Journal of Computational and Applied Mathematics 270 (2014) 484–495

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Highly flexible and reusable finite element simulations with ViennaX

Josef Weinbub *, Karl Rupp, Siegfried Selberherr

Institute for Microelectronics, Technische Universität Wien, Gußhausstraße 27-29/E360, A-1040 Wien, Austria

ARTICLE INFO

Article history:
Received 30 September 2013
Received in revised form 4 December 2013

Keywords:
Finite elements
Component framework
Plugins
Simulation

ABSTRACT

An approach for increasing the flexibility and reusability of finite element applications for scientific computing is investigated by utilizing the ViennaX framework. Implementations are decoupled into components, allowing for extensible application setups as well as convenient changes in the simulation flow. The feasibility of our approach is shown by decoupling finite element implementations, provided by the ViennaFEM and the deal.II library, respectively. A ViennaFEM elasticity problem is highly decoupled into separate components, whereas an adaptive mesh refinement example provided by the deal.II library is used to show ViennaX's support for execution loops. Finally, we underline the high level of flexibility and reusability by outlining the transformation from the depicted applications into a finite volume-based solution of the Poisson equation.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

The ever-increasing availability of finite element (FE) related software tools puts challenges on the software development of simulation applications. Albeit the fact that feature-rich packages such as the deal.II library [1] are available, solely utilizing a static set of tools for an application limits the capabilities of the implementation with respect to flexibility and reusability [2,3]. For instance, exchanging the linear solver typically requires actual coding and thus implies in depth knowledge of the code base. A usual way to tackle these challenges are component frameworks, such as the Cactus framework [4]. However, the available approaches either offer an almost prohibitively high entry-level for users or focus on domain-specific applications, like computational fluid dynamics (CFD). We introduce an approach based on our component execution framework ViennaX [5], aiming for general applicability in scientific computing. We extend previous work [6] by focusing solely on decoupling FE-based applications and depicting the thus increased level of flexibility and reusability of the respective FE-based implementations. In this context, flexibility refers to the fact that individual simulation components can be exchanged with a minimum of effort. Reusability is achieved by the component approach, as a component can be utilized in several ViennaX simulation flows. A particularly noteworthy aspect is the fact that our approach allows the user to – over time – grow a set of ready-to-use components. Based on this growing set of components, new simulation flows can be defined without coding efforts by selecting a subset of the available components. This fact, for obvious reasons, underlines that our approach reduces long-term implementation efforts of scientific simulations, particularly useful to FE-based simulations.

This work is organized as follows. Section 2 puts the work into context by discussing related research. Section 3 provides an overview of ViennaX. Section 4 outlines the high degree of flexibility and reusability of ViennaX by discussing various application scenarios.

* Corresponding author. Tel.: +43 15880136053.
E-mail addresses: weinbub@iue.tuwien.ac.at (J. Weinbub), rupp@iue.tuwien.ac.at (K. Rupp), selberherr@iue.tuwien.ac.at (S. Selberherr).

0377-0427/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cam.2013.12.013

2. Related work

First, we give an overview of relevant research work. We focus on free open source tools due to their natural affinity with our research work, due to accessible source code. Section 2.1 lists FE libraries, typically utilized in various areas of scientific computing. Section 2.2 discusses available component frameworks. Section 2.3 summarizes the presented related work, depicting the motivation for this work.

2.1. Finite element libraries

Here, we introduce a selection of freely available open source FE libraries, utilized to discretize and solve partial differential equations (PDEs) in various applied fields of scientific computing.

deal.II is a C++ library for the solution of PDEs using adaptive FEs [1,7]. The source code of the library is freely available under the GNU Lesser General Public License (LGPL). deal.II supports line, quadrilateral, and hexahedral meshes as well as different parallelization strategies, such as small-scale multi-threading-based applications and large-scale applications via the message passing interface (MPI) [8]. Furthermore, local mesh refinement is provided based on error indicators and error estimators. A variety of finite elements are available, such as Lagrange, Nedelec, and Raviart–Thomas elements of any order. Extensive interfaces to external libraries, like Trilinos, PETSc, and METIS, are provided as well as comprehensive documentation and tutorials.

The *Distributed and Unified Numerics Environment (DUNE)* is a modular C++-based toolbox for solving PDEs with grid-based methods [9]. Various discretization schemes are supported, i.e., FE, finite volume (FV), and finite difference. DUNE is freely available under the GNU General Public License (GPL).

A central aspect of DUNE is the modular structure, meaning that functionality is shipped in separate modules. A set of so-called core modules is provided, containing, for instance, mesh storage support. Additional modules are available, such as the DUNE-FEM module [10].

libMesh provides a C++ framework for the numerical simulation of PDEs using arbitrary unstructured discretizations [11,12]. libMesh is available under the GNU LGPL. The focus of the library is on supporting parallel adaptive mesh refinement applications. Mixed FEs on hybrid unstructured grids are available for one-, two-, and three-dimensional problems. Multi-physics as well as multi-threaded and MPI-based applications are supported.

The *FEniCS* project is a collection of free open source software components, aiming for the automated solution of PDEs [13,14]. The components are available under the GNU LGPL. Several core components are offered, most importantly the DOLFIN library, being a C++/Python library [15]. DOLFIN provides data structures and algorithms required for building FE simulations. Shared- and distributed memory parallel approaches are supported via OpenMP and MPI. A separate preprocessor is provided to enable convenient problem definitions, such as mesh generation and formulation of PDEs.

