Engineering with Computers (2014) 30:651–668
DOI 10.1007/s00366-013-0314-1

ORIGINAL ARTICLE

ViennaX: a parallel plugin execution framework for scientific computing

Josef Weinbub · Karl Rupp · Siegfried Selberherr

Received: 17 August 2012 / Accepted: 28 January 2013 / Published online: 13 February 2013

Abstract We present the free open source plugin execution framework ViennaX for modularizing and parallelizing scientific simulations. In general, functionality is abstracted by the notion of a task, which is implemented as a plugin. The plugin system facilitates the utilization of both, already available functionality as well as new implementations. Each task can define arbitrary data dependencies which are used by ViennaX to build a task graph. The framework supports the execution of this dependence graph based on the message passing interface in either a serial or a parallel fashion. The applied modular approach allows for defining highly flexible simulations, as plugins can be easily exchanged. The framework's general design as well as implementation details are discussed. Applications based on the Mandelbrot set and the solution of a partial differential equation are investigated, and performance results are shown.

Keywords Task graph · Parallel execution · Plugin system software reuse · Framework

1 Introduction

The field of scientific computing is based on modeling various physical phenomena. A promising approach to improve the quality of this modeling is to combine highly specialized simulation tools [23] and is commonly practiced in fields like computational fluid dynamics (CFD) [37]. In short, important tasks are to couple simulations which model relevant phenomena on a different physical level, thus performing multiphysics computations. Although several multiphysics tools are publicly available, the implementations are typically based on assumptions with respect to the field of application. For example, a specific discretization method is used, such as the finite element methods (FEMs) utilized by the Elmer[1] framework. Aside from combining different simulators to perform multiphysics simulations, decoupling a simulation into smaller parts is of high interest. The ability to reuse these parts for different simulation setups significantly increases reusability and, thus reduces long-term implementation efforts.

The available frameworks applied in the field of distributed high-performance scientific computing usually focus on the data parallel approach based on the message passing interface (MPI). Typically, a mesh datastructure representing the simulation domain is distributed, thus the solution is locally evaluated on the individual subdomains. This approach is referred to as domain decomposition [28], and is reflected by a data parallel approach. As such, the tasks to be executed by the framework are typically processed in a sequence, whereas each plugin itself utilizes the MPI to distribute the work among the compute units, for example, to utilize an MPI-based linear solver.

ViennaX[2] does not restrict itself to such an execution behavior, in fact its focus is on providing an extendible set of different schedulers to not only support data parallel approaches, but also serial and task parallel implementations. In this

J. Weinbub (✉) · S. Selberherr
Institute for Microelectronics, Technische Universität Wien, Vienna, Austria
e-mail: weinbub@iue.tuwien.ac.at

K. Rupp
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA

[1] http://www.csc.fi/english/pages/elmer/.

[2] http://viennax.sourceforge.net/.

context, serial execution refers to the execution of the tasks on a shared-memory machine, enabling to execute the tasks sequentially, however, the plugins can indeed have parallelized implementations based on, for example, OpenMP or CUDA. Such an approach becomes more and more important, due to the broad availability of multi-core CPUs by simultaneously stagnating clock frequencies [10, 16, 20]. On the contrary, task parallel approaches can be used to parallelize data flow applications, for instance, wave front [15] or digital logic simulations [29].

ViennaX facilitates the setup of flexible scientific simulations by providing an execution framework for plugins. The decoupling of simulations into separate components is facilitated by the framework's plugin system. Functionality is implemented in plugins, supporting data dependencies. Most importantly, the plugin system enables a high degree of flexibility, as exchanging individual components of a simulation is reduced to switching plugins by altering the framework's configuration data. Consequently, no changes in the simulation's implementation must be performed, thus avoiding recompilation and knowledge of the code base. Furthermore, decoupling simulation components into plugins also increases the reusability significantly. For example, a file reader plugin for a specific file format can be utilized in different simulations. Ultimately, the effort of changing parts of the simulation is greatly reduced, strongly favoring long-term flexibility and reusability.

In this work we depict the applicability of ViennaX (Version 1.2.0) for scientific computing. We extend previously published material [35, 36] by a rigorous discussion of the library details, framework design, user-interface, and execution performance. The framework is written in the C++ programming language and is available under the MIT free open source license. ViennaX is available for Linux and Mac OS X platforms. Different scheduler kernels are available relying on a task graph [30] approach, modeling the data dependencies between the individual tasks. The implementation heavily relies on already available functionality, such as the Boost libraries[3], and on advanced programming techniques, like generic programming [17], to realize a maintainable and extendible implementation by simultaneously upholding a small code base.

This work is organized as follows: Sect. 2 puts the work into context. Section 3 provides an overview of the framework. Section 4 shows an exemplary plugin implementation. Section 5 discusses implementation details. Section 6 introduces application results whereas Sect. 7 lists shortcomings and possible future extensions.

[3] http://www.boost.org/.

2 Related work

Various research has already been conducted to investigate either a generalized approach to connect and generalize scientific simulations and/or to apply graph-based approaches in computations. This section provides a selection of related research. At the end of this section, a table of comparison is provided (Table 1).

The intel threading building blocks (Intel TBB) library[4] is written in C++, licensed under the GNU General Public License (GPL) and utilizes a shared-memory approach to enable parallelism. One of the core features is the so-called flow graph, which allows for task parallel implementations. A flow graph can be used to send messages, representing arbitrary data, between components. Our approach utilizes a task-based parallelism approach similar to the TBB library. However, we employ a distributed approach based on the MPI, which enables the scaling of our framework beyond a single node of, for instance, a commodity cluster, thus providing considerable flexibility with respect to the computing environment.

The common component architecture (CCA)[5] is a standard and applies so-called component-based software development to encapsulate units of functionality into components [4, 8]. Data communication between components is implemented via so-called ports. An interface definition language (IDL) 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 [8], such as the high-performance computing framework CCAFFEINE [3] and the distributed computing frameworks XCAT [19], Legion [26], and SCIRun2 [38]. Our approach is similar to the CCA with respect to the component system. However, we solely focus on the C++ language, therefore avoiding additional overhead. This simplification is reasonable, as C++ can be coupled with other regularly used languages in scientific computing, like Python, Fortran, and C. The strength of our framework is the straightforward applicability due to the fact that the entry level for users is low. For example, defining ports, and utilizing the plugin with the framework, does not require an IDL. Utilizing available implementations is reduced to coupling the datastructures with a plugin's sockets, representing the plugin interface. Although the approach proposed by the CCA is of high importance to introduce reusability for scientific computations, in our opinion the additional effort is primarily justified for large-scale projects. On the other hand, we

[4] http://threadingbuildingblocks.org/.

[5] http://www.cca-forum.org/.

Table 1 An overview of the features of related projects is given

	Intel TBB	CCA	Cactus	ESMF	COOLFluiD	Uintah	DAGuE	ViennaX
Graph	•		•	•		•	•	•
MPI		•	•	•	•	•	•	•
Threads	•	•	•	•	•	•	•	•
Micro	•	•					•	
Macro	•	•	•	•	•	•		•
Generic	•	•	•					•
Task parallelism	•			•		•	•	•
Data parallelism		•	•	•	•	•		•

Graph denotes whether the package utilizes a task graph to model data dependencies. MPI and Threads refer to application of the respective parallelization layer for the execution of the individual components. Micro and Macro denote whether the projects' components focus on containing small or large parts of an algorithm, respectively. Generic refers to the general applicability in arbitrary fields of scientific computing. Task and data parallelism denotes the focus of the individual approach for the respective parallelization approach

believe that ViennaX is especially of interest for applications on a daily basis aiming for small to medium-scale scenarios.

Cactus[6] is a multi-purpose framework, which has its roots in the field of relativistic astronomy [18]. The framework is available under the GNU lesser general public license (LGPL) and focuses on data parallel approaches. The design follows a modular approach and supports different target architectures as well as collaborative code development. The central part of the framework (called "flesh") connects the individual typically user-supplied application modules (called "thorns"), typically containing the implementations of the actual simulation. Communication between thorns is realized via the framework's API. Connections between thorns are provided by the user in configuration files, which are processed during compile-time. Cactus provides its own make system, enabling automatic compilation for different hardware architectures and configuration options. Data dependencies of components have to be defined in specification files, using the Cactus Configuration Language (CCL). The ViennaX component communication layer is realized by a so-called socket system which does not require an introduction of an additional configuration language to setup data dependencies. Aside from data parallelization ViennaX also supports applications with a focus on task parallel and serial execution.

The earth system modeling framework (ESMF)[7] provides the setup of flexible, reusable, and large-scale simulations in climate, weather, and data assimilation domains [21]. The source code is publicly available under the University of Illinois-NCSA License. The software design is based on a component approach, enabling to separate functionality into reusable components offering a unified interface. The parallelization layer is abstracted by a virtual machine approach and focuses on data parallel and basic task parallel approaches. Although the latter is supported in principal, no automatic control mechanisms with respect to data dependencies are provided. 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 datatypes. ViennaX has improved support for task parallel execution approaches, as the execution of the individual plugins is automatically handled with respect to the data dependencies. Furthermore, ViennaX supports arbitrary datatypes for the data communication between the plugins. With respect to the parallelization layer, ViennaX directly exposes the parallelization backend to the user. We believe that such a straightforward approach is both sufficient and important. Sufficiency relates to the fact that MPI is the de facto standard for distributed computing. Furthermore, the approach is important in the sense that abstracting the parallel layer of an MPI library results in reimplementation and additional maintenance overhead to keep the interfaces updated.

