ViennaX

User Manual
v1.2.0

Institute for Microelectronics
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria

Main author(s):

Josef Weinbub

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

Phone	+43-1-58801-36001
FAX	+43-1-58801-36099
Web	`http://www.iue.tuwien.ac.at`

Contents

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 [1] and is common practice in fields like computational fluid dynamics (CFD) [2]. 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 framework [3]. 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 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 [4], 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 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 [5]. In this 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 [6][7][8]. On the contrary, task parallel approaches can be used to parallelize data flow applications, for instance, wave front [9] or digital logic simulations [10].

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.

Chapter 1 provides a quick start tutorial, information about dependencies, and building the examples, as well as building and executing the tests. Chapter 2 describes the parts of ViennaX, which are most relevant to the user. The individual components are introduced with code examples.

Chapter 1

Building Instructions

This chapter depicts how ViennaX can be utilized in a project as well as how examples and tests are built and executed. If you experience any troubles, please let us know in the online forum:

`http://sourceforge.net/p/viennax/discussion/`

In general, ViennaX is a header only library, and as such does not require building. However, the examples and tests have to be built, and for this purpose a CMake environment has been provided.

This section first discusses the overall dependencies, like external libraries, and later on depicts how the examples and tests can be built and executed.

1.1 Quick Start

For running a quick test whether ViennaX builds and executes fine on your system, execute the build script in the root folder, like depicted in the following.

```
$> cd /your-ViennaX-path/
$> ./build.sh
```

This script automatically detects the number of available CPU cores and builds the examples in parallel.

The following executes an example with the serial mode (SM) scheduler (Section 2.2.1).

```
$> cd build/examples/
$> ./vxsm ../../examples/inputs/basic.xml plugins-serial/
```

The following executes an example with the MPI-based distributed task parallel mode (DTPM) scheduler (Section 2.2.2).

```
$> cd build/examples/
$> mpirun -np 2 ./vxdtpm ../../examples/inputs/basic.xml plugins-mpi/
```

This example launches 2 MPI processes, thus two ViennaX plugins can be executed in parallel.

If there is no valid MPI environment detected by CMake, the MPI schedulers are not generated. Investigate the configuration phase of CMake to identify possible problems.

See Section 1.3 for further information on building and executing the examples.

1.2 Dependencies

ViennaX has been tested on Linux 64 Bit (Funtoo Linux: Kernels 2.6.39, 3.2.1; CentOS: Kernel 2.6.18) and Mac OS X 64 Bit (10.7.4). In general, ViennaX has the following dependencies:

- C/C++(/Fortran) compiler, i.e., GNU GCC 4.1.2 or above (Section 1.2.1)
- The Boost Libraries 1.44 or above (Section 1.2.2)
- MPI implementation, i.e., Open MPI 1.3.3 or above (optional) (Section 1.2.3)
- CMake 2.8 or above (optional) (Section 1.2.4)
- Extrae 2.2.1 or above (optional) (Section 1.2.5)
- METIS 5.0 (shipped) (Section 1.2.6)
- Lua 5.1.5 (shipped) (Section **??**)
- ViennaUtils (shipped) (Section 1.2.7)

The *optional* tag indicates, aside of not being a prerequisite, that the dependency is an external dependency, thus has to be provided by the user. The *shipped* keyword relates to the fact, that the dependency is provided by the ViennaX package, and is therefore no concern to the user.

1.2.1 C/C++(/Fortran) Compiler

A C/C++ compiler is required to utilize ViennaX. We successfully tested the following compilers:

- GNU GCC 4.1.2 or above [11]
- Clang 3.1 [12]

ViennaX does not require a Fortran compiler to be present, however, in case plugins utilize Fortran code, a suitable compiler has to be provided.

Utilize non-default compilers by using the FC, CC, and CXX environment variable to set the Fortran, C, and C++ compiler, respectively, before configuring with CMake.

If you experience a run-time exception with the error `locale::facet::_S_create_c_locale name not valid`, try setting the `LC_ALL` environment variable to `C`.

1.2.2 The Boost Libraries

ViennaX heavily utilizes already available functionality to reduce the overall code base. Version 1.44 or above of the Boost Libraries is required. If the Boost Library is not available in the system path, or a different location should be used, the CMake flags `BOOST_INCLUDEDIR` and `BOOST_LIBRARYDIR` can be used. For an overview of the possible and most common CMake parameters for ViennaX have a look at the `config.sh` script in the root folder. ViennaX requires the following compiled boost libraries:

- System
- Filesystem
- Serialization *
- MPI *

Note that libraries tagged with * are only required, if the MPI-based schedulers are enabled (Section 1.3.3).