Open Source Field Operation And Manipulation (OpenFOAM) is a C++-based toolbox primarily for the field of CFD [16,17]. The toolbox is available under the GNU GPL. Different discretization approaches are supported, such as FV and FE methods. A multitude of functionality is provided, such as (in)compressible and multi-phase flows as well as combustion-related investigations. A prominent feature of OpenFOAM is a mechanism for implementing arbitrary PDEs in a natural manner, i.e., the implementation closely resembles the mathematical formulation.

FreeFem++ is a partial differential equation solver [18,19] coded in C++ with an own scripting language for solving two- and three-dimensional non-linear multi-physics problems. The library is publicly available under the LGPL. Among the features are a large variety of finite elements, such as linear and quadratic Lagrangian elements, the ability to interpolate data between different meshes automatically, a variety of linear solvers, and parallel implementations based on MPI.

ViennaFEM is a free open source C++ library, implementing the FE method for the solution of PDEs [20]. ViennaFEM is built on top of a symbolic math kernel, allowing for a convenient specification of the strong and weak formulation, from which the assembly routines are derived automatically. The software package is available under the Massachusetts Institute of Technology (MIT) License. A library-centric software design is applied, as such ViennaFEM is built on top of several orthogonal libraries, like ViennaCL [21] for the iterative linear solver and ViennaMath [22] for the symbolic math kernel.

2.2. Component frameworks

This section discusses representative available component frameworks and approaches, aiming for introducing flexibility and reusability in the field of scientific computing.

The *Common Component Architecture (CCA)* applies component-based software development to encapsulate units of functionality into components [23–25]. Data communication between components is implemented via so-called ports. An interface definition language is used to describe the interfaces of components by simultaneously being independent of the underlying programming language. The actual connection mechanism of the individual components via the interfaces requires end-user interaction. The CCA standard has been applied in several projects [25], such as the high-performance computing framework CCAFFEINE [26] and the distributed computing frameworks XCAT [27], Legion [28], and SCIRun2 [29].

Cactus is a multi-purpose framework, which has its roots in the field of relativistic astronomy [4,30]. The framework is available under the GNU LGPL and focuses on data parallel approaches. The central part of the framework (called "flesh")

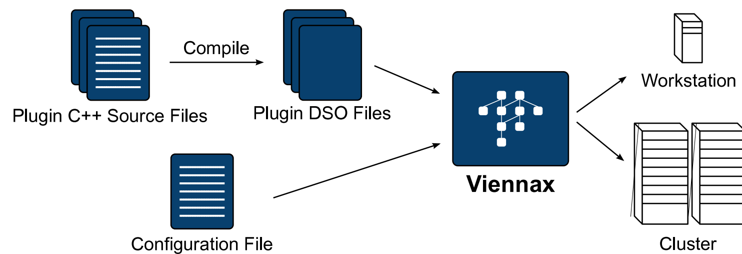

Fig. 1. The schematic utilization of ViennaX is shown. Source files modeling the plugin concept are compiled into Dynamic Shared Objects (DSOs). The DSOs as well as the configuration file are loaded into the framework during the framework's startup phase. Based on the plugins' individual dependencies a task graph is generated. The plugins are executed according to the dependencies, until the graph has been processed.

connects the individual typically user-supplied application modules (called ''thorns''), containing the implementations of the actual simulation. Communication between thorns is realized via the framework's application programming interface. Connections between thorns are provided by the user in configuration files which are processed during compile-time. Data dependencies of components have to be defined in specification files, using the Cactus configuration language.

The *COOLFluiD* project enables multi-physics simulations based on a component framework and is primarily designed for data parallel applications in the field of CFD [31,32]. The source code is available under the LGPL. The core is a flexible plugin system, coupled with a data communication layer based on so-called data sockets. Each plugin, representing a component, can set up data sockets which are in turn used to generate a dependence hierarchy driving the overall execution.

The *Earth System Modeling Framework (ESMF)* provides the setup of flexible, reusable, and large-scale simulations in climate, weather, and data assimilation domains [33,34]. The source code is publicly available under the University of Illinois-NCSA License. The parallelization layer is abstracted by a virtual machine approach and focuses on data parallelism. The component interface is based on three functions, responsible for initialization, execution, and finalization of the respective component. The data exchange between components is based on ESMF-specific data types.

The *Uintah* project is a large-scale multi-physics computation framework available under the MIT License [35–37]. Uintah solves reacting fluid–structure problems on a structured adaptive mesh. Task and data parallel applications are supported. The primary area of application is computational mechanics and CFD. The framework is based on a directed acyclic graph (DAG) representation of parallel computation and communication to express data dependencies between multiple components. Each node in the graph corresponds to components which in turn represent a set of tasks. The data dependencies between components are modeled by edges in the DAG.

2.3. Summary

A plethora of open source FE libraries is available, supporting different approaches essential for FE simulations, such as physical modeling, discretizing PDEs, and solving the assembled system of equations. On the other hand, many of these parts could and should be decoupled into reusable and exchangeable components by component frameworks. This decoupling enables to alter a FE simulation with a minimum of effort, allowing, for instance, to investigate particular physical models or the solution behavior of different linear solvers without coding effort. The components are connected with a communication mechanism for inter-component data transfer, for example, to forward the system matrix from an assembler component to a linear solver component. However, as already indicated, the discussed frameworks either offer an almost prohibitively high entry-level for users, such as a specialized configuration language, or focus on domain-specific applications, like CFD. Instead our approach based on ViennaX does not suffer from such restrictions, thus serving as a flexible platform for implementing reusable and exchangeable simulations in the general area of scientific computing.

