CONCEPTS FOR HIGH PERFORMANCE GENERIC SCIENTIFIC COMPUTING

R. Heinzl, P. Schwaha, T. Grasser, CDL for TCAD, Institute for Microelectronics, TU Vienna, Austria
M. Spevak, Institute for Microelectronics, TU Vienna, Austria

Corresponding author: R. Heinzl
TU Vienna, Gusshausstr. 27-29/E360, A-1040 Vienna, Austria
Phone: +43 1 58801-36053, Fax: +43 1 58801-36099
email: heinzl@iue.tuwien.ac.at

Abstract. We present concepts for a generic environment for high performance scientific computing that impose no restrictions on geometry, topology, or discretization schemes. Therewith algorithms and discretization schemes can be formulated in a dimension and topology neutral way.

1. Motivation

The scientific computing approach is used to gain an understanding of scientific and engineering problems by the analysis of mathematical models implemented in computer programs and solved by numerical techniques. Due to the diversity of the mathematical structures, combined with efficiency considerations, in particular in three dimensions, the development of high performance simulation software is quite challenging. In the field of TCAD, as in many others, the numerical simulation results are based on different discretization schemes such as finite differences, finite elements, and finite volumes. Each of these schemes has its merits and shortcomings and is therefore more or less suited for different classes of equations. All of these methods have in common that they require a proper tessellation and adaptation of the simulation domain [1, 2], so-called unstructured meshes or structured grids.

Testing and validation are major problems in the development process of software for numerical applications. Errors are often not obvious to detect. It may already require a lot of experience to decide if a result from a simulation is erroneous or not. If the result is not correct, it may be due to a great number of reasons, e.g. to a programming bug, a logical error in the program flow, or a badly chosen parameter. Therefore the availability of already tested and proven modules can not be underestimated.

We have identified the following issues for an environment suitable for scientific computing:

- Sophisticated realtime visualization possibilities
- High performance
- Abstract solver interface
- Automatic treatment of discretization schemes
- Algebraic support for an easy implementation of equations

To deal with all of these issues, our institute has developed different simulation environments, libraries, and applications during the last decade:

- Wafer-State-Server [3] is a complete geometrical and topological library with geometrical algorithms, interpolation mechanisms and consistency checks for simplex objects, especially in three dimensions.
- STAP [4] is based on a set of high-speed simulation programs for two- and three-dimensional analysis of interconnect structures. The simulators are based on the finite element method and can be used for highly accurate capacitance extraction, resistance calculation, transient electric and coupled electro-thermal simulations.
- Minimos-NT [5] is a general-purpose semiconductor device simulator providing steady-state, transient, and small-signal analysis of arbitrary device structures. In addition, Minimos-NT offers mixed-mode device/circuit simulation to embed numerically simulated devices in circuits with compact models.
- FEDOS [6] is a finite-element based simulator for oxidation and diffusion phenomena with integrated mesh adaptation.

However, none of these simulators has proven to be perfect for the rapid progress in scientific software development. Even the reuse of simple code parts is difficult, due to the non-generic-library approach.

We have extracted the main concepts from our own simulation tools and have combined them with the most promising techniques proposed by other groups. The result of this work is the multidimensional and multi-topological *generic scientific simulation environment (GSSE)*. We compare related work in the next section.

Generic library means that each part of GSSE can be used separately. The complete GSSE is based on header files only and therefore only the required mechanisms have to be included without incurring additional dependencies. This approach guarantees a great enhancement for building libraries in terms of interoperability and performance.

Multidimensional means two and three dimensions for unstructured topology and arbitrary dimensions for structured topology. Multitopological means structured and unstructured topology.

- GSSE uses the C++ Boost libraries [7] and the C++ STL only. There are no additional dependencies on other libraries.
- All sub-modules are capable of complete self-tests. There are complete overall test files and application test files for each application.
- With meta-programming, functional programming, and the generic programming approach with partial specialization very high performance can be achieved.

2. Related Work

Various research groups have put a lot of effort into the development of libraries for scientific computing or for sub-problems occurring in scientific computing. Here we want to review and compare libraries with respect to GSSE, which can be used in the field of TCAD or scientific computing.