1.2.3 MPI Implementation

ViennaX provides MPI-based schedulers, which require an operational MPI environment. The implementation of the framework is based on the Boost MPI Library [13], which is known to work best with Open MPI [14]. We tested successfully against Version 1.3.3, 1.4.3, and 1.6 of Open MPI. However, the Boost MPI Library additionally supports other MPI implementations, like MPICH2 [15] or LAM/MPI [16].

1.2.4 CMake

CMake is used as the build system of choice and as such we provide a configuration file to build the examples and tests. ViennaX requires a CMake Version 2.8 or above.

1.2.5 Extrae

Beginning from Version 1.1.1 ViennaX ships with scripts and configuration files to support trace-based performance analysis with Paraver [17]. Paraver is the visualization front-end the actual trace data generated by the Extrae tool [17]. For details on using Paraver/Extrae on ViennaX refer to Section 2.5.6.

1.2.6 METIS

Beginning from Version 1.1.0 ViennaX supports the revised MPI-based scheduler DTPM (Section 2.2.2), which utilizes the graph partitioning library METIS [18][19]. For convenience, the library is shipped with ViennaX and built automatically.

Although METIS is shipped with the ViennaX package, METIS does have its own license, located in `external/METIS/LICENSE.txt`. Obviously, utilizing ViennaX simultaneously with METIS requires fulfilling the license imposed by METIS.

1.2.7 ViennaUtils

ViennaUtils is a C++ library, providing auxiliary functionality, like timers. ViennaX utilizes plenty of the available mechanisms for various parts. An implementation of ViennaUtils is shipped with ViennaX, and therefore by default no external dependency has to be fulfilled. The library path is: `external/ViennaUtils`.

1.3 Examples

ViennaX provides serial and parallel schedulers, where the latter are based on the MPI. An example application for each of the scheduler approaches (Table1.1) is provided and can be executed with different configurations (Table1.2). These implementations represent examples, and as such only depict the utilization of the ViennaX framework.

The build process for Linux (Section 1.3.1) and Mac OS X (Section 1.3.2) is described in the following sub-sections.

File	Purpose
`sm.cpp`	Demonstrates the serial mode (SM) scheduler
`dtpm.cpp`	Demonstrates the distributed task parallel mode scheduler (DTPM)
`ddpm.cpp`	Demonstrates the distributed data parallel mode scheduler (DDPM)
`app.cpp`	Demonstrates a generic application implementation, providing a various scheduler kernels via a single application

Table 1.1: Overview of the ViennaX application examples, path: `examples/src/`.

File	Purpose
app.xml	Demonstrates the utilization and control of the generic application.
basic.xml	Demonstrates a basic dependency task graph.
bigdata.xml	Demonstrates a basic dependency task graph where the nodes exchange significant amounts of data.
clone.xml	Demonstrates the clone functionality to conveniently generate multiple instances of a single plugin.
container.xml	Demonstrates how to pass an STL container datastructure between plugins.
deal.II.step40.xml	Demonstrates an exemplary implementation based on the deal.II library.
err_one_sink_two_sources.xml	Demonstrates that connecting one sink socket with more than one source socket yields an ViennaX error.
err_unplugged_sink.xml	Demonstrates that an unplugged sink socket yields an ViennaX error.
fortran.xml	Demonstrates the utilization of Fortran code in the plugins.
jacobi.xml	Demonstrates the loop mechanism for the Jacobi method.
loop.xml	Demonstrates the ability for loops in task graphs.
mandelbrot.xml	Demonstrates an example based on the Mandelbrot set for the DTPM scheduler.
mass_sockets.xml	Demonstrates that a plugin can have a diverse field of sockets.
mpi_plugins.xml	Demonstrates the utilization of the MPI communicator in the plugins used for the DDPM scheduler.
no_sockets.xml	Demonstrates that a plugin with not input and/or output dependencies can also be utilized.
non-contiguous.xml	Demonstrates the ability of the DTPM's partitioner to handle non-contiguous partitions.
one_to_one.xml	Demonstrates a single connection between two plugins.
one_to_two.xml	Demonstrates a plugin providing individual data sockets for two plugins.
one_to_two_share.xml	Demonstrates a plugin providing a shared data socket for two plugins.
parallel.xml	Demonstrates a simple parallel execution.
two_to_one.xml	Demonstrates two plugins reducing to one plugin.
user_defined.xml	Demonstrates how to pass a user-defined datastructure between plugins.
warn_unplugged_source.xml	Demonstrates that an unplugged source socket yields a ViennaX warning.