3. Overview of ViennaX

ViennaX is a component execution framework, coded in C++ and available under the MIT License [5,6]. The decoupling of simulations into separate components is facilitated by the framework's plugin system. Functionality is implemented in components by plugins supporting data dependencies. For the remainder of this work, the term plugin is used as a synonym for a component. The components and the data dependencies are used to generate a so-called task graph [38], where the components represent the nodes and the dependencies relate to the edges of the graph. The task graph is used by the available scheduler mechanisms to drive the overall execution (Fig. 1). Our task graph is a DAG, which is required for generating a prioritized queue of tasks via the topological sort graph algorithm [39].

This requirement is based on the fact that an edge in a DAG is directed, i.e., it points from a source to a target vertex. In addition, the graph must not contain cycles, also known as loops. If this DAG condition is satisfied, a topological sort algorithm is used to generate a linear sequence of vertices – representing the ViennaX components – which upon execution guarantees that the input dependencies of each component are met. Despite the acyclic requirement, ViennaX supports loops by identifying the loop entry vertex and the loop exit vertex. The loop is then broken up to satisfy the DAG condition,

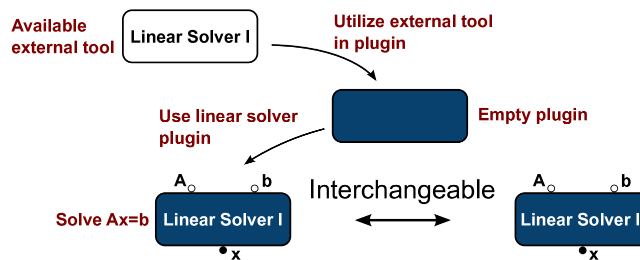

Fig. 2. A component is used to wrap available functionality, like different linear solver implementations. Input sockets (empty bullets) and output sockets (solid bullets) are referred to as sink and source sockets, respectively. Due to the abstraction mechanism provided by the socket concept, components can be exchanged. In this case, two linear solver components wrapping different implementations are exchanged in a straightforward manner, as both components offer the same socket setup.

followed by executing the topological sort algorithm to determine the linear execution sequence. ViennaX manually triggers a re-execution of the sub-graph representing the loop part via the previously identified loop vertices.

Various task graph schedulers are provided, enabling serial or concurrent execution in a data or task parallel manner based on the MPI. In the data parallel mode all components work on a subset of the data, thus one component is processed after another. On the contrary, in the task parallel mode components as a whole are – if the input dependencies are satisfied – executed in parallel, i.e., each MPI process executes a subset of components. Previous work discusses these parallelization scenarios in detail as well as the negligible computational overhead of approximately one second introduced by the component approach [6].

Components are loaded at run-time according to the framework's input configuration supplied via the extensible markup language (XML). This input configuration mechanism is also used to forward specific parameters from input files to individual components, such as a tolerance break-off threshold used in iterative linear solvers. ViennaX automatically handles data communication between components via the socket system, even within an MPI-based distributed computing environment. Therefore, no additional development effort is required when deploying a component in a serial or in a parallel setting.

Components can have input and output dependencies, implemented by the socket system. Input and output sockets are called sink and source sockets, respectively. Fig. 2 depicts the setup and exchange of a plugin. In general, the data associated with the sockets can either be already available, thus no copying is required, or the data object can be generated automatically by the respective socket generation method. The latter is depicted in the following, where an object of type `double` is instantiated and associated with a source socket identified with the string "`val`".

```
1 create_source<double>("val");
```

Each socket is uniquely identified by the type, e.g., `double`, of the data it represents and the key string, e.g., "`val`". Based on this unique socket identification tuple, the source and sink sockets of the simulation components are automatically connected by ViennaX. The data of the socket is accessed by the respective access methods:

```
1 double& value = access_source<double>("val");
```

In contrast, an already instantiated data object can be *linked* to a source socket by using – instead of the `create_source` method – the `link_source` method:

```
1 double value;
2 link_source(value, "val");
```

The access mechanism remains the same. Note that similar implementations for sink sockets are available.

With respect to the component implementation, each component is in fact based on a plugin, following a class hierarchy to impose a unified interface. Each component offers an initialization, execution, and finalization method. The initialization part is responsible for preparing data structures required during the execution as well as setting up sink and source sockets. The execution method holds the actual computation and may be executed several times, if the component is part of a loop. The finalization part is used for cleanup, such as deallocating memory.

Our component framework enables a high degree of flexibility, as changing parts of a simulation is reduced to switching plugins by solely altering the framework's input XML-based configuration data. Consequently, no changes in the implementations are required, thus avoiding recompilation and knowledge of the code base. The major advantage, however, is the profound increase in long-term reusability. Over the time a set of components is implemented which will find a new purpose in future simulation setups. The setup of simulations is thus restricted to merely selecting the components to be utilized via the XML-based configuration data, instead of actually coding the implementation.

4. Applications

This section presents application cases of ViennaX for FE-based simulations. We investigate examples shipped with open source FE libraries, being ViennaFEM and deal.II, respectively. The individual implementations are initially analyzed, followed by a decoupled implementation based on ViennaX. The steps required to perform this decoupling are discussed via code snippets to elaborate on usability. In this work we focus on sequentially executed FE applications, however, ViennaX fully supports large-scale task and data parallel applications based on MPI [6].

Section 4.1 discusses a fine-grained decoupling of an elasticity problem provided by the ViennaFEM library. Section 4.2 depicts ViennaX's loop mechanism via decoupling an adaptive mesh refinement application shipped with the deal.II library. Section 4.3 further underlines the advantages of using ViennaX by solving the Poisson equation with the FV method.

