Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 108C (2017) 2171–2179

Procedia Computer Science

International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland

Performance Analysis of Parallel Python Applications

Michael Wagner[1], Germán Llort[1,2], Estanislao Mercadal[1,2], Judit Giménez[1,2], and Jesús Labarta[1,2]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
michael.wagner@bsc.es
[2] Universitat Politècnica de Catalunya (UPC), Barcelona, Spain

Abstract

Python is progressively consolidating itself within the HPC community with its simple syntax, large standard library, and powerful third-party libraries for scientific computing that are especially attractive to domain scientists. Despite Python lowering the bar for accessing parallel computing, utilizing the capacities of HPC systems efficiently remains a challenging task, after all. Yet, at the moment only few supporting tools exist and provide merely basic information in the form of summarized profile data. In this paper, we present our efforts in developing event-based tracing support for Python within the performance monitor Extrae to provide detailed information and enable a profound performance analysis. We present concepts to record the complete communication behavior as well as to capture entry and exit of functions in Python to provide the according application context. We evaluate our implementation in Extrae by analyzing the well-established electronic structure simulation package GPAW and demonstrate that the recorded traces provide equivalent information as for traditional C or Fortran applications and, therefore, offering the same profound analysis capabilities now for Python, as well.

Peer-review under responsibility of the scientific committee of the International Conference on Computational Science

Keywords: Performance Analysis, Tracing, Tools, HPC, Parallel, Python, Extrae, Paraver

1 Introduction

While the HPC landscape of programming languages is essentially monopolized by Fortran and C, other programming languages that are popular outside the HPC community are striving to gain ground. Python is a widely-used, high-level programming language with the self-proclaimed goal to allow fast and easy program development. Among others, it features are a simple syntax, dynamic data types, powerful data structures and a large standard library. Combined with third party libraries for scientific computing, such as NumPy [16] and SciPy [14], it is well comprehensible that Python is especially attractive to domain scientists.

For parallel computing with Python exist, among others, packages from the Python standard library like the *multiprocessing* module and external Python interfaces to parallel runtimes

1877-0509
Peer-review under responsibility of the scientific committee of the International Conference on Computational Science
10.1016/j.procs.2017.05.203

like MPI4Py [9] for message passing. MPI4Py provides bindings of MPI [12] with an object-oriented interface similar to C++ and allows the communication of arbitrary Python objects. In addition, MPI4Py supports optimized communication of NumPy arrays with a speed close to that of communication directly in C or Fortran.

Combining the benefits of Python (e.g. fast development and high-level algorithms) with an implementation of performance critical parts in C (e.g. main numerical kernels with BLAS, LAPACK, NumPy; and MPI communication with MPI4Py) has been the most successful approach in HPC computing, so far. For instance, GPAW [10] has demonstrated about 25 % peak performance and good scaling up to tens of thousands of cores [11].

Despite the fact that Python is consolidating itself more and more in the HPC world, suitable tool support is still scarce. Currently, for the purpose of performance analysis there exists some basic profiling support in the form of summarized function-level or line-level timing information provided, for instance, by the Python modules *profile* and *cProfile* [6]. However, more advanced tool support, such as event-based tracing, is still missing. In contrast to summarized profiling, event-based tracing records runtime events, such as entering/leaving a function or communication operations, individually. As a result, while profiles may lack crucial information and hide dynamically occurring effects, event-based tracing allows capturing the dynamic interaction between thousands of concurrent processing elements and enables the identification of outliers from the regular behavior. Thus, event-based tracing allows a more detailed and profound analysis and assists developers not only in identifying performance issues within their applications but also in understanding their behavior on the complex and increasingly heterogeneous HPC systems.

In this paper, we share our efforts in developing event-based tracing support for Python within the performance monitor Extrae [2]. Our contributions in this work are, first, concepts to capture entry and exit of functions in Python; second, a method to record the complete communication behavior for Python applications using either MPI4Py or custom MPI bindings; and, third, we demonstrate the capabilities of our prototype implementation with GPAW [10, 4], a well-establish software package for electronic structure simulations, which is implemented in Python and C and massively parallelized with MPI.

