International Journal of Parallel, Emergent and Distributed Systems
Vol. 24, No. 6, December 2009, 505–520

Taylor & Francis
Taylor & Francis Group

GUIDE: Parallel library-centric application design by a generic scientific simulation environment

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

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

(*Received 7 January 2009; final version received 19 January 2009*)

Techniques for library-centric application design have already proven to be very useful in the past. The current gain in computer performance is shifted towards the utilisation of multi-core processors which extends the importance of this type of application design in the field of scientific computing, which also poses new difficulties. A parallel generic scientific simulation environment has been developed to ease this transition from single-core to multi-core systems without additional development activity.

Keywords: parallel library centric application design; parallel computing; generic environment; generic programming; high performance programming; scientific computing

1. Introduction

The growing complexity of physical models leads to an increase of the amount of source code, thereby significantly increasing the importance of efficient code development and maintenance. This issue can be addressed by providing modular building blocks which can be tested and refined independently of each other and seamlessly integrated into the desired applications. Hence, the concept of library-centric application design [13,16] and the availability of a set of high performance libraries considerably eases the development of highly scalable applications.

The evolution of complexity leads to a growing number of software packages for different types of problems. Additionally, this multitude of packages is usually not organised in a way that allows immediate algorithmic reusability. Different projects have contributed components to the field of scientific computing, but up to now, no general set of generic data structures or algorithms suitable for scientific computing in general has been developed. Some works have developed modules, e.g. for generic grid components [6], which was a major contribution to software components for scientific computing with an in-depth analysis of the problem related to algorithms operating on computational grids. This effort analysed the relationship between data and algorithms based on topological and combinatorial concepts. A small set of kernel components was developed, which greatly eases the specification of algorithms for grid applications.

The current trend, however, is to combine several programming languages, resulting in multi-language applications [21]. Different languages are utilised, each within the field where it performs best. Languages such as Python are used to connect different modules. However, problems of interface specification and implementation arise with the combination

*Corresponding author. Email: heinzl@iue.tuwien.ac.at

ISSN 1744-5760 print/ISSN 1744-5779 online

DOI: 10.1080/17445760902758545
http://www.informaworld.com

of several programming languages, further complicating matters. The handling of different languages on different platforms is even more difficult. In the field of scientific computing, the performance aspects must be handled orthogonally to the development of applications. Optimisations can thereby be treated separately. With the multi-language approach performance aspects cannot be considered orthogonally because of the use of compiled modules which require an interface layer in order to build applications.

To secure further gain in computing performance the semiconductor industry has shifted the upcoming processor upgrades to multi-core systems, where the gain in processing power is obtained by using an increased number of processing cores in the CPUs. Current single desktop computer systems can already handle large scientific simulation problems locally. Nevertheless, industrial problems and parameter settings often require large scale simulations which have to be performed on supercomputers, where the individual nodes of these supercomputers often do not differ in the execution speed from the desktop computers anymore. Instead they are heavily parallelised with a large amount of shared memory. Only for the final parameter setting, the application has to run on a supercomputer, where the amount of CPUs is drastically increased, as is the amount of available memory. This scaling also has to be reflected in current application design.

While scientific simulations have been among the first applications to embrace parallelisation, still not all fields of scientific computing make use of it. Most related work focuses on parallel toolkits within their frameworks [20]. Our approach is based on providing modular blocks which can be used on top of existing libraries. Also, utmost emphasis is placed on the issue that already tested and stable code has to be parallelised without modification of existing source code. This is not only important to speed up the development of parallelly executing applications, but also to preserve the already invested time to develop, debug and calibrate an application.

We therefore present the library-centric application design approach by the generic scientific simulation environment (GSSE [13,16])[1], domain-specific embedded language (DSEL) concepts and their realisation in C++ as well as two parallel approaches usable by the GSSE:

- Various multi-threading libraries are used in conjunction with the topological partitioning provided by the GSSE to subdivide the amount of topological objects. Several discretisation schemes and the assembly times benefit greatly from this approach.
- Recent developments towards parallel STL [33] techniques can easily be incorporated, which require a recompilation step, where all STL algorithms and the GSSE algorithms built on top of these algorithms are then executed in parallel. These techniques are already on their way of being incorporated into the GCC [32].

These approaches enable the utilisation of several parallelisation techniques without altering the developed, tested and calibrated applications by simple re-compilation steps.