4.1. ViennaFEM: elasticity simulation

In this section we investigate a fine-grained decoupling of an elasticity example provided by the ViennaFEM library [20]. More concretely, we solve the weak formulation of the Lamé equation, describing linear elasticity. The task is to find u in a suitable space of trial functions U such that $\int_\Omega \varepsilon(u) : \sigma(v)\, d\Omega = \int_\Omega F \cdot v\, d\Omega$, for all vector-valued test functions v from a suitable space V, for example, $U = V = H_0^1(\Omega)^3$. Note that: represents the so-called double dot product, $\varepsilon(u)$ denotes the strain tensor, $\sigma(v)$ relates to the stress tensor, and F refers to the force. We analyze the initial implementation (Section 4.1.1), followed by a detailed description of a decoupled ViennaX approach (Section 4.1.2).

4.1.1. The initial implementation

The initial implementation consists of a single C++ source file, containing all required parts for conducting the simulation. In the following we give an overview of the implementation.

The mesh file is imported into the mesh data structure, provided by the ViennaGrid library [40].

```
1 Domain         domain;             // The mesh data structure
2 Segmentation   segments(domain);  // Support mesh partitions
3 netgen_reader  reader;            // NetGen mesh file reader
4 reader(domain, segments, file);   // Import the mesh file
```

Note that we utilize a mesh file reader for the NetGen meshing tool [41].

The stress and strain tensors are generated by using the symbolic math library ViennaMath [22]. In particular, the strain tensor $\varepsilon_{i,j} = \frac{1}{2}(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i})$ and the stress tensor $\sigma = 2\mu\varepsilon + \lambda \mathrm{tr}(\varepsilon(v))I$ are used, where μ and λ are the Lamé coefficients and I refers to the identity matrix. Next the implementation of the strain tensor is outlined. The stress tensor is implemented in a corresponding manner.

```
1 // a 3x3 matrix holds the strain tensor
2 ExprVec strain(9);
3 FuncSymVec u, v;
4 strain[0] =       diff(u[0], x);
5 strain[1] = 0.5 * (diff(u[0], y) + diff(u[1], x));
6 strain[2] = 0.5 * (diff(u[0], z) + diff(u[2], x));  /* .. */
```

ExprVec refers to a vector of dynamic ViennaMath expression. The data type FuncSymVec denotes a vector-valued unknown function type. u denotes one of the two unknown vector-valued functions, being evaluated by the solution process. x,y,z are the coordinate variables, whereas diff denotes ViennaMath's differentiation mechanism.

The weak formulation of the Lamé equation is implemented by applying a force in z-direction $F = (0, 0, 1)^T$:

```
1 Equ equ = make_equation(
2  integral(symbolic_interval(),tensor_reduce(strain,stress)),
3  integral(symbolic_interval(),rt_constant<double>(1.)*v[2]));
```

Line 2 represents the left hand side of the equation, and Line 3 holds the right hand side. The tensor_reduce function implements the double dot product.

Dirichlet boundary values are assigned on the mesh elements of choice, for instance, along a particular surface area.

```
1 viennafem::set_dirichlet_boundary(storage, vertex, value);
```

storage represents a quantity database, allowing to store data associated with, for instance, the elements of a mesh. vertex relates to a particular point of the mesh, which for boundary conditions is obviously located on the boundary. value refers to the actual Dirichlet boundary value.

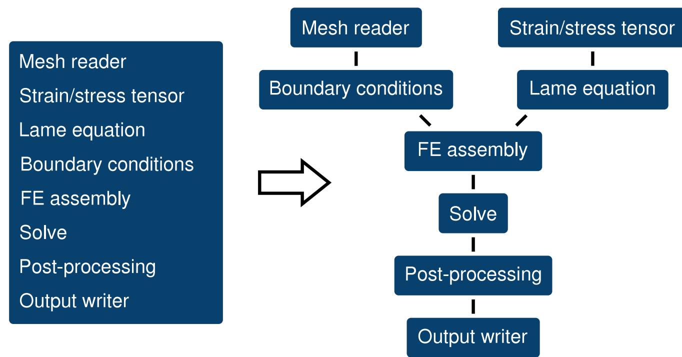

Fig. 3. The schematic overview of decoupling the implementation of an elasticity simulation into a flexible and reusable ViennaX simulation. The individual simulation parts of the initial implementation (*left*) are separated into reusable components and connected according to their dependencies (*right*), forming the actual simulation. For the sake of simplicity, the sockets are abstracted by single edges.

The system matrix as well as the load vector are being assembled by the respective assembly mechanism.

```
1 fem_assembler(make_linear_pde_system(equ,u),domain,
2  system_matrix,load_vector);
```

The assembled system of linear equations is solved by utilizing the ViennaCL library [21].

```
1 Vector displacements = solve(system_matrix, load_vector);
```

`displacements` hold the final solution of the simulation, which is the displacement vector field.

The solution is used to move the mesh points according to the displacement field, to further improve the significance of the visualization.

```
1 // move the point vectors of each vertex according
2 // to the displacement vector field
3 apply_displacements(domain, storage, displacements);
```

Finally, the mesh including the solution quantity is exported using the VTK file format which is supported by various visualization tools such as ParaView.

```
1 write_solution_to_VTK_file(displacements, name, domain,
2  segments, storage);
```

4.1.2. Simulation decoupling

The implementation discussed in Section 4.1.1 can be modularized in several ways. For instance, the linear solver or the file reader method can be decoupled. Where the first would allow for utilizing different linear solver implementations – such as the serial iterative linear solvers provided by PETSc [42] – the latter would enable to support various additional input file formats. However, the granularity of the modularization can be significantly extended, further increasing the level of reusability and flexibility of the entire simulation. Increasing the number of simulation components consequently also increases the number of exchangeable simulation parts, thereby significantly increasing, for example, the possible ways of altering a simulation flow. Although such an approach requires additional initial development effort – due to the implementation of the plugins – the effort pays off in the long run. Having access to a set of ready-to-use components significantly reduces the setup time of future simulations, as these components can be reutilized by selecting them via the ViennaX's XML-based input configuration data. We investigate a decoupling as depicted in Fig. 3. For the sake of simplicity, we focus the investigation on two exemplary components. The remainder of the components are implemented in a similar manner. The discussion on the plugin implementation is followed by presenting the required ViennaX XML input configuration file which drives the overall execution.

