Advanced Python Performance Monitoring with Score-P

Andreas Gocht [ID], Robert Schöne, and Jan Frenzel

Abstract Within the last years, Python became more prominent in the scientific community and is now used for simulations, machine learning, and data analysis. All these tasks profit from additional compute power offered by parallelism and offloading. In the domain of High Performance Computing (HPC), we can look back to decades of experience exploiting different levels of parallelism on the core, node or inter-node level, as well as utilising accelerators. By using performance analysis tools to investigate all these levels of parallelism, we can tune applications for unprecedented performance. Unfortunately, standard Python performance analysis tools cannot cope with highly parallel programs. Since the development of such software is complex and error-prone, we demonstrate an easy-to-use solution based on an existing tool infrastructure for performance analysis. In this paper, we describe how to apply the established instrumentation framework Score-P to trace Python applications. We finish with a study of the overhead that users can expect for instrumenting their applications.

1 Introduction

Python is one of the Top 5 programming languages,[1] and it is not surprising that more and more scientific software is written in Python. But the standard implementation CPython interprets Python source code, rather than compiling it. Hence, it is deemed to be less performant than other programming languages like C or C++. Moreover, as

---

[1]According to the TIOBE Index Oktober 2019: https://www.tiobe.com/tiobe-index/.

A. Gocht (✉) · R. Schöne · J. Frenzel
Center for Information Services and High Performance Computing (ZIH),
Technische Universität Dresden, 01062 Dresden, Germany
e-mail: andreas.gocht@tu-dresden.de

R. Schöne
e-mail: robert.schoene@tu-dresden.de

J. Frenzel
e-mail: jan.frenzel@tu-dresden.de

CPython employs a Global Interpreter Lock (GIL) [1], it is often stated that Python does not support parallelism. While there are different Python implementations like pypy[2] or IronPython,[3] which try to counter these drawbacks, these approaches do not represent the standard implementation.

However, CPython is easily extensible, e.g., by using its C-API or foreign function interfaces. These interfaces allow programmers to exploit the parallelism of a problem with traditional programming languages like C without losing the flexibility and the power of the standard Python implementation. Moreover, it is possible to offload computation to accelerators like graphic cards. Nevertheless, these extensions and the Python source code itself need to be optimised to exploit the full performance of a computing system. To optimize the application, it has to be monitored. To monitor the application, performance-related information has to be collected and recorded.

While collecting performance information is possible to some extent with tools that are part of the standard Python installation, none of these tools makes it easy to gain knowledge about the efficiency of thread parallel, process parallel, and accelerator-supported workloads. However, such tools exist for traditional programming languages used in High Performance Computing (HPC). Here, Score-P [2], Extrae [3], TAU [4], and others allow users to record the performance of their applications and analyze them with scalable interfaces.

In this paper, we present the Python bindings for Score-P, which make it easy for users to trace and profile[4] their Python applications, including the usage of (multi-threaded) libraries, MPI parallelism and accelerator usage. The paper is structured as follows: We describe our concept and implementation in Sect. 2 and evaluate the overhead in Sect. 3. We present related work in Sect. 4 and finalize this paper with a conclusion and an outlook in Sect. 5.

2 The Score-P Python Bindings

The Python module, which is used to invoke Score-P and allows tracing and profiling of Python code, is called *Score-P Python bindings*. The module can be split into three basic blocks, which are used in two phases: The *initialisation*, which is executed in a preparation phase, prepares the measurement and executes the application. The *instrumenter* is registered with the Python instrumentation hooks and used during execution. The *Score-P C-bindings* connect Python with C and Score-P and are also used during execution. The workflow of the overall process, including preparation phase and the execution phase, is depicted in Fig. 1.

---

[2]https://pypy.org/.