The remainder of the paper is structured as followed. In Section 2 we introduce related research and distinguish our work. In Section 3 we highlight the concepts and implementation to record function entry/exit and communication events. In Section 4 we evaluate our current implementation and show its capabilities for a performance analysis of parallel Python applications. Finally, we conclude our work in Section 5.

2 Related Work

The most commonly used method to generate basic summarized information are the Python modules *profile* and *cProfile* in combination with *pstats* [6]. *Profile* and *cProfile* provide statistics for accumulated duration and number of invocations for various parts of the program. Both export the same information and are mostly interchangeable; with the main difference being that *cProfile* is a C extension with less overhead but also less compatibility. A function can be profiled by calling `cProfile.run(<function>)` instead of `<function>` within any Python script that imports the profiling module. *CProfile* can also be invoked as a script to profile another script by adding `-m cProfile` to the Python command. The generated statistics can be formatted into simple text reports via the *pstats* module. In a parallel execution, the output is generated for each process and the output is intermingled, which requires some additional post-processing to provide meaningful results.

The TAU performance system allows to profile a parallel Python application and produces a similar results to *cProfile* but with a GUI representation [11]. Additionally, the authors discuss some further problems that arise for running and profiling parallel Python applications, e.g., the fact that each module is imported redundantly for each process creating a remarkable overhead in the start-up time. Unfortunately, the authors only mention the use of TAU without details on how the performance information was collected.

Next to these, the commercial tools Allinea MAP [8] and Intel's Vtune [13] support mixed Python and C/Fortran applications. Allinea MAP provides basic system information over time, e.g., CPU and memory utilization; which can be accessed without specific support for Python. Intel VTune's summarized information is based on periodic sampling for mixed Python and C/Fortran applications but they mention some limitations in accuracy and the collection of Python data. Nevertheless, we were unable to locate further details on the available Python support for these tools at the moment of writing this work.

All the above mentioned tools have in common that they provide basic summarized profiling information. In contrast, we present an event-based tracing approach that allows a detailed and profound analysis, in particular, of the parallel behavior and inter-process dependencies. In addition, we describe the methods to acquire the performance data, which is missing in previous works. Furthermore, we evaluate and discuss the analysis capabilities for a well-known, massively parallel application. To the best of our knowledge, this has not been previously done.

3 Implementation

In this section we discuss the extensions developed for the Extrae instrumentation package [2] to support event-based tracing of function entry/exit and communication events for Python. Extrae is an open-source tracing framework that provides instrumentation and sampling mechanisms to collect performance measurements from the most common parallel programming models automatically (e.g. MPI, OpenMP, POSIX threads, etc.). The information captured by Extrae typically includes the activity of the parallel runtime (e.g. message exchanges in MPI), as well as performance counters and call-stack information to correlate the measurements with the actual source code. Furthermore it is possible to manually or automatically instrument source code functions. Likewise, the two main targets for event-based monitoring of Python applications is, first, to capture the calls to the parallel runtime and, second, to provide the according source code context.

3.1 Instrumentation of MPI

Event-based tracing for conventional (C or Fortran) MPI applications is a well-known technique supported by most performance analysis tools. The most common approach relies on the MPI standard profiling interface (PMPI) [12]. Each MPI function can be called with an MPI_ or PMPI_ prefix, which allows the tools to intercept the program's calls to MPI by rewriting the functions with the MPI_ prefix. The new functions can capture performance data and then perform the according message-passing operation by calling the associated PMPI function.

A convenient method for the function replacement is bundling all the new *wrapper* functions together in a shared library to be loaded at runtime. This enables replacing the MPI calls from a dynamically linked binary without having access to the source code nor having to relink. This can be primarily effected by setting the `LD_PRELOAD` environment variable to the path of the shared object that redefines the MPI symbols. This library will be loaded before any other library, replacing in turn the original implementation from the MPI runtime. Overriding

standard library's functions with your own version of these functions through the `LD_PRELOAD` environment variable is supported in most UNIX systems, which makes it a method of choice to perform symbol substitutions transparently. Moreover, it allows tracing of MPI from Python programs out-of-the-box since MPI-for-Python packages like MPI4Py [9] provide C-bindings of the MPI standard to an external shared object that can be dynamically interposed.