Table 1.2: Overview of the ViennaX configuration examples, path: examples/input/

1.3.1 Linux

To build the examples, open a terminal and change to:

```
$> cd /your-ViennaX-path/
```

Create a build directory

```
$> mkdir build
$> cd build/
```

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make && make install
```

to build the examples.

Note that `make install` is required as the plugins are built in MPI and serial mode. To avoid confusion, the respective versions are moved to special subfolders, being `plugins-serial` and `plugins-mpi`.

Speed up the building process by using multiple concurrent jobs, e.g. `make -j4`.

To execute the examples, change to the example build sub-folder:

```
$> cd examples/
```

and run the executables, for example the SM scheduler

```
$> ./vxsm ../../examples/inputs/basic.xml plugins-serial/
```

or the DTPM scheduler with four processes:

```
$> mpirun -np 4 ./vxdtpm ../../examples/inputs/basic.xml plugins-mpi/
```

The MPI-based schedulers are automatically built by default, if an MPI environment is detected by CMake. To switch off automatic generation of the MPI scheduler, use the `-D USE_MPI=OFF` parameter during the CMake configuration step.

Make sure that the correct plugins subfolder is used as plugin-path. Using, for example, the serial path (`plugins-serial`) for a MPI scheduler, results in runtime errors.

1.3.2 Mac OS X

The tools mentioned in Section 1.2 are available on Macintosh platforms too. For the GNU GCC compiler the Xcode [20] package has to be installed. To install CMake, external portation tools such as Fink [21], DarwinPorts [22], or MacPorts [23] have to be used.

Most importantly, though, the build process on Mac OS X platforms is similar to Linux.

1.3.3 Build Configurations

CMake tries to find the appropriate tools used to build the examples and/or tests in the system paths. The most important tools are: the C++ compiler, the MPI environment, the Boost library, etc. Upon execution of the CMake configuration, an overview of the discovered tools is given. If an error occurred in case a tool has not been found, the configuration is halted and no Makefiles are generated.

Aside of the default build configuration (`cmake ..`), additional configuration parameters can be utilized to specialize the building process. Table 1.3 depicts the different parameters and their purpose. These parameters can be passed to the CMake build environment by using the `-D key=value` syntax, for example, `cmake -D CMAKE_BUILD_TYPE=Debug ../`.

The `config.sh` script in the root directory of ViennaX contains the introduced set of configurations for convenient use.

Parameter	**Value**	**Purpose**
`CMAKE_BUILD_TYPE`	Release|Debug	Build examples in release or in debug mode.
`USE_MPI`	ON|OFF	Build the MPI-based scheduler.
`BUILD_TESTS`	ON|OFF	Build the tests. Building the examples is omitted.
`BUILD_EXAMPLES`	ON|OFF	Build the examples.
`BUILD_DOXYGEN_DOCS`	ON|OFF	Build the Doxygen documentation.
`BOOST_INCLUDEDIR`	/path/to/Boost/include/	Use an external Boost include path. By default, CMake uses the library in the system paths.
`BOOST_LIBRARYDIR`	/path/to/Boost/lib/	Use an external Boost library path. By default, CMake uses the library in the system paths.

Table 1.3: Overview of the ViennaX CMake configuration parameters, underlined values denote default values.

1.4 Tests

To ensure the continuing quality of ViennaX a couple of test implementations have been implemented, enabling convenient checks whether the library operates flawlessly. The tests are located in the `tests/` sub-folder and are handled by CMake and CTest. One can issue the compilation and execution of the tests by executing the `run_regression.sh` script in the `tests/` sub-folder. Upon completion, an overview of the passed and failed tests will be provided.

By default the regression tests require the availability of all optional libraries.

Chapter 2

The Framework

This chapter introduces the ViennaX framework. In its essence, ViennaX is a plugin execution framework. Available simulation tools or components can be wrapped by plugins and are therefore re-used. These plugins can have data input and outputs, which are used to form a dependence graph[1]. Scheduler execute this graph, if possible, in parallel. Runtime configuration for the framework as for the plugins is based on the Extensible Markup Language (XML). In the following sections, the functionality of ViennaX is described in detail.

The general design of ViennaX is discussed in Section 2.1. The utilization of the framework is shown in Section 2.2. The plugin system which enables to utilize available functionality by wrapping around existing implementations is depicted in Section 2.3. The framework configuration approach based on the XML enables a flexible means to setup arbitrary execution flows. The configuration mechanisms are discussed in Section 2.4.

[1] As the vertices of this graph contain actual tasks, the graph is typically also referred to as task graph.

2.1 Design

ViennaX builds upon three major parts, being the scheduler, the plugin system, and the configuration (Figure 2.1).

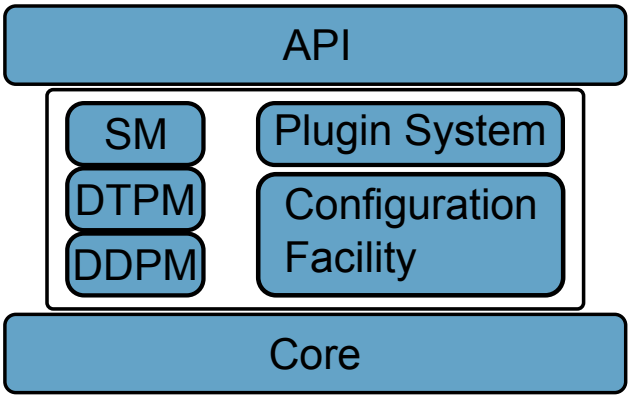

Figure 2.1: Design of ViennaX. An API provides access to the supported different scheduler kernels, being serial mode (SM), distributed-task-parallel-mode (DTPM), and distributed-data-parallel-mode (DDPM). Additionally, the plugin system, and the configuration facility can be accessed by the user. The core part provides fundamental functionality utilized throughout the framework, such as a task graph implementation.

The scheduler[2] performs the execution of the task graph, as depicted in Figure **??**. Section 2.2 gives some details on the available schedulers. Serial and parallel, MPI-based schedulers are provided, supporting data and task parallelism.

The power of a plugin system is due to its wrapping capabilities, as depicted in Figure 2.2. This introduces two substantial advantages. First, it enables re-usability, by wrapping already available tools. Second, it allows for exchangeability, as all plugins which have the same interfaces can be used. The plugin interfaces are realized by so-called data sockets, which are introduced in Section 2.3.2.

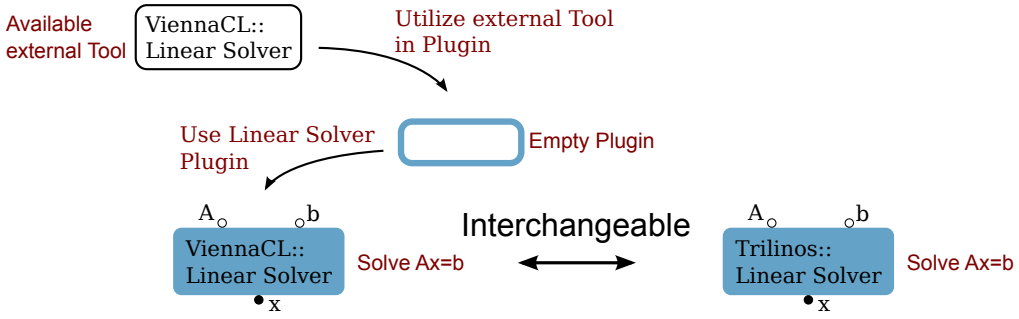

Figure 2.2: A plugin can be used to wrap available functionality. Due to the abstraction mechanism provided by the socket input/output dependencies, plugins can be exchanged with other plugins. In this case, a linear solver implementation provided by ViennaCL [24] is interchanged with an implementations of Trilinos [25].

[2] Actually there are more than one scheduler implementations available.

The configuration enables ViennaX to setup the task graph, which is the fundamental entity used to process all the plugins (see Section 2.4 for more details). The configuration is forwarded to the plugins, where the user has the opportunity to create data dependencies (also referred to as sockets) based on the parameters in the configuration. Additionally, the configuration provides plugin specific parameters, which can be accessed from within the plugin body.

In the following sections, an in-depth overview is presented on how to utilize the provided framework to setup and execute intricate task graphs.

2.2 Scheduler

ViennaX is a header only library, and as such it can be directly utilized in a project. Exemplary utilization is depicted in the examples folder, which holds source files for different schedulers.