2. Related work

In the last decade, many approaches towards implementing a (parallel) simulation environment for sub-parts of scientific computing, e.g. the solution of partial differential equations has been taken. Most of the tools resulting from these attempts use topological structures which are specialised to work with a particular discretisation scheme. This reduces resource use but comes at the cost of greatly diminishing the flexibility

of topological traversal. However, for some reasons it might be advantageous to implement discretisation schemes based on a mixed finite element/finite volume scheme which requires such traversal operations. The following brief overview shows some significant steps towards a more flexible and generic application design:

- 1995: multi-paradigm language's run-time performance (C++) equal to Fortran [35]
- 1997: emergence of the generic programming paradigm by C++'s STL [3,25]
- 2002: generic data structure interface and generic traversal operations [7]
- 2002: generic graph library [31] with the introduction of the concept of a property map
- 2003: concepts for separation of traversal and data access [2]

A major step towards a more flexible use of data structures was developed by the Boost Graph Library (BGL [31]). This library implements a generic interface to enable access to arbitrary graph structures but hides the details of the actual implementation. The interfaces make it possible for any graph library based on these interfaces to be interoperable with the BGL. The approach is similar to the one taken by the C++ STL to ensure the interoperability of the various algorithms and containers. The property map concept [31] was also introduced. Unfortunately, the BGL was designed for graphs only and neither lower nor higher dimensional data structures can be handled.

The Computational Geometry Algorithm Library (CGAL [9]) is another important collection of reusable components for a great number of geometrical algorithms and data structures in a generic library-centred approach, such as two- and three-dimensional modules for mesh generation, Voronoi diagrams and surface mesh simplification. The main contribution of CGAL is the concept of an algebraically parametrised kernel [24] related to the actual implementation robustness of mathematical operations.

The Grid Algorithms Library (GrAL [6]) was one of the first contributions to the unification of data structures of arbitrary dimensions for the field of scientific computing. A common interface for grids with dimensionally and topologically independent access and traversal was designed. Mathematical concepts for topological spaces were introduced and applied to grids. Applications for the field of solving PDEs were presented, but no concrete implementation was given.

The Sophus C++ library [12] aims at coordinate-free formulations. This library implements grid components for sequential and parallel high performance computing. A field layer for numerical discretisation schemes, a tensor layer to handle various quantities related to a coordinate systems, and finally an application layer with solver interfaces were developed. However, this approach suffers from severe abstraction penalties and requires a code transformation tool [4].

deal.II [5] provides a framework for finite element methods and adaptive refinement for finite elements. It uses modern programming techniques of C++ and enables the treatment of a variety of finite element schemes in one, two or three spatial dimensions and of time-dependent problems. Modern finite element algorithms, using, among other aspects, sophisticated error estimators, and adaptive meshes can be developed easily.

The FEniCS project [22], which is a unified framework for several tasks in the area of scientific computing, is a great step towards generic modules for scientific computing. Up to now most of the modules are in a prototype state.

The template numerical toolkit (TNT [26]) is a collection of interfaces and reference implementations of numerical objects (matrices) in C++. The toolkit defines interfaces

for basic data structures, such as multidimensional arrays and sparse matrices, commonly used in numerical applications.

These libraries are an important step towards library-centric application design. But most of these libraries were not 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.

3. Library-centric software design

Libraries have become a central part of all major programming efforts connected to scientific computing. As a consequence the possible library-centric software design can be regarded as a methodology for designing applications as an assembly of single components with a low degree of coherence and a high degree of orthogonality, where the following basic principles are essential for successful generic components [23]:

- Functions should not depend on the global status but only on the arguments.
- Every function is either general or application-specific.
- Every function that could be made general should be made general.
- The global state should be documented by describing both the semantics of individual variables and the global invariants.

An important step towards library design is the definition of interfaces based only on concept requirements [11,30] in order to avoid monolithic application development which always leads to redevelopment of parts or complete applications. Instead already existing concepts and modules which have already proven successful can be used. Essential requirements related to an optimal library development can be summarised as follows:

- A set of libraries has to be complete and must provide a systematic taxonomy [18,23] to guide the design of an application. Many different types of applications can be written using these libraries and adaptors.
- Libraries should be generic which means that they are usable [3] for a broad range of different applications. Each of the software components is not only written for a very specific purpose, but for a manifold of problems.
- Constraints on performance are required for each of the libraries [11] to obtain an overall high performance application.
- The interoperability of a library is not adversely affected by its completeness [30]. Even if a library is complete by itself, it provides standardised interfaces which guarantee compatibility for data structures which have not been foreseen in its initial design.

In the past, the main drawback related to library-centric design was the absence of programming paradigms supporting this type of design. An example can be seen in one of the features most used in programming: loops. The imperative style of using loops offers a great degree of flexibility during the development process, with regard to maintainability and side-effects. Nevertheless, simple loops require local variables to maintain a state, and only the imperative style of data access is available. These issues result in codes that lack maintainability, scalability and lead to unnecessarily error-prone implementation bodies. Another issue which also does not support the concept of component reuse by itself is the object-oriented programming paradigm, which complicates the interoperability of its software modules. An advantage of imperative and object-oriented programming is that no sophisticated programming techniques have to be taught and learned to be able

to understand or extend the code. But it also results in an additional drawback, which directly stems from the missing expressiveness of the code. Highly optimised code sections can barely be extended by other developers.

Reusability, orthogonality, enhancement capabilities and performance are all issues which can be eased by using paradigms other than imperative and object-oriented programming. The most important issues related to library-centric application design are the transitions from these programming paradigms to generic programming and the efficient implementation in programming languages which then offer generality and specialisation at the same time.