Stress/strain tensor component. This particular component offers four outgoing dependencies which are the stress and strain tensors as well as two vector-valued unknown functions (`u,v`). We thus set up the following data structures and implement the according source sockets:

```
1 // Component state
2 FuncSymVec u, v;
3 ExprVec    strain, stress;
4 void init() {  // Initialization method
5  link_source(strain,"strain");
```

```
6  link_source(stress,"stress");
7  link_source(u,      "u");
8  link_source(v,      "v");/*allocate u,v,strain,stress*/}
```

The implementation of the execution implementation is as follows:

```
1 bool execute(std::size_t call) {
2  strain[0] =         diff(u[0], x);
3  strain[1] = 0.5 * (diff(u[0], y) + diff(u[1], x));
4  strain[2] = 0.5 * (diff(u[0], z) + diff(u[2], x));
5  /* .. */ return true; }
```

The implementation is similar to the initial ViennaFEM version. As the source sockets are linked to the respective data objects, updating the objects themselves automatically updates the data available via source sockets. The variable `call` (Line 1) provides the current loop iteration of the component, which is not required in the presented application. The Boolean return value (Line 5) is used for loop iterations in order to notify the framework whether the loop should be exited. As this application does not utilize a loop, we return `true` by default.

Note that we do not require any deallocation mechanisms, an implementation of the finalization method is therefore not required.

Equation component. The equation object sets up the Lamé equation based on the stress and strain tensors. We thus implement the following initialization method.

```
1 void init() {
2  create_sink<ExprVec>     ("strain");
3  create_sink<ExprVec>     ("stress");
4  create_sink<FuncSymVec> ("v");
5  create_source<Equ>       ("equation"); }
```

We set up three sink sockets, holding the stress and strain tensors as well as the vector-valued unknown function `v`. One source socket is defined for forwarding the equation object to the assembly component.

The execution method accesses the data objects associated with the sockets and utilizes these objects for the computation.

```
1 bool execute(std::size_t call) {
2  ExprVec    & strain = access_sink<ExprVec>    ("strain");
3  ExprVec    & stress = access_sink<ExprVec>    ("stress");
4  FuncSymVec & v      = access_sink<FuncSymVec>("v");
5  Equ        & equ    = access_source<Equ>      ("equation");
6  equ = make_equation(
7   integral(symbolic_interval(),tensor_reduce(strain,stress)),
8   integral(symbolic_interval(),v[2]));
9  return true; }
```

XML input configuration. Each component is identified with a specific `key`. A plugin's name is defined via a macro in the respective source file. For instance, the mesh reader implementation offers the macro `#define PLUGIN_NAME ViennaGridReader`. ViennaX is notified, which of the available components to utilize by referring to the respective name. To set up the simulation flow provided by the initial implementation, the following configuration file is used for this particular application.

```
1 <plugins>
2    <plugin><key>ViennaGridReader</key>
3       <inputfile>meshfile.data</inputfile></plugin>
4    <plugin><key>ViennaMathLameStressStrain</key></plugin>
5    <plugin><key>ViennaMathLameEquation</key></plugin>
6    <!-- et cetera -->
7 </plugins>
```

ViennaX automatically determines the dependencies of each plugin and generates a task graph accordingly. No user input is required to ensure a proper connection of components. Also, additional component-specific properties can be provided via the configuration file, e.g., the path of the input mesh file is passed to the mesh file reader (`ViennaGridReader`, Line 3). The user is able to introduce arbitrary XML tags for storing such parameters, for instance, this example used the tag `inputfile`. A component accesses these properties in its methods via an access function. In the case of the mesh reader component, the

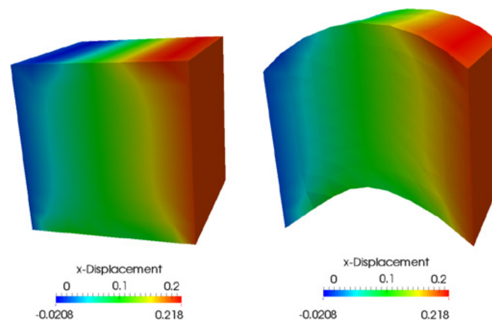

Fig. 4. The displacement in x-direction is shown for the ViennaFEM elasticity problem. *Left:* the solution based on the initial mesh. *Right:* the solution after the mesh is adapted according to the displacement field. Vector-valued Dirichlet boundary conditions are applied on the opposing left (blue) and right (red) sides, being $(0, 0, 0)^T$ and $(0.2, 0, 0)^T$, respectively. The applied additional force in z-direction results in an upwards bending.

input file parameter is retrieved by using an XPath expression. Note that each component has only access to its own XML region, i.e., its own specific parameters.

```
1 std::string file = query_value("inputfile");
```

Fig. 4 shows the simulation results of the discussed elasticity problem.

4.2. deal.II: adaptive mesh refinement

This section discusses another typical requirement for FE simulations, namely adaptive mesh refinement (AMR). The general idea is to improve the solution process by locally adapting the mesh during the course of the simulation. Usually, a first solution is computed based on an initial mesh. Based on this solution, the mesh is locally adapted.

Typically, a refinement is conducted in regions with large gradients or large curvatures in the solution [43,44]. Increasing the resolution, i.e., adding mesh elements in areas of interest, tends to increase the overall accuracy. The adaptation and solution steps are repeated, until a certain threshold level is reached as indicated by appropriate error estimators [45].

