A High Performance Generic Scientific Simulation Environment

René Heinzl, Michael Spevak, Philipp Schwaha, and Siegfried Selberherr

Institute for Microelectronics, TU Wien,
Gusshausstrasse 27-29, 1040 Wien, Austria
heinzl@iue.tuwien.ac.at

Abstract. A generic scientific simulation environment is presented which imposes minimal restriction regarding topological, dimensional, and functional issues. Therewith complete discretization schemes based on finite volumes or finite elements can be expressed directly in C++. This work presents our multi-paradigm approach, our generic libraries, some applications based on these libraries, and performance aspects.

1 Introduction

In the last decades numerous software environments and libraries have been developed to handle the many areas in the field of scientific computing. Due to the diversity of the mathematical structures, combined with efficiency considerations, the development of high performance simulation software is quite challenging. These challenges are becoming more difficult to meet, when the purpose of the software is to validate novel algorithms and complex methods, or to investigate physical phenomena that have not yet been fully understood. High performance computations have turned the attention especially to C++, since Blitz++ has shown that the run-time behavior is comparable to Fortran [1], the traditional language for scientific computing. In addition, distinct programming paradigms and their respective advantages can be combined in a multi-paradigm language, such as C++.

2 Motivation

Many library approaches [1,2,3] focus on topics such as expression templates, high performance matrix operations and calculations, and discretization of differential operators. The nature of dealing with different types of partial differential equations (PDEs) with the inherent coupling of topological traversion and functional description complicates the use of these libraries.

Our main area of work is focused on Technology Computer-Aided Design (TCAD), which deals with the assembly of large equation systems by utilizing discretized partial differential equations from different fields of physics. All types of PDEs (parabolic, elliptic, hyperbolic) and their discretization schemes such

B. Kågström et al. (Eds.): PARA 2006, LNCS 4699, pp. 996–1005, 2007.

as the finite element method [4] or the finite volume method [5] have to be considered for the diverse types of problems. Types of topological cell complexes, different dimensions, and solving strategies have to be considered during application development [5,6,7]. Most of these applications use data structures such as `list` and `array` as well as triangles, quadrilaterals, tetrahedra, cuboids, each with their own access mechanisms, traversal operations, and data storage.

These issues demand great care to ensure high software quality while also addressing performance issues, when source code is being written. This is the primary motivation to develop generic libraries for high performance applications in the field of scientific computing. Generic library design deals with the conceptual categorization of computational domains, the reduction of algorithms to their minimal conceptual requirements, and strict performance guarantees. The benefits of this approach are the re-usability and the orthogonality of the resulting software.

3 Related Work

Various research groups have put a lot of effort into the development of libraries for sub-problems occurring in scientific computing. We briefly review the most important library approaches suitable for application design:

- The Boost Graph Library (BGL [2]) is a generic approach to the topic of graph handling and traversal with a standardized generic interface.
- The Computational Geometry Algorithms Library (CGAL [8]) implements generic classes and procedures for geometric computing with generic programming techniques.
- The Grid Algorithms Library (GrAL [9]) is a generic library for grid and mesh data structures and algorithms operating on them.
- deal.II [3] provides a framework for finite element methods and adaptive refinement for finite elements.
- ExPDE [10] collects efficient high-performance libraries for PDEs using the C++ technique of expression templates

Our analysis has revealed that, up to now, no related work can be used directly. All of these libraries have not been developed with emphasis on interoperability. This issue complicates the transition from one library to another. Therefore, our approach (Section 4.1) introduces a common layer with data structure definition and access routines, where all of these libraries can be used. With the generic programming paradigm and the implementation with templates in C++ the abstraction penalty can be minimized (Section 6).

Deal.II and the ExPDE library collection offer support for application design in the field of scientific computing. These libraries are an important step into library centric application design. But, as mentioned before, none of these libraries were developed with interoperability as a necessary constraint. As a consequence, additional code has to be introduced which slows the development process down and impedes the execution speed of the final application.