The COOLFluiD project[8] enables multiphysics simulations based on a component framework [32] and is primarily designed for data parallel applications in the field of CFD. 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 can set up data sockets which are in turn used to generate a dependence hierarchy driving the overall execution. We adopted the plugin system of COOLFluiD, which enables us to conveniently wrap already available functionality. Additionally, the communication layer utilized by the so-called socket system has been simplified to fit our needs. As COOLFluiD focuses on a data parallel

6 http://cactuscode.org/.

7 http://www.earthsystemmodeling.org/.

8 http://coolfluidsrv.vki.ac.be/trac/coolfluid/.

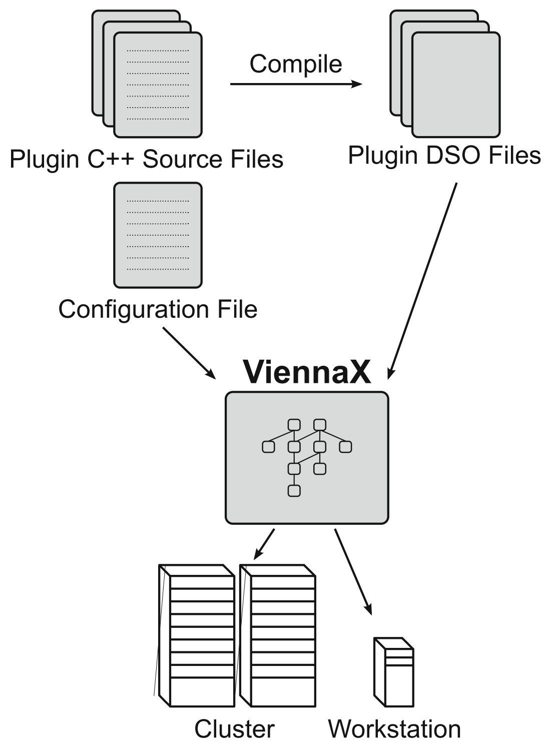


Fig. 1 Schematic utilization of ViennaX. C++ source files modeling the ViennaX plugin concept are compiled into DSOs. The DSOs as well as the configuration file are loaded into ViennaX. The plugins are loaded during run-time and based on the dependencies a task graph is generated. The plugins are executed according to the dependencies until the graph has been processed

approach, the library also provides its own distributed data structures, like the mesh data structure. Therefore, COOLFluiD performs an automatic partitioning and distribution of the mesh data structure via MPI primarily for domain decomposition methods. Although ViennaX does not provide its own distributed datastructures, already available implementations can be utilized such as the PETSc[9] or the p4est[10] library. However, aside from supporting data parallelism ViennaX supports serial and task parallelism.

The Uintah project[11] is a large-scale multi-physics computation framework [9, 13] available under the MIT license. Uintah solves reacting fluid-structure problems on a structured adaptive mesh refinement mesh. Uintah supports task and data parallel applications. The primary area of application is computational mechanics and fluid dynamics. 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 represents a set of tasks. The data dependencies between components are modeled by edges in the DAG. Our approach utilizes a similar task graph approach for modeling the data dependencies. However, we do not enforce an underlying mesh data structure, as we aim for a more general area of application. Nevertheless, Uintah's implementation of a hybrid task graph scheduler is of high interest for future extensions of our scheduler implementation, as it is capable of scaling up to 100,000s of cores.

The directed acyclic graph unified environment (DAGuE)[12] enables scientific computing on large-scale distributed, heterogeneous environments [11]. The source code is available under a BSD-similar license. The basis of DAGuE is a DAG-based scheduling engine, where the nodes are sequential computation tasks and the edges refer to data movements between the tasks. Computational tasks are encapsulated into sequential kernels. A DAGuE-specific language is used to describe the data flow between the kernels. Instead of focusing on executing relatively small parts of an algorithm within one node of the DAG, our approach also aims at utilizing full-scale simulation tools within a task. Additionally, ViennaX supports conventional distributed data parallel execution and does not require a specific language to describe the input and output dependencies.

Aside from the discussed projects, other frameworks in this area are available, such as Palm[13] [12], Overture [7], AMP [31], FastFlow[14] [2], Cilk Plus[15], FEniCS[16] [27], and OpenFOAM[17] [22].

3 The framework

In its essence, ViennaX is a plugin execution framework. Available simulation tools or components can be wrapped by the plugins and are, therefore, reused. A ViennaX application is constructed by executing a set of plugins. The input configuration file based on the extensive markup language (XML) contains information indicating the plugins to be utilized during the course of the execution. Additionally, parameters can be provided by this configuration file, which are forwarded to the respective plugins by ViennaX.

[9] http://www.mcs.anl.gov/petsc/.
[10] http://www.p4est.org/.
[11] http://www.uintah.utah.edu/.

[12] http://icl.cs.utk.edu/dague/.
[13] http://www.overtureframework.org/.
[14] http://calvados.di.unipi.it/dokuwiki/doku.php?id=ffnamespace:about.
[15] http://software.intel.com/en-us/intel-cilk-plus.
[16] http://fenicsproject.org/.
[17] http://www.openfoam.org/.

Plugins can have data dependencies, which are internally represented by a task graph and handled by the so-called socket system. ViennaX provides different standalone applications, utilizing the different available scheduler kernels. Each of the kernels focuses on different execution approaches, being serial, task parallelism, and data parallelism, respectively. These applications can be used to execute the graphs generated from the input XML file.

Figure 1 schematically depicts the general execution flow of the framework. ViennaX plugins are implemented and compiled as dynamic shared objects (DSOs), which are forwarded to the framework's application. In addition to the plugins, the input configuration file is passed to the application. The ViennaX schedulers automatically generate and execute the task graph according to the data dependencies. The intended target platforms are workstations or clusters, which are supported by different distributed scheduler kernels based on the MPI. More specifically, the Boost MPI Library[18] is utilized.

ViennaX does not wrap the parallel execution layer of the target platform like MPICH[19]. As such, ViennaX is executed as a typical application utilizing the respective parallelization library. For instance, to execute an MPI capable ViennaX scheduler application, the following expression is used.

```
mpiexec -np 4 ./vxscheduler config.xml plugins/
```

As already stated, the MPI layer is not abstracted, therefore the scheduler application is executed according to the utilized MPI library. In this case the mpiexec command is used, which spawns the execution of four instances. vxscheduler relates to the application, whereas configuration.xml refers to the XML input file holding, for instance, the required information to build the task graph. The final parameter plugins refers to the directory path, containing the ViennaX-valid plugins to be utilized during the execution.

Although ready-to-use applications are provided, ViennaX provides an application programming interface (API). Therefore, ViennaX can be utilized in external applications in a straightforward manner.

Figure 2 depicts the fundamental design of ViennaX. An API exposes different scheduler kernels as well as the plugin system and the configuration facility to the user. The design of the framework allows for different task execution modes implemented by the respective scheduler kernels to support, for instance, different parallel task graph execution strategies.

[18] http://www.boost.org/libs/mpi/.

[19] http://www.mpich.org/.

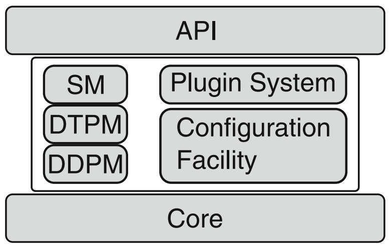


Fig. 2 Design of ViennaX. An API provides access to the supported different scheduler kernels, being serial mode (SM), distributed-task-parallel-mode (DTPM), and distributed-data-parallel-mode (DDPM). Additionally, the plugin system, and the configuration facility can be accessed by the user. The core part provides fundamental functionality utilized throughout the framework, such as a task graph implementation

Table 2 Overview of graph and plugin execution modes supported by the ViennaX scheduler kernels

	Graph execution	Plugin execution
SM	Serial	Serial/shared-memory
DTPM	Distributed	Serial/shared-memory
DDPM	Serial	Distributed

Table 2 depicts the currently provided scheduler kernels, being the serial mode (SM), distributed-task-parallel-mode (DTPM), and distributed-data-parallel-mode (DDPM) scheduler. The SM-based kernel processes one plugin at the time, where the individual plugins run either serial and/or parallel shared-memory-parallelized implementations restricted to a single process, such as OpenMP. The DTPM kernel models the task parallel concept in an MPI context, where plugins are executed in parallel by different MPI processes, if the respective dependencies are satisfied. Consequently, applications with parallel paths in the graph can benefit from such a scheduling approach, for instance, wave front simulations [15].

Finally, the DDPM kernel allows for a data parallel approach, where, although each plugin is processed consecutively, the plugin's implementation follows an MPI-based parallelization approach. Such an approach allows, for instance, to utilize an MPI-based linear solver component within a plugin, such as PETSc. Figure 3 schematically compares the principles of the different execution modes.