The combination of different programming paradigms fits the scenario of scientific computing exceptionally well. The generic programming paradigm establishes homogeneous interfaces between algorithms and data structures without sub-typing polymorphism. Functional programming eases the specification of equations and offers extendable expressions while retaining the functional dependence of formulae by higher order functions. Also, this type of specification of access, algebraic manipulation and traversal enable to circumvent the problems of an imperative implementation. The features of meta-programming offer the embedding of domain-specific terms and mechanisms directly into the host language as well as compile-time algorithms to obtain optimal run-time. Developments toward an alternative compilation model and active library design are also an important step [36,39]. However, reusability of traditional libraries is often extremely limited due to the following issues:

- *Numerical data types*. There are numerous well-known numerical data types which also are often optimised for special applications in order to yield high performance. Only with generic interfaces can these performance-enhancing measures be used in different kinds of applications.
- *Topologies*. Numerical schemes often require different underlying topological data structures. While some applications perform well using structured grids, other applications require unstructured meshes with varying local feature sizes. Although the nature of these topologies is totally different, standardised interfaces for all topological data types have to be provided.
- *Different dimensions*. Special symmetries that are encountered in many problems of scientific computing can be used to reduce the effective dimension of a calculation. Even though all problems can be treated in their full dimension, an enormous gain in performance by using lower dimensional data structures cannot be neglected.
- *Equation system assembly*. Most of the solver mechanisms require an initialisation of the values of their own interfaces. Therefore, an interface which abstracts these specialties and makes the solvers accessible in a general manner is required. With such an interface the governing equations can be formulated independently of the actual data structures of the solver.
- *Solution of large equation systems*. A lot of problems result in large equation systems which have to be solved. There are solvers available for various special cases, which perform well under certain circumstances but fail to converge sometimes. Therefore, interface design has to guarantee that different solvers can be used.

To circumvent the stated issues, a set of requirements for library-centric application design in the field of scientific computing is given in the following to allow the transformation of the concepts for scientific computing into generically applicable and efficient software

components. The achieved library-centric design is facilitated, as the following criteria are met:

- The environment is complete, so all applications can be written exclusively using its libraries (as well as standard libraries). Indeed, completeness increases usability enormously, because no components have to be added while existing components can be adapted.
- The components of the environment are usable for a broad range of different applications.
- The interoperability of the environment is not affected by its completeness. Even though all the libraries can be used by themselves, they provide standardised interfaces, which guarantees compatibility for data structures which have not been foreseen in the initial design.

An additional requirement for application design is related to an efficient use of programming paradigms. A basic layer can be identified, which has to implement a formal topological interface for container properties as well as for traversal, thereby establishing a consistent interface for data structures and quantity storage. The object-oriented and generic programming paradigms are best suited to accomplish this. On top of this layer, functional expression specification facilities are required, which can be modelled best by the functional programming paradigm. The concept of a DSEL in C++ requires the additional concept of meta-programming resulting in an active library concept to implement and guarantee an overall high performance, as described next.

4. Domain-specific embedded languages

Closely related to the development approaches which have been discussed above is an important concept for high-level languages, such as C++, DSELs. First, a domain-specific language (DSL) is defined by a set of symbols as well as by a well-defined set of rules specifying how the symbols are used to build well-formed compositions [1]. The domain-specific part of a DSL can then be employed by a simplified grammar and an increased expressiveness. The possibility of writing code in terms close to the level of abstraction of the initial problem domain is the characteristic property of DSLs.

The next relevant issue regarding a DSL is interoperability. A possible way of dealing with interoperability is to integrate the DSL into a host language, finally obtaining a DSEL, e.g. YACC [19]. All software engineering tasks, such as designing, implementing and maintaining the DSL, are reduced to implementing and maintaining a library of the host language. More importantly, the interoperability can be enhanced to the highest level, due to the fact that both concepts are now available in one host language. All the libraries already developed and their functionality are available at the same time. To summarise, the advantages of DSELs are:

- Abstracting the underlying language/system/compiler in the direction of the user/domain expert.
- The overhead of learning or adopting a new language is greatly reduced.
- Reduction of documentation due to expressive names and self-documentation.
- Validation of semantics, e.g. by a compiler.

The advent of having multiple paradigms available in a single programming language creates new possibilities for DSLs. The advantages and disadvantages of several programming paradigms and the reduction of specification and implementation effort clearly suggest the use of DSELs. Maintenance and further development for an extra

Table 1. Language comparison.