In the following, an example application provided by the deal.II library is investigated. Concretely, we analyze the `step-8` example, dealing with an elasticity problem coupled with an iterative AMR scheme. In contrast to the previous application case (Section 4.1), we do not focus on a high degree of decoupling of the simulation itself. Instead, we investigate ViennaX's loop mechanism which is required to model the iterative nature of the AMR algorithm. We analyze the initial implementation (Section 4.2.1), followed by a detailed description of a decoupling scheme (Section 4.2.2).

4.2.1. The initial implementation

The initial implementation is based on a single C++ source file, containing a class implementation (`ElasticProblem`), which provides methods to utilize the deal.II library for solving the elasticity problem. We only show the class interface instead of the full implementation.

```
1 template <int dim>
2 class ElasticProblem { // ..
3  void run ();
4  void setup_system ();
5  void assemble_system ();
6  void solve ();
7  void refine_grid ();
8  void output_results (const unsigned int cycle) const; };
```

The structure of the class implementation provides an intuitive description of the entire simulation process. The class itself is parameterized via the compile-time template parameter `dim` (Line 1), indicating the dimensionality of the problem. The `run()` method (Line 3) drives the overall simulation by performing the iteration process, in turn calling the individual methods required for the computation (Lines 4–8). As the focus of this particular application example is on implementing the AMR algorithm via the ViennaX loop mechanism, the implementation of the `run()` method deserves closer investigation:

```
1 template <int dim>
2 void ElasticProblem<dim>::run () {
3  for (unsigned int cycle=0; cycle<6; ++cycle) {
4   if (cycle == 0) {    // Initial mesh generation
5    hyper_cube (triangulation, -1, 1);
```

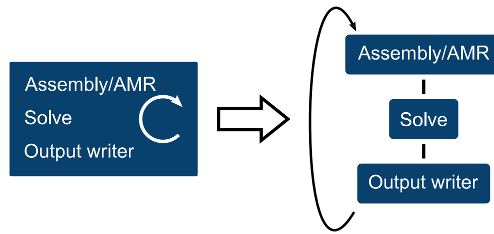

Fig. 5. The schematic overview of the initial deal.II example (*left*) and our proposed decoupling into a loop-based execution graph (*right*) is shown. For the sake of simplicity, the sockets are abstracted by single edges.

```
6      triangulation.refine_global (2); }
7     else refine_grid (); // AMR
8     setup_system ();
9     assemble_system ();
10    solve (); } }
```

Note that the AMR loop is implemented via the for-loop and the `refine_grid` method (Lines 3–7). In the first iteration an initial mesh is generated (Lines 4–6). During each loop iteration all steps required for this particular FE simulation are conducted, being the assembly of the linear system of equations and the linear solution process (Lines 8–10).

4.2.2. Simulation decoupling

An exemplary decoupling of this particular simulation is schematically depicted in Fig. 5. Note that this particular `ElasticProblem` class implementation had to be slightly altered to allow external access to the data structures and methods. In the following, we focus on the implementation aspects of the `Assembly/AMR` and `Output writer` components.

Assembly/AMR component. The initialization phase of the `Assembly/AMR` component is implemented as follows.

```
1 // Component state
2 ElasticProblem<2>    sim; /* .. */
3 void init() { // Initialization method /* .. */
4 link_source(sim, "sim");
5 create_sink<bool> ("loopback");  }
```

We omit the instantiation of the data structures for the mesh, the system matrix, and the load vector. These data structures – kept in the state of the component – are forwarded to the constructor of the `sim` object. An additional sink socket is used for the loop mechanism (Line 5). A corresponding source socket is used within the last component in the loop sequence, being the `Output writer` component.

The implementation of the execution part is similar to the original version.

```
1 bool execute(std::size_t call) {
2  if (call == 0)    {
3   hyper_cube (triangulation, -1, 1);
4   triangulation.refine_global (4);  }
5  else sim.refine_grid ();
6  sim.setup_system ();
7  sim.assemble_system ();
8  return true; }
```

No explicit loop mechanism is required, such as a `for`-loop, as ViennaX automatically issues a re-execution of the components within the loop. However, to keep track of the loop iteration within the component, the `call` variable is used (Lines 1,2), replacing the originally utilized `cycle` loop parameter. Note that the `sim` object is used to call the respective methods for assembling the linear system of equations (Lines 5–7).

Output writer component. The `Output writer` component is – aside of generating the simulation output files – responsible for notifying ViennaX whether to continue the loop. Aside of the obvious sink sockets required to access the result data to be written to a file, this particular component has to provide a complementary source socket to close the loop, being defined in the initialization phase:

```
1  create_source<bool>   ("loopback");
```

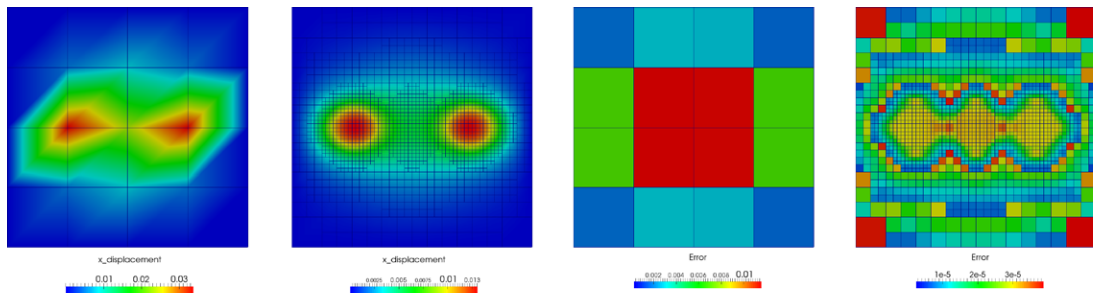

Fig. 6. *Left half:* the x-displacement for the deal.II example for the initial (left) and the six-times adapted mesh (right). *Right half:* the corresponding error estimates show a drop of the maximum error from 0.012 to 3.9E−5. The depicted error estimates for the sixth iteration would trigger a refinement in the corners in the subsequent iteration step, as due to the continuing reduction in the error the relative small solution gradients become relevant.