4 The GSSE

Based on the experience of developing high performance applications a *generic scientific simulation environment (GSSE)* with an overall high performance was developed, which does not impose restrictions on topological treatment or functional description. Different programming paradigms were used for the non-trivial and highly complex scenario of scientific computing. The generic programming paradigm realized by the template mechanism offers homogeneous interfaces between algorithms and data structures by the iterator/cursor pattern. Functional programming enables an efficient means to specify equations and offers extensible expressions.

Investigations of the applications developed at our institute have shown that the topological structures can be abstracted and generalized into a generic topology library (GTL), which is presented in more detail in Section 4.1. This new approach of a generalized topology enables a new functional description for the discretization of PDEs without sacrificing any performance. This is accomplished in our generic discretization library (GDL, Section 4.2), which fulfills the requirements for scientific computing, especially TCAD [11].

To introduce the base libraries of the GSSE, we examine different topological traversal operations without any assumption about the dimension. This dimension independent programming eases software development considerably and reduces the probability of errors.

4.1 Generic Topology Library: GTL

Topological functionality is mostly needed to fulfill the requirements of discretization schemes. All these schemes need a set of neighboring elements based on the topological property of **incidence**. Therefore the GTL [12] provides comprehensive incidence traversal and orientation operations with a generic interface similar to the C++ STL [13] and is based on GrAL [9]. Problems which can be formulated only with difficulty using existing libraries, can thereby be handled easily.

The following example presents an incidence traversal mechanism starting with an arbitrary cell iterator which is evaluated on a cell complex (grid). Then a vertex on cell iterator is initialized with a cell of the cell complex (cell container). The topological traversal is started with the `for` loop. During this loop an edge on the vertex iterator is created and initialized with the evaluated vertex. This edge iterator starts the next topological traversal. The `valid()` mechanism is used, because there is no `end()` iterator on inter dimensional objects.

```
cell_iterator c_it = cell_complex.cell_begin();
for(cell_vertex voc_it(*c_it); voc_it.valid(); ++voc_it)   {
  for(vertex_edge eov_it(*voc_it); eov_it.valid(); ++eov_it)  {
    //operations on edges
  }
}
```

4.2 Generic Discretization Library: GDL

One of the base operations in the context of PDEs is the assembly procedure of discretized equations into a matrix. To ease this procedure the GDL was developed. This library implements the ability to specify whole equations in a concise yet expressive way. The availability of a topology or traversal library, e.g. the GTL, is a fundamental requirement to specify equations in a functional way. Together these two libraries offer mechanisms to separate discrete mathematics into topological and numerical operations. To present the application of the GDL, we examine a simple equation:

$$\sum_{\mathrm{v}\to\mathrm{e}} (\Delta_{\mathrm{e}\to\mathrm{v}}\ \mathrm{quan}) = 0$$

where $\mathrm{v} \to \mathrm{e}$ denotes the traversal of all edges incident to the vertex, $\mathrm{e} \to \mathrm{v}$ denotes the traversal of the vertices incident to the edge, quan denotes the quantity located on a vertex, and Δ denotes the difference of this quantity.

The implementation with the GTL and the GDL without any dependence on the dimension or the type of cell complex is presented in the next code snippet. The addition over all edges incident to the given vertex are traversed by the `vertex_edge` expression with the `sum` functor. All elements have a standard local orientation, e.g. an edge provides a source and a sink vertex. We define an orientation function $\mathcal{O}(a, b)$ between an edge and a vertex, which returns $+1$ if a vertex coincides with the source and -1 if the vertex coincides with the sink (Figure 1).

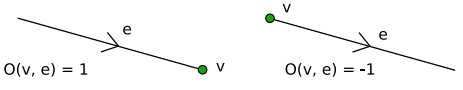

Fig. 1. Orientation of an edge. The orientation function returns either $+1$ or -1 depending on whether the vertex is the sink or the source of the edge.