Language	*Operator overloading*	*Parametric polymorphism*	*Functional mechanisms*	*Time of evaluation*
C	No	Partial (macros)	No	Compile/run-time
D	Partial	Yes	No	Compile/run-time
Java	No	Partial (version)	No	Run-time
C#	No	Partial (version)	No	Run-time
Haskell	Yes	Partial (version)	Yes	Run-time
C++	Yes	Yes	Yes	Compile/run-time

transformation tool is thereby greatly reduced, compared to, e.g. the transformation tool for Sophus [4]. But up to now, the implementation of a DSEL for non-trivial areas is far from simple or user-oriented. Also, DSLs require various mechanisms from the host languages, where the following features are almost mandatory:

- *operator overloading* means that different operators, such as $+$, $-$ can be overloaded.
- *parametric polymorphism* means that a kind of template system is available.
- *functional mechanism* means that higher order and lambda objects are available.
- *time of evaluation* means that program code can be specified for compile and for run-time.

As can be seen from the brief comparison of languages provided in Table 1, only a few languages are available which allow an efficient modelling of operators. Languages such as Haskell were not developed with a focus on an overall high performance, but they still offer the definition of new operators which extend the built-in language syntax. Based on this overview, the only language offering the best support for DSELs with syntactic expressiveness as well as run-time efficiency is C++:

- static type system
- ability to achieve near-zero abstraction penalty [28,39]
- powerful optimisers
- template system that can be used to:
  - generate new types and functions
  - perform arbitrary computations at compile-time
  - dissect existing program components
- set of built-in symbolic operators[2] that can be overloaded.

This currently unique combination of features results in C++ being the only possible language for the concepts given here.

In addition to using C++ as a host language for a DSEL, the multi-paradigm approach offers a large degree of freedom in the implementation of high performance scientific code and even applications [29,37].

An important issue of the generic programming paradigm realised in C++ is that optimisation of a whole application generates a run-time performance for various tasks without comparison. One major advantage is the technique of template meta-programming of C++, which means that a DSEL can function as an extension to the general-purpose host language. Meta-programming and DSEL are also actively developed areas [38]. An example is given next which illustrates the embedded nature of a grammar specification in C++. The first part yields the normal grammar specification used by YACC:

```
group       ::= '(' expression ')'
factor      ::= integer | group
term        ::= factor (('*' factor) | ('/' factor))*
expression  ::= term (('+' term) | ('-' term))*
```

Next, an example of Spirit [8] is given. Spirit is an object-oriented, recursive-descent parser and output generation framework enabling target grammars written entirely in C++ as a DSEL.

```
group       = '(' >> expression >> ')';
factor      = integer | group;
term        = factor >> *(('*' >> factor) | ('/' >> factor));
expression  = term >> *(('+' >> term) | ('-' >> term));
```

Another advantage of the DSEL concept is the continuous improvement of compiler technology, which drastically increases the performance of high-level code. Recent performance analyse have shown, that each compiler generation increases the overall run-time performance [14].

In combination with an application library, any general-purpose programming language can act as a DSL. DSLs were developed in the first place, because they can offer domain-specificity in better ways:

- Appropriate or established domain-specific notations are usually beyond the limited user-definable operator notation offered by general-purpose languages. A DSL offers domain-specific notations from the start. Their importance cannot be overestimated, as they are directly related to the suitability for end user programming and, more generally, an improved programmer productivity can be associated with the use of DSLs.
- Appropriate domain-specific constructs and abstractions cannot always be mapped in a straightforward way to functions or objects that can be put in a library. This means a general-purpose language using an application library can only express these constructs indirectly. Again, a DSL would incorporate domain-specific constructs from the start.

The following issues should be addressed when building a DSEL:

- Interoperability: One of the most important parts in a heterogeneous environment such as scientific computing.
- Declarativeness and expressiveness: Scientists are used to a mathematical notation and not to the constructs of a programming language. The distance from the mathematical notation should be reduced as much as possible.
- Efficiency: A simple example within the DSEL should compile without any overhead.
- Static type safety: The host compiler should catch as many problems as possible.
- Maintainability and scalability: Simple changes to the problem should only result in simple changes to the model within the host language. If the environment does not support a feature, it should be extensible easily.

5. Generic scientific simulation environment

The required change in application design to cope with current requirements of high-performance distributed computing already indicates the shift to library-centric application design. This final change offers the possibility of DSEL techniques to suit different groups of users by providing a layer of domain specific elements for scientific computing. As has already been demonstrated [15,16], the goals of high run-time performance and genericity do not have to be contradictory.

Our approach deals with the identification and implementation of building blocks for multi-physic simulation with special consideration regarding easy specification of various types of discretised differential equation as well as run-time performance. Several tasks in scientific computing can be considerably eased by the utilisation of functional programming. Unfortunately several tasks defy the nature of stateless description. All different types of storage mechanisms as well as streaming processes cannot be easily described by functions. Here, the actually stored elements are the important parts and not their functional description.

Based on these components [13],[3] the GSSE is specified by a direct implementation of the concepts given. An overview is depicted in Figure 1 where the GTL represents the traversal library whereas the GFL overviews the components of the functor library.