The currently implemented parallel scheduler focuses on the distributed MPI. To better support the ongoing development of continually increasing core numbers per computing target, scheduler kernels utilizing shared-memory parallelization approaches are planned for future extensions. These future extensions are supported by the introduced naming scheme for the scheduler kernels as well as by the modular kernel approach provided by ViennaX.

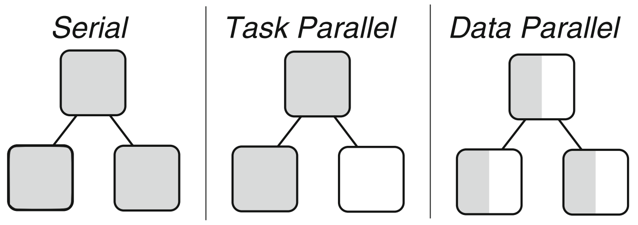


Fig. 3 Different task graph execution models, where each vertex of the graph represents a plugin. *Gray and white shaded* plugins denote different compute-units, e.g., MPI processes. In serial mode, one compute-unit executes all the tasks but only one task at a time. In task parallel mode, different compute-units are responsible for subsets of the task graph. In data parallel mode, each task is executed by every available compute-unit, where each compute-unit processes only a subset of the data

4 Exemplary plugin implementation

For the sake of clarity, an exemplary plugin implementation is provided, which not only depicts the utilization of ViennaX with respect to using the already available implementations in ViennaX plugins but is also a reference for the subsequent investigation of the implementation details.

The plugin wraps an iterative linear solver implementation of the header-only ViennaCL[20] library, thus providing a high-performance reusable linear solver component to ViennaX. Utilizing ViennaCL in this example additionally underlines the straightforward applicability of our framework with respect to utilizing already available implementations.

In the following, the full implementation of a simplistic ViennaCL-powered iterative solver plugin is given.

```
 1 // ViennaCL Includes
 2 #include "viennacl/linalg/bicgstab.hpp"
 3 // ViennaX Includes
 4 #include "viennax/core/plugin.hpp"
 5 // Plugin Name
 6 #define PLUGIN_NAME ViennaCLLinSol
 7 // Plugin Class Implementation
 8 namespace viennax {
 9 struct PLUGIN_NAME : public plugin {
10 INIT_VIENNAX_PLUGIN
11 // Initialization: Setup Sockets
12 void init() {
13   create_sink  <Matrix>("A");
14   create_sink  <Vector>("b");
15   create_source<Vector>("x");
16 }
17 // Execution: Perform computation
18 bool execute(std::size_t call) {
19   // Access socket data
20   Matrix& A = access_sink  <Matrix>("A");
21   Vector& b = access_sink  <Vector>("b");
22   Vector& x = access_source<Vector>("x");
23   // Solve the system
24   x = solve(A, b, bicgstab_tag());
25   return true;
26 } }; FINALIZE_VIENNAX_PLUGIN }
```

External header files are included required for utilizing ViennaCL in the plugin (Line 2). The plugin's name (Line 6), class definition (Lines 8, 9, 28), and required macros are implemented (Lines 10, 28). The data dependencies are set up in the initialization part (Lines 13–15). Two input sockets (A, b) and one output socket (x) are provided, relating to the linear system $Ax = b$. ViennaCL is utilized in the execution part (Lines 18–28). The data associated with the sockets are accessed (Lines 20–22) and the system is solved (Line 24). The result vector x is automatically available to other plugins via the outgoing data connection.

5 Implementation details

This section provides implementation details to the core components of ViennaX.

Section 5.1 introduces the plugin system, Sect. 5.2 outlines the configuration mechanism based on the XML, and Sect. 5.3 discusses the scheduler kernel implementations.

5.1 Plugins

In general, a plugin approach introduces a high level of reusability by wrapping already available functionality into components with a specific, unified interface. The plugins can contain core parts of simulations, such as a linear solver implementation, but also full-fledged simulators in their own right. This approach is highly flexible; for example, simulation tools may be combined to form multiphysics simulation flows, but they may also be decomposed into smaller components, enabling specific exchanges of functionality by switching the respective plugins.

Figure 4 depicts the setup and exchange of a plugin. If the process of interchanging plugins is compared to the one of conventional simulation tools, it becomes clear that the conventional approach requires actual coding, and as such in-depth knowledge of the implementation at hand. For obvious reasons, this fact impedes the implementation of changing functionality. With our plugin-based approach, the exchange can be realized conveniently by only adjusting the input configuration data accordingly.

In the following, implementation details on the core parts of our plugin system are provided, being the plugin registration mechanism, interface, and socket system.

5.1.1 Factory

One of the core capabilities of a plugin framework is to discover, load, and execute the plugins. In general, we apply the so called self-registering approach, enabling the

[20] http://viennacl.sourceforge.net/.

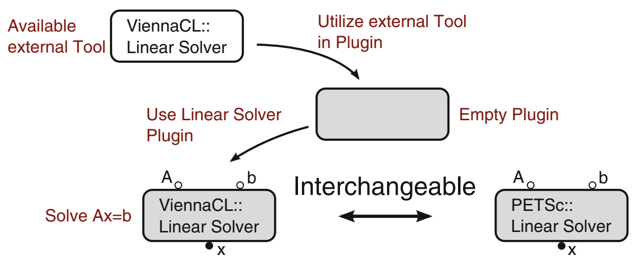


Fig. 4 A plugin can be used to wrap available functionality, like linear solver implementations. Due to the abstraction mechanism provided by the socket input/output dependencies, plugins can be exchanged by other plugins. In this case, a linear solver implementation provided by ViennaCL as depicted in Sect. 4 is interchanged with an implementations of PETSc

plugins to register themselves in a global plugin database upon loading the DSOs into the ViennaX application by the portable operating system interface (POSIX) dlopen command. The implementation is based on the so-called template factory design pattern [24, 32], which can be seen as an extension of the abstract factory design pattern with C++ templates.

Figure 5 depicts a simplified class diagram of our registration mechanism. The Base and Concrete template parameters refer to a base and a derived class of a class hierarchy, respectively. This hierarchy in turn relates to the ViennaX-provided base and user-provided derived plugin class, holding the actual functionality. Due to the increased genericity introduced by the template factory design pattern, class hierarchies of arbitrary type can be stored. However, the derived class has to satisfy a so-called registrable concept. This concept requires the derived class to provide a static function named ID returning an identification (ID) string and to offer a member type named Base holding the type of the base class. The need for the registrable concept is discussed in the following.

Each plugin source file holds aside from the implementation of the derived plugin (ViennaCLLinSol) a static object of the type Provider<ViennaCLLinSol>. The Provider class is part of the ViennaX factory mechanism and provides automatic registration within the factory's database. This automatism is based on the fact that static objects are generated during the start-up phase of the application, thus the registration related code provided by the Provider class is automatically executed before the main ViennaX application is executed.

The constructor of Provider<ViennaCLLinSol> utilizes the registrable concept induced interface to access the base class type and the ID string. This information is forwarded to the ProviderBase<Base> constructor, which in turn registers itself in the instance of the singleton pattern-based factory class. Using the factory's get method, a

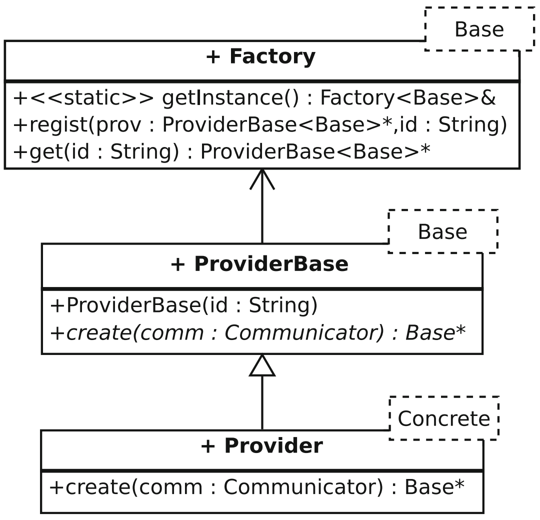

Base
+ Factory
+<<static>> getInstance() : Factory<Base>&
+regist(prov : ProviderBase<Base>*,id : String)
+get(id : String) : ProviderBase<Base>*
Base
+ ProviderBase
+ProviderBase(id : String)
+create(comm : Communicator) : Base*
Concrete
+ Provider
+create(comm : Communicator) : Base*

Fig. 5 Class diagram of the implemented template factory design pattern. The constructor of the ProviderBase class registers instances of itself into the singleton Factory class

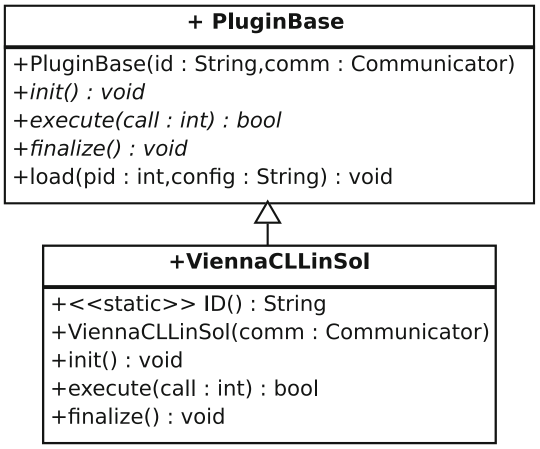

+ PluginBase
+PluginBase(id : String,comm : Communicator)
+init() : void
+execute(call : int) : bool
+finalize() : void
+load(pid : int,config : String) : void
+ViennaCLLinSol
+<<static>> ID() : String
+ViennaCLLinSol(comm : Communicator)
+init() : void
+execute(call : int) : bool
+finalize() : void

Fig. 6 Class diagram of the plugin interface. The user-side plugin ViennaCLLinSol models the registrable concept and derives from the ViennaX provided base plugin class PluginBase

specific plugin's Provider class can be retrieved and created with the respective create method.