```
1 examples/src/
```

In the following the available scheduler implementations are given.

2.2.1 Serial Mode

An exemplary implementation for utilizing the ViennaX serial scheduler can be found here:

```
1 examples/src/sm.cpp
```

The required parts to run ViennaX powered by the SM scheduler are introduced by the following code snippet.

```
1  // The ViennaX include files
2  //
3  #include "viennax/version.h"
4  #include "viennax/core/plugin_loader.hpp"
5  #include "viennax/scheduler/sm.hpp"
6
7  // The core implementation lines, issuing the execution of the framework
8  //
9  std::cout << "ViennaX version: " <<
10   ViennaX_MAJOR << "." << ViennaX_MINOR << "." << ViennaX_PATCH << std::endl;
11 viennax::PluginLoader  pl("plugin/location/");
12 viennax::scheduler(config, viennax::backend::sm(), std::cout);
```

The ViennaX version number can be accessed via macros (Lines 9,10), which require the inclusion of `version.h` (Line 3).

An automatic plugin discovery and load tool is utilized to register available ViennaX plugins (Lines 4,11). Section 2.3.6 gives more information on this tool.

The ViennaX SM scheduler is called, ultimately initiating the execution (Line 12). Note that this function expects three parameters. The first parameter is the filename of the XML configuration file, which is not only used to setup the dependence task graph, but also to configure the individual plugins. Section 2.4 gives more details on the setup of such a configuration file. The second parameter is a tag indicating the type of scheduler kernel.

The third parameter is optional and can be used to re-route debug output to a certain C++ stream. In this case the standard output is used.

2.2.2 Distributed Task Parallel Mode

An exemplary implementation for utilizing the ViennaX DTPM scheduler can be found here:

```
examples/src/dtpm.cpp
```

In the following the required parts to run the DTPM scheduler are introduced.