Finally, the `_e` and `_1` are local variables (placeholders). The `_e` variable passes the edge into the next context `[ quan * orient(_1, _e) ]`, and `_1` stands for the local vertex.

```
for (v_it  = cell_complex.vertex_begin();
     v_it != cell_complex.vertex_end(); ++v_it)  {
  equation = sum<vertex_edge>
  [
    sum<edge_vertex>(0.0, _e) [ quan * orient(_1, _e) ]
  ](*v_it)
 // .. evaluate the equation object
}
```

The complex resulting from this mapping is completed by specifying the current vertex object `*v_it` at run-time, which clearly demonstrates the compile-time and run-time border. The datatype `equation` is explained in more detail in Section 4.4.

4.3 Matrix Assembly

Differential equations are discretized on elements of the topological structure and entered in a matrix by assembly methods. In general there exist two main methods of equation assembly which differ in the type of sub-matrices assembled in one step.

Element-wise assembly (Figure 2) is typically used in the finite element method. All cells (finite elements) are traversed and for each of the cells a local matrix is calculated. This matrix introduces coupling factors between the various shape functions. As each shape function is mapped to an element of the underlying cell complex, couplings between functions can be seen as couplings between values on elements. The local matrix entries are written into the global matrix according to a global vertex/cell numbering scheme. The finite volume scheme uses a different

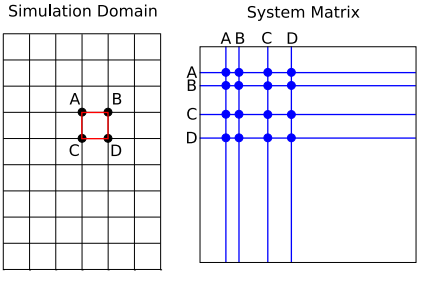

Fig. 2. Element-wise assembly. All cells are traversed and sub-matrices are calculated. The sub-matrices are inserted into the system matrix.

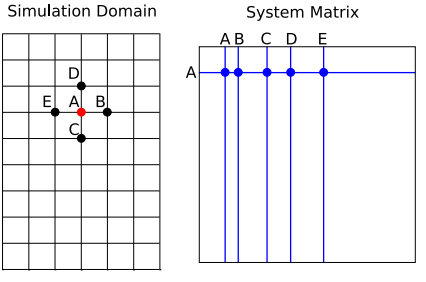

Fig. 3. Line-wise assembly. All vertices are traversed and linear equations are assembled. Linear equations are inserted into the system matrix.

technique of matrix assembly (Figure 3). The differential equation is discretized on a vertex of the simulation domain. Couplings to other values are described by sums over topological elements. One of the major advantages of this method is that each degree of freedom causing a matrix entry has its own governing equation. This also implies that the matrix regions of assembly are disjoint, which allows a larger degree of parallelization, because each assembling element has exclusive access to matrix lines.

4.4 Linear Functions

In order to simplify the line-wise assembly method, e.g. for finite volumes, we introduce the notion of a linear function data type. We consider some differential operator $\mathcal{L}(\psi)$ and the differential equation $\mathcal{L}(\psi) = 0$. Figure 4 shows a one-dimensional simulation domain with a quantity distribution. While the depicted values are not the solution of the considered equation, the residual can be determined by using the finite volume formulation. We consider not only the residual value but also the effects of linear changes of single values on the residuum. In

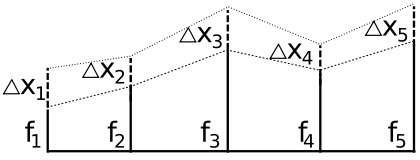

Fig. 4. Linear equation. The linear equation data structure stores the value as well as the dependence on the unknown variables Δx_i.

order to provide an environment which is able to handle small variations, we assign an index i to all variables. The small variations are called Δx_i and the function value is denoted by ψ_i. A function value including dependencies thus yields $\psi_i + \Delta x_i$. Based on these considerations we introduce addition.

$$(\psi_i + \Delta x_i) + (\psi_j + \Delta x_j) = (\psi_i + \psi_j) + (\Delta x_i + \Delta x_j) \; . \tag{1}$$

In order to define a closed algebraic structure, we use the concept of a linear function. This function can be written in the following manner

$$\sum_i a_i \cdot \Delta x_i + c_i \; . \tag{2}$$

The relations described in Equation 1 can be generalized to algebraic expressions. A data structure which implements these operations consequently, provides the following property: specifying the residuum and replacing the function value ψ_i by $\psi_i + \Delta x_i$ results in a linear equation for the values of Δx_i.