5.1.2 *Interface*

This section discusses the plugin interface, which has to be modeled by a ViennaX plugin. Additionally, the general class hierarchy and the access for the ViennaX scheduler kernels is introduced. The class diagram shown in Fig. 6 gives an essential overview of the relations between the ViennaX base plugin class and a user-provided plugin implementation.

A straightforward dynamic polymorphism approach via virtual functions is used to specialize the functionality for each plugin. The implementation of the static function and the constructor are handled by macros to increase the level of convenience. For instance, by providing a macro holding the plugin name in the plugin's source file, a convenience macro can be utilized, automatically generating the required code. The following code snippet depicts the required code by the user's plugin implementation.

```
1 #define PLUGIN_NAME ViennaCLLinSol
2 struct PLUGIN_NAME : public plugin {
3 INIT_VIENNAX_PLUGIN
4   // Plugin interface functions ..
5 };
```

We deliberately do not automatically generate the C++ structure definition by macros (Lines 2, 5), to raise the awareness that the plugin structure is in fact a common C++ class, and can thus be used by the plugin developers to, for instance, store objects in its state.

ViennaX offers a three-stage interface model, enabling an initialization, execution, and finalize step realized by the init(), execute(), finalize() functions, respectively. Such an approach allows for multiple executions of a plugin within a simulation introduced by, for instance, loops. Utilizing a three-step interface is a common approach in scientific frameworks, such as the ESMF, and known to handle most application scenarios. However, more sophisticated needs cannot be covered by such an approach, for instance, additional communication between the individual components. Therefore, improving the interface for more intricate cases is a part of future extensions.

Aside from the three virtual interface functions, the scheduler kernels use the load method to initialize the plugin with the plugin specific configuration data and with a unique plugin ID integer. The constructors are used by the factory mechanism to instantiate the plugins as well as providing the plugins with a Communicator object. If ViennaX is compiled with MPI support, the communicator refers to a Boost MPI communicator, otherwise it maps to an integer value, enabling to compile ViennaX on non-MPI targets without any changes.

5.1.3 Sockets

Essential to a scientific plugin framework is the ability to exchange data of arbitrary type and dimension between the different plugins utilized in a simulation. For instance, a scalar field representing the result of a simulation conducted in a plugin might be used as an initial guess for another simulation performed by a subsequent plugin. The sockets have to be first defined in the initialization phase of the plugin, and can then be utilized in the execution part. We use a similar approach for the data communication layer as introduced by the COOLFluiD framework. This section introduces first user-level code, and then further delves into aspects of the respective implementation.

The socket system supports input and output data ports, called sink and source sockets, respectively. In general, the data associated to the sockets can either be already available, thus no copying is required, or it can be generated automatically during the course of the socket creation. The following user-level code snippet creates a source socket, generating the associated data object automatically.

```
create_source<Vector>("x");
```

The data of the socket can be accessed by the following.

```
Vector& x = access_source<Vector>("x");
```

If a data object is already available, the socket can be linked to it.

```
Vector x;
link_source(x, "x");
```

Note that similar implementations for socket creation and access are available for sink sockets.

Figure 7 gives an overview of the socket implementation via a class diagram. In general, the socket hierarchy utilizes a socket ID class and a database class to store the data associated with the sockets (Fig. 8). Sockets can be compared to enable matching validation tests. The remainder of this section discusses the database implementation and the socket class hierarchy.

The DataBase class provides a centralized, generic storage facility for the data associated with the sockets. This storage additionally provides access and lookup mechanisms for retrieving and deleting the data objects of a given socket. The storage internally uses an associative container, mapping a string ID value to a void-pointer, thus being able to hold pointers of arbitrary type. The ID string is generated from the name of the socket and the type string, thus as long as the names are unique, the data can be clearly identified even if the types are the same.

The following code snippet depicts the essential internal parts of the database storage, being adding and retrieving data.

```
1 m_database[name+typeid(Data).name()] = data;
2 Data* data = static_cast<Data*>(
3   m_database[name+typeid(Data).name()]);
```

By generating the key string from the socket name and the type string, a unique lookup can be set up (Line 1). The

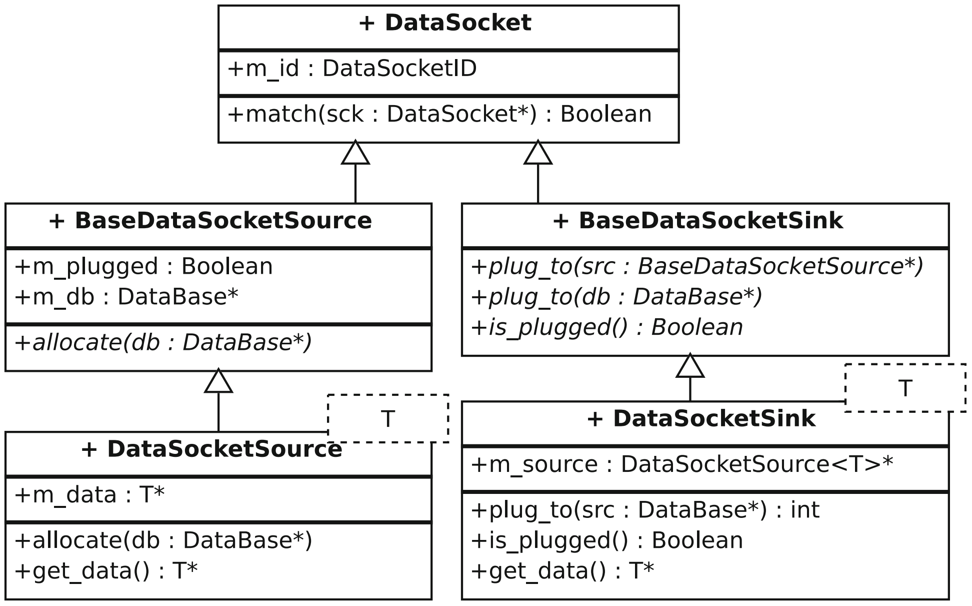


Fig. 7 Class diagram of the socket system

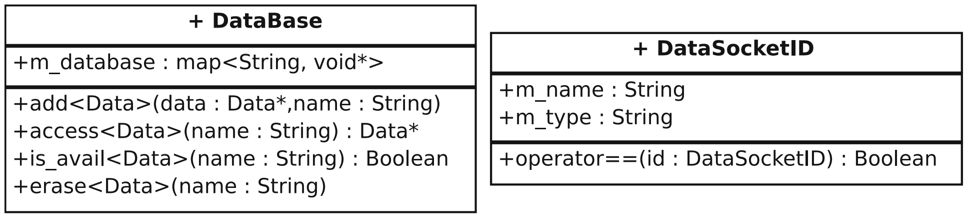

+ DataBase
+m_database : map<String, void*>
+add<Data>(data : Data*,name : String)
+access<Data>(name : String) : Data*
+is_avail<Data>(name : String) : Boolean
+erase<Data>(name : String)
+ DataSocketID
+m_name : String
+m_type : String
+operator==(id : DataSocketID) : Boolean

Fig. 8 Class diagram of the socket database and ID class

data pointer (data) can be accessed again by using the appropriate key information and cast operation (Lines 2, 3). The applied socket data storage approach decouples the actual storage related tasks from the actual socket implementations, thus improving maintainability and extendability as, for instance, possible future extensions to the socket storage layer can be conducted without interfering with the socket implementations.

To enable storing source and sink sockets, and holding data of arbitrary types in a homogeneously typed data structure, a virtual inheritance approach is applied. As such, source and sink sockets are generalized by the BaseDataSocketSource/Sink classes, respectively. The derived, type-aware socket class specializations DataSocketSource/Sink , provide access to the associated data object via the get_data function. In general, a source socket holds the actual data pointer (m_data), whereas the sink socket merely points to the corresponding source socket (m_source). A sink socket has, thus, to be linked to a source socket via the plug_to method, which is explained in the following.

Before working with the socket data, the source sockets have to be allocated and the sink sockets have to be linked to their respective source counterparts. This step is automatically handled by ViennaX by using the allocate and plug_to methods. The allocate function requires a pointer to an already available socket database object, which is then used for the allocation implementation, as depicted in the following.

```
db->add(m_data, m_name);
m_db = db;
```

In Line 1 the data pointer (m_data) is added to the socket database (db), whereas in Line 2 the externally provided database pointer is stored locally for future references.

