A Parallel Generic Scientific Simulation Environment

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

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

Abstract. Upcoming parallelization with multi-core processors as well as the availability of large networks with high-speed interconnects require a change in the used programming paradigms and techniques to develop scalable applications. Techniques for library-centric application design have already proven to be very useful in the past and are of utmost importance in the field of parallel and scalable application design. However, the utilization of multi-core processors places new difficulties in application design for scientific computing. A parallel generic scientific simulation environment enables parallelization by facilitating a suitable combination of multiple programming paradigms and by following modern application design guidelines, which is necessary in order to keep development on multi-core processors as simple as possible, while not forsaking any of their computational power. Inherent parallel assembly mechanisms for large equations systems and multi-dimensional topological traversal mechanism are presented.

1 Introduction

To secure a further a gain in computing performance the semiconductor industry has shifted the upcoming processor upgrades to multi-core computer systems where the gain in processing power is obtained by using an increasing number of processing cores in the CPUs. For application developers, especially in the field of scientific computing, the additional complexity of developing software which performs well on a varying number of available processing units is thereby introduced. This evolutionary shift of how the computational power of systems increases also requires the adoption of new programming methodologies and libraries. While these have already been developed for multiprocessing systems in supercomputers for several years, they have yet to find widespread adoption. Only by applying the concepts of parallelism in everyday programming can the full power of the new multi-core processors be unlocked.

In the field of scientific computing, the increasing complexity of underlying physical models often also brings an increase of the complexity and amount of source code. Current single desktop computer systems can already handle a great amount of scientific simulations locally. Development of new methods, new models, and new techniques is thereby greatly eased due to local and fast execution. Industrial problems and settings often require large scale simulations which have to be performed on supercomputers. However, the individual nodes of these supercomputers often do not differ in the execution speed from the desktop computers anymore. Instead they are heavily parallelized with a great amount of shared memory and are based on a large number of CPUs, which is going to further increase in the near future. This scaling also has to be reflected in nowadays application design. Most initial developments of scientific applications do not require more than a processor with, e.g., ten cores and 8 GByte of memory, and can thereby be easily performed on a workstation. Only for the final industrial setting or final example, the application has to run on a supercomputer, where the amount of CPUs is drastically increased, as is the amount of available memory.

The concept of library-centric application design and the availability of a set of high performance libraries significantly eases the development of highly scalable applications. While scientific simulations have been among the first applications to embrace parallelization, still not all fields of scientific computing make use of it, as it is perceived as an additional and often overly complex task. The efficiency of developing and maintaining source code is an increasingly important issue. It 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. A simple work-flow present in most applications includes the time needed to calculate the distribution of quantities of interest in a given simulation domain, which is mostly allocated to the assembly of an equation system and its subsequent solution. How the time requirement is allocated depends on the complexity of the equations being assembled and the respective numerical condition.

Most related work focuses on parallel toolkits within their frameworks [1]. Our approach is based on providing modular blocks which can be used on top of existing libraries, such as the Boost graph library (BGL [2]), CGAL [3], GrAL [4], to unify these interfaces. Also utmost emphasis is placed on the issues that already tested and stable code has to be parallelized without modification of existing code. This is not only important to speed up the development of parallel executing applications, but also to preserve the already invested time to develop, calibrate, and debug an application. We therefore present two approaches with our Generic Scientific Simulation Environment (GSSE) [5, 6], which enables the use of several parallelization techniques without altering the existing algorithms.

Firstly, various multi-threading libraries are used in conjunction with the topological partitioning provided by GSSE to subdivide the amount of topological objects. Several discretization schemes and the assembly times benefit greatly from this approach. Secondly, parallel STL [7–9] techniques or libraries can easily be incorporated, which require just a recompilation step, where all STL algorithms and our GSSE algorithms, which are built on top of these algorithms, are then executed parallely. These techniques are already going to be incorporated into the GCC [10]. Finally, several MPI [11] implementations are also possible, e.g., the Trilinos MPI mechanisms [12].

To present the application of the parallelization techniques we choose the area of Technology Computer Aided Design (TCAD), which serves as the semiconductor industry's branch of scientific computing. The increasing complexity of underlying physical models often brings an increase of the complexity and amount of source code. Therefore, the efficiency of developing and maintaining source code is an increasingly important issue. It 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.