The following example uses finite volume schemes in order to assemble a simple Laplace equation. We use the linear equation data type instead of the standard numerical types (e.g. `double`) and replace the function values by linear equations. The constructor of the linear equation contains the value ψ_i as well as the index i and returns an equation object with the meaning $\psi_i + \Delta x_i$. The following code snippet presents an application of the introduced linear function concept to obtain the formulation of the Laplace equation:

```
linear_equation laplace_eqn;
vertex_edge eov_it(vertex)
for(; eov_it.valid(); ++eov_it) {
 linear_equation  equation;
 edge_vertex voe_it(*eov_it);

 for(; v_it.valid(); ++v_it)
 {
   equation += linear_equation(f(*v_it), i(*v_it)) *
                                orient(*v_it, *eov_it);
 }
 equation *= A(*eov_it) / d(*eov_it);
 laplace_equ += equation;
}
```

The linear equation can also be specified using functional programming.

```
laplace_equ = sum<vertex_edge>
[ sum<edge_vertex>(_e)
  [ lineqn(psi(_1), i(_1)) * orient(_1, _e)
  ] *  A(_1) / d(_1)
](vertex);
```

5 Application Design Using the GSSE

We demonstrate the development of simple applications which solve partial differential equations based on the facilities of the GTL and the GDL. These equations can be divided into elliptic equations, such as the Laplace or Poisson equation, parabolic equations which describe diffusive processes, and hyperbolic equations, such as wave equations. Examples for each of these equations are often found in TCAD. The first example presents the capability of reducing complex discretization to simple topological iterations, mostly based on the GTL. The second example utilizes the GDL based on a piecewise construction of functional parts to form a complex equation.

5.1 Maxwell's Equation

We present an implementation using Yee's algorithm for Maxwell's equations [14] in the following. While the Yee formulation makes use of staggered grids, the application on structured topologies based on the GTL and inter-dimensional iterators causes an enormous simplification. Instead of special grids, we employ different dimensional elements such as edges and faces for the representation of the electrical field strength and magnetic field. It turns out that the tensorial character of the quantities fits into the dimensionality concept of the topological elements. We present the special case of a transversal magnetic mode only. The following formulation can be derived by applying the Yee discretization scheme.

$$\begin{aligned} E_z^{n+1}(i,j) = E_z^n(i,j) \; &+\Delta t \, \frac{l}{\Delta x}[H_y^{n+1/2}(i+\frac{1}{2},j) - H_y^{n+1/2}(i-\frac{1}{2},j) \\ &-\Delta t \, \frac{l}{\Delta y}[H_x^{n+1/2}(i,j+\frac{1}{2}) - H_x^{n+1/2}(i,j-\frac{1}{2}) \end{aligned}$$

With the transfer of all index calculations to topological iteration and traversal mechanisms, e.g. the electric field quantity to edges E_e and the magnetic field quantity to facets H_f the formula can be rewritten as:

$$E_\mathrm{e} - E_\mathrm{e}^\mathrm{old} = \Delta t \; l_\mathrm{e} \Delta_{\mathrm{e}\to\mathrm{f}}[\frac{H_\mathrm{f}}{A_\mathrm{f}}],$$

where $\mathrm{e} \to \mathrm{f}$ denotes the traversal of all facets incident to the edges, and A_f represents the area of the corresponding facet. The evaluation of all quantities on their corresponding dimension and topological objects is completed automatically. The final source code is presented in the following code snippet. The minimal requirement to specify such complex equations can be seen clearly.

```
equation_E += dt * l * sum<edge_facet>(0.0, _e)
  [
    H / A * orient(_e, _1)
  ]
```