The socket linking step, required for accessing the data of sink sockets, is implemented by the plug_to method, which prior to updating the internal source socket pointer verifies socket compatibility.

```
if(match(src)) m_source =
  static_cast<DataSocketSourceT*>(src);
```

Therefore, a suitable external source socket has to be provided by the calling instance utilizing the DataSocketID information.

Aside from the exchange of data between the plugins, our data communication layer inherently supports an approach to handle physical units in a straightforward manner. Units are a major concern in scientific computing, as mixing the units between functions, obviously results in a major corruption of the computational result [34]. As such it is of utmost interest to introduce automatic layers of protection to ensure that required data is given in the expected units.

ViennaX tackles this challenge by coupling the unit information to the string-based ID of the sockets. As the automatic socket plugging mechanism of ViennaX requires the sink and source socket to have not only the same type but also the same ID string, a sink and a source socket with different ID will not be connected. The string-based approach allows for coupling arbitrary properties to the sockets, making it a highly versatile system to impose correctness on the plugin data connections. It can also be used to implement an automatic conversion mechanism, enabling a self-acting adjustment of data according to the required physical units. Such a conversion is not yet provided, but scheduled for future work.

5.2 Configuration

ViennaX is based on run-time configuration to allow user-input to actively drive the execution of the task graph and relies on the XML to build a flexible configuration environment. Using an XML-based approach for the configuration of the framework is also important with respect to a future graphical user interface (GUI) extension. A GUI will have to generate well-formed XML data, which is then used to drive the framework execution in ViennaX.

A basic configuration file with the sole purpose of executing a single plugin named ViennaCLLinSol is depicted in the following.

```
1 <plugins>
2    <plugin>
3       <key>ViennaCLLinSol</key>
4       <tol>1.0E-10</tol>
5    </plugin>
6 </plugins>
```

The general plugins region contains the set of all plugins, which should be utilized during the execution (Lines 1–6). Each plugin is defined within its own region (Lines 2–5), which enables to pass parameters to the plugin instance (Fig. 9). The name of the plugin has to be mentioned within the key region (Line 3), and needs to match the name as provided by the static ID method provided by the respective plugin (Sect. 5.1.2).

The presented configuration must be supplemented with additional plugins connected to the two input sockets (system matrix A and the right hand-side x) of the ViennaCLLinSol plugin. Otherwise, the configuration is incomplete and ViennaX will shutdown with a corresponding error message.

Internally, the pugixml[21] library is used, which aside from providing an XML datastructure also supports the XPath query language. This query language enables to conveniently access the parameters forwarded from the input configuration data. For instance, based on the previous XML configuration snippet, a string containing the value of the parameter tol is obtained by the following user-level code.

```
std::string step = query("plugin/tol/");
```

This tolerance parameter (tol) can be forwarded to ViennaCL's linear solver implementation, to set the required accuracy for the iterative convergence process.

The corresponding implementation of this query is essentially based on utilizing the following pugixml-based algorithm.

```
pugi::xpath_node_set query_result =
  xml.select_nodes(xpath_query_str.c_str());
```

The xml object is the data structure holding the entire XML configuration data. xpath_query_str contains the XPath query expression, based on the previous user-level example, the string would contain "plugin/step/".

ViennaX additionally provides numerical conversion routines and mathematical string-expression evaluations based on the Lua[22] library. These features enable to directly utilize the string-based query results in numerical expressions. For instance, the following user-level snippet depicts the evaluation feature.

```
std::string value = "3*5";
double val = eval_mathstr<double>(value);
// val == 15.
```

5.3 Scheduler kernels

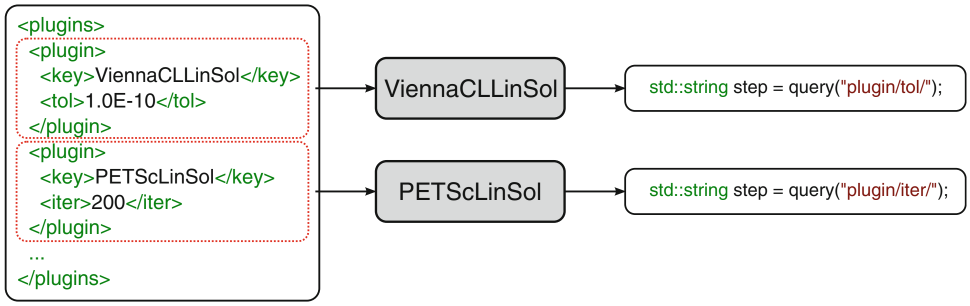
This section discusses the different scheduler kernels provided by ViennaX. The general design as well as implementation background is provided.

5.3.1 Serial mode

The SM kernel is used for serial task graph execution on a shared-memory machine. Although the task graph is processed in a serial manner, the individual plugins can indeed

[21] http://pugixml.org/.

[22] http://www.lua.org/.



```
<plugins>
  <plugin>
    <key>ViennaCLLinSol</key>
    <tol>1.0E-10</tol>
  </plugin>
  <plugin>
    <key>PETScLinSol</key>
    <iter>200</iter>
  </plugin>
  ...
</plugins>
```

```
std::string step = query("plugin/tol/");
std::string step = query("plugin/iter/");
```

Fig. 9 Each plugin possesses its own configuration region within the input configuration file. This data can be accessed from within the plugin by querying the configuration object

utilize shared-memory parallelization approaches, such as OpenMP. The task graph implementation is based on the Boost Graph Library[23], which not only provides the data structure but also graph algorithms, such as topological sort [33].

The serial scheduler is based on the list scheduling technique [25]. Informally, this technique uses a prioritized sequence of tasks, which is then processed consecutively. Figure 10 depicts the major steps of execution flow.

The plugins are loaded according to the input configuration file (Sect. 5.2) by the factory mechanism (Sect. 5.1.1). Each plugin is configured based on the parameters listed in the input file. Based on these parameters the input and output dependencies are defined. For instance, the following code snippet depicts the implementation of the plugin configuration as well as the allocation of the plugin sockets.

```
1 boost::shared_ptr<DataBase> db(new DataBase);
2 std::size_t pID = 0;
3 for(PluginIter piter = plugins.begin();
4     piter != plugins.end(); piter++)
5 {
6   (*piter)->load(pID, configurations[pID]);
7   (*piter)->init();
8   Sources& sources = (*piter)->provides();
9   for(SourcesIter srciter = sources.begin();
10      srciter != sources.end(); srciter++)
11    get_socket(srciter)->allocate(db);
12  pID++;
13 }
```

A smart pointer is used to create the database object (Sect. 5.1.3), ensuring proper memory handling (Line 1). The set of plugins is traversed and preinitialized with the configuration data and the unique plugin integer, as discussed in Sect. 5.1.2 (Lines 3–6). During the plugin initialization step, the sockets are created in the individual plugins (Line 7). If source sockets have been created, the associated data has to be instantiated by allocating the data in the socket database (Line 8–11).

[23] http://www.boost.org/libs/graph/.

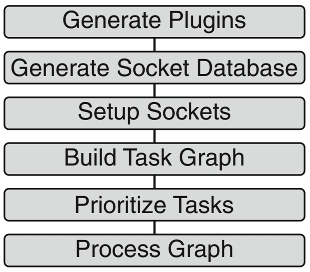


Fig. 10 Flow diagram of the SM scheduler kernel

A task graph meshing algorithm connects the various plugins based on their dependencies. The meshing procedure is based on plugging the sink sockets of the plugins to valid source sockets of other plugins (Sect. 5.1.3). Validity is ensured by comparing the socket IDs. The generated task graph is used for building the prioritized sequence, generated by the Boost Graph's topological sort graph algorithm, as depicted in the following.

```
std::list<Vertex>   plist;
boost::topological_sort(graph,
  std::front_inserter(plist));
```

The graph object is a directed graph datastructure, which is used to hold the entire task graph. The prioritized sequence of tasks is processed consecutively by traversing the result container plist and executing the individual plugins via the plugin's execute interface method (Sect. 5.1.2).

Note that the linear solver plugin introduced in Sect. 4 can be utilized with this scheduler.

5.3.2 Distributed task parallel mode

The DTPM scheduler kernel enables applications focusing on a task parallel approach. In general, the scheduler follows a static scheduling approach, based on load balancing indicated by optional plugin weights. Similar workload distribution approaches are available, focusing on dynamic scheduling implementations based on, for instance, work-stealing [1, 14]. The execution of the individual plugins is distributed among the available MPI processes. Therefore, considerable speedup of the task execution can be achieved, if the task graph offers parallel paths. Figure 11a depicts the flow diagram of the scheduler.

The DTPM scheduler has two peculiarities: First, the global task graph is partitioned and ultimately the individual subgraphs are processed by different MPI processes. Second, as the plugins sharing a data connection might be executed on different MPI processes, an extension to the socket data communication layer incorporating the distributed memory environment is required.

The distribution of the workload is based on the METIS[24] graph partitioning library.