5.1 GSSE: Topological traversal

The C++ STL offers great mechanisms for sequential containers and the corresponding algorithms, but for more complex data structures a common way of accessing data or iteration is not available at the moment. Different developments, such as the BGL or CGAL, offer their own mechanisms derived from the STL. The GSSE offers traversal mechanisms for all different types of data structures, such as simple sequential containers and meshes for any dimension. Topological traversal describes the iteration over elements of a data structure. The incidence information for cells is stored in a connection matrix.

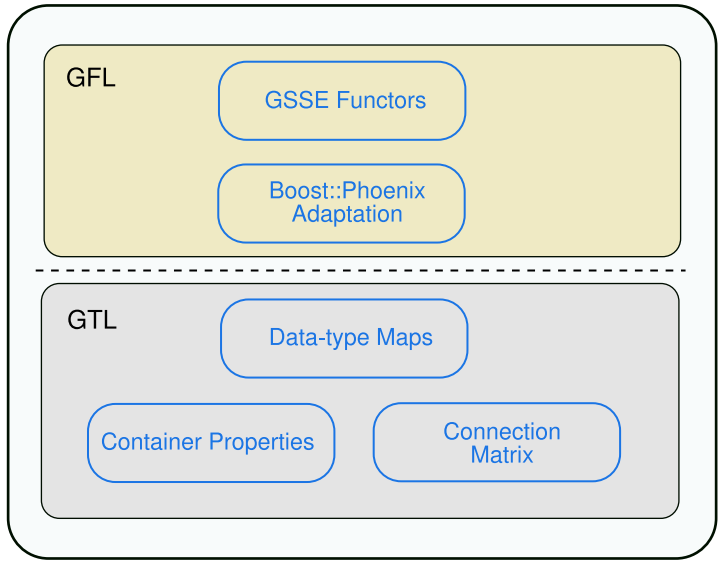

Figure 1. Building blocks of the GSSE.

It is basically a two-dimensional data structure which stores different types of topological objects, e.g. edges or cells, and their connection to other objects, e.g. vertices. Container properties, such as dimension and type of cell, e.g. tetrahedral cell, are also stored.

The topological traversal library provides incidence and adjacence traversal operations for various topological elements. Thereby enabling the necessary topological operations for various tasks of scientific computing, e.g. traversal concepts required by discretisation schemes. This traversal concept allows the formulation of algorithms based on this interface independently of the actual implementation of the topological data structure or the dimension considered. Such a consequent use of the topological interface leads to dimensionally and topologically independent application design and is a key issue for parallel application design.

Figure 2 provides examples of such traversal types and the possible relationships between topological elements of different dimensions. The first row shows all edges, faces and cells which are incident with the same base vertex (a)–(c), while the first column shows vertices which are incident with one base edge, face or cell (d,g,j).

The connection matrix is the underlying data structure for these types of operations. To specify these types of traversal efficiently, a concept called *data-type map*, where each container specifies its traversal possibilities, is used. The following example depicts a vertex on cell traversal, where the containers from `boost::mpl` are used.

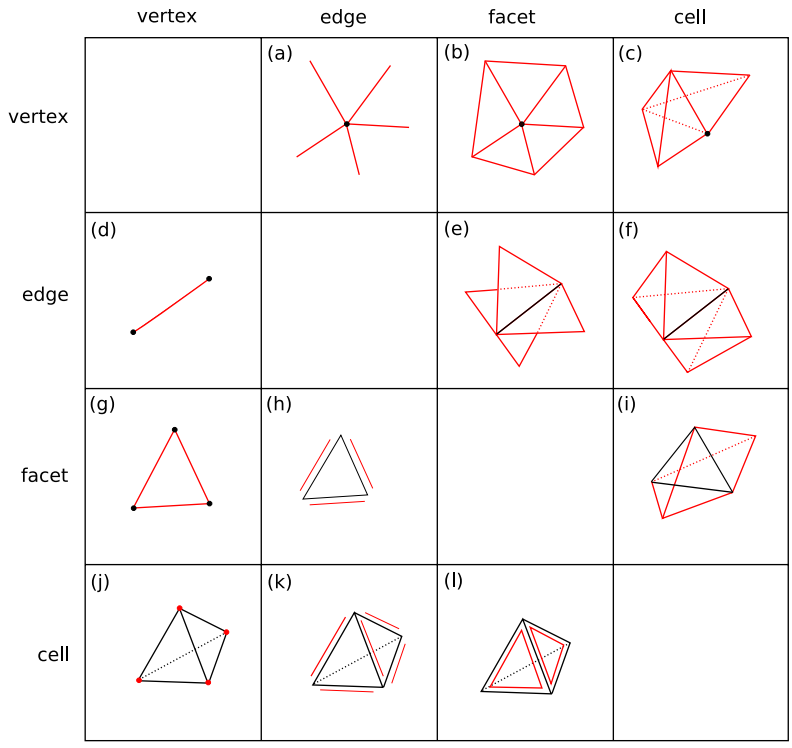

Figure 2. Traversal methods of a three-dimensional tetrahedral cell complex. The rows illustrate traversal schemes of the same base element, whereas columns depict traversal schemes of the same traversal element.

```
map< pair<pair<typename CC::vertex_type, typename CC::edge_type >,
                typename CC::vertex_on_edge_iterator  >,
     pair<pair<typename CC::vertex_type, typename CC::cell_type >,
                typename CC::vertex_on_cell_iterator  >
   > datatype_map;
```

5.2 GSSE: Functional description

The approach of a generalised topological access to data structures makes a functional description for algorithms possible which are a natural choice for reuse in the field of scientific computing. The functional specification of algorithms thereby derived is used to develop and implement a minimal base of generic functors for the DSEL.

