Synergies in Scientific Computing by Combining Multi-Paradigmatic Languages

Philipp Schwaha, René Heinzl, Franz Stimpfl, and Siegfried Selberherr

Institute for Microelectronics, TU Wien, Gußhausstraße 27-29, Vienna, Austria

Abstract. In recent years run-time languages such as Python have enjoyed a growing community of developers in several programming areas, e.g., scientific computing. The multi-paradigm approach of Python is certainly one of the major advantages as well as its simple syntax and semantics. Rapid prototyping is thereby possible. As major deficiency remains, the low overall performance, especially in the area of scientific computing. We therefore introduce a link between the multi-paradigm Python and a high-performance multi-paradigm environment in C++.

1 Introduction

Besides specifying the outlines of simulation domains the ability to manipulate attributes or quantities stored on such domains is crucial for the field of scientific computing. This should be provided by easy to use facilities with which new simulations can be setup quickly. Furthermore the ability to use analytical models or to use virtually arbitrary values for specification should also be included.

The Python programming language [1], which has enjoyed a growing community of developers and users due to the easy availability of several different programming paradigms within a single language and its, by design, simple syntax and semantics, has many of the required features but lacks the essential feature of consistent and efficient traversal of simulation domains as well as quantity storage mechanisms. By providing such mechanisms and filling this void, a very powerful tool is obtained, as numerous Python modules addressing many issues from several diverse areas have already been created. A key feature of this approach is the ability to rapidly develop applications, efficiency however is limited by the Python interpreter and the performance of the individual modules. While the efficiency of the interpreter continues to evolve, it is of high importance to provide efficient modules in order insure acceptable run times. This is especially true for the field of scientific computing, where processing of considerable amounts of data is commonplace.

Therefore Python modules are often based on highly optimized libraries written in a variety of compiled languages, which are made available to Python by a wrapping layer. The multi-paradigmatic nature of both C++ and Python makes combinations of these two languages very appealing. Not only does C++ offer several paradigms concurrently, it also offers a high degree of run-time efficiency [2, 3]. However, the great multitude of features and possibilities offered by C++ along with its strong typing mechanisms is often perceived as an obstacle, especially by beginners to programming, for rapid

implementations of prototypes. On the other hand Python has become known as an easy to learn language suitable for rapid prototyping.

The field of scientific computing requires not only the topological and geometrical outlines of the simulation domain, but also requires the quick and easy specification of quantities within these domains to set parameters and boundary conditions alike. Python provides many such facilities but inherently lacks efficient complex traversal mechanisms. By providing these capabilities a powerful tool for the setup and even conduction of simulations is created.

2 Multi-Paradigm Development in C++

The developed generic scientific simulation environment [4] has been further enhanced by various functional as well as generic modules to not only support the close interaction with Python, but also the interoperability with the C++ STL and parallel STL [5], BGL [6], GrAL [7], and CGAL [8].

Basic data structures, such as the STL containers, can already be modeled by a simple topological space and hence by a simple topological traversal. A more complex data structure, e.g., a one-dimensional graph, can also be traversed by simple traversal mechanism, e.g., all vertices, all edges, vertex on edge, and edge on vertex. Here the differentiation between so-called intrinsic traversal (vertices within a container) and deduced traversal (edge on vertex traversal) is becoming important. Higher dimensional topological spaces, e.g., two- and three-dimensional meshes and grids, require a more complex combinatorial traversal hierarchy.

By providing a formal and common traversal interface for different types of libraries, interoperability is greatly supported. An example for using STL data structures, e.g. an array, is presented next. Higher dimensional topological objects, such as edges, facets, cells, are not available directly in STL containers.

```
traverse<vertex>()  [  quan = quan_gen(1) ] (container);
```

A more complex traversal on a higher-dimensional space can be traversed in the following way for all different libraries, in this case STL, CGAL, GrAL,and GSSE [9]:

```
traverse<segment>()
[
  traverse<cell>()  [  traverse<edge> [  /* functor */  ] ]
](domain);
```

Another example is given next, where all geometrical points with a special coordinate functor are marked:

```
traverse<segment>()
[
  traverse<vertex>()
  [
    if_(coord[x] > 5.3) [  quan = 1  ]
  ]
](domain);
```

3 The GSSE::Typhoon Module

Next we present a package, called *Typhoon*, which brings the efficient topological traversal and quantity storage available in the GSSE [4, 10, 9] C++ environment to the Python programming language. It introduces the same multi-dimensional and multi-topological traversal of the GSSE to the Python language.

