Synergies in Scientific Computing by Combining Multi-Paradigmatic Languages for High-Performance Applications

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

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

1 Introduction

High-performance application design for scientific computing has always been a challenge regarding the applied programming paradigms. An additional development in recent years regarding interpreted languages such as Python [1] raised the complexity in application design, but also introduced new possibilities, especially concerning rapid prototyping. The multi-paradigm approach of Python is certainly one of the major advantages as well as its simple syntax and semantics. A 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++ suitable for scientific computing.

The Python programming language 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. A key feature of Python is the ability to rapidly develop applications, efficiency however is limited by the interpreter, and the performance of the individual modules. The absence of multi-threading capabilities and inherent parallelization is another severe issue currently not rigorously treated. Creating a high-performance link between the convenient Python control and parallel execution paths is therefore of utmost importance, as presented here [2]. While the capabilities of the interpreter continue to evolve, it is of high importance to provide efficient modules in order to insure acceptable run-times. This is especially true for the field of scientific computing, where processing of considerable amounts of data is commonplace.

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 support for several paradigms concurrently, it also provides meta-programming mechanisms to orthogonally optimize and even parallelize developed code without modification. 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 also in the field of scientific computing.

2 Multi-Paradigm Development in C++

The developed Generic Scientific Simulation Environment (GSSE [3]) incorporates 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 [4], BGL [5], GrAL [6], and CGAL [7].

Basic data structures, such as the STL containers, already model a simple topological space and hence provide elementary topological traversal mechanisms. More complex data structures, e.g., a one-dimensional graph, can also be traversed by simple traversal mechanisms, 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 significantly enhanced. An example of using STL data structures with GSSE [3, 8] concepts, e.g. an array, is presented next. Higher dimensional topological objects, such as edges, facets, or cells, are not available directly in STL containers.

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

More complex traversal on a higher-dimensional space can be traversed in the following way for several libraries, in this case STL, CGAL, GrAL, and GSSE, where all geometrical points with a special coordinate functor are marked:

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

This example can be executed on an arbitrary number of cores due to the functional specification. Only a simple recompilation step with, e.g., the parallel STL is required.

3 The GSSE::Typhoon Module

Next to the developed traversal and quantity storage mechanism of the GSSE, *Typhoon* links these mechanisms to the Python programming language, thereby enabling the highly efficient and parallel multi-dimensional and multi-topological traversal of the GSSE for the run-time scheme. The *Typhoon* Python module has been implemented using Boost Python [9], 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 the implementation effort, the resulting compile times cannot be neglected, as all desired facilities for all required dimensions must be instantiated at compile time in order to be available at run-time. In a C++ application, the correct dimension is automatically selected at compile time, while in the case of Python applications of dimensions one through three, the single code base results in three separate Python modules, with their proper selection automatically performed 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 cells using the identifier "quan_1". Then a sample code is given to present, 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 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*.

Deploying Typhoon

Using the *Typhoon* Python module it is possible to also rapidly develop application prototypes by combining it with one of the numerous scientific packages already available for Python. Figure 1 schematically shows the interaction of the GSSE, *Typhoon*, and Python.

The availability of a wide variety of traversal mechanisms in conjunction with high performance solver packages such as Trilinos [10], which is made available in Python by PyTrilinos [11], is a particularly interesting combination, as it enables even complete implementations of simulations directly in Python. Typically, the input is used to assemble an equation system which is subsequently solved to compute an approximate solution of the problem under investigation. By using the *Typhoon* module, the Python code remains unchanged even for different topologies and dimensions, as the underlying GSSE takes care of equalizing the interfaces.