Blitz++

The Blitz++ library [8] implements basic n-dimensional arrays along with algebraic operations on them. The author has developed techniques which enables C++ to rival and in some cases even exceed the speed of Fortran for numerical computing, while preserving an object-oriented interface. The technique of *expression templates* allows expressions to be passed to functions as an argument and inlined into the function body. These results are not obtained by better optimizing compilers, preprocessors, or language extensions, but through the use of template techniques. This library was the first approach to use templates cleverly to accomplish optimizations such as loop fusion, unrolling, tiling, and algorithm specialization automatically at compile time.

GSSE implements its own functional layer to get the benefits of partial and lazy evaluation.

Matrix Template Library and Iterative Template Library

The Matrix Template Library (MTL, [9]) is a comprehensive, high-quality library of re-usable matrix components and linear algebra operations for scientific computing with containers, iterators, adaptors, and function objects. A similar approach is employed by the C++ STL [10] which guarantees high performance together with easy extensibility.

The Iterative Template Library (ITL, [11]) is a generic component library that provides iterative methods for solving linear systems. ITL also provides numerous preconditioners and uses an abstract interface of matrix-vector, vector-vector, and vector-scalar operations from the MTL. ITL is able to use other packages such as Blitz++ and BLAS with the same abstract interface.

The data types from the MTL library can easily be used with GSSE due to the parametrization of data types within all GSSE layers and the abstract solver interface.

Boost Graph Library

The Boost Graph Library (BGL, [12]) is a generic approach to the topic of graph handling and traversion with a standardized generic interface. Therewith all different kinds of graph algorithms can be implemented without taking care of the actual implementation. This interface makes it possible for any graph library that implements the interface to interoperate with the BGL. The approach is similar to the C++ STL approach for interoperability of algorithms and containers.

The BGL can be used quite easily for example to store the topology information of GSSE.

Computational Geometry Algorithms Library

CGAL [13] implements generic classes and procedures for geometric computing with generic programming techniques. It is based on a three-layer structure, where the kernel layer consists of constant-size non-modifiable geometric primitive objects and operations on these objects. The second layer consists of a collection of basic geometric data structures and algorithms. This layer includes polygons, half-edge data structures, polyhedral surfaces, topological maps, planar maps, arrangements of curves, triangulations, convex hulls, alpha shapes, optimization algorithms, dynamic point sets for geometric queries, and multidimensional search trees. The third layer consists of non-geometric support facilities, such as support for number types, and circulators.

Geometrical calculations are not part of GSSE but of CGAL.Therefore it can be used orthogonally with GSSE.

Grid Algorithms Library

GrAL [14] is a generic library for grid and mesh data structures and algorithms operating on them. GrAL was initially developed to solve PDEs numerically. The underlying mathematical structure of abstract cellular complexes is very general.

GSSE can be seen as a direct successor of GrAL with enhancements in the used programming paradigms and the functional layer, types of topology, solver interface, and realtime visualization.

Prophet

Prophet [15] is an environment for the solution of PDEs. It introduces four different levels of abstraction. The first and base library layer consists of a database for models of coefficients and material parameters, macros for expressions, grid routines, and a linear solver. The PDE layer consists of an assembly control, discretizations, and modeling modules. The third layer consists of different modules, for example modules for solving. The fourth layer is the user layer which consists of an input parser and the visualization mechanism. Prophet separates the geometric information and physic information. The user should not need to know about the spatial discretization.

Prophet was originally developed for semiconductor process simulation and was enhanced for device simulation afterwards.

deal.II

deal.II [16] provides a framework for finite element methods and adaptive refinement for finite elements. It uses the modern programming techniques of C++ and enables the treatment of a variety of finite element schemes in one, two, or three spacial dimensions, as well as time-dependent problems. The main aim of deal.II is to enable development of modern finite element algorithms, using, among other aspects, sophisticated error estimators and adaptive meshes. Writing such programs is a non-trivial task, and applications tend to become very large and complex.

deal.II and GSSE are similar in the way, how the finite elements discretization can be handled. But GSSE offers a functional layer for all different discretization schemes.

3. Base Concepts of GSSE

The approach used for the development of GSSE fundamentally relies on the generic programming and functional programming paradigm. Generic programming, started with the development of the STL, is an especially attractive paradigm for developing libraries for high-performance computing because it emphasizes generality and efficiency simultaneously. With this approach libraries can be coupled much more tightly to compilers, allowing libraries to provide highly-reusable data types and algorithms, as specific optimization possibilities as well.

The fundamental principle of generic programming is to separate algorithms from the concrete data structure on which they operate. This allows the algorithms and data structures to interoperate freely. From a generic point of view, algorithms do not manipulate concrete data structures directly, but instead operate on abstract interfaces defined for entire classes of data structures.