[3]https://ironpython.net/.
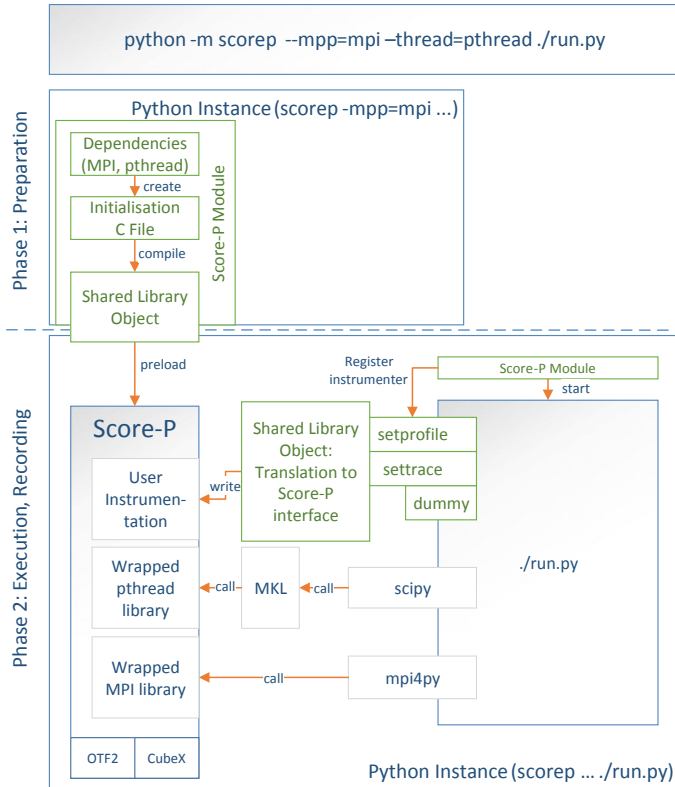
[4]As defined in [5, Sect. 2].

Fig. 1 Overview of the instrumentation process with Score-P. In the first phase, the Score-P Python module initializes the Score-P measurement system and attaches Score-P libraries. In the second phase, the bindings use the preloaded Score-P libraries and instrument the Python code to record events with Score-P. In addition to the Python instrumentation, other parts of the application, such as MPI, pthreads, and CUDA functions, are automatically instrumented by Score-P (not depicted)

2.1 Preparation Phase

Since version 2.5, Python allows running modules as scripts [6]. This approach can be used to record traces of a Python application. Instead of starting the Python application directly, the script and its parameters are passed as arguments to the Score-P Python module. The recording can be configured by prefixing additional parameters to the parameter specifying the original Python application. An example is given in Listing 1.

In the first step, all Score-P related parameters are parsed. Score-P supports a variety of programming models like OpenMP, MPI, and CUDA. However, increasing the monitoring detail leads to more information in a trace or profile but also to a higher instrumentation overhead. Therefore, we allow the user to choose which functionality should be monitored. Based on the chosen features, the Score-P ini-

```
1 # mpirun -n 2         -> run two parallel MPI processes
2 # python              -> each of these runs python
3 # -m scorep           -> run 'scorep' module before
     actual script
4 # --mpp=mpi --thr...   -> use MPI & pthread
     instrumentation
5 # ./run.py            -> the script or application to
     run
6 # -app-arg            -> an argument to ./run.py
7 mpirun -n 2 \
8 python -m scorep --mpp=mpi --thread=pthread ./run.py -
     app-arg
```

Listing 1 Calling an MPI-parallel application using the Score-P Python bindings

tialisation code is generated. This code is then compiled and added together with some dependencies to the LD_PRELOAD environment variable. As LD_PRELOAD is evaluated by the linker, the whole Python interpreter needs to be restarted, which is done using os.execve() [7].

Once restarted, the module starts the second step: The instrumenter is created, and the arguments, which are succeeding the Score-P arguments are utilised. The first non-Score-P argument is the Python application that shall be executed, followed by its arguments. The Python application is read, compiled, and executed [8], and its arguments are passed to the application.

2.2 Execution Phase

As described before, the execution phase uses two different software parts: the instrumenter and the Score-P C-bindings that hand over the events from the instrumenter to Score-P.

2.2.1 The Instrumenter

The instrumenter represents a component that is registered with CPython and supposed to be called for specific events during the execution of an application. Python offers two registration alternatives for such callback functions: sys.settrace() and sys.setprofile() [9]. However, different events are raised and forwarded to the instrumenter depending on which of these functions is used. A summary of these events is shown Table 1. Obviously, both functions can be used to instrument function calls, but both also offer different functionality. While sys.setprofile() can be used to trace also calls to C-functions, sys.settrace() can be used to record lines of code or operations executed.