5.2 Applications for Device Simulation

Good understanding of electron transport in semiconductors is one key requirement for the progress in microelectronic devices. For device simulation we present the drift-diffusion model which reads in a simple form for electrons [5]:

$$\begin{aligned} \operatorname{div}(\varepsilon \operatorname{grad}(\Psi)) &= \mathrm{q}(n - N_{\mathrm{D}}) \\ \operatorname{div}(\mathbf{J}_{\mathrm{n}}) - \mathrm{q}\,\partial_t n &= \mathrm{q}R \\ \mathbf{J}_{\mathrm{n}} &= \mathrm{q}\,n\,\mu_{\mathrm{n}} \operatorname{grad}(\Psi) + \mathrm{q}\,D_{\mathrm{n}} \operatorname{grad}(n) \end{aligned} \tag{3}$$

Discretizing the continuity relation (as presented in Equation 3) using the Scharfetter-Gummel scheme [15] results in:

$$\sum_{\mathrm{v}\to\mathrm{e}} \mathrm{q}\,\mu_{\mathrm{n}} U_{\mathrm{t}} \frac{A}{d} \sum_{\mathrm{e}\to\mathrm{v}} n \operatorname{Bern}\Big[\sum_{\mathrm{e}\to\mathrm{v}} \frac{\psi}{U_{\mathrm{t}}}\Big] \tag{4}$$

$\operatorname{Bern}(x) = x/(e^x - 1)$ represents the Bernoulli function and U_{t} describes the thermal voltage. This discretized form can be transformed into C++ code using our formalism to yield:

```
// Poisson equation
equation_poiss=
sum<vertex_edge>
[ A / d * eps *
  sum<edge_vertex>(0.0, _e) [pot*orient(_1, _e)]
] - q*(n-nD)

// Continuity equation for electrons
equation_n = sum<vertex_edge>
[ q * mu_n * U_t * A / d  *
  sum<edge_vertex>(0.0,_e)
  [
    orient(_e, _1)*n(_1)*
    Bern(locate(_e) [sum<edge_vertex>[pot*orient(_1,_e)]/U_t])
]]
```

Due to the functional specification, a special mechanism has to be introduced `locate(_e)` to obtain the edge information in the innermost loop. The main problem with this type of programming in C++ is that the expression between `[]` opens up a new scope and new local variables. The `locate` function passes the edge information from the second `sum` to the `Bern` function.

5.3 Number of Code Lines

To implement a complete application, one line of code is used to import a given cell complex (mesh) from a file and another line is used to assemble the linearized functions into a generic matrix interface. The overall number of lines of code to implement an application can thereby be greatly reduced. This results in a minimization of maintenance of source code as well as the learning time for new developers.

6 Performance

The basic parts of how to achieve high performance in C++ are based on the usage of templates. Therewith the compiler's data-type-based function selection at compile time leading to a global optimization with inlined function blocks is possible. Additionally lightweight object optimization [16] with frequent allocation to registers is made available.

To circumvent the problems with benchmarking different techniques, we restrict the performance analysis on a simple but often used detail, namely the addition of several small matrices which occur in the discretization schemes described before. This simple expression can be compared to other benchmark studies [17]. The test is performed using the vector addition $A_f = A_b + A_c + A_d$, evaluated with different vector sizes on a Pentium 4 (2.4 GHz) with the GCC 4.1.0. Several approaches, a naive C++ implementation with `std::vector<T>`, Blitz++, a simple version of expression templates [18] , the C++ `std::valarray`, and finally the GDL approach are compared to a hand-optimized Fortran 77 implementation (F77) as can be seen in Figure 5 . Although functional and generic programming