Generic programming has proven its usefulness for a class of mathematically simple, but very important concepts, namely the linear sequences of the STL. Genericity can be explained easily in terms of the STL:

- Separation of algorithms and data structures: Due to the separation of algorithms and data structures complete interoperability can be achieved. Therewith a single generic function can operate on many different data structures. The code size can be reduced from $\mathrm{O}(n \cdot m)$ to $\mathrm{O}(n + m)$ where n is the number of algorithms and m is the number of data structures.
- Extension through function objects to adapt and customize generic algorithms.
- Element type parametrization for all containers.

In contrast to the algorithms and data structures found in the STL, scientific computing shows a lot more diversity with respect to the needed concepts. The STL is based on linear data sequences like arrays, lists, and maps. There is always a trivial topological connection inside these data sequences. Scientific computing has no restrictions on topological connections or dimensions. We have to operate on vertices, edges, rectangles, triangles, cubes, or tetrahedra.

The language of choice was and is C++ due to a manifold of reasons. It has been shown [17] that all other languages have significant problems with generic implementations in the field of numeric algorithms. Only the multi-paradigm language C++ shows excellent efficiency.

The implementation of generic programming concepts in C++ is done with parametric polymorphism [12] with features like *template specialization, partial specialization, partial ordering of function templates*. These features also allow techniques like *meta-programming* [18]. These techniques guarantee a performance behavior similar to Fortran code [8].

Functional programming techniques are available in C++ such as Lambda (unnamed functions [19]) and Currying (partial function evaluation [20]) with the Spirit-Phoenix library [21]. These techniques are used in our function layer which connects the cursor and property map concepts to the discretization concepts.

The base concept of GSSE is the separation of topology, in this case a finite CW complex, and quantities, which means all different kinds of attached properties. Geometric information, for example coordinates of a vertex, can be seen as a quantity. Figure 1 illustrates the separation of the cursor concept and the property map concept, where the cursor concept uses only topological information. The abstract interface mechanism to the visualization and the solver capabilities are also included in the figure.

Section 3 explains the motivation for the separation in more detail.

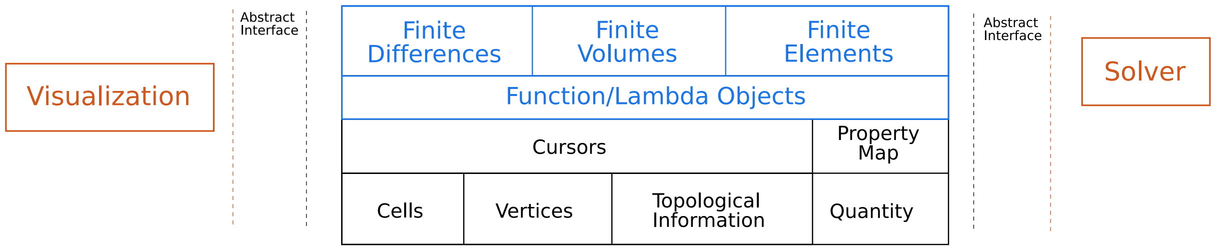

Figure 1: Complete conceptual view of GSSE with interfaces to additional external libraries.

Vertex and Cell Concepts

A simulation domain D which is part of a space $\mathbf{R}^n$ where $0 < n < \infty$ is described by vertices (0-dimensional objects) and cells (n-dimensional objects) and the topological information about the connections. The minimal topological information that has to be stored is *vertex on cell* connections. For an efficient traversion through all sub-dimensions, we store the *cell on vertex* information as well.

Topological Information Concept

In general we use spatial tesselations which fulfill the concept of a CW complex and have a finite number of cells. In general we need to store the complete information of the complex in order to enable a convenient handling such as storing quantities as well as traversal with cursors. We use two different kinds of topologies which can be handled differently due to their interior structure.

- Structured topology: A widespread approach to spatial discretization is to divide the simulation domain into a structured assembly of quadrilateral cells, with the topological information being apparent from the fact that each interior vertex has exactly the same number of neighboring cells. This kind of discretization is called *structured grid* or simply *grid*.
- Unstructured topology: The alternative approach is to divide the computational domain into an unstructured assembly of more or less arbitrarily formed cells. The topological information can not be deduced implicitly from the elements and has to be stored explicitly. This kind of discretization is called *unstructured mesh* or simply *mesh*.