Table 1 Supported events for Python profiling/debugging interfaces

Event	Description	Supported by `sys_set…`	
		`…profile()`	`…trace()`
Call	A function is called	✓	✓
Return	A code block (e.g., a function) is about to return	✓	✓
C_call	A C function is about to be called	✓	✗
C_return	A C function has returned	✓	✗
C_exception	A C function has raised an exception	✓	✗
Line	The interpreter is about to a new line of code or re-execute the condition of a loop	✗	✓
Exception	An [Python] exception has occurred	✗	✓
Opcode	The interpreter is about to execute a new opcode	✗	✓

Please note that *tracing* has different meanings in the Python documentation and in the HPC community. In the former, tracing describes the investigation of per line execution of the source code, which can be used to implement debuggers [9]. In contrast, the HPC community understands tracing as the recording of events like entering or exiting a region over time [5]. In this paper, we use the term tracing for the HPC notion of tracing. If we refer to the python notion of tracing we use `sys.settrace()`.

However, for each callback, `sys.settrace()` and `sys.setprofile()`, Python also issues the Python frame causing the event and some additional arguments. The Python frame holds information like the current line number of the associated module. The instrumenter passes this information to the Score-P C-bindings.

2.2.2 Score-P C-bindings

The *Score-P C-bindings* between Python and Score-P use the Python C-interface [10] and the user instrumentation from Score-P [11, Sect. J.1.2]. The bindings do not only forward events regarding entering or exiting of functions, but also group these functions based on their associated module. Moreover, they also pass information like line number or the path to the source file to Score-P. Score-P then uses these instrumentation events to create Cube4-profiles, OTF2-traces or to call substrate
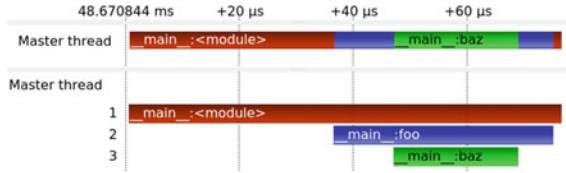
Fig. 2 Trace of a simple application using the Score-P Python bindings and Vampir. `__main__` indicates that the function is part of the currently run script
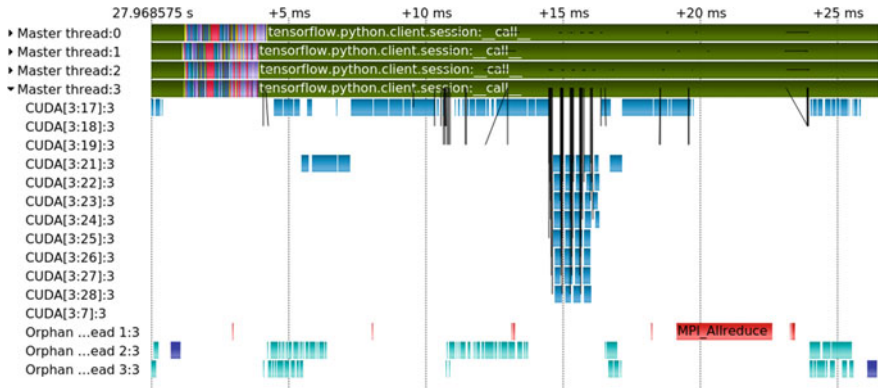

Fig. 3 Trace of a Python application [13] using CUDA and MPI. Traced using the Score-P Python bindings. Green are TensorFlow functions; red are MPI operations; blue are CUDA operations; black lines are CUDA communication

plugins for an online interpretation. Resulting traces can be viewed in Vampir [12], as shown in Fig. 2 for the small example code in Listing 2. A more complex parallel application is visualized in Fig. 3.

```python
def baz():
    print("Hello
    World")
def foo():
    baz()
if __name__ == \
    "__main__":
    foo()

```

Listing 2 Simple Python example

3   Performance Evaluation

To evaluate the overhead caused by the instrumentation, we designed two test cases. The first test case, shown in Listing 3, increments a value in a loop. We expect that the overhead introduced by the `sys.setprofile()` instrumenter does not depend on the number of iterations about this loop, since no functions are entered or exited. In contrast, we expect that the instrumenter using `sys.settrace()` causes an overhead depending on the iterations, since it is called for each executed line.