This enables to capture entry and exit of MPI calls in Python scripts or mixed Python + C/Fortran scenarios in an analogous manner as for pure C or Fortran codes along with typical performance characteristics. However, pure MPI instrumentation provides only limited information due to the lack of application context. Therefore, the second target is to complement the MPI information with source code instrumentation.

3.2 Instrumentation of Source Code Context

A common approach for tools to provide application context is to store the native call-stack (or parts of it) to attribute the performance to the source code. However, this approach can not be applied in the context of an interpreted execution, as backtraces of the native call-stack do not refer to the actual script's execution flow but to the interpreter's parsing process. Another possibility to add context to the trace is to instrument the (main) functions of the program. There are many approaches to this problem, ranging from source-to-source compilers, automatic compiler instrumentation with `-finstrument-functions`, binary rewriting or dynamic injection of monitoring probes. Yet again, these methods can not be applied on interpreted executions as the target program is not the actual script but the Python interpreter.

In order to support tracing of function entry/exit in Python we rely on the *sys* module. This module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter, including profiling capabilities [5]. The `sys.setprofile` method allows to implement a Python source code profiler within Python by setting a callback routine that is invoked on call and return events of the scripts' functions. It receives three arguments: *frame*, *event* and *arg*, which identify, respectively, the current stack frame, whether it is an entry or exit event, and additional arguments. This enables keeping track of the functions called within the script, and store the activity in the trace.

Next to that, in a mixed Python + C/Fortran execution it is furthermore possible to record the source code context from the C/Fortran parts via already existing means, e.g. store the calling context of MPI calls via stack unwinding methods.

3.3 The Extrae Python Module

To support capturing MPI events, user functions, and both combined we developed a new Python module that can be directly imported into the user's Python script. This module conveniently bundles the three main requirements to facilitate the trace recording. First, it provides dynamic interposition of the tracing library based on *ctypes*. *Ctypes* is a foreign function library for Python that provides C compatible data types and allows calling functions in shared libraries [1]. *Ctypes* exports the `cdll` object for loading dynamic link libraries, which allows hooking a tracing library through the `cdll.LoadLibrary` method, with an effect comparable to pre-loading the library into a binary executable. With this we can load the Extrae library that intercepts the MPI library to capture the calls to this runtime.

Second, it activates the Python profiler to track the user functions that are called, allowing to selectively record a subset of the functions during the execution of the Python script. This way, the Extrae Python module enables monitoring the function calls without any source code modification by simply providing a list of function names in a plain text file.

Third, it provides an interface that allows to pass information from the Python profiler to the tracing library. We rely on Extrae's API to emit punctual, user-defined events in the trace (i.e. `Extrae_event` method). We developed bindings that make the API calls available in Python, which are called from within the profiler callback to store function entry and exit events. This way, we can provide user function information to complement MPI communication activity with the application context. In addition, the interface allows manual instrumentation of further user events, e.g. iterations, phases, or execution states.

3.4 Other Usage Scenarios

While MPI-for-Python packages are widely used to develop parallel codes, this is not the only approach. It is also very frequent to use the *multiprocessing* package, which supports spawning processes and offers local and remote concurrency. Tracing of Python programs based on multiprocessing's *Process* module is also supported through the mechanisms explained above. We use the system profiler to track the spawning of processes started from the *multiprocessing* module. Whenever we detect calls to a subprocess with a new *process ID* (i.e. PID), we interpose the tracing library on the new process through `cdll`. This way, we can record the process' life-span by emitting events through the Extrae API. The support for programs using multiprocessing's *Pool* module is currently still under development. Finally, serial executions are seamlessly supported through Extrae's python module by transparently loading the tracing library and recording of function calls and user-defined punctual events.