```
// The ViennaX include files
//
#include "viennax/core/plugin_loader.hpp"
#include "viennax/scheduler/dtpm.hpp"

// Prepare the MPI environment
//
boost::mpi::environment   env;
boost::mpi::communicator  world;

// The core implementation lines, issuing the execution of the framework
//
viennax::PluginLoader  pl("plugin/location/");
viennax::scheduler(config, viennax::backend::dtpm(), std::cout);
```

The DTPM scheduler focuses on task parallelism, similar to a data flow model. The graph is initially generated, analyzed, and partitioned. The partitions are distributed to available processes, whereas communication is realized directly between the individual processes.

Always verify the applied partitioning by investigating the dumped task graph 2.5.1.

Try to improve the plugin distribution by adding weights to the plugins 2.3.4.

2.2.3 Distributed Data Parallel Mode

An exemplary implementation for utilizing the ViennaX DDPM scheduler can be found here:

```
examples/src/ddpm.cpp
```

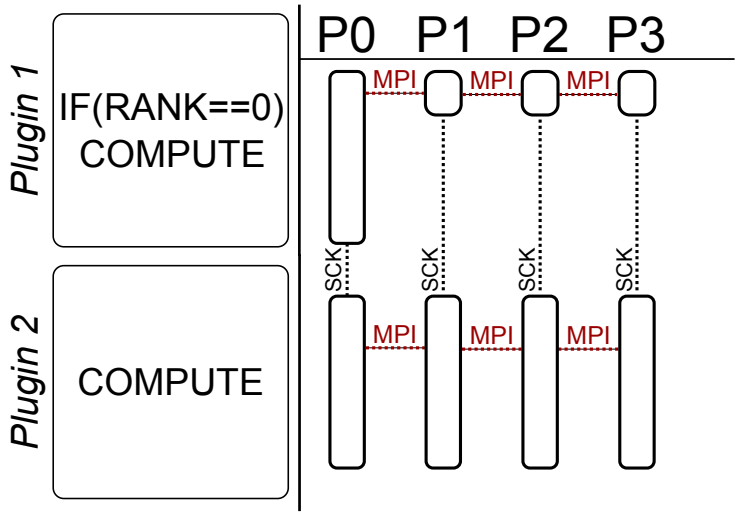

Figure 2.3: Exemplary execution behavior of the DDPM scheduler based on two plugins and four MPI processes. The bars in the right part of the figure indicate computational load. Each MPI process executes the individual plugin. Additionally, each plugin has access to an MPI communicator object, enabling not only classical data parallel execution modes but also plugin inter-process communication. Inter-plugin communication is realized by the socket mechanism (SCK).

In the following the required parts to run the DDPM scheduler are introduced.

```
1  // The ViennaX include files
2  //
3  #include "viennax/core/plugin_loader.hpp"
4  #include "viennax/scheduler/ddpm.hpp"
5
6  // Prepare the MPI environment
7  //
8  boost::mpi::environment   env;
9  boost::mpi::communicator  world;
10
11 // The core implementation lines, issuing the execution of the framework
12 //
13 viennax::PluginLoader  pl("plugin/location/");
14 viennax::scheduler(config, viennax::backend::ddpm(), std::cout);
```

The DDPM scheduler focuses on data parallelism. The graph is initially generated, analyzed, and the entire task set is distributed to all processes. Each MPI process executes all tasks in a predetermined manner. A plugin instance has access to an MPI communicator.

Fig. 2.3 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 inter-process communication is supported by the MPI library. A similar parallel communication model has already been applied by the Common Component Architecture [26].

2.3 Plugins

This section provides detailed information about the plugin system. Section 2.3.1 introduces the required steps to implement a ViennaX plugin. Section 2.3.2 discusses how to transfer data between plugins via the socket system. Section 2.3.3 depicts loop handling. Section 2.3.4 shows how to indicate the computational and communication load of the plugin, to support the schedulers ability to distribute the work efficiently. Section 2.3.6 investigates the plugin discovery and load tool.

2.3.1 General

A plugin is required to be implemented in a separate source file. The following code snippet depicts the complete implementation of an empty, but working plugin. It essentially represents the implementation of the example plugins which can be found here:

`examples/plugins/dummyX/dummyX.cpp` .

`X` denotes different integer numbers, for example, 3, which are used to uniquely separate the set of examples.