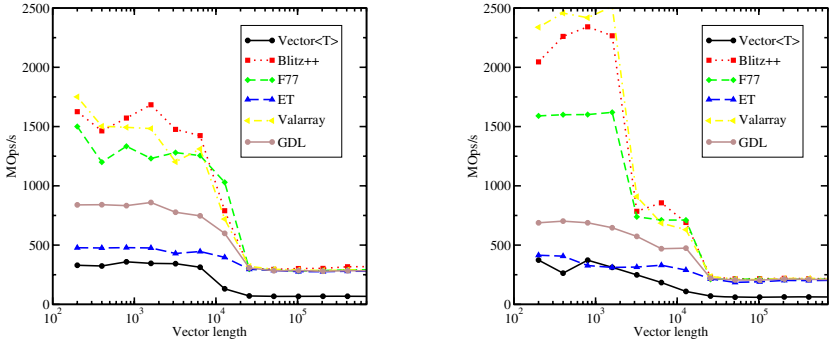

Fig. 5. Performance of the evaluated expression on a P4 (left) and an AMD64 (right)

support the parallelization significantly, the currently used computer technology restricts this effort due to their architecture. For vector lengths smaller than 10^4, cache hits reveal the full computation power of the CPU, longer vectors show the limits imposed by memory band width.

7 Conclusion

The application of several modern programming paradigms solves the problem of portability while insuring high performance by providing orthogonal means of optimization. Currently no language other than C++ offers sufficient support for all the necessary programming paradigms to enable this high-level abstraction at unmatched performance. As we have demonstrated in the complex field of TCAD, applications can be developed with a reasonable amount of effort.

References

1. Veldhuizen, T.L.: Using C++ Template Metaprograms. C++ Report 7(4), 36–43 (1995) (Reprinted in C++ Gems, S. Lippman (ed.))
2. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, London (2002)
3. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II Differential Equations Analysis Library, Technical Reference, `http://www.dealii.org/`
4. Zienkiewicz, O.C., Taylor, R.L.: The Finite Element Method. McGraw-Hill, Berkshire, England (1987)
5. Selberherr, S.: Analysis and Simulation of Semiconductor Devices. Springer, Heidelberg (1984)
6. Sabelka, R., Selberherr, S.: A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. Microelectronics Journal 32(2), 163–171 (2001)
7. IμE: MINIMOS-NT 2.1 User's Guide. Institut für Mikroelektronik, Technische Universität Wien, Austria (2004), `http://www.iue.tuwien.ac.at/software/minimos-nt`
8. Fabri, A.: CGAL- The Computational Geometry Algorithm Library (2001), `http://citeseer.ist.psu.edu/fabri01cgal.html`
9. Berti, G.: GrAL - The Grid Algorithms Library. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science - ICCS 2002. LNCS, vol. 2331, pp. 745–754. Springer, Heidelberg (2002)
10. Härdtlein, J., Linke, A., Pflaum, C.: Fast Expression Templates - Object Oriented High Performance Programming. Eng. with Comput. 3515, 1055–1063 (2005)
11. Heinzl, R., Spevak, M., Schwaha, P., Grasser, T.: Concepts for High Performance Generic Scientific Computing. In: Proc. of the 5th MATHMOD, Vienna, Austria (2006)
12. Heinzl, R., Spevak, M., Schwaha, P., Selberherr, S.: A Generic Topology Library. In: Library Centric Sofware Design, OOPSLA, Portland, OR, USA, pp. 85–93 (2006)
13. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)
14. Yee, K.S.: Numerical Solution of Initial Boundary Value Problems involving Maxwell's Equations in Isotropic Media. IEEE Trans. Antennas and Propagation 14(1), 302–307 (1966)
15. Scharfetter, D., Gummel, H.: Large-Signal Analysis of a Silicon Read Diode Oscillator. IEEE Trans. Electron Dev. 16(1), 64–77 (1969)
16. Siek, J., Lumsdaine, A.: Mayfly: A Pattern for Lightweight Generic Interfaces. In: Pattern Languages of Programs (1999)
17. Heinzl, R., Schwaha, P., Spevak, M., Grasser, T.: Performance Aspects of a DSEL for Scientific Computing with C++. In: Proc. of the POOSC Conf., Nantes, France, pp. 37–41 (2006)
18. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley, London (2004)