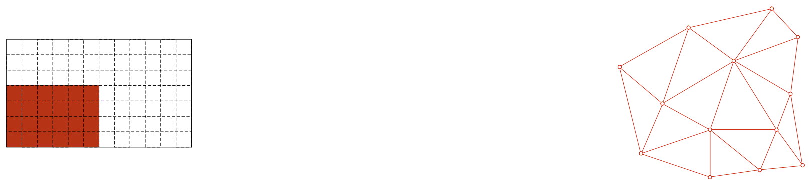

Figure 2: Structured topology (left) and unstructured topology (right)

Quantity Concept

A quantity means all kinds of attributes or properties, which can be attached to objects (topological or geometrical). In the area of scientific computing and especially in the field of TCAD the handling of a large number of different quantities is required. These quantities need to be stored on various objects (vertices, edges, facets, cells). On the one side, we have developed a completely generic quantity library which is capable of storing various mathematical structures in a dense and a sparse format: scalar values, vector values, matrices, and tensors.

On the other side, we have developed specializations of all of these mathematical structures to provide high performance calculations, for instance for small fixed size vectors and matrices.

Of course, all other libraries, such as the MTL, can be used due to the use of parametric polymorphism.

Cursor and Property Map Concepts

In scientific computing a lot of data can be associated with objects. The *iterator* concept [10] can only handle one associated data item due to the linear data sequence concept of the STL. This concept is not suitable for a scientific computing environment. Therefore we have separated the traversion of the topology from the access to the quantities (similar to [22]). To support a wide variety of traversion mechanisms, there are several hierarchies of cursors, which are based on [14], but are implemented quite differently:

- Base cursors: vertex, edge, facet, cell cursor
- Adjacency cursors: Vertex to vertex, cell to cell cursor
- Incidency cursors: Cell on vertex, vertex on cell cursor
- Special cursors: Boundary vertex/edge/facet cursor

As the iterator concept in the STL, the cursor concept of GSSE is the glue between the data structures (meshes and grids) and the algorithms. Therewith algorithms can be specified multidimensionally and multitopologically. To insure high performance, all needed types of cursors are generated by meta-programming techniques [18] at compile-time for each dimension and topology so there is no runtime overhead at all.

The access to quantities is provided by several property maps. The geometric coordinates and the associated data are kept in separate property maps.

```
vertex_cursor vcs = segment.vertex_begin();
while (vcs.valid())
{
  segment.store_quantity(*vcs, quantity_name, value);
  ++vcs;
}
```

Figure 3: Multidimensional and multitopological cursor usage.

Functional Layer Concept

To highlight the behavior of the functional programming paradigm, we present a complex topic of the STL: the use of generic algorithms on arbitrary data structures. The next sample of code demonstrates the output from a generic container:

```
  std::vector<int> data;

  copy(data.begin(), data.end(), ostream_iterator<int>(std::cout, "\n"));
```

The complexity is hidden in the behavior of the third argument to the generic copy algorithm. With functional programming (especially the Lambda part) we can write things more intuitively:

```
  for_each(data.begin(), data.end(), std::cout << phoenix::arg1 << '\n');
```

Therefore we have developed a functional layer which builds a convenient access to all of the underlying concepts.

Abstract Interfaces

For all visualization tasks, we have developed an abstract visualization interface. For now, IBM's data explorer [23] can be used due to its multidimensional and multitopological visualization capabilities. We have extended it by an additional realtime visualization module.

We have developed an abstract solver interface. At the moment, we use modules from the Trilinos project [24] with the LAM/MPI [25], which provides high performance on small single CPU clusters to large SMP machines with high speed networks, even in heterogenous environments.

4. Applicability

To demonstrate the applicability of GSSE we briefly present a few examples. We want to emphasize the coupling of different concepts used in GSSE to support simple and robust software development in the area of scientific computing.

Support for several spatial dimensions is inherently included in such a way that programs can be written independently of the spatial dimension without unreasonable penalties on run-time and memory consumption. The key feature to achieve this abstraction is the cursor concept. With the functional layer, a very high level of abstraction to all different kinds of data structures and functionality can be provided. Algorithms can be specified in a dimension independent and data structure neutral way.