4 Evaluation

In this section we evaluate the new Python support with the well-established software package for electronic structure simulations GPAW [10, 4]. GPAW is implemented in Python and C and is parallelized with MPI. We consider this a well-suited example since it has demonstrated about 25 % peak performance and good scaling up to tens of thousands of cores [11].

We recorded a basic example from the manual [3] of a calculation for a single H_2 molecule using a supercell of size $6.0 \times 6.0 \times 6.0$ Å, i.e., $32 \times 32 \times 32$ grid points. We used GPAW in version 1.1.0 with Python 2.7 running in MareNostrum [7], an HPC system at Barcelona Supercomputing Center (BSC) based on Intel SandyBridge processors. For the recording we used Extrae 3.4.3 [2] that adds the Python support.

Ahead of the actual measurement run, we execute the script with the *profile* module enabled to get an overview of the most common Python functions in the script. Out of these we select the functions with the longest accumulated duration. The list with the function names is passed to Extrae, which intercepts the entry/exit of the functions to add them to the trace.

During the recording we also tracked the additional overhead that is introduced by the trace recording. For this measurement we observed a total overhead of about 5 %. Obviously, the overhead depends strongly on the granularity of the recorded functions. For the instrumentation of function calls we found an average overhead 4.6 us per entry/exit event, i.e. 9.2 us per function call. This overhead can be broken down in 2.4 us originating from the Python profiler, 2.0 us for the collection of eight hardware performance counters from PAPI, and 0.2 us for the tracing library. Recording MPI events shows equivalent costs regarding the access to hardware performance counters and the emission of events to the tracing library.

Figure 1 shows screenshots of the analysis of the resulting trace with the performance analyzer Paraver [15]. Paraver presents performance information with two main displays that provide qualitatively different types of information. The timeline displays represent the behavior

