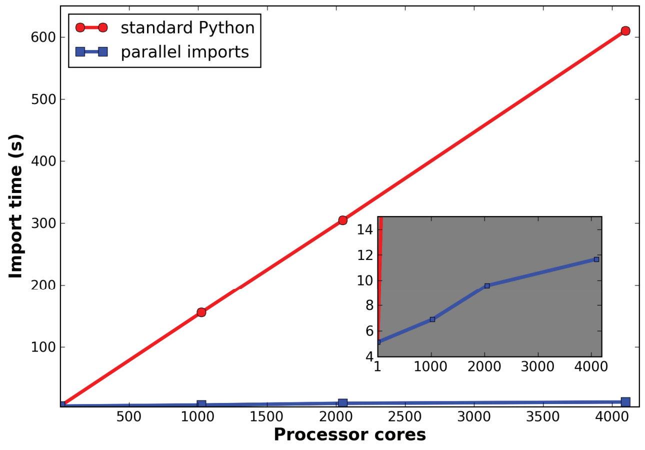


Figure 4: The reduction in Python startup time with "parallel" imports when importing NumPy and GPAW modules.

5. Performance

GPAW has parts whose computational intensity is constant with the input data size N, or increase as $O(N)$, $O(N^2)$, and $O(N^3)$. The performance overhead due to Python is most relevant in the constant and $O(N)$ terms, while the C-extensions and linear algebra libraries dominate the $O(N^2)$ and $O(N^3)$ parts. Typically, from medium size calculations up the Python overhead is less than 10 %. Especially in large scale calculations GPAW obtains very good floating performance, as an example on Cray XT5 a 2048 core GPAW calculation reaches 4.8 TFLOP/s which is 25 % of peak floating point performance.

Neglecting the I/O bottleneck in import statements, Python based implementation does not raise any special issue in parallel calculations. All the limitations of parallel scalability originate from the algorithms and general MPI considerations. As GPAW has been designed from the beginning to large scale parallel calculation, the parallel scalability is generally good as demonstrated in Figure 5.

6. Conclusions

GPAW is versatile and efficient software package for electronic structure calculations. While electronic structure simulation packages have traditionally been implemented using compiled languages, GPAW uses a more novel approach where the majority of the code is implemented in Python, and only the most time consuming parts have been implemented in C-extensions, or preferably in highly optimized libraries. NumPy package is crucial part of the approach, as it provides an efficient data structure for multidimensional arrays as well as efficient numerical operations for these arrays. Most importantly, NumPy allows easy transfer of array data between Python code and C-code.

The use of Python allows rapid application development for the most parts of the software. Since the numerical intensive kernels have been implemented in C, optimization of higher level algorithms and addition of new features can be programmed largely with Python.

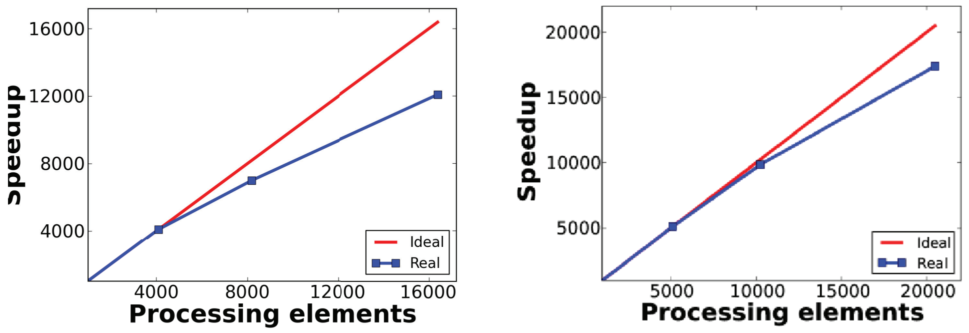


Figure 5: Parallel scaling of GPAW. a) ground state DFT calculation of 561 Au atom cluster (∼ 6200 electrons) on Blue Gene P. b) TD-DFT calculation of 702 atom Si cluster (∼ 2800 electrons) on Cray XT5.