By traversing the vertices of the input structure the system matrix is assembled using *Typhoon*. The high level of abstraction provided by the GSSE is retained without restrictions on the topology or dimension of the specified problem. In contrast to a C++ implementation, where the, often time consuming, recompilation of the program is required to obtain an executable from the source code, the Python implementation is available immediately. A caveat, however, is that the *Typhoon* module had to be compiled including the appropriate dimension and topology, which is automatically taken care of with C++ during compilation.
The following source code demonstrates the combination of *Typhoon*'s traversal mechanisms, which are used to assemble the matrix, and the PyTrilinos solver interface, which is employed to calculate the solution:

```
for segment in segments(domain):
    for element in vertices(segment):
        assign_matrix_value(matrix_A, element) = calc(element)
        assign_matrix_value(matrix_B, element) = calc_rhs(element)
Solver.SetMatrix(matrix_A)
Solver.SetVectors(X, matrix_B)
Solver.Solve()
```

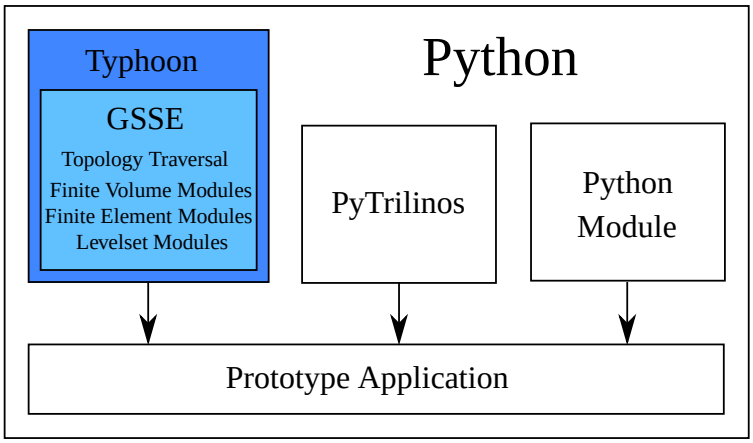

Fig. 1: The Typhoon module brings the topology and traversal mechanisms of the GSSE to Python. The combination with additional Python modules enables a rapid implementation of prototype applications.

4 Benchmarks

To further compare the two different approaches controlled by *Typhoon*, we give benchmark results for topological traversal. The benchmarks 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 multi-dimensional structures than Python. Again Python's native memory requirements surpassed those of *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 elements	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 an AMD Phenom 9600).

5 Application Example

We present an example of TCAD's device simulation applications [12], where the most basic model, the drift-diffusion model, is comprised of four coupled partial differential equations which need to be assembled. In this case the assembly time is usually small compared to the time spent on the solution of the equation system. More sophisticated and complex models such as energy transport or higher transport models, however, spend an increasing amount of time on equation assembly. Parallelism of not only the assembly procedure but also the pre- and post-processing algorithms is therefore of increasing importance to control computation times. To demonstrate not only the functional equation specification mechanism of the GSSE, but also the parallel approach mechanism the following code snippet demonstrates the actual C++ code for the electron temperature `T_n` for a hydro-dynamic device simulation application [13]. Equation 1 shows the energy flux equation for electrons, which is solved self-consistently with Poisson's equation and the current relations [14].

$$\operatorname{div}\left(\alpha_n \operatorname{grad}(nT_\mathrm{n}^2) + \operatorname{grad}\varphi\, n\, T_\mathrm{n}\right) = -\operatorname{grad}\varphi \cdot \mathbf{J}_\mathrm{n} - \beta_n\, n(T_\mathrm{n} - T_\mathrm{Lattice}) \qquad (1)$$

The following source code snippet reflects the functional specification for a finite volume discretization scheme in actual C++ code. Each `sum` is automatically parallelized by the parallel STL, where the full matrix line is assembled in parallel by a vertex partitioning mechanisms executed by a multi-threading mechanism:

```
(sum<edge>()
[
  let(_x = Bern(edge_log<vertex>(T_n)) / T_n *
      sum<vertex>() [ phi ] +
      sum<vertex>() [ T_n ]
     )
   [
      alpha_n * T_n / Bern(edge_log<vertex>(T_n))  *
      sum<vertex>() [  Bern( _x )  *  n  *  T_n  ] *
      area / dist
   ]
]
+ sum<edge>()
[
  sum<vertex>() [ phi ] / dist * J_n
]  * vol  +
   beta_n * n * (T_n - T_lattice ) * vol
) (vx);
```

A benchmark for a simple drift-diffusion and hydro-dynamic simulation for a two-dimensional pn-diode with different compiler (GCC 4.2) optimization levels and different numbers of concurrent threads is presented next.