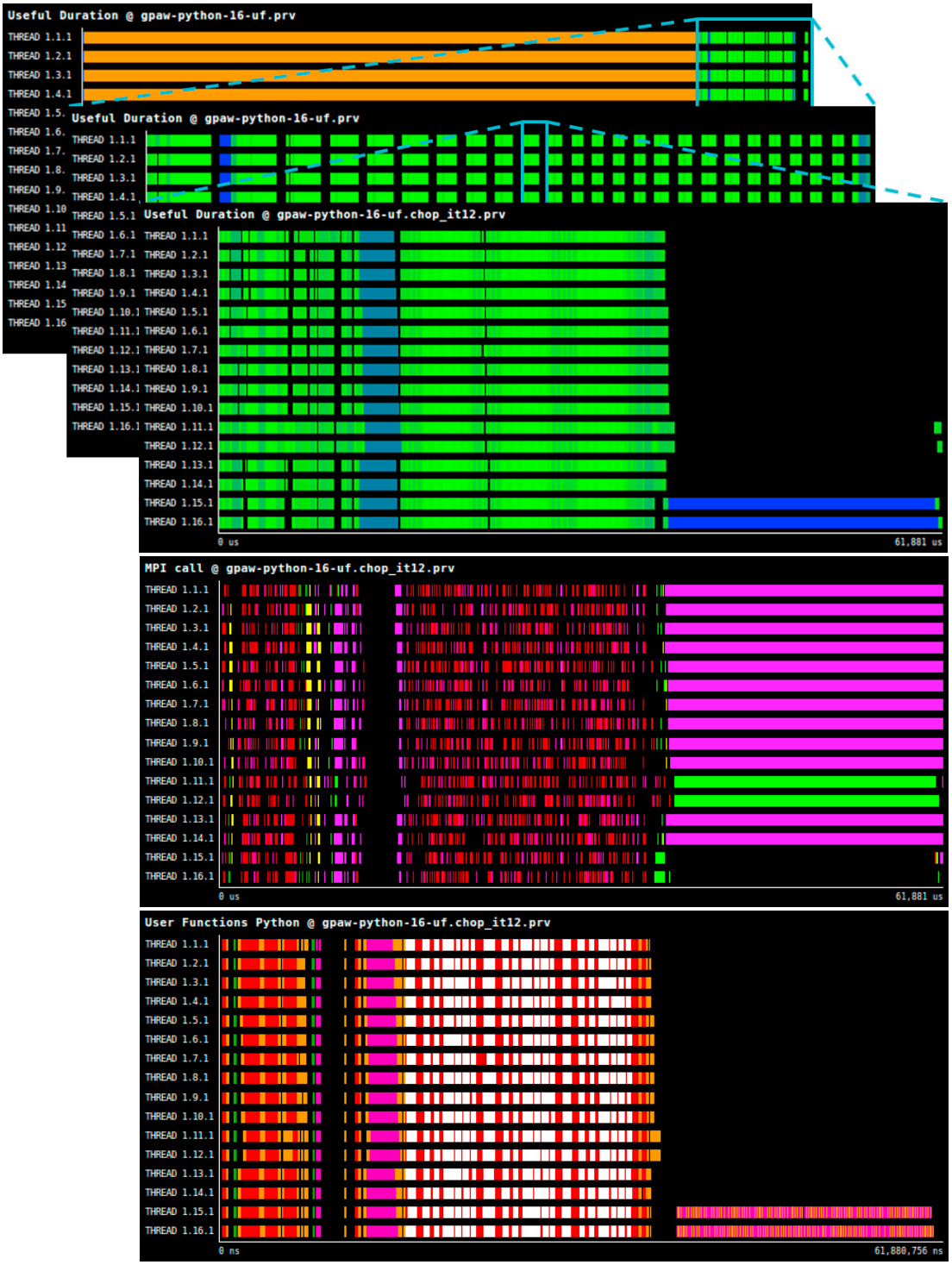

Figure 1: Analysis of the H2 molecule example with Paraver. Top: Useful duration for the entire application (background), a zoom to the iterative phase, and a zoom into the 12th iteration. Center: Display of the MPI activity. Bottom: Timeline of Python user functions.

of an application along time and processes/threads and provides a general understanding of the application behavior and simple identification of phases and patterns. The statistics display, on the other hand, allows a numerical analysis of the data that can be applied to any user selected region, helping to draw conclusions on where and how to focus the optimization effort.

The timelines in Figure 1 highlight different behavioral aspects evolving over time in a two-dimensional chart. The vertical axis includes the executing processes; the horizontal axis represents the runtime of the recorded application. At the top, three screenshots depict the metric *Useful Duration*, i.e., time spent for computation outside of the parallel runtime; whereas the color gradient from green to blue represents the length of each compute phase from short to long, respectively; black marks time outside of useful computation, i.e., time in the parallel runtime. The three screenshot visualize the application behavior for the entire application (background), a zoom to the iterative phase (middle), and a zoom into the 12th iteration (front). It can be seen that there is a relatively long start-up time (orange), which is overrepresented in this case, since the actual computation phase for the H_2 example is rather short. The iterative phase shows 26 iterative steps before the simulation converged. The zoom to the 12th iteration reveals three main phases within each iteration; each about one third of time. The second phase contains another three nested iterations running the Poisson solver, while the third phase shows a huge imbalance due to only the last two processes are computing, which reduced the overall load balance within the iteration to 58 %.

The timeline in the center shows the MPI communication, whereas the different colors represent different MPI calls. The first and second phase exhibit multiple non-blocking point-to-point communication (red = MPI_Isend, pink = MPI_Irecv, dark red = MPI_Wait) while the last phase is mainly spent in the synchronizing calls to MPI_Allreduce (pink) and MPI_Waitall (light green). Both calls last very long since all processes have to wait for the last two processes to finish their computation. In total the iteration spends on average only 42 % in computation and the rest in communication, with the MPI_Allreduce covering about 31 % followed by MPI_Wait with 12 % and MPI_Isend and MPI_Irecv with each 4 % (all average values).

The last timeline highlights the recorded Python functions with the first phase prevailed by calls to *apply* (red) and *integrate* (orange), the second phase dominated by calls to *relax* (white) and *apply* (red), and in the third phase mainly calls to *calculate* (pink) and *integrate* (orange). Based on this information, general statistics like accumulated function duration similar to cProfile can be computed.