A weighting approach is implemented enabling the user to assign a weight to the plugin implementation indicating the computational load of the respective plugin. This load is used by METIS, aiming to equalize the computational effort over the generated partitions. The following code snippet depicts the user-level expression which sets the computational load of a plugin. This statement has to be placed inside the plugin's initialization method (Sect. 5.1.2).

```
computation_level() = 10;
```

Internally, ViennaX converts the graph datastructure to a compressed sparse row format, which is required by the METIS API. In the following, the utilization of the METIS partitioner is shown. For the sake of simplicity, only the most important parameters are depicted.

```
1 METIS_PartGraphKway(
2   &n,               // Number of vertices
3   &(xadj[0]),       // CSR indices
4   &(adjncy[0]),     // CSR adjacencies
5   &(weights[0]),    // Vertex weights
6   &nparts,          // Number of partitions
7   &(part[0])        // Partitions
8 );
```

The total number of graph vertices is provided (Line 1). The graph is passed to METIS in the CSR format (Lines 2, 3). The vertex weights and the number of desired partitions, which equals the number of available MPI processes, are forwarded (Lines 5, 6). Upon completion, the result container maps each vertex to a specific partition (Line 7).

The second peculiarity of the DTPM scheduler, being the incorporation of a distributed memory environment into the socket data communication layer, is based primarily on the non-blocking point-to-point communication capabilities of the MPI layer. The graph partitioning step yields, aside from the MPI process assignments of the plugins, a

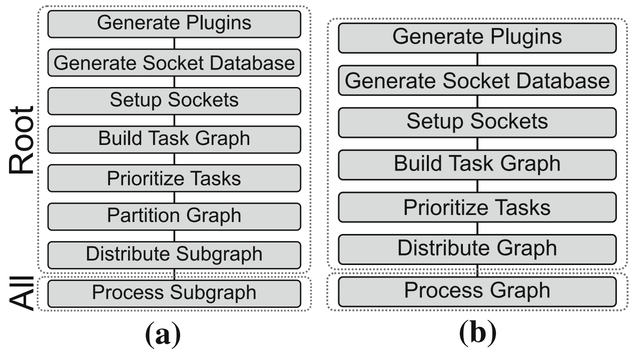


Fig. 11 **a** Flow diagram of the DTPM scheduler kernel. The root MPI process is responsible for preparing and distributing the workload evenly between the compute units. All available compute units process their distinct parts of the graph. **b** Flow diagram of the DDPM scheduler kernel. Similar to the DTPM scheduler, the root MPI process prepares the entire task graph. However, the entire workload is distributed to all MPI processes, as each process executes the entire task set represented in the task graph

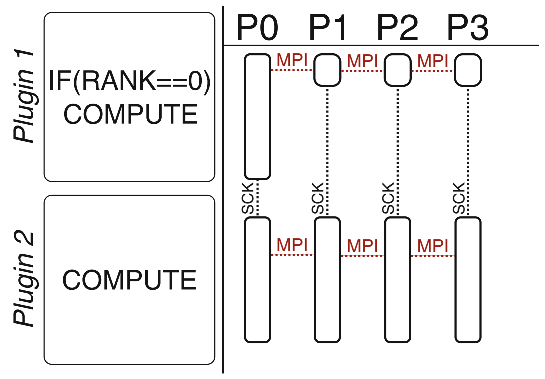


Fig. 12 Exemplary execution behavior of the DDPM scheduler based on two plugins and four MPI processes. The *bars* in the *right part* of the figure indicate computational load. Each MPI process executes the individual plugin. Additionally, each plugin has access to an MPI communicator object, enabling not only classical data parallel execution modes but also plugin inter-process communication. Inter-plugin communication is realized by the socket mechanism (SCK)

lookup table for the socket communication. Each MPI process holds its own socket database, and utilizes the communication lookup table to determine the corresponding transmission sources and sinks. For instance, after a plugin has been executed on an MPI process, its source sockets requiring outbound inter-process communication are traversed and the transmission is initiated.

In general, we utilize the non-blocking point-to-point methods to increase execution performance. This is crucial, as, for instance, in the optimal case, an MPI process should not wait for an outgoing transmission to be finished before it executes another plugin. Such an approach is typically referred to as overlapping communication with computation. However, using a pure MPI approach and therefore

[24] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

non-blocking communication methods, such an overlap is rarely achieved. In fact, specialized hardware and software is required to achieve a reasonable overlap, for instance, Cray's XE6 with Gemini interconnects is capable of delivering such an overlap [20]. A possible future extension would be a hybrid approach, utilizing MPI and Threads to implement a true asynchronous approach, thus introducing a much more improved overlap of communication and computation.

The linear solver plugin introduced in Sect. 4 can be utilized with this scheduler, as each plugin is executed by one process. Therefore, one MPI process does access the available computational resources via the parallel accelerator layer.

5.3.3 Distributed data parallel mode

The DDPM scheduler kernel enables simulations based on the data parallel approach. Figure 11b depicts the flow diagram of the scheduler implementation. Contrary to the DTPM scheduler, the graph is not partitioned as all plugins are processed by all MPI processes in the same sequence. The root process prepares the task graph and generates a prioritized list of plugins. This list is distributed to all MPI processes each processing the graph in its entirety. As with the DTPM scheduler, each MPI process holds its own socket database responsible to store the data associated with the sockets on the local process.

A peculiarity of the DDPM scheduler kernel is the fact that each plugin has access to an MPI communicator object via the `comm` method, providing access to the entirety of the MPI environment. The following code snippet depicts an exemplary utilization in a plugin's implementation to evaluate the rank of the current MPI process.

```
if (comm().rank() == 0) {
  // Root code
}
```

A Boost MPI communicator object offers implicit conversion to a raw MPI communicator, ensuring interoperability with non-Boost MPI implementations.

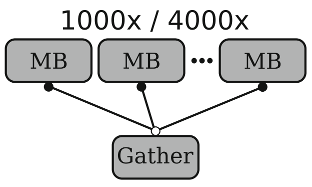


Fig. 13 The graph for the Mandelbrot benchmark. 1,000 and 4,000 instances of the Mandelbrot plugin (MB) have been used. The partial results are gathered, which forces MPI communication

Figure 12 shows the execution behavior of the scheduler. Each plugin is processed by all MPI processes and has access to an MPI communicator. Inter-plugin communication is provided by the socket data layer, whereas interprocess communication is supported by the MPI library. A similar parallel communication model has already been applied by the CCA [8].

The current approach of providing the plugins with an MPI communicator offers access to all MPI processes which enables single instruction multiple data implementations (SIMD), but prohibits multiple instruction multiple data (MIMD) execution. Extensions could implement a grouping of MPI processes into new split MPI communicator objects, thus supporting MIMD application scenarios.

It should be noted that the utilization of the linear solver plugin introduced in Sect. 4 is not reasonable here, as in this case each process would perform the computation, thus massively overburdening the compute unit beyond reasoning. For the scheduler at hand, an MPI-powered linear solver implementation is the proper choice, as is provided by, for instance, the PETSc library.

6 Applications

This section presents application results with a focus on distributed parallel execution scenarios. Therefore, results for the DTPM and the DDPM scheduler are depicted. The target system for the benchmarks is HECToR[25], a Cray XE6 supercomputer with a Gemini interconnect.

6.1 Distributed task parallel mode

This section investigates the scalability of the DTPM scheduler (Sect. 5.3.2) by a Mandelbrot benchmark as implemented by the FastFlow framework. Implementation details of the benchmark are provided as well as performance results.

6.1.1 The benchmark

The DTPM scheduler is used to compute the Mandelbrot set for parts of the application domain by using different instances of a Mandelbrot plugin. The partial results are gathered at the end of the simulation, which provides insight into the communication overhead (Fig. 13). The implementation of the Mandelbrot plugin performs a partitioning of the simulation domain. Thus, each instance is responsible for a subdomain, which is identified by the plugin index. In its essence, this approach follows a data parallel approach, which also shows that the DTPM can be

[25] http://www.hector.ac.uk/.

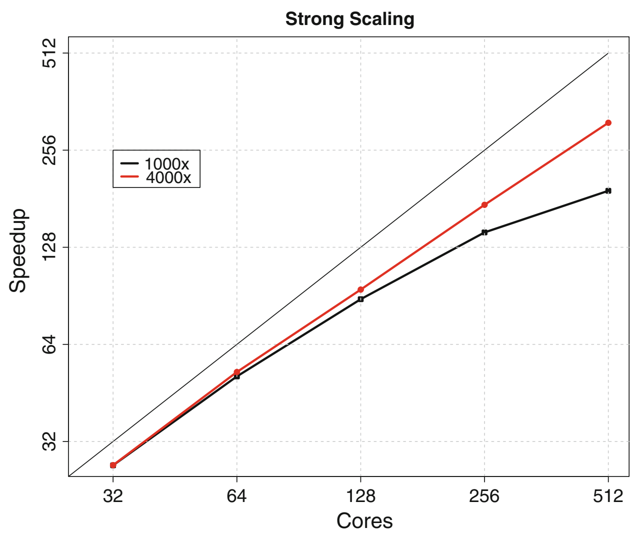


Fig. 14 The strong scaling behavior of the Mandelbrot benchmark. Increasing the computational load on the plugins shifts the scaling saturation towards higher core numbers. The smaller problem scales well for up to 256 cores, however, for higher core numbers the communication overhead outpaces the computational load on the plugins. Both benchmarks depict a non-optimal speedup for small core numbers, which is due to communication overhead and insufficient load balancing

used for such kind of tasks, despite of its inherent focus on task parallelism.