Even though Python speeds up the development process, the approach is not without challenges, especially in the context of parallel calculations. The actual parallel programming can be performed efficiently by calling MPI routines either in C-extensions or in Python code, however, parallel debuggers and performance analysis tools support typically only C or Fortran, which can make debugging and parallel optimization more challenging. In our opinion the most severe issue is, however, the non-scalability of CPython's import mechanism, which triggers extremely high metadata load for the filesystem when using large number of processes. We have implemented workarounds for this problem, either by using ramdisk, or by using special MPI-enabled wrappers for C stdio-functions in selected parts of Python.

GPAW shows currently good parallel scalability up to thousands of CPU cores achieving at best ∼25 % of peak floating point performance. A future challenge is the increasing number of CPU cores per node, which is normally addressed by hybrid thread and message passing based parallelization. Whereas message passing can be used transparently by both on Python and C level, the global interpreter lock in CPython limits the thread based parallelization to the C-extensions only. We are currently investigating hybrid OpenMP/MPI implementation with the hope that limiting threading to only C-extension provides enough performance.

Acknowledgements

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

[1] J. J. Mortensen, L. B. Hansen, K. W. Jacobsen, Real-space grid implementation of the projector augmented wave method, Phys. Rev. B 71 (2005) 035109.
[2] J. Enkovaara, C. Rostgaard, J. J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. A. Hansen, H. H. Kristoffersen, M. Kuisma, A. H. Larsen, L. Lehtovaara, M. Ljungberg, O. Lopez-Acevedo, P. G. Moses, J. Ojanen, T. Olsen, V. Petzold, N. A. Romero, J. Stausholm-Møller, M. Strange, G. A. Tritsaris, M. Vanin, M. Walter, B. Hammer, H. Häkkinen, G. K. H. Madsen, R. M. Nieminen, J. K. Nørskov, M. Puska, T. T. Rantala, J. Schiøtz, K. S. Thygesen, K. W. Jacobsen, Electronic structure calculations with gpaw: a real-space implementation of the projector augmented-wave method, Journal of Physics: Condensed Matter 22 (25) (2010) 253202.
URL `http://stacks.iop.org/0953-8984/22/i=25/a=253202`
[3] https://wiki.fysik.dtu.dk/gpaw.
[4] T. E. Oliphant, Guide to NumPy, Provo, UT (Mar. 2006).
URL `http://www.tramy.us/`
[5] http://numpy.scipy.org/.
[6] P. Hohenberg, W. Kohn, Inhomogeneous electron gas, Phys. Rev. 136 (3B) (1964) B864–B871. doi:10.1103/PhysRev.136.B864.
[7] W. Kohn, L. J. Sham, Self-consistent equations including exchange and correlation effects, Phys. Rev. 140 (4A) (1965) A1133–A1138. doi:10.1103/PhysRev.140.A1133.
[8] E. Runge, E. K. U. Gross, Density-functional theory for time-dependent systems, Phys. Rev. Lett. 52 (1984) 997 – 1000.
[9] P. E. Blöchl, Projector augmented-wave method, Phys. Rev. B 50 (24) (1994) 17953–17979. doi:10.1103/PhysRevB.50.17953.

[10] P. E. Blöchl, C. J. Först, J. Schimpl, The projector augmented wave method: ab-initio molecular dynamics with full wave functions, Bull. Mat. Sci. 26 (2003) 33.

[11] http://www.mpi-forum.org/.

[12] http://mpi4py.scipy.org/.

[13] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[14] S. Shende, A. D. Malony, The tau parallel performance system, International Journal of High Performance Computing Applications 20 (2006) 287.

[15] D. M. Beazley, P. Lomdahl, Feeding a large-scale physics application to python, in: 6th International Python Conference, USENIX, 1997.