Next to that, we recorded hardware performance counters to characterize, for instance, the compute intensity with the derived metric instructions per cycle (IPC) and the communication data rates. Figure 2 shows two examples of correlating all three types of information (MPI, user functions, and hardware performance counters) with each other. The statistics display (histogram) on the left highlights the number of instructions per cycle (IPC) per process and function, which indicates how efficiently the different functions compute results. Thereby, the IPC is considered only for computation phases within the functions, i.e. outside of MPI calls. In this case, the function *calculate* shows the best compute performance with an average IPC of 1.5; others exhibit performance of drastically lower IPC, e.g. *iterate2*. In addition, it can be seen that *iterate2* shows a high variability in the achieved IPC with a balance of only 72 %. The histogram on the right correlates the user functions with the MPI activity; in this example for the function *iterate2*. It can be seen that, first, the function spends on average 39 % of its time in *MPI_Allreduce* and, second, that the distribution of computation and communication shows a large variability. In fact, the variability of the computation time is a direct result of the imbalance in IPC discussed above since the number of executed instructions (not shown) is almost perfectly balanced within the function.

	relax	apply	calculate	iterate2	gemm	integrate
THREAD 1.1.1	0.53	0.61	1.59	0.31	0.93	0.55
THREAD 1.2.1	0.55	0.62	1.46	0.33	0.93	0.57
THREAD 1.3.1	0.55	0.62	1.49	0.33	0.91	0.56
THREAD 1.4.1	0.56	0.64	1.50	0.36	0.94	0.58
THREAD 1.5.1	0.61	0.69	1.50	0.44	0.95	0.66
THREAD 1.6.1	0.63	0.71	1.50	0.49	0.95	0.70
THREAD 1.7.1	0.64	0.71	1.50	0.31	0.94	0.69
THREAD 1.8.1	0.65	0.73	1.50	0.54	0.95	0.73
THREAD 1.9.1	0.61	0.70	1.48	0.49	0.95	0.70
THREAD 1.10.1	0.63	0.72	1.50	0.57	0.96	0.72
THREAD 1.11.1	0.63	0.77	1.49	0.55	0.94	0.76
THREAD 1.12.1	0.65	0.80	1.50	0.62	0.95	0.78
THREAD 1.13.1	0.56	0.63	1.48	0.40	0.97	0.58
THREAD 1.14.1	0.58	0.65	1.49	0.43	0.93	0.62
THREAD 1.15.1	0.58	0.65	1.30	0.43	0.95	0.63
THREAD 1.16.1	0.59	0.67	1.29	0.47	0.93	0.64
Total	9.53	10.91	23.57	7.07	15.06	10.48
Average	0.60	0.68	1.47	0.44	0.94	0.66
Maximum	0.65	0.80	1.59	0.62	0.97	0.78
Minimum	0.53	0.61	1.29	0.31	0.91	0.55
StDev	0.04	0.05	0.07	0.09	0.01	0.07
Avg/Max	0.92	0.86	0.93	0.72	0.97	0.84

	Outside MPI	MPI_Allreduce
THREAD 1.1.1	42.45 %	57.55 %
THREAD 1.2.1	45.46 %	54.54 %
THREAD 1.3.1	46.28 %	53.72 %
THREAD 1.4.1	48.54 %	51.46 %
THREAD 1.5.1	60.61 %	39.39 %
THREAD 1.6.1	66.11 %	33.89 %
THREAD 1.7.1	42.69 %	57.31 %
THREAD 1.8.1	74.08 %	25.92 %
THREAD 1.9.1	67.88 %	32.12 %
THREAD 1.10.1	81.92 %	18.08 %
THREAD 1.11.1	74.58 %	25.42 %
THREAD 1.12.1	85.27 %	14.73 %
THREAD 1.13.1	54.13 %	45.87 %
THREAD 1.14.1	60.37 %	39.63 %
THREAD 1.15.1	60.61 %	39.39 %
THREAD 1.16.1	67.61 %	32.39 %
Total	978.58 %	621.42 %
Average	61.16 %	38.84 %
Maximum	85.27 %	57.55 %
Minimum	42.45 %	14.73 %
StDev	13.32 %	13.32 %
Avg/Max	0.72	0.67

Figure 2: Histograms showing the IPC per process and function (left) and the time spent in computation and communication in the function *iterate2*.