Multidimensional Laplace Code

For instance, using finite volume discretization the Laplace equation can be formulated as:

$$\sum_{\text{edge vertex}} \left(\Psi_j - \Psi_i\right) \frac{A_{\text{ij}}}{d_{\text{ij}}} = 0$$

We present the source code for the Laplace equation for one, two, or three-dimensions and for structured or unstructured meshes. The compiler will always generate the most suitable code for each architecture and application. Therewith an overall high performance can be achieved easily.

The main focus here is on the powerful equation assembly. Therefore the handling of boundaries and the matrix assembly are omitted.

```
for (vertex_cursor vcu = (*segit).vertex_begin();
                  vcu != (*segit).vertex_end(); ++vcu)
{ //boundary handling..

  equation = (gsse::sum<vertex_edge>[gsse::diff<edge_vertex>
        [pot_quan] * (edge_area_quan / edge_dist_quan)]) (*vcu);

} // matrix assembly ..
```

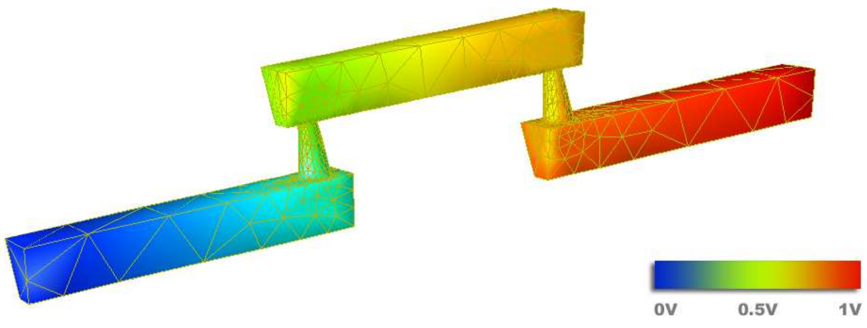

Figure 4: Visualization of the potential distribution.

First we explain the $\Psi_j - \Psi_i$ term in terms of GSSE code:

```
 gsse::diff<edge_vertex>[pot_quan] * ()]
```

A lambda function `gsse::diff` is used to calculate the difference of the potential `pot_quan` which is located on two vertices on an edge. GSSE code for the term $\sum(..)$ can be formulated like this:

```
 gsse::sum<vertex_edge>[gsse::diff<edge_vertex> [..](..)] (*vcu);
```

The sum is generated by a `sum` lambda function which is parametrized by a `vertex_on_edge` cursor and initialized by the actual vertex (dereferenced vertex cursor, `*vcu`).

Each result of the `gsse::diff<edge_vertex>[..](..)` is summed up and is inserted into the left equation object.

Multidimensional Diffusion Code

A diffusion equation with a constant diffusion coefficient and forward Euler time discretization can be formulated:

$$\mathrm{c}_t = \mathrm{c}_{t-1} + \Delta\mathrm{t} \cdot \mathrm{D} \cdot \sum_{\text{edge vertex}} (\mathrm{c}_{j,t-1} - \mathrm{c}_{i,t-1}) \frac{A_{\mathrm{ij}}}{d_{\mathrm{ij}}}$$

The source code for the diffusion equation for one, two, or three-dimensions and for structured or unstructured meshes can be formulated as:

```
for (vertex_cursor vcu = (*segit).vertex_begin();
                  vcu != (*segit).vertex_end(); ++vcu)
{ // boundary handling...

 (pot_quan=pot_old_quan + delta_t * diff_coeff *
  gsse::sum<vertex_edge>(0.0) [gsse::diff<edge_vertex>[pot_old_quan] *
         (edge_area_quan / edge_dist_quan)]) (make_const(*vcu));
}
```

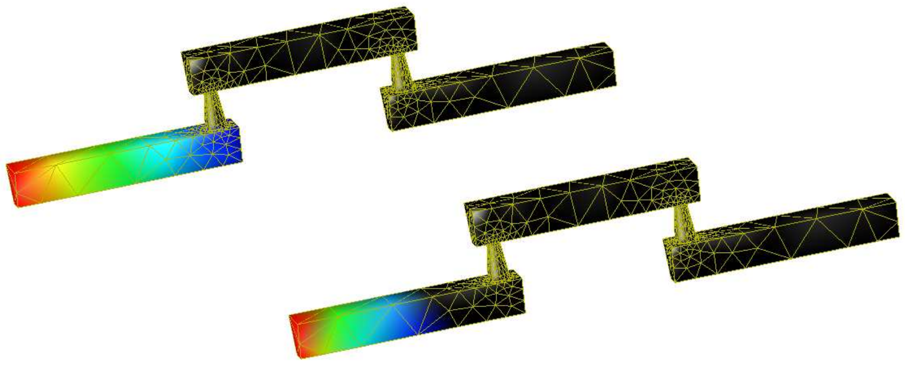

Figure 5: Visualization of two different time steps during simulation.