Two different simulation domains have been investigated, being a grid of 1,000 × 1,000 and 4,000 × 4,000 to demonstrate the influence of varying computational load on the scaling behavior. Each plugin processes one line of the grid, consequently 1,000 instances for the smaller and 4,000 instances for the larger case have been used.

The following depicts the configuration file for the smaller benchmark.

```
<plugins>
  <plugin>
    <key>Mandelbrot</key>
    <clones>1000</clones>
    <dim>1000</dim>
    <niters>10240000</niters>
  </plugin>
  <plugin>
    <key>MandelbrotGather</key>
  </plugin>
</plugins>
```

The clones entry triggers an automatic duplication of the plugin in ViennaX, thus in this case 1,000 instances of the Mandelbrot plugin are generated. The parameters dim and niters are forwarded to the plugins and relate to the number of grid points in one dimension and the number of iterations used in the Mandelbrot algorithm, respectively. The configuration of the larger benchmark is similar, however, the number of clones and dimensions is increased to 4,000, respectively.

During the Mandelbrot plugin's initialization phase, the output socket has to be generated as well as the computational level of the plugin has to be set to balance the computational load over the computational resources. A peculiarity of computing the Mandelbrot set is the fact that the computational load in the center is larger than on the boundary. Therefore, an exemplary weighting scheme has been applied to increase the load balancing over the MPI processes. The plugin instances in the center are assigned a higher computational weight (Sect. 5.3.2) as depicted in the following.

```
1 void init() {
2   double current = pid();
3   if( (current >= dim*0.45) &&
4       (current <= dim*0.55) )
5     computation_level() = 100;
6   // ..
7 }
```

In Line 1 the current plugin ID is extracted, which has been provided by the scheduler during the preparation phase (Sect. 5.1.2). As each plugin is responsible for a single line of the simulation grid, its ID is tested whether it is responsible for the center part. If so, the computational level is set to a high value, which is forwarded to the graph partitioner.

As each plugin computes a subset of the simulation result, the output port has to be localized with respect to the plugin instance to generate a unique socket. As already stated, the socket setup has to be placed in the initialization step.

```
create_source_local<Vector>("vector");
```

This socket generation method automatically generates sockets of the type Vector and the name vector including an attached string representing the plugin ID, retrieved from the pid method. This approach ensures the generation of a unique source socket for each Mandelbrot plugin instance.

The execution part accesses the data associated with the source socket and uses the datastructure to store the computational result. The plugin ID is used as an offset indicating the responsible matrix line which should be processed.

```
bool execute(std::size_t call) {
  Vector & vector =
    access_source_local<Vector>("vector");
  i = pid();
  for (j = 0; j < dim; j++) {
    // compute k as a function of i and j
    vector[j] = k;
  }}
```

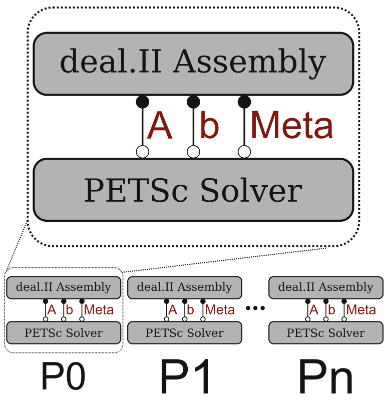


Fig. 15 The graph of the deal.II benchmark, which is executed by all MPI processes (P0-Pn). The system matrix (*A*) and the *right-hand side* (*b*) as well as meta information (*meta*) is forwarded to the solver plugin via the socket communication layer

On the receiving side, the Gather plugin's initialization method generates the required sink sockets using a convenience function.

```
void init() {
  create_sink_set<Vector>("vector");
}
```

The above method automatically generates one sink socket for $0\ldots n-1$ predecessor plugins, where n represents the current plugin ID. Therefore, all previous output ports generated by the Mandelbrot plugins can be plugged to the respective sink sockets of the Gather plugin.

The execution phase is dominated by a gather method, similar to the MPI counterpart.

```
bool execute(std::size_t call) {
  std::vector<Vector> matrix;
  gather<Vector>("matrix", matrix);
}
```

The partial results of the $0\ldots n-1$ predecessor plugins are stored consecutively in a vector container, thus forming the result matrix where each entry corresponds to a point on the two-dimensional simulation grid.

6.1.2 Results

Figure 14 depicts the strong scaling results, i.e., a fixed problem size is investigated for different core numbers.

Reasonable scaling is achieved, although communication overhead and load balancing problems are identifiable already for small core numbers. However, increasing the computational load on each plugin and the number of plugins to be processed by the MPI processes, further shifts the scaling saturation towards higher core numbers.

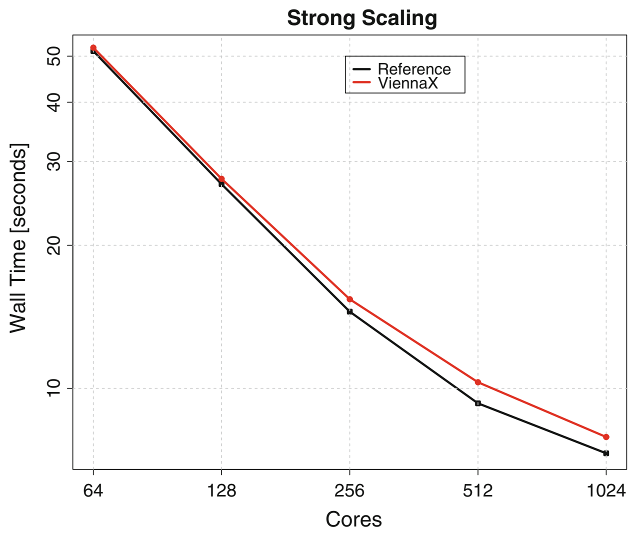


Fig. 16 The execution performance of the Laplace benchmark is compared to the reference implementation provided by the deal.II library. ViennaX is approximately 1 s slower than the reference implementation throughout the core spectrum

For the smaller problem an efficiency of 38 % and for the larger problem an efficiency of 60 % for 512 cores is achieved. Improving the load balancing via the plugin weighting approach as well as introducing a hybrid scheduler to improve communication and computation overhead will further improve the scaling behavior.

6.2 Distributed data parallel mode

The DDPM scheduler (Sect. 5.3.3) is investigated by comparing the execution performance to a reference implementation provided by the deal.II[26] library [6]. This benchmark not only shows that an available implementation using external high-performance libraries can be transferred to the ViennaX framework in a straightforward manner but also that the execution penalty of using the framework is negligible.

6.2.1 The benchmark

A large-scale 2D Laplace test case of the deal.II library offering 67 million degrees of freedom is considered, which utilizes components representing important aspects of large-scale high performance computing applications [5]. For instance, the datastructure holding the mesh and the linear system is fully distributed by using datastructures provided by the PETSc library and the p4est library. The equations are discretized using biquadratic finite elements and solved using the conjugate gradient method

[26] http://www.dealii.org/.

preconditioned by an algebraic multigrid method provided by the Hypre[27] package and accessed via PETSc.

The reference implementation is split into two functional parts, being the assembly by the deal.II library and the solution of the linear system via the PETSc library. Therefore, two plugins have been implemented which also underlines the reusability feature (Fig. 15). For instance, the linear solver plugin can be replaced with a different solver without changing the implementations.

The following depicts the configuration file for the benchmark.

```
<plugins>
   <plugin>
      <key>deal.II.Assemble</key>
   </plugin>
   <plugin>
      <key>PETSc.Solve</key>
   </plugin>
</plugins>
```

For the sake of simplicity, no parameters are provided to the plugins. In production code, however, additional parameters would be made accessible via sinks.

The assembly plugin's initialization phase prepares the distributed datastructures as well as provides the source sockets. However, as the PETSc and deal.II libraries require a call to their own initialization methods which are required for the MPI mode, a pre-initialization step is required. Therefore, ViennaX supports passing a functor to the plugin's constructor via an additional convenience macro, which is executed before the plugin-instance's creation. This step is vital as these initialization methods have to be called before the library specific MPI datastructures are instantiated. The following code snippet is placed inside the plugin's body to trigger an correct initialization order. Although ViennaX already initialized the MPI environment, the call to PetscInitialize is required.

```
struct PreInit {
  void operator()() {
    PetscInitialize(0,0,0,0);
    dealii::deallog.depth_console (0);
}};
INIT_VIENNAX_PLUGIN_PREINIT(PreInit())
```

The datastructures required for the computation are generated and kept in the plugin's state. Therefore, the source sockets are linked to the already available objects, instead of created from scratch during the socket creation. The execute method of the assembler plugin forwards the simulation datastructures to the implementation provided by the deal.II example, which performs the actual distributed assembly. This fact underlines the straightforward utilization of already available implementations by the ViennaX plugins. The following code snippet gives a basic overview of the assembly plugin's implementation.

```
1 // Plugin state
2 Matrix     matrix;
3 Vector     rhs;
4 Mesh       triangulation;
5 Simulation sim;
6
7 void init() {
8   link_source(matrix, "matrix");
9   link_source(rhs,    "rhs");
10  // ..
11 }
12
13 bool execute(std::size_t call) {
14   GridGenerator::hyper_cube (triangulation);
15   triangulation.refine_global (10);
16   sim.assemble_system ();
17 }
```

The plugin state holds various data objects (Lines 2–5). The sim objects hold references of the other objects, thus the internals of the simulation class can access the datastructures. The source sockets link to already available data objects (Lines 6, 7). The simulation grid is generated (Lines 14, 15) and the linear system (matrix, rhs) is assembled (Lines 16).