The second test case (Listing 4) uses a function to increment the value. Here, we expect a strong dependency on the number of iterations for both instrumenters.

```
1  import sys
2
3  result = 0
4
5  iterations = \
6      int(sys.argv[1])
7
8  iteration_list = \
9      list(range(iterations
       ))
10
11 for i in iteration_list:
12     result += 1
13
14 assert(result ==
       iterations)
```

Listing 3 Test case 1: loop only

```
1  import sys
2
3  def add(val):
4      return val + 1
5
6  result = 0
7  iterations = int(sys.argv
       [1])
8  iteration_list = \
9      list(range(iterations)
       )
10
11 for i in iteration_list:
12     result = add(result)
13
14 assert(result ==
       iterations)
```

Listing 4 Test case 2: function calls

We performed our experiments on the Haswell partition of the Taurus Cluster at TU Dresden. Each node is equipped with two Intel Xeon CPU E5-2680 v3 with 12 cores per CPU, and at least 64 GB of main memory per node [14]. Measurements are taken for each instrumenter, i.e. `sys.setprofile()` and `sys.settrace()`, as well as without the Score-P module, marked with `None`. Each experiment is repeated 51 times. The results are depicted in Fig. 4. We use linear interpolation to calculate the costs for (a) enabling instrumentation and (b) using the instrumentation. While the former includes setting up the Python environment and starting and finalizing Score-P, the latter represents the costs to execute one loop iteration. We disabled the Score-P measurement substrates profiling and tracing to represent only the overhead of instrumenting the code. The linear interpolation uses the median of each measurement and the `polyfit` function from numpy to create $t = \alpha + \beta N$ where t represents the runtime, N is the number of iterations, α is the one-time overhead for enabling the instrumentation and β is the cost per loop iteration. The results of this interpolation are presented in Table 2.

For the first test case (Fig. 4a), we see that the instrumentation cost is about 0.6 s. This cost will apply every time the instrumentation is enabled. Executing one
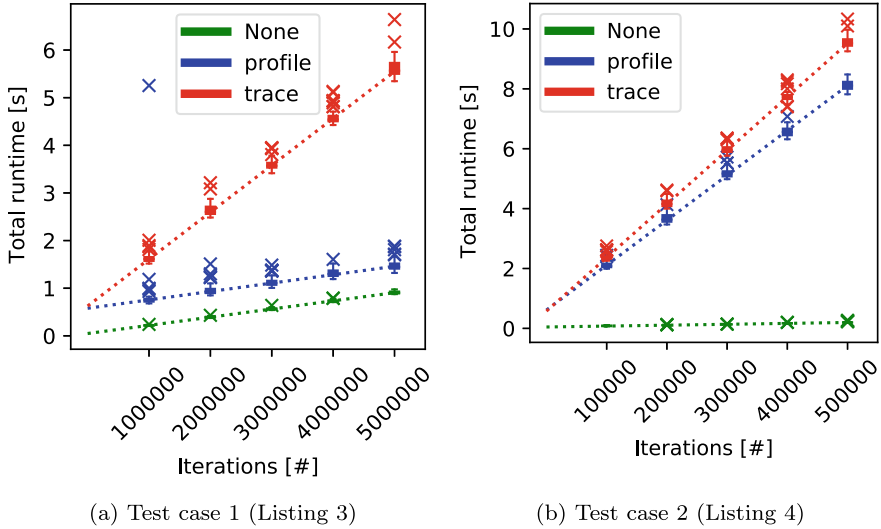
(a) Test case 1 (Listing 3)   (b) Test case 2 (Listing 4)

Fig. 4 Runtime of two instrumenters and non-instrumented code (`None`) for different test cases. Dotted lines represent a linear interpolation of the medians of each measurement point. The overhead for setting up the measurement and starting the Python environment is 0.6 s and independent of the instrumenter. Please note the different x-axis

Table 2 Overhead for test cases (median results): α: constant overhead; β: per loop iteration overhead

	Test case 1		Test case 2	
Instrumenter	α (s)	β (us)	α (s)	β (us)
None	0.05	0.17	0.05	0.3
sys.setprofile()	0.58	0.18	0.61	15.0
sys.settrace()	0.63	0.98	0.58	17.9

loop will consume about $0.17\,\mu s$. Capturing the loop execution on a per-line scale without forwarding the information to Score-P costs additionally $0.8\,\mu s$. This cost only appears for the `sys.settrace()` instrumenter.

For the second case (Fig. 4b), we see the same initial costs. However, the per-iteration costs are higher since we call functions. The general overhead without instrumentation (`None`) increases by about $0.13\,\mu s$ to about $0.3\,\mu s$. The overhead for function instrumentation increases even more. Here each function call adds about $14.7\,\mu s$ (for `sys.setprofile()`). Due to the per-line overhead, we can say that `sys.settrace()` should not be used in the current implementation where the same data is given to Score-P by both available instrumenters. Therefore, we choose to set `sys.setprofile()` as default instrumenter. In future versions of our software, we plan to include information on exceptions or executed lines in

profiles and traces. The user will have to choose whether the additional information is important enough for the added overhead.