```
1  #include "viennax/core/plugin.hpp"
2
3  #define PLUGIN_NAME Dummy9
4
5  namespace viennax {
6
7  struct PLUGIN_NAME : public plugin {
8
9    INIT_VIENNAX_PLUGIN
10
11   void init() {
12     std::cout << pname() << " initializing .. " << std::endl;
13     ...
14   }
15
16   bool execute(std::size_t call) {
17     std::cout << pname() << " executing .. " << std::endl;
18     ...
19     return true;
20   }
21
22   void void finalize() {
23     std::cout << pname() << " clean up .. " << std::endl;
24   }
25 };
26 FINALIZE_VIENNAX_PLUGIN
27 }
```

The plugin system is based on common C++ dynamic polymorphism techniques, as such each plugin needs to be derived from the base class. The base class is defined in the `viennax/core/plugin.hpp` header file and as such requires inclusion into the plugin source file (Line 1). Each Plugin requires a unique plugin name, which has to be set via the macro `PLUGIN_NAME` (Line 3). Note that the actual plugin implementation is

within the ViennaX namespace (Lines 5,27). Each plugin is represented by its own class, which is derived from the ViennaX plugin base class (Line 7). The convenience macros `INIT_VIENNAX_PLUGIN` (Line 9) and `FINALIZE_VIENNAX_PLUGIN` (Line 26) have to be used to take care of boiler plate code required for a ViennaX plugin. Each plugin has three regions where the user adds the implementation, namely the `init()` (Lines 11-14), the `execute()` (Lines 16-20), and the `finalize()` (Lines 22-24) function.

The initialization method enables the user to prepare the data dependencies. The execution method provides one parameters and one return value. `call` represents the number of executions of this plugin. This value might be interesting in case of loop-wise executions, where a plugin might get executed more than once. This value provides the ability to keep track of the loop execution state. See Section 2.3.3 for details on loops. Finally, a boolean value is returned. This does not indicate that the execution was successfully, it does, however, provide ViennaX with information, whether a possible subsequent plugin can be called or an optional loop within the task graph has to be re-entered. The finalization method can be used to, for instance, deallocate previously defined objects, stored in the plugin's state.

It is important to note that the initialization function is called by the framework prior to the execution. Each plugin is given the chance to configure itself, meaning to create data sockets (see Section 2.3.2 for details). Based on the generated data sockets, ViennaX is able to connect the required plugins accordingly, which in turn enables the scheduler to run all the plugins by calling the execute function.

One can place type definitions into the body of the plugin, to setup declarations which hold both for the configuration as for the execution method.

In case a code block has to be executed before the plugin is instantiated, the `INIT_VIENNAX_PLUGIN_PREINIT` macro is provided. In the following, this macro is used instead of the default `INIT_VIENNAX_PLUGIN` macro to ensure code execution before the plugin's constructor is called.

```
1 struct PreInit {
2   void operator()() {
3     // pre-initialize something
4   }
5 };
6 INIT_VIENNAX_PLUGIN_PREINIT(PreInit())
```

2.3.2 Sockets

ViennaX plugins can get or provide data via so called data sockets. For the following in-depth investigation on sockets, the plugin implementation presented in Section 2.3.1 is re-used.

ViennaX supports two different types of sockets:

- **Sink:** A sink socket receives data from a source socket, thus representing data input.
- **Source:** A source socket sends data to a sink socket, thus representing data output.

A sink and source socket is created by utilizing the socket creation functions:

```
template<DataType>
void create_sink  (std::string const& identifier);
template<DataType>
void create_source(std::string const& identifier);
```

The `DataType` refers to the actual datatype of the data passing in and out of the socket. The `identifier` is a string object, uniquely identifying the socket. For a sink socket to be plugged into a source socket, the datatype `DataType` and the socket identifier (`identifier`) have to match. Sockets are created during run-time in the configuration phase (Figure 2.4), as such the implementation has to be done within the body of the plugin's configuration function.

The data associated with sockets can be accessed within the execution phase, by using the access functions:

```
template<typename DataType>
DataType& access_sink  (std::string const& identifier)
template<typename DataType>
DataType& access_source(std::string const& identifier)
```

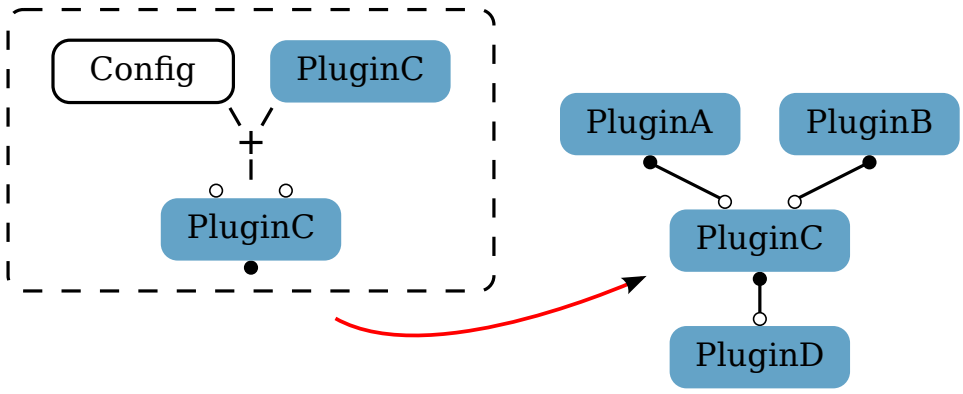

Figure 2.4: The data sockets of a plugin are created during run-time based on the configuration. The sockets can be used to exchange data with other plugins.

In the following example, the two counterparts, a sink and a source socket, are implemented to share the data of a `std::vector<double>` container. First, the plugin, providing the data, is discussed.