The *Typhoon* Python module has been implemented using Boost Python [11] which greatly simplifies the interfacing of C++ and Python. Care has to be taken to correctly transfer the high flexibility awarded to the GSSE by employing several programming paradigms in concert with the ones available to Python. A particular difficulty is the fact that the static polymorphism used in C++ for performance and consistency reasons must be transferred to the dynamically typed world of Python. While generic programming techniques are used to minimize implementation effort, the resulting compile times cannot to be neglected, as all desired facilities for all required dimensions must be instantiated at compile time in order to be available at run time. The proper selection is performed automatically by Python's dynamic type system and by function overloading.

The following short code snippet demonstrates the application of the traversal mechanisms, where the same traversal mechanism is used as in the C++ example. First all segments in a domain are traversed, followed by the traversal of the cells of the traversed segment. A quantity is stored on all of the traversed cell using the identifier "quan_1". The following part of sample code shows, how the traversal mechanisms can be combined with Python's lambda function facilities to obtain a powerful selection mechanism. The result of such a selection is again compatible with *Typhoon*'s facilities, as is shown in the very last two lines of code.

```
for segment in segments(domain):
    for element in cells(segment):
        store_cell_quan(domain,segment,element,"quan_1", 1.0)

    selection = filter(lambda x:
                       filter (lambda y: coordinates(d,y)[0] > 5.3,
                               vertices(x)),
                       cells(segment))

    for selected in selection:
        sum = 0.0
        for v in vertices(selected)
              sum += retrieve_vertex_quan(domain, v,"quan_3")

        store_cell_quan(domain,segment,selected,"quan_2",
                        sum / len(vertices(selected)))
```

Here the actual traversal is executed by the GSSE traversal library, where the control of run-time selections is handled by *Typhoon*.

4 Benchmarks

To further compare the two different approaches controlled by *Typhoon*, we give benchmark results for topological traversal. The benchmarks given here are obtained by simple traversal of an array as equivalent data structures are available in Python and *Typhoon*. No quantities where stored on the traversed objects. It should be noted that memory consumption was much higher in native Python than when using *Typhoon* and even prohibited the traversal of more than the given 10^8 elements.

The second and third columns are for multi-dimensional arrays and show that *Typhoon* is always faster when traversing multidimensional structures than Python. Again Python's native memory requirements surpassed those of when using *Typhoon*.

The memory issue is expected to become even more pronounced when quantities are to be stored on the traversed structures as *Typhoon* inherently makes use of the GSSE's quantity handling capabilities.

#of elemnts	10^8	10000×10000	$100 \times 1000 \times 1000$
Python	9m26s	3m35s	3m57s
Typhoon	2m44s	1m18s	2m29s

Table 1: Comparisons of the traversal times of data structures from Python and Typhoon (times obtained on a AMD Phenom 9600).

5 Conclusion

The field of scientific computing requires in addition to the topological and geometrical outlines of the simulation domain, a comprehensive yet convenient specification of quantities within these domains to set parameters and boundary conditions alike. Python offers many such facilities but inherently lacks complex traversal mechanisms and performance. By providing these capabilities with our *GSSE::Typhoon* module a powerful tool for the setup and conduction of simulations is obtained.

References

1. Python Software Foundation: Python Programming Language. http://www.python.org/.
2. Gregor, D., Järvi, J., Kulkarni, M., Lumsdaine, A., Musser, D., Schupp, S.: Generic Programming and High-Performance Libraries. Intl. J. of Parallel Prog. **33**(2) (June 2005)
3. Heinzl, R., Spevak, M., Schwaha, P.: Concepts for High Performance Generic Scientific Computing. In: Proc. of the 12th Conf. Student EEICT 2006. Volume 4., Brno, Czech Rep. (April 2006) 446–450
4. Heinzl, R., Schwaha, P., Selberherr, S.: A High Performance Generic Scientific Simulation Environment. In et al., B.K., ed.: Lecture Notes in Computer Science. Volume 4699/2007. Springer, Berlin / Heidelberg (2007) 781–790
5. Singler, J., Sanders, P., Putze, F.: The Multi-Core Standard Template Library. In: Lecture Notes in Computer Science. Volume 4641/2007. Springer, Berlin / Heidelberg (2007) 682–694
6. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
7. Berti, G.: GrAL - The Grid Algorithms Library. In: ICCS '02: Proc. of the Conf. on Comp. Sci. Volume 2331., London, UK, Springer (2002) 745–754
8. Fabri, A.: CGAL - The Computational Geometry Algorithm Library (2001)
9. Heinzl, R., Schwaha, P.: GSSE. (2007) http://www.gsse.at.
10. Heinzl, R., Spevak, M., Schwaha, P., Grasser, T.: A High Performance Generic Scientific Simulation Environment. In: Proc. of the PARA Conf., Umea, Sweden (June 2006) 61
11. Boost: Boost Python. (2006) http://www.boost.org/.