2 The Parallel GSSE

As has already been demonstrated [5, 13], 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 an easy specification of all different types of discretized differential equations, reducing error prone tasks such as index calculations or the evaluation of the elements of the Jacobian matrix, while at the same time not sacrificing run-time performance. The most basic building block for the transition of continuous function spaces to discrete spaces is the operation of topological traversal.

Several tasks in scientific computing can be greatly eased by the utilization of functional programming, e.g., using functional programming in the design of parallel applications. Unfortunately several tasks defy the nature of stateless description, e.g., loading a file. 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. Therefore the whole application design cannot be reduced to a purely functional description. However, most purely functional programming languages, such as Haskell, ML [14] do not equally support other programming paradigms, such as imperative or object-oriented programming. Due to these reasons, our approach is based on a multi-paradigm approach, where each paradigm is used where it performs best:

- The object-oriented paradigm is used where hierarchies of data types are relevant, e.g., data type selections or additional properties of data. All containers or storage mechanism can easily be covered by this paradigm because of the information transportation between objects.
- The functional programming paradigm is best at describing functional expressions, in our case discretized and linearized projections of the continuous function spaces.
- The generic programming paradigm couples the object-oriented and functional paradigm by, e.g., the parametric polymorphism of C++. Generic programming excels at the abstract treatment of objects, called concepts.

2.1 Topological Traversal

Topological traversal describes the iteration over elements of a data structure, e.g., a doubly-linked list. 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 a common topological approach, the generic topology library (GTL [6]), where all data structures are mapped to a cell complex and the corresponding mechanisms derived from algebraic topology. Where the STL offers support for various simple data-structures, the GSSE offers all different types of mesh and grid data-structures of all dimensions. The generic functor library (GFL [6]) built on top of the topological interface is then dimensionally and topologically independent.

For a parallel environment the most important issue for the transition of continuous function spaces to discrete spaces is the operation of topological traversal. Most of the parallelization is accomplished by accessing data structures in a parallel way. The parallel version of the GSSE therefore introduces additional mechanisms on top of the already existing libraries in a non-intrusive way. Next, a source snippet for C++ code is given where a finite volume discretization of a generic Poission equation is discretized:

```
equ = (sum<edge>()
[
  sum<vertex>()  [  phi   ]    * area     / dist
] + rho * vol
)  (*v_it);
```

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

space. An example is given next, where geometrical points are selected by a coordinate functor:

```
for (vertex_iterator v_it = (iter_part).vertex_begin(threadID);
                     v_it != (iter_part).vertex_end(threadID); ++v_it)
      // algorithm
```

The various discretization schemes differ with respect to which quantities they compute and in which fashion. The crucial difference concerning assembly, however, is the distinction of the required traversal operation. Actual access to quantities and insertion into the matrix itself remains identical. The assembly of the Jacobian matrix is the final crucial part, where the functional specification of the discretized equations is combined with the required traversal operations. The clean separation of these two steps, guaranteed by our approach, also provides a clear interface for parallelization. The traversal of the discretized simulation domain is responsible for the global assembly of the Jacobian matrix, whereas the assembly module takes care of mapping the local operations of each topological object to the global Jacobian matrix.

The assembly algorithms remain unchanged for parallelization, only the topological traversal is partitioned automatically, e.g., an environment variable changes the number of parallel execution tasks.

2.2 Parallel STL

Various approaches extend the C++ STL by parallel execution paths. The basic mechanism of the STL's algorithms is given next by a simple example of a `for_each`.

```
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 traverses all vertices / nodes of an arbitrary container and assigns a value.