```
typedef std::vector<double>   Container;

void configure()
{
  ..
  create_source<Container>( "cont");
}

bool execute(std::size_t call)
{
  ..
  Container& data = access_source<Container>( "cont" );
  // data is a common std::vector<> datatype,
  // no proxies or wrappers are utilized
  //
  data.resize(10);
  for(std::size_t i = 0; i < data.size(); i++)
    data[i] = i;
  return true;
}
```

As can be seen, the access function provides the user with the actual data object.

In the following the socket relevant implementation parts of the body of the sink plugin is shown.

```
typedef std::vector<double>   Container;

void configure()
{
  ..
  create_sink<Container>( "cont");
}

bool execute(std::size_t call)
{
  ..
  Container& data = access_sink<Container>( "cont" );
  // data is a common std::vector<> datatype,
  // no proxies or wrappers are utilized
  //
  for(std::size_t i = 0; i < data.size(); i++)
    assert( data[i] == i );
  return true;
}
```

A similar utilization of sockets can be found in the examples, `dummy12` and `dummy13`.

The provided `create` functions automatically instantiate an object of the associated type holding the data. This requires the type-related class to be default constructable. If this

is not supported, or the data object should simply be not instantiated for some reason, an already available object can be linked to a socket. Therefore, the `link_socket` method can be used, as depicted in the following.

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

2.3.2.1 Physical Units

Units are a major concern in scientific computing, as mixing the units between, for example, functions, obviously results in a major corruption of the computational result. As such it is of utmost interest to introduce automatic layers of protection to ensure that required data offer the expected units.

In ViennaX we tackle this challenge by coupling the unit information to the string based identifier of the sockets by the `make_datakey()` method. In the following, we extend the already introduced creation of a source socket by an additional unit information.

```
1 void configure()
2 {
3   create_source<Container>( make_datakey("cont", "Vs"));
4 }
```

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 identifier string, a sink and a source socket with different identifiers will not be connected. The string based approach allows to couple arbitrary properties to the sockets, making it a highly versatile system to impose correctness on the plugin data connections.

It is important to retrieve the unit string objects from a versatile unit system, which also allows to ultimately encode the unit information into the type system. We suggest to utilize the Boost Units Library [27].

Until C++11 is widely adopted by the compilers, thus variadic templates can be used, we support up to four string objects via the `make_datakey()` method.

2.3.2.2 User-Defined Datatypes

The ViennaX socket system supports arbitrary datatypes, as depicted in the following code snippet.

```
1  struct Foo
2  {
3    double  val;
4  };
5
6  void configure()
7  {
8    create_source<Foo>( "foo" );
9  }
10
11 bool execute(std::size_t call)
12 {
13   Foo& foo = access_source<Foo>( "foo" );
14   foo.val = 2.0;
15   return true;
16 }
```

Make sure your namespaces, which might contain your user-defined datatype, match when creating sockets based on this datatype in different plugins.

However, if the plugin is supposed to be executed in an MPI environment, the user-defined datatype needs to provide additional serialization information. As we utilize the Boost MPI Library [13], which in turn utilizes the Boost Serialization Library [28], each datatype used by the ViennaX sockets has to be Boost serializable. For in-depth information about serialization and how to handle intricate cases visit the Boost Serialization project [28].

In the following, we make our previously depicted user-defined datatype serializable.