While the given screenshots can only hint at the analysis capabilities provided by the recorded traces, they demonstrate that the Python support in Extrae provides equivalent information as for traditional C or Fortran applications. The recorded information allows to, first, analyze and understand the behavior of the application over time, second, use the statistical tools within Paraver to quantify the performance behavior and potential issues like compute intensity or load balance, and, third, correlate the identified behavior and issues to the source code location within the Python scripts.

5 Conclusions

Python, already a widely-used programming language, is becoming more and more popular within the HPC community since its simple syntax, large standard library, as well as powerful third-party libraries for scientific computing make it very attractive, especially, to domain scientists. Despite Python lowering the bar for accessing parallel computing, utilizing the capacities of HPC systems efficiently remains a challenging task, after all.

Consequently, support from appropriate tools like performance analyzers is inevitable, yet, currently the few tools that exist only offer basic feedback in the form of summarized profile data. In this paper, we share our efforts in developing event-based tracing support for Python within the performance monitor Extrae to provide detailed information and enable a profound performance analysis. We explored common methods to instrument the MPI runtime for C/Fortran applications and applied analogous mechanisms relying on library pre-loading and dynamic linking to make information from the parallel runtime available for Python programs. In order to complement this information with the according application context, we developed selec-

tive user function instrumentation based on Python's system profiler and provide C-bindings to transfer the data to an external tracing library. We created a Python module that can be imported into any Python script to conveniently incorporate this functionality.

We evaluate our implementation by analyzing the electronic structure simulation package GPAW. The recorded information allows to analyze and understand the behavior of the application, identify performance issues, and correlate them to the source code. We demonstrate that the recorded traces provide equivalent information as for traditional C or Fortran applications and, therefore, offering the same profound analysis capabilities now for Python, as well.

References

[1] ctypes - A foreign function library for Python. https://docs.python.org/2/library/ctypes.html [Online; accessed 2017-01-01].

[2] Extrae instrumentation package. http://tools.bsc.es/extrae [Online; accessed 2017-02-07].

[3] GPAW Manual. https://wiki.fysik.dtu.dk/gpaw/documentation/manual.html [Online; accessed 2017-02-08].

[4] GPAW website. https://wiki.fysik.dtu.dk/gpaw [Online; accessed 2017-02-06].

[5] sys - System-specific parameters and functions. https://docs.python.org/2/library/sys.html [Online; accessed 2017-01-01].

[6] The Python Profilers. https://docs.python.org/2/library/profile.html [Online; accessed 2017-02-07].

[7] MareNostrum III User's Guide, 2016. https://www.bsc.es/support/MareNostrum3-ug.pdf.

[8] Allinea. How to debug and profile those mixed Python and Fortran codes. https://www.allinea.com/blog/201502/how-to-debug-and-profile-mixed-python-and-fortran-codes [Online; accessed 2017-02-07].

[9] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[10] J Enkovaara, C Rostgaard, J J Mortensen, J Chen, M Duak, 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, and 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):253202, 2010.

[11] Jussi Enkovaara, Nichols A. Romero, Sameer Shende, and Jens J. Mortensen. GPAW - Massively Parallel Electronic Structure Calculations with Python-based Software. *Procedia Computer Science*, 4:17–25, 2011.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1, 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[13] Intel. A Performance Analysis of Python* Applications with Intel® VTune™ Amplifier. https://software.intel.com/en-us/videos/performance-analysis-of-python-applications-with-intel-vtune-amplifier [Online; accessed 2017-02-07].

[14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2017. http://www.scipy.org [Online; accessed 2017-02-06].

[15] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A Tool to Visualize and Analyze Parallel Code. *Transputer and occam Developments*, pages 17–32, 1995. https://tools.bsc.es/paraver.

[16] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.