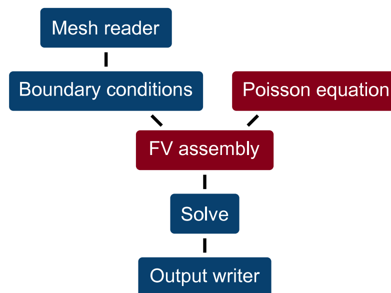

Fig. 7. The Poisson equation is solved with the finite volume method. Blue components are reutilized from the ViennaFEM example and the red components are new. In essence, the structure of the simulation flow is similar to the presented ViennaFEM example, however, by changing the equation and assembly component, the intended solution of the Poisson equation is implemented with a minimum of effort. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Note that this socket is not required for actual data communication, it is merely required to inform ViennaX that there is a backward-dependence.

The execution phase utilizes the Boolean return value to indicate whether the loop has to be continued. In the following example, after six executions, the loop is exited.

```
1 bool execute(std::size_t call) { /* .. */
2   if(call == 6) return true;
3   else return false; }
```

Note that by convention returning `true` forces ViennaX to exit the loop, whereas returning `false` triggers a new iteration.

Fig. 6 depicts exemplary simulation results for a three-dimensional simulation based on the introduced decoupled implementation.

4.3. ViennaFEM: Poisson equation

This section underlines the core advantages of our component approach, being the high degree of reusability and flexibility. We outline a simulation solving the FV-discretized Poisson equation by not only introducing new components but also by reusing several components utilized in the ViennaFEM application presented in Section 4.1. We focus on the high level utilization rather than the actual implementation. Fig. 7 depicts the simulation flow. Note the high degree of flexibility, as, for instance, by exchanging the equation component implementing the Lamé equation with a Poisson equation component, the physical modeling approach is changed. Additionally, by exchanging the FE-based assembly component with its FV counterpart, the discretization scheme of the simulation is switched. The high level of reusability is obvious due to the fact that four of six components are reused, which is further underlined, if a FE discretization of the Poisson equation is considered. In this particular case the original FE assembly component would be utilized, reusing five of six components. The coding efforts for the new components are primarily focused on the components' execution parts, as the socket setup remains essentially the same.

5. Summary

We depicted the applicability of ViennaX for increasing the flexibility and reusability of FE applications for scientific computing. Example implementations, shipped by the ViennaFEM library and the deal.II library, have been analyzed and decoupled into component-based ViennaX applications.

Overall, it has been shown that creating ViennaX applications based on already available components is straightforward. The advantage of implementing a set of ready-to-use components, which can be utilized in different simulation flows, has been discussed.

Acknowledgment

This work has been supported by the European Research Council (ERC) through the grant #247056 MOSILSPIN.