The functor library built on top of the topological interface is then dimensionally and topologically independent. An example of these two libraries is given next, where a finite volume discretisation of a generic Laplace equation is discretised:

```
sum<edge>()
[
  sum<vertex>()  [ equ_pot ] * area / dist
] (vx);
```

The functional body can be arbitrarily extended by other traversal operations, calculations or assignments. This example is inherently fully parallelisable due to the functional specification, where `vx` represents a vertex, an object from the traversal space.

This implementation of this library is based on an adaptation layer for Boost::Phoenix to read the data-type maps of the traversal part. Several new functors have been developed to ease not only the traversal, but also the quantity access. The given functors, `equ_pot, area, dist`, are all quantity access functors in the spirit of the C++ cursor and property map concept [2]. In the given example, the functors `sum` $<>$ specifies the traversal part, where

```
sum<edge>
[
   sum<vertex>[  ]
] (vx);
```

represents a vertex on edge traversal, where the context of vertex on edge is created by the data-type map and the corresponding meta-program. The final traversal sequence is then started by using the actual vertex `(vx)` provided by any container which models the concept of vertices and edges.

Due to the compile-time evaluation of the functional specification, an overall high-performance is established for the application. In addition, the functional specification of algorithms permits unrestricted parallel execution.

By using the comprehensive traversal capability of the GSSE and the capability to describe algorithms functionally, scalability and parallelity of an application is simply achieved by using either a partition of the traversal elements by GSSE or new approaches such as the parallel STL.

5.3 Parallel applications

The basic mechanism for parallelisation is given by STL's separation of traversal of data structures and algorithms, which can be briefly explained by:

```
std::for_each(container.begin(),
              container.end(),
              functor());
```

The GSSE offers the same concept but in a more general way which separates the discrete topological space (elements of a data structure) and the access to quantities. The following example illustrates the calculation for all edge lengths of a more complex container structure.

```
traverse<edge>()
[
  dist = norm ( sum<vertex>() [  coord  ] )
](container);
```

The implementations of these traversal mechanisms use the C++ STL algorithms internally and are thereby automatically parallelised by utilising one of the parallel STL approaches. A linear speed-up corresponding to the number of cores can be accomplished by just a recompilation step and adjusting a run-time environment variable.

A manual partitioning of the traversal space can also be accomplished for discretisation schemes and the required assembly steps. The following example illustrates the parallelisation capabilities for a finite-volume discretisation scheme (`ts` respresents the traversal space):

```
for (vertex_iterator v_it = (ts).vertex_begin(threadID);
                     v_it != (ts).vertex_end(threadID); ++v_it)
{
  // ..
}
```

By simply using partitions of the vertex space, the whole application remains unchanged and just a recompilation step is required to use the application parallely.

6. Examples

To present the application of the parallelisation techniques offered by the GSSE we choose the area of technology computer aided design (TCAD), which serves as the semiconductor industry's branch of scientific computing.

Here we present a parallel combined Delaunay and advancing front mesh generation and adaptation approach for TCAD's process simulation. The complete hull is pre-processed separately to comply with the Delaunay property [34]. This guarantees a volume mesh generation approach, where each segment can be meshed concurrently. The following snippet of code shows a central part of the mesh generation application:

```
gsse::for_each(container.segment_begin(),
               container.segment_end(),
               generate_mesh(thread_id++));
```

A GSSE container is used as an interface for segments which are fed to a functional meshing routine. Table 2 summarises two different meshing examples from TCAD on two different computer systems: AMD's X2 6000 and AMD X4 Phenom 9600 (quad-core).

Table 2. Comparisons of the mesh generation and included mesh adaptation times (in s) on AMD's X2 6000 and AMD X4 Phenom 9600 (quad-core).

Example	*Sequential mesh (s)*	*Dual-core (s)*	*Quad-core (s)*	*No. of points*
MOSFET	172	101	46	1.7e6
Levelset	20	13	11	3.6e5

Furthermore, we present an example of TCAD's device simulation application. The most basic model, the drift-diffusion (DD) model, consists of a set of coupled partial differential equations which need to be assembled and solved. 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 the here presented energy transport or higher transport models, however, spend an increasing amount of time on equation assembly.