4 Related Work

There are different tools to profile or trace Python code. The most common ones are the built-in profiling tools *profile* and *cProfile* [15]. While both share the same command-line interface, cProfile is preferable, since it is implemented in C and therefore faster. The output of both tools is usually written to the command line, but can also be re-directed to a file. The output can be converted and visualised by several third-party tools. For example, pyprof2calltree [16] enables users to convert the output for later analysis with Kcachegrind [17]. An alternative is SnakeViz [18], which visualises the output of the built-in profilers in a web application.

All these tools are only focussed at single node analysis and do not support parallel programming paradigms used in HPC, like MPI or OpenMP. This is different for Extrae [3] and TAU [4]. Extrae uses `sys.setprofile()` to register callbacks from Python. The developers implemented their interface using ctypes, which is a foreign function interface for Python. TAU version 2.28.1 utilises `PyEval_SetProfile` from the C-API and register a callback function that is written in C.

5 Conclusion and Future Work

In this paper, we introduced a module that enables performance engineers to instrument Python applications with Score-P. We described and justified different design decisions that we encountered during development. To quantify the runtime overhead, we presented measurements of two benchmark kernels. Based on these measurements, we decided to use `sys.setprofile()` as the default instrumenter, as the runtime overhead is smaller than the overhead caused by `sys.settrace()`.

Further work might include ways to control the runtime overhead, besides manual instrumentation. One approach could be to sample Python applications.

The Score-P Python bindings are available online at https://github.com/score-p/scorep_binding_python.

References

1. Python 3 Documentation: Glossary (2019). https://docs.python.org/3/glossary.html
2. Knüpfer, A., Rössel, C., Mey, D.a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., Wolf, F.: Score-P: a joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. Tools for High Performance Computing 2011 (2012). https://doi.org/10.1007/978-3-642-31476-6_7
3. Wagner, M., Llort, G., Mercadal, E., Gimenez, J., Labarta, J.: Performance analysis of parallel python applications. Procedia Comput. Sci. (2017). https://doi.org/10.1016/j.procs.2017.05.203
4. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. (2006). https://doi.org/10.1177/1094342006064482
5. Ilsche, T., Schuchart, J., Schöne, R., Hackenberg, D.: Combining instrumentation and sampling for trace-based application performance analysis. Tools for High Performance Computing 2014 (2015). https://doi.org/10.1007/978-3-319-16012-2_6
6. Coghlan, N.: Executing modules as scripts. PEP 338, Python (2004). https://www.python.org/dev/peps/pep-0338/
7. Python 3.7: os 'Miscellaneous operating system interfaces. https://docs.python.org/3/library/os.html#os.execve
8. Python 3.7: Built-in Functions. https://docs.python.org/3/library/functions.html
9. Python 3.7: sys 'System-specific parameters and functions'. https://docs.python.org/3/library/sys.html
10. Python 3.7: Extending and Embedding the Python Interpreter (Oct.). https://docs.python.org/3/extending/index.html
11. Score-P User Manual 6.0 (2019). http://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/pdf/scorep.pdf
12. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The vampir performance analysis tool-set. Tools for High Performance Computing (2008). https://doi.org/10.1007/978-3-540-68564-7_9
13. Horovod: tensorflow_keras_mnist.py. GitHub (2019). https://github.com/horovod/horovod/blob/master/examples/tensorflow_keras_mnist.py, sha:c6ed366
14. HPC System Taurus (2019). https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HardwareTaurus
15. Python 3.7: The Python Profilers. https://docs.python.org/3/library/profile.html
16. Waller, P., Dufresne, J., Grisel, O., Benjamin, Z.: pyprof2calltree (2019). https://github.com/pwaller/pyprof2calltree/
17. Weidendorfer, J.: Sequential performance analysis with callgrind and kcachegrind. Tools for High Performance Computing (2008). https://doi.org/10.1007/978-3-540-68564-7_7
18. Davis, M., Bray, E.M., Schlömer, N., Xiong, Y.: SnakeViz (2019). https://github.com/jiffyclub/snakeviz/