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.

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 © 2017 The Authors. Published by Elsevier B.V.
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.