Discretised using finite-volume schemes, it can be assembled in parallel by partitioning the traversal space due to the line-wise entries into the matrix. Equation (1) shows the energy flux equation for electrons, which is solved self-consistently with Poisson's equation and the current relations [27]. The following code snippet demonstrates the actual C++ code for the electron temperature `T_n` determined by Equation (1) [10].

$$\operatorname{div}\left(\alpha_n \operatorname{grad}(nT_n^2) + \operatorname{grad}\varphi\, n\, T_n\right) = -\operatorname{grad}\varphi \cdot \mathbf{J}_n - \beta_n \; n(T_n - T_{\text{Lattice}}). \tag{1}$$

Each `sum` can be automatically parallelised by the parallel STL, where the full matrix line is assembled in parallel by a vertex partitioning mechanism executed by spawned threads:

```
(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);
```

Several existing multi-thread libraries can be used orthogonally due to the separate partitioning of the vertex space (e.g. the Boost thread library [8]). This method can then be used for all finite volume codes.

To present the results obtained by using different machines and not altering the actual code parts, a benchmark[4] is given for a simple DD and hydro-dynamic (HD) simulation for a two-dimensional pn-diode with 4000 elements. The bias voltage is stepped from -0.03 to 1 V. An abstract matrix–solver interface enables the application of various solver packages, such as Trilinos [17], which is also available to the GSSE. For the actual benchmarks, a sequential implementation of the Trilinos linear solver, a variant of a BiCGStab algorithm, was used, whereas the non-linear solver part is implemented by the GSSE.

Table 3. Comparisons of the simulation times for DD and HD simulation of a pn-diode with GCC 4.2 on an Opteron Cluster with 4xDual-Core 8222 SE.

Example	*Sequential (s)*	*Two threads (s)*	*Four threads (s)*	*Eight threads (s)*
DD	11.6	10.1	9.1	8.3
HD	17.8	14.7	10.5	8.9

Table 3 reviews the benchmark results from four AMD 4 × Dual-Core Opteron Cluster 8222 SE 3 GHz with 2 × 32 GB and 2 × 16 GB RAM.

7. Conclusion

Library-centric design does not only ease the development of applications significantly by providing building blocks centralised in a generic environment, the GSSE, but also greatly facilitates the evolution of single-processing applications into parallel applications suitable for multi-core processors by parallel components, thereby simplifying development, scalability, stabilisation, further support and parallelisation. Only applying the concepts of parallelism in everyday programming can unlock the full power of the new multi-core processors.

Acknowledgement

This work has been supported by the Austrian Science Fund FWF, project P19532-N13, and the Intel Corporation.

Notes

1. The GSSE uses an open source license (Boost [8]) and is available at http://www.gsse.at
2. In C++ there are currently 48 operators which can be overloaded.
3. A detailed overview of the GSSE is available at: http://www.reneheinzl.net/gsse/
4. GCC 4.2, -O3 -march=k8 -mtune=k8.

References

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques From Boost and Beyond (C++ in Depth Series)*, Addison-Wesley, Boston, MA, 2004.
[2] D. Abrahams, J. Siek, and T. Witt, *New iterator concepts*, Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++ (2003).
[3] M.H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, Boston, MA, 1998.
[4] O. Bagge, *CodeBoost: A framework for transforming C++ programs*, Master's thesis, University of Bergen, P.O.Box 7800, N-5020 Bergen, Norway, 2003.
[5] W. Bangerth, R. Hartmann, and G. Kanschat, *deal.II Differential Equations Analysis Library,* Technical Reference.
[6] G. Berti, *Generic software components for scientific computing*, Dissertation, Technische Universität Cottbus, 2000.
[7] G. Berti, *GrAL – The grid algorithms library*, in *Proceedings of the Computational Science ICCS*, Vol. 2331, Springer, London, UK, 2002, pp. 745–754.
[8] Boost: *Boost C++ Libraries*, http://www.boost.org.
[9] A. Fabri, *CGAL – The computational geometry algorithm library*, in *Proceedings of the 10th International Meshing Roundtable,* CA, USA, 2001, pp. 137–142.