```
traverse<vertex>()  [  quan = 0   ] (container);
```

Another example is given next, where all edge lengths of a more complex container structure is calculated. Here, the container has to model the concept of a data structure with dimension greater than zero, e.g., a graph or a triangulation.

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

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

3 Examples

3.1 Device Simulation

We present an example of TCAD's device simulation applications, where the most basic model, the drift-diffusion model, is comprised of four coupled partial differential equations which need to be assembled. In this case the assembly time is usually small compared to the time spent on the solution of the equation system. More sophisticated and complex models such as energy transport or higher transport models however spend an increasing amount of time on equation assembly. Parallelism of not only the assembly procedure but also the pre- and post-processing algorithms is therefore of increasing importance to decrease computation times. To demonstrate not only the functional equation specification mechanism of the GSSE, but also the parallel approach mechanism the following code snippet demonstrates the actual C++ code for the electron temperature `n_te` for hydro-dynamic device simulation application. Equation 1 show the energy flux equation for electrons which is solved self-consistently with Poisson's equation and the current relations [15].

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

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

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

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

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

Table 1: Comparisons of the simulation times for drift-diffusion and hydro-dynamic simulation of a pn-diode with different optimization level (GCC 4.2) on AMD X2 6000 CPU's (dual-core) and AMD X4 Phenom 9600 (quad-core).

3.2 Mesh Generation and Adaptation

Another example for the utilization of a parallel topological traversal is given by the task of Delaunay mesh generation and adaptation. Here we present a parallel combined Delaunay and advancing front mesh generation and adaptation approach. The complete hull is pre-processed separately to comply with the Delaunay property [16, 17]. 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, using a GSSE container, parametrized to a specific data type, as an interface for segments which are fed to a functional meshing routine.

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

Figure 1 depicts a three-dimensional device structure (MOSFET), which can be calculated on a workstation computer, whereas the full device is then simulated on four AMD 4xDual-Core Opterons 8222 SE with 2x32 GByte and 2x16GByte RAM.

Example	Sequential mesh	Dual-core	Quad-core	Num. points	Num. segments
MOSFET (industrial)	172 s	101s	101 s	1.7e6	7
MOSFET	70 s	42s	21 s	3.6e5	7

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

4 Conclusion

By using the concept of library-centric application design in the area of parallel environments, it can be shown that the whole simulation process can be easily separated into small building blocks. The appropriate realization of each of these blocks guarantees not only an impressive performance, but also eases development, scalability, stabilization, further support, and parallelization.