References

[1] deal.II. URL http://www.dealii.org/ (accessed 05.09.13).
[2] D.E. Bernholdt, R.C. Armstrong, B.A. Allan, Managing complexity in modern high end scientific computing through component-based software engineering, in: Proceedings of the HPCA Workshop on Productivity and Performance in High-End Computing, P-PHEC, 2004.
[3] C. Szyperski, Component Software: Beyond Object-Oriented Programming, second ed., Addison-Wesley Longman Publishing Co., Inc., 2002.
[4] Cactus. URL http://cactuscode.org/ (accessed 05.09.13).
[5] ViennaX. URL http://viennax.sourceforge.net/ (accessed 05.09.13).
[6] J. Weinbub, K. Rupp, S. Selberherr, ViennaX: a parallel plugin execution framework for scientific computing, Eng. Comput. (2013) http://dx.doi.org/10.1007/s00366-013-0314-1.
[7] W. Bangerth, R. Hartmann, G. Kanschat, deal.II—a general-purpose object-oriented finite element library, ACM Trans. Math. Softw. 33 (4) (2007) 24:1–24:27. http://dx.doi.org/10.1145/1268776.1268779.
[8] W. Bangerth, C. Burstedde, T. Heister, et al., Algorithms and data structures for massively parallel generic adaptive finite element codes, ACM Trans. Math. Softw. 38 (2) (2011) 14:1–14:28. http://dx.doi.org/10.1145/2049673.2049678.
[9] DUNE. URL http://www.dune-project.org/ (accessed 05.09.13).
[10] A. Dedner, R. Klöfkorn, M. Nolte, et al., A generic interface for parallel and adaptive discretization schemes, Computing 90 (3–4) (2010) 165–196. http://dx.doi.org/10.1007/s00607-010-0110-3.
[11] B.S. Kirk, J.W. Peterson, R.H. Stogner, et al., libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations, Eng. Comput. 22 (3–4) (2006) 237–254. http://dx.doi.org/10.1007/s00366-006-0049-3.
[12] libMesh. URL http://libmesh.sourceforge.net/ (accessed 05.09.13).
[13] FEniCS. URL http://fenicsproject.org/ (accessed 05.09.13).
[14] A. Logg, K.A. Mardal, G. Wells, Automated Solution of Differential Equations by the Finite Element Method, in: Lecture Notes in Computational Science and Engineering, vol. 84, Springer, 2012, http://dx.doi.org/10.1007/978-3-642-23099-8.
[15] A. Logg, G.N. Wells, DOLFIN: automated finite element computing, ACM Trans. Math. Softw. 37 (2) (2010) 20:1–20:28. http://dx.doi.org/10.1145/1731022.1731030.
[16] OpenFOAM. URL http://www.openfoam.com/ (accessed 05.09.13).
[17] H. Jasak, A. Jemcov, Ž. Tuković, OpenFOAM: a C++ library for complex physics simulations, in: Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics, 2007, pp. 47–66.
[18] FreeFem++. URL http://www.freefem.org/ff++/ (accessed 03.12.13).
[19] F. Hecht, New development in FreeFem++, J. Numer. Math. 20 (3–4) (2012) 251–265. http://dx.doi.org/10.1515/jnum-2012-0013.
[20] ViennaFEM. URL http://viennafem.sourceforge.net/ (accessed 05.09.13).
[21] ViennaCL. URL http://viennacl.sourceforge.net/ (accessed 05.09.13).
[22] ViennaMath. URL http://viennamath.sourceforge.net/ (accessed 05.09.13).
[23] CCA. URL http://www.cca-forum.org/ (accessed 05.09.13).
[24] R. Armstrong, D. Gannon, A. Geist, et al. Toward a common component architecture for high-performance scientific computing, in: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing, HPDC, 1999, pp. 115–124.
[25] D.E. Bernholdt, B.A. Allan, R. Armstrong, et al., A component architecture for high-performance scientific computing, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 163–202. http://dx.doi.org/10.1177/1094342006064488.
[26] B.A. Allan, R.C. Armstrong, D.E. Bernholdt, et al., The CCA core specification in a distributed memory SPMD framework, Concurr. Comput.: Pract. Exper. 14 (5) (2002) 323–345. http://dx.doi.org/10.1002/cpe.651.
[27] M. Govindaraju, M.R. Head, K. Chiu, XCAT-C++: Design and Performance of a Distributed CCA Framework, in: Lecture Notes in Computer Science, vol. 3769, 2005, pp. 270–279. http://dx.doi.org/10.1007/11602569_30.
[28] M.J. Lewis, A.J. Ferrari, M.A. Humphrey, et al., Support for extensibility and site autonomy in the Legion grid system object model, J. Parallel Distrib. Comput. 63 (5) (2003) 525–538. http://dx.doi.org/10.1016/S0743-7315(03)00012-1.
[29] K. Zhang, K. Damevski, V. Venkatachalapathy, et al. SCIRun2: a CCA framework for high performance computing, in: Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments, HIPS, 2004, pp. 72–79. http://dx.doi.org/10.1109/HIPS.2004.1299192.
[30] T. Goodale, G. Allen, G. Lanfermann, et al., The Cactus framework and toolkit, in: High Performance Computing for Computational Science—VECPAR 2002, in: Lecture Notes in Computer Science, vol. 2565, 2003, pp. 197–227. http://dx.doi.org/10.1007/3-540-36569-9_13.
[31] COOLFluiD. URL http://coolfluidsrv.vki.ac.be/trac/coolfluid/ (accessed 05.09.13).
[32] T. Quintino, A component environment for high-performance scientific computing, Ph.D. Thesis, Katholieke Universiteit Leuven, 2008.
[33] ESMF. URL http://www.earthsystemmodeling.org/ (accessed 05.09.13).
[34] C. Hill, C. DeLuca, V. Balaji, et al., The architecture of the earth system modeling framework, Comput. Sci. Eng. 6 (1) (2004) 18–28. http://dx.doi.org/10.1109/MCISE.2004.1255817.
[35] Uintah. URL http://www.uintah.utah.edu/ (accessed 05.09.13).
[36] M. Berzins, Status of Release of the Uintah Computational Framework, Tech. Rep. UUSCI-2012-001, Scientific Computing and Imaging Institute, University of Utah, 2012.
[37] J. Davison de St Germain, J. McCorquodale, S.G. Parker, et al. Uintah: a massively parallel problem solving environment, in: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing, HPDC, 2000, pp. 33–41. http://dx.doi.org/10.1109/HPDC.2000.868632.
[38] A. Miller, The task graph pattern, in: Proceedings of the 2nd Workshop on Parallel Programming Patterns, ParaPLoP, 2010, pp. 8:1–8:7. http://dx.doi.org/10.1145/1953611.1953619.
[39] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 2009.
[40] ViennaGrid. URL http://viennagrid.sourceforge.net/ (accessed 05.09.13).

[41] J. Schöberl, NETGEN an advancing front 2D/3D mesh generator based on abstract rules, Comput. Vis. Sci. 1 (1) (1997) 41–52. http://dx.doi.org/10.1007/s007910050004.
[42] PETSc. URL http://www.mcs.anl.gov/petsc/ (accessed 05.09.13).
[43] P. Colella, J. Bell, N. Keen, T. Ligocki, et al., Performance and scaling of locally-structured grid methods for partial differential equations, J. Phys. Conf. Ser. 78 (1) (2007) 012013. http://dx.doi.org/10.1088/1742-6596/78/1/012013.
[44] M. Möller, D. Kuzmin, Adaptive mesh refinement for high-resolution finite element schemes, Internat. J. Numer. Methods Fluids 52 (5) (2006) 545–569. http://dx.doi.org/10.1002/fld.1183.
[45] T.J. Barth, A posteriori error estimation and mesh adaptivity for finite volume and finite element methods, in: Adaptive Mesh Refinement—Theory and Applications, in: Lecture Notes in Computational Science and Engineering, vol. 41, 2005, pp. 183–202. http://dx.doi.org/10.1007/3-540-27039-6_13.