[10] T. Grasser, T. Tang, H. Kosina, and S. Selberherr, *A review of hydrodynamic and energy-transport models for semiconductor device simulation*, Proc. IEEE 91(2) (2003), pp. 251–274.
[11] D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp, *Generic programming and high-performance libraries*, Int. J. Parallel Prog. 33(2) (2005), pp. 145–164.
[12] M. Haveraaen, H. Friis, and T. Johansen, *Formal Software Engineering for Computational Modeling*, Nordic J. Comput. 3(6) (1999), pp. 241–270.
[13] R. Heinzl, *Concepts for scientific computing*, Dissertation, Technische Universität Wien, Austria, 2007.
[14] R. Heinzl, P. Schwaha, M. Spevak, and T. Grasser, *Performance aspects of a DSEL for scientific computing with C++*, in *Proceedings of the POOSC Conference,* Nantes, France, 2006, pp. 37–41.
[15] R. Heinzl, M. Spevak, P. Schwaha, and S. Selberherr, *A generic topology library*, in *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications Conf., Portland,* OR, USA, 2006, pp. 85–93.
[16] R. Heinzl, P. Schwaha, and S. Selberherr, *A high performance generic scientific simulation environment*, in *Lecture Notes in Computer Science*, in B. Kaagström, et al., eds., Vol. 4699, Springer, Berlin, 2007, pp. 781–790.
[17] M. Heroux, R. Bartlett, V.H.R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, H. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams, *An overview of trilinos*, Technical report SAND2003-2927, Sandia National Laboratories, 2003.
[18] M. Jazayeri, *Component programming – A fresh look at software components*, in *Proceedings of the Software Engineering (ESEC'95)*, in W. Schäfer & P. Botella eds., Springer, Berlin, 1995, pp. 457–478.
[19] S.C. Johnson, *YACC: Yet another compiler compiler*, in *UNIX Programmer's Manual*, Vol. 2, Holt, Rinehart, and Winston, New York, NY, 1979, pp. 353–387.
[20] B. Kagstrom, E. Elmroth, J. Dongarra, and J. Wasniewski, eds., *Applied parallel computing*, *State of the Art in Scientific Computing*, Springer, Berlin/Heidelberg, 2007.
[21] H.P. Langtangen and X. Cai, *Mixed language programming for HPC applications*, in *Proceedings of the PARA Conference,* Umea, Sweden, 2006, p. 154.
[22] A. Logg, T. Dupont, J. Hoffman, C. Johnson, R.C. Kirby, M.G. Larson, and L.R. Scott, *The FEniCS project*, Technical Report 2003-21, Chalmers Finite Element Center, 2003.
[23] D.R. Musser and A.A. Stepanov, *Generic programming*, in *Proceedings of the ISSAC'88 on Symb. and Alg. Comp*, Springer, London, UK, 1988, pp. 13–25.
[24] S. Pion and A. Fabri, *A generic lazy evaluation scheme for exact geometric computations*, in *Proceedings of the Object-Oriented Programming Systems, Languages, and Applications Conference,* Portland, OR, USA, 2006, pp. 75–84.
[25] P. Plauger, M. Lee, D. Musser, and A.A. Stepanov, *C++ Standard Template Library*, Prentice Hall PTR, Upper Saddle River, NJ, 2000.
[26] R. Pozo, *Template numerical toolkit for linear algebra: high performance programming with C++ and the standard template library*, Int. J. High Perfor. Comput. Appl. 11(3) (1997), pp. 251–263.
[27] P. Schwaha, M. Schwaha, R. Heinzl, E. Ungersboeck, and S. Selberherr, *Simulation methodologies for scientific computing*, in *Proceedings of the 2nd ICSOFT 2007,* Barcelona, Spain, 2007, pp. 270–276.
[28] J. Siek and A. Lumsdaine, *Mayfly: A Pattern for Lightweight Generic Interfaces*, 1999.
[29] J. Siek and A. Lumsdaine, *Software Engineering for Peak Performance*, C++ Report, 2000, pp. 23–27.
[30] J.G. Siek and A. Lumsdaine, *Concept checking: Binding parametric polymorphism in C++*, in *Proceedings of the First Workshop on C++ Template Programming,* Erfurt, Germany, 2000.
[31] J. Siek, L.Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley, Boston, MA, 2002.
[32] J. Singler and B. Kosnik, *The libstdc++ parallel mode: Software engineering considerations*, in *Proceedings of* IWMSE, Leipzig, Germany, 2008, pp. 15–22.
[33] J. Singler, P. Sanders, and F. Putze, *The multi-core standard template library*, in *Lecture Notes in Computer Science*, Vol. 4641, Springer, Berlin, 2007, pp. 682–694.

[34] F. Stimpfl, R. Heinzl, P. Schwaha, and S. Selberherr, *High performance parallel delaunay mesh generation and adaptation*, in *Proceedings of the PARA Conference,* Trondheim, Norway, 2008.
[35] T.L. Veldhuizen, *Expression templates*, C++ Report 7(5) (1995), pp. 26–31, Reprinted in C++ Gems, ed. Stanley Lippman.
[36] T.L. Veldhuizen, *Five compilation models for C++ templates*, in *1st Workshop on C++ Template Programming*, 2000.
[37] T.L. Veldhuizen, *Domain-Specific Program Generation. Lecture Notes in Computer Science*, Vol. 3016, Springer, Berlin, 2004, pp. 306–324.
[38] T.L. Veldhuizen, *Stage-preserving embeddings of languages*, in *The 16th Nordic Workshop on Programming Theory (NWPT'04)*, 2004, pp. 5–8.
[39] T.L. Veldhuizen and D. Gannon, *Active libraries: Rethinking the roles of compilers and libraries*, in *Proc. of the SIAM Workshop on Obj.-Oriented Methods for Inter-Operable Sci. and Eng. Comp. (OO'98)*, SIAM, Philadelphia, 1998.