Example	Sequential	Dual-core	Quad-core	Num. elements
DD, O1	32 (s)	9 (s)	6 (s)	1e4
DD, O3	11 (s)	8 (s)	6 (s)	1e4
HD, O1	41 (s)	15 (s)	7 (s)	1e4
HD, O3	20 (s)	9 (s)	6 (s)	1e4

Table 2: Comparisons of the simulation times for drift-diffusion and hydro-dynamic simulation of a pn-diode with different optimization levels (GCC 4.2) on AMD X2 6000 CPU's (dual-core) and AMD X4 Phenom 9600 (quad-core).

By making these efficient C++ implementation available to Python via *Typhoon* even complicated applications and simulation flows can be assembled quickly. Not only can the efficiency of C++ be carried over to Python, Python's shortcomings regarding multi-threading can also be addressed by using *Typhoon*. The sections intended to be parallelized are implemented in C++ using GSSE and stored in a table. Source code of this procedure is given in the following:

```
thread_count = 4
for segment in segments(domain):
   parallel_apply_to(segment, thread_count, hydro_assemble)
```

The single Python thread is used to control the application, referencing the available functions with Python remaining oblivious to the parallel nature of the underlying implementation.

6 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, performance, and parallelization. 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. Heinzl, R., Schwaha, P., Stimpfl, F., Selberherr, S.: Parallel Library-Centric Application Design by a Generic Scientific Simulation Environment. In: Proc. of the POOSC Conf., Paphos, Cyprus (July 2008) Submitted
3. Heinzl, R., Schwaha, P., Selberherr, S.: A High Performance Generic Scientific Simulation Environment. In B. Kaagström et al., ed.: Lecture Notes in Computer Science. Volume 4699/2007. Springer, Berlin (2007) 781–790
4. Singler, J., Sanders, P., Putze, F.: The Multi-Core Standard Template Library. In: Lecture Notes in Computer Science. Volume 4641/2007. Springer, Berlin (2007) 682–694
5. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
6. Berti, G.: GrAL - The Grid Algorithms Library. In: Proc. Computational Science ICCS. Volume 2331., London, UK, Springer (2002) 745–754
7. Fabri, A.: CGAL - The Computational Geometry Algorithm Library. In: Proc. of the 10th Intl. Meshing Roundtable, CA, USA (2001) 137–142
8. Heinzl, R., Schwaha, P.: GSSE. (2007) http://www.gsse.at/.
9. Boost: Boost Python. (2006) http://www.boost.org/.
10. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., K., T.G., L., R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An Overview of the Trilinos Project. ACM Transactions on Mathematical Software **31**(3) (2005) 397–423
11. Sala, M., Spotz, W., Heroux, M.: PyTrilinos: High-Performance Distributed-Memory Solvers for Python. ACM Transactions on Mathematical Software **34**(2) (March 2008) 1–33
12. Grasser, T.: Advanced Device Modeling and Simulation. World Scientific Publishing Co. (2003)
13. Grasser, T., Tang, T., Kosina, H., Selberherr, S.: A Review of Hydrodynamic and Energy-Transport Models for Semiconductor Device Simulation. Proc. IEEE **91**(2) (2003) 251–274
14. Schwaha, P., Schwaha, M., Heinzl, R., Ungersboeck, E., Selberherr, S.: Simulation Methodologies for Scientific Computing – Modern Application Design. In: Proc. of the 2nd ICSOFT 2007, Barcelona, Spain (July 2007) 270–276