5 Acknowledgment

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

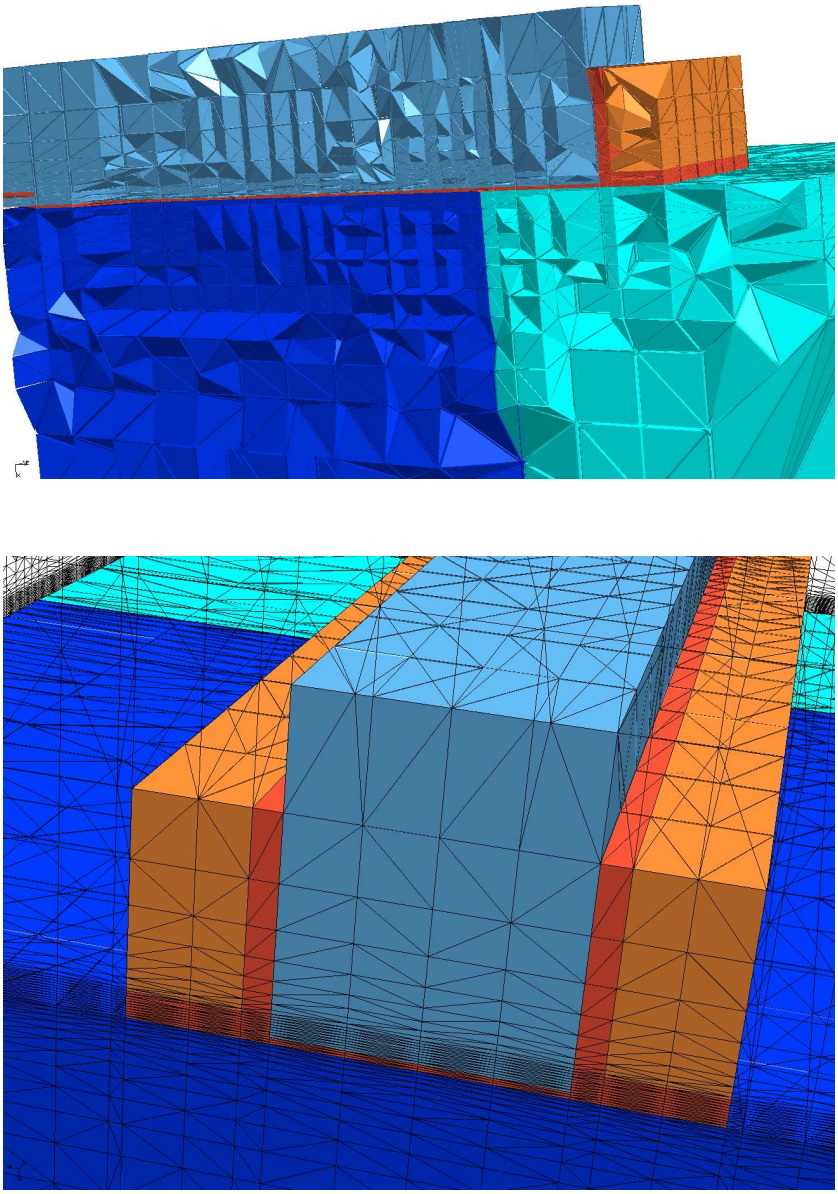

Fig. 1: A three-dimensional device structure for a MOSFET with an additional externally supplied point cloud. The important part is the regularity of the elements in the channel region (red). The different aspect ratios, e.g., the thin red oxide part and the large blue silicon part, are also an additional complication for the mesh generation algorithm.

References

1. Kagstrom, B., Elmroth, E., Dongarra, J., (ed.), J.W.: Applied Parallel Computing. State of the Art in Scientific Computing. Berlin / Heidelberg (2007)
2. Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley (2002)
3. Fabri, A.: CGAL - The Computational Geometry Algorithm Library. In: Proc. of the 10th Intl. Meshing Roundtable, CA, USA (2001) 137–142
4. Berti, G.: Generic Software Components for Scientific Computing. Dissertation, Technische Universität Cottbus (2000)
5. Heinzl, R., Schwaha, P., Selberherr, S.: A High Performance Generic Scientific Simulation Environment. In B. Kaagström et al., ed.: Lecture Notes in Computer Science. Volume 4699/2007. Springer, Berlin (2007) 996–1005
6. Heinzl, R.: Concepts for Scientific Computing. Dissertation, Technische Universität Wien, Austria (2007)
7. Rauchwerger, L., Arzu, F., Ouchi, K.: Standard Templates Adaptive Parallel Library (STAPL). In: Proc. Intl. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, London, UK, Springer (1998) 402–409
8. Putze, F., Sanders, P., Singler, J.: MCSTL: The Multi-Core Standard Template Library. In: Proc. Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, ACM (2007) 144–145
9. Singler, J., Sanders, P., Putze, F.: The Multi-Core Standard Template Library. In: Lecture Notes in Computer Science. Volume 4641/2007. Springer, Berlin (2007) 682–694
10. Singler, J., Kosnik, B.: The libstdc++ Parallel Mode: Software Engineering Considerations. In: Proc. of IWMSE 2008. (2008)
11. Squyres, J.M., Lumsdaine, A.: A Component Architecture for LAM/MPI. In: Proc., 10th European PVM/MPI Users' Group Meeting. Number 2840 in Lecture Notes in Computer Science, Venice, Italy, Springer (2003) 379–387
12. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., K., T.G., L., R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An Overview of the Trilinos Project. ACM Transactions on Mathematical Software **31**(3) (2005) 397–423
13. Heinzl, R., Spevak, M., Schwaha, P., Selberherr, S.: A Generic Topology Library. In: Proc. of the Object-Oriented Programming Systems, Languages, and Applications Conf., Portland, OR, USA (2006) 85–93
14. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: An Extended Comparative Study of Language Support for Generic Programming. J. of Functional Programming **17**(2) (2007) 145–205
15. Schwaha, P., Schwaha, M., Heinzl, R., Ungersboeck, E., Selberherr, S.: Simulation Methodologies for Scientific Computing. In: Proc. of the 2nd ICSOFT 2007, Barcelona, Spain (2007) 270–276
16. Stimpfl, F., Heinzl, R., Schwaha, P., Selberherr, S.: A Multi-Mode Mesh Generation Approach for Scientific Computing. In: ESM 2007, St. Julians, Malta (2007) 506–513
17. Stimpfl, F., Heinzl, R., Schwaha, P., Selberherr, S.: High Performance Parallel Delaunay Mesh Generation and Adaptation. In: Proc. of the PARA Conf., Trondheim, Norway (2008)