The solver plugin generates the corresponding sink sockets in the initialization method and utilizes the PETSc solver environment in the execution method. As each MPI process holds its own instance of the socket database, the pointers of the distributed datastructures are forwarded from the assembler to the solver plugin. The PETSc internals are, therefore, able to work transparently with the distributed datastructures without the need for additional copying operations. The following depicts the crucial parts of the solver plugin's implementation.

```
void init() {
  create_sink<Matrix>   ("matrix");
  create_sink<Vector>   ("rhs");
}

bool execute(std::size_t call) {
  Matrix &matrix=access_sink<Matrix>("matrix");
  Vector &rhs   =access_sink<Vector>("rhs");
  // compute solution vector x
}
```

6.2.2 Results

Figure 16 compares the execution performance of the ViennaX implementation with the deal.II reference

[27] http://acts.nersc.gov/hypre/.

implementation. A system of 67 million degrees of freedom is investigated, which due to it's memory requirements does not fit on one compute node. Generally, excellent performance is achieved, however, an overall constant performance hit of about 1 s for all core numbers is identified. This performance hit is due to the run-time overhead introduced by the plugin framework, such as the time required to load the plugins, generate the task graph, and perform virtual function calls. The relative difference for 1,024 cores is around 8 %, however, it is reduced to 1.5 % for 64 cores, underlying the fact that the framework's overhead becomes more and more negligible for larger run-times. In our opinion this overhead is acceptable, as the simulations we are aiming for have run-times way beyond 50 s, as is the case for 64 cores.

Although the relative difference is significant for short run-times, a delay of 1 s hardly matters in real world, day-to-day applications. On the other hand, accepting this performance hit introduces a significant increase in flexibility to the simulation setup due to the increased reusability of ViennaX's component approach.

7 Possible future extensions

Aside from the features that ViennaX currently provides, the following limitations and future extensions have been identified. The communication and computation overlap of the DTPM scheduler is improvable by extending the current pure-MPI implementation by a hybrid MPI/Threads approach. The DDPM scheduler currently only supports SIMD parallelism models. To also support MIMD approaches, the scheduler needs to be extended by essentially creating new communicator objects, being responsible for subgroups of MPI processes. An automatic conversion mechanism for the socket communication layer could be implemented, enabling coupling of data sockets offering different but convertible physical units. The current implementation of ViennaX solely focuses on the distributed parallelization. Although this implementation can be used on shared-memory multi-core targets, it does likely not optimally utilize the system resources with respect to memory allocation. Therefore, future extensions should incorporate shared-memory parallelization models for the scheduler kernels.

8 Conclusion

We presented the free open source, parallel plugin execution framework ViennaX. Details about the library as well as core features have been discussed. Application examples have been introduced and performance results have been shown. Our framework enables the introduction of a high level of flexibility for scientific simulations using a plugin approach. Parallel execution allows for the execution of large work packages on distributed computing nodes, thus tremendously improving overall performance.

Acknowledgments This work has been supported by the European Research Council (ERC) through the grant #247056 MOSILSPIN. Additionally, this work has been partially supported by the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission—Capacities Area—Research Infrastructures. This work made use of the facilities of HECToR, the UK's national high-performance computing service, which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme.

References

1. Agrawal K, Leiserson CE et al (2010) Executing task graphs using work-stealing. In: Proceedings of the 24th IEEE international symposium on parallel and distributed processing (IPDPS), pp 1–12
2. Aldinucci M, Danelutto M et al (2013) FastFlow: high-level and efficient streaming on multi-core. In: Programming multi-core and many-core computing systems, parallel and distributed computing, Wiley
3. Allan BA, Armstrong RC et al (2002) The CCA core specification in a distributed memory SPMD framework. Concurr Comput 14(5):323–345
4. Armstrong R, Gannon D et al (1999) Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE international symposium on high performance distributed computing (HPDC)
5. Bangerth W, Burstedde C et al (2011) Algorithms and data structures for massively parallel generic adaptive finite element codes. ACM Trans Math Softw 38(2):14:1–14:28
6. Bangerth W, Hartmann R et al (2007) deal.II–a general-purpose object-oriented finite element library. ACM Trans Math Softw 33(4):24:1–24:27
7. Bassetti F, Brown D et al (1998) Overture: an object-oriented framework for high performance scientific computing. In: Proceedings of the 1998 ACM/IEEE conference on supercomputing, pp 1–9
8. Bernholdt DE, Allan BA et al (2006) A component architecture for high-performance scientific computing. Int J High Perform Comput Appl 20(2):163–202
9. Berzins M (2012) Status of release of the Uintah computational framework. Technical Report, Scientific Computing and Imaging Institute, University of Utah
10. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54(5):67–77
11. Bosilca G, Bouteiller A et al (2012) DAGuE: A generic distributed DAG engine for high performance computing. Parallel Comput 38(1–2):37–51
12. Buis S, Piacentini A et al (2006) Palm: a computational framework for assembling high-performance computing applications. Concurr Comput 18(2):231–245
13. de St. Germain D, McCorquodale J et al (2000) Uintah: a massively parallel problem solving environment. In: Proceedings of the 9th IEEE international symposium on high performance distributed computing (HPDC)

14. Dinan J, Larkins DB et al (2009) Scalable work stealing. In: Proceedings of the conference on high performance computing networking, storage and analysis (SC), pp 53:1–53:11
15. Dios A, Asenjo R et al (2011) High-level template for the task-based parallel wavefront pattern. In: Proceedings of the 18th international conference on high performance computing (HiPC), pp 1–10
16. Dongarra J, van der Steen A (2012) High-performance computing systems. Acta Numerica 21:379–474
17. Dos Reis G, Järvi J (2005) What is generic programming? In: Proceedings of the 1st international workshop on library-centric software design (LCSD)
18. Goodale T, Allen G et al (2003) The Cactus framework and toolkit. In: High performance computing for computational science–VECPAR 2002, Lecture Notes in Computer Science, vol 2565, pp 197–227
19. Govindaraju M, Head MR et al (2005) XCAT-C++: design and performance of a distributed cca framework. In: Proceedings of the 12th annual IEEE international conference on high performance computing (HiPC), pp 18–21
20. Hager G, Wellein G (2010) Introduction to high performance computing for scientists and engineers, CRC Press, Boca Raton, FL
21. Hill C, DeLuca C et al (2004) The architecture of the earth system modeling framework. Comput Sci Eng 6(1):18–28
22. Jasak H, Jemcov A et al (2007) OpenFOAM: a C++ library for complex physics simulations. In: Proceedings of the international workshop on coupled methods in numerical dynamics
23. Keyes D, McInnes LC et al (2011) Multiphysics simulation. Technical Report ANL/MCS-TM-321, Argonne National Laboratory
24. Kharrat D, Quadri S (2005) Self-registering plug-ins. In: Proceedings of the 18th Canadian conference on electrical and computer engineering (CCECE), pp 1324 –1327
25. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput Surv 31(4)
26. Lewis MJ, Ferrari AJ et al (2003) Support for extensibility and site autonomy in the Legion grid system object model. J Parallel Distrib Comput 63(5):525–538
27. Logg A, Mardal KA et al (2012) Automated solution of differential equations by the finite element method, Lecture Notes in Computational Science and Engineering, Springer
28. Magoules F (2008) Mesh partitioning techniques and domain decomposition methods, Saxe-Coburg Publications, UK
29. Miczo A (2003) Digital logic testing and simulation, Wiley, New York
30. Miller A (2010) The task graph pattern. In: Proceedings of the 2nd workshop on parallel programming patterns (ParaPLoP), pp 8:1–8:7
31. Philip B, Clarno KT et al (2010) Software design document for the AMP nuclear fuel performance code. Technical Report, Oak Ridge National Laboratory
32. Quintino T (2008) A component environment for high-performance scientific computing, PhD thesis, Katholieke Universiteit Leuven
33. Siek JG, Lee LQ et al (2001) The Boost Graph Library. Addison-Wesley Professional, Reading, MA
34. Stroustrup B (2012) Software development for infrastructure. Computer 45(1):47–58
35. Weinbub J, Rupp K et al (2012a) A flexible execution framework for high-performance TCAD applications. In: Proceedings of the 17th international conference on simulation of semiconductor processes and devices (SISPAD)
36. Weinbub J, Rupp K et al (2012b) Towards a free open source process and device simulation framework. In: Proceedings of the 15th international workshop on computational electronics (IWCE)
37. Wesseling P (2001) Principles of computational fluid dynamics, Springer, Berlin
38. Zhang K, Damevski K et al (2004) 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), pp 72–79