5. Conclusion

We have presented concepts for high performance scientific computing. We have presented the main paradigms, the generic and functional approach, and the applicability. With GSSE it is possible to build scientific applications with few lines of code. The reduction to ”real” scientific work, not covered with complex programming issues or language ”hacks” is thus the major benefit. Due to the shift of most of the calculations to compile time, runtime performance is excellent. The most important observation is that the so-called abstraction penalty of the generic approach is negligible.

Details can be found at `http://www.iue.tuwien.ac.at/software/gsse`.

References

[1] R. Heinzl and T. Grasser, “Generalized Comprehensive Approach for Robust Three-Dimensional Mesh Generation for TCAD,” in *Proc. SISPAD*, (Kobe, Japan), pp. 211–214, Sept. 2005.

[2] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser, “Coupling Three-Dimensional Mesh Adaptation with an A Posteriori Error Estimator,” in *Proc. SISPAD*, (Kobe, Japan), Sept. 2005.

[3] A. Hössinger, R. Minixhofer, and S. Selberherr, “Full Three-Dimensional Analysis of a Non-Volatile Memory Cell,” in *Proc. SISPAD*, (Munich, Germany), pp. 129–132, Sept. 2004.

[4] R. Sabelka and S. Selberherr, “A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures,” vol. 32, no. 2, pp. 163–171, 2001.

[5] *MINIMOS-NT 2.1 User’s Guide*. Institut für Mikroelektronik, Technische Universität Wien, Austria, 2004. http://www.iue.tuwien.ac.at/software/minimos-nt.

[6] H. Ceric, *Numerical Techniques in Interconnect and Process Simulation*. Dissertation, Technische Universität Wien, 2004.

[7] `Boost C++ Libraries`. `http://www.boost.org`.

[8] T. L. Veldhuizen, “C++ Templates as Partial Evaluation,” in *Proc. of PEPM’99.*, pp. 13–18, University of Aarhus, Dept. of Computer Science, Jan. 1999.

[9] J. G. Siek and A. Lumsdaine, “The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra,” in *ECOOP Workshops*, pp. 466–467, 1998.

[10] M. H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[11] `Iterative Template Library (ITL)`. `http://www.lsc.nd.edu/research/itl`.

[12] J. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[13] A. Fabri, “CGAL- The Computational Geometry Algorithm Library,”, 2001. `citeseer.ist.psu.edu/fabri01cgal.html`.

[14] G. Berti, “GrAL - The Grid Algorithms Library,” in *ICCS ’02: Proceedings of the International Conference on Computational Science-Part III*, (London, UK), pp. 745–754, Springer-Verlag, 2002.

[15] C. S. Rafferty and R. K. Smith, “Solving Partial Differential Equations with the Prophet Simulator,” Bell Laboratories, Lucent Technologies, 1996.

[16] `deal.II` *Differential Equations Analysis Library, Technical Reference*. `http://www.dealii.org`.

[17] R. Garcia, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock in *Proc. of the 18th Annual ACM SIGPLAN*, (New York, NY, USA), pp. 115–134, ACM Press, 2003.

[18] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[19] C. Böhm, ed., *Lambda-Calculus and Computer Science Theory, Proceedings of the Symposium Held in Rome, March 25-27, 1975*, vol. 37 of *Lecture Notes in Computer Science*, Springer, 1975.

[20] M. Sperber and P. Thiemann, "Realistic Compilation by Partial Evaluation.," in *PLDI*, pp. 206–214, 1996.

[21] `Phoenix`. `http://spirit.sourceforge.net/`.

[22] D. Abrahams, J. Siek, and T. Witt, "New Iterator Concepts," Tech. Rep. N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003.

[23] *IBM visualization Data Explorer*, third ed., Feb. 1993.

[24] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software*. For TOMS special issue on the ACTS Collection.

[25] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," in *Proceedings, 10th European PVM/MPI Users' Group Meeting*, no. 2840 in Lecture Notes in Computer Science, (Venice, Italy), pp. 379–387, Springer-Verlag, September / October 2003.