```
1  #include <boost/archive/text_oarchive.hpp>
2  #include <boost/archive/text_iarchive.hpp>
3
4  struct Foo
5  {
6    double  val;
7
8  private:
9    friend class boost::serialization::access;
10   template<class Archive>
11   void serialize(Archive & ar, const unsigned int version)
12   {
13     ar & val;
14   }
15 };
```

Note that the Boost Archive Library has to be included (Lines 1-2). Each user-defined datatype has to notify the Boost Serialization mechanisms how to serialize the data. In this case we only have to write/read the member `val` to/from the archive (Lines 8-14).

The example `user_defined.xml`, which utilizes the example plugins `Dummy12` and `Dummy13`, depicts the utilization of user-defined datastructures.

If a data type has a fixed amount of data stored at fixed field positions (as is the case in our example), optimization operations can be employed by Boost MPI. More details are given in the chapter *User-defined data types* of the Boost MPI tutorial [13].

The compile error `...has no member named serialize` indicates, that the serialization extensions are missing or corrupted.

2.3.3 Loops

The SM and the DDPM scheduler kernel support support loops in the task graph, to enable the execution of, for instance, optimization processes. An example of a loop setup can be found in the `loop.xml` example, and the utilized plugins: `Dummy5,Dummy6,Dummy7,Dummy8`

Figure 2.5 depicts the setup of the loop example, and the applied data flow. According to this figure, `PluginB` and `PluginC` are the entry and the exit plugin of the loop, and as such have to contain additional logic for a successful execution.

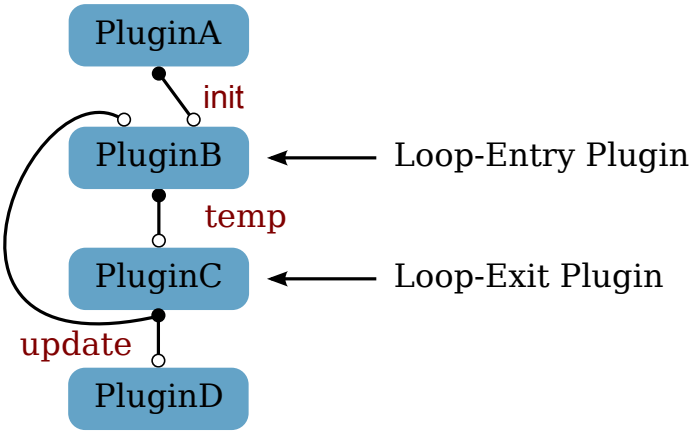

Figure 2.5: An exemplary loop execution is depicted. `PluginB` and `PluginC` handle the loop logic. The loop is only exited, if `PluginC` decides so.

2.3.3.1 Loop Entry

The first important part is the entry plugin, as there are two sink sockets, where one of these only contains data on the second and higher loop iteration. Therefore, one must not access data from this socket on the first iteration. The `call` counter, introduced in Section 2.3.1, can be used as trigger information, whether a certain socket should be accessed or not.

Such an application can be seen in the following, where the important part of the `PluginB` (`Dummy6` in the examples) implementation is shown.

```
1  int& source = access_source<int>( make_datakey("temp") );
2  if(call == 0)
3  {
4    int& data = access_sink<int>( make_datakey("init") );
5    ...
6  }
7  else
8  {
9    int& data = access_sink<int>( make_datakey("update") );
10   ...
11 }
```

The source socket can be accessed, regardless of the execution state (Line 1). In case it is the first call, meaning it is the first loop execution, the data of the `init` socket has to be accessed (Lines 2-6). In case it is a subsequent loop execution, the other input socket `update` has to be accessed (Lines 7-11).

2.3.3.2 Loop Exit

The loop exit, on the other side, has to utilize the boolean return value of the execution function, introduced in Section 2.3.1.

In the following, the implementation of `PluginC` (`Dummy7` in the examples) is depicted.

```
1  // Exemplary loop-exit condition
2  if(sink <= 5) return false;
3  else          return true;
```

A common condition is used to decide whether `true` or `false` has to be returned. In case of `false`, the loop is continued, where in case of `true`, the loop is exited.

Currently loops are only supported with the serial and the centralized MPI scheduler.

2.3.4 Weights

Beginning with Version 1.1.1, ViennaX supports user provided information to indicate the computation and communication load of a plugin.

This is only important in case the plugin is executed with the DTPM scheduler, as this particular scheduler performs a task graph partitioning.

For this purpose new functions have been introduced (`computation_level()` and `communication_level()`), which should be called during the configuration step. Both functions accept integer of values greater than zero. The higher the value, the higher the respective load, as depicted in the following.

```
1 void configure() {
2   ..
3   computation_level()    = 10;
4   communication_level() = 1;
5 }
```

The expression in Line 3 sets the computation level to 10, indicating a severe computational effort. Line 4 expresses low amount of communication volume, thus the amount of data received or transmitted is small.

Always try to provide this information, as it helps the scheduler to derive a reasonable partition of the tasks.

2.3.5 Programming Language Support

Aside of C++ ViennaX is known to support plugins utilizing C and Fortran code. In general, the plugin interface has to be always implemented in C++ using the introduced interface. However, external non-C++-based implementations can be called from within the plugins, by employing the usual techniques of calling other languages from within C++ code. For instance, the `fortran_dummy` plugin depicts the utilization of Fortran-based code from within a ViennaX plugin.

2.3.6 Loader

The ViennaX plugins apply the so-called self-registering approach [29], and as such are only required to be loaded by a run-time system. The plugin system takes care of notifying the framework with respect to which plugins are available. The convenience class `PluginLoader` has been implemented to automatically load and handle the plugins. To register plugins within the ViennaX framework, the class has to be utilized, as depicted in the example applications:

```
1 /examples/src/
```

In the following, the `PluginLoader` class is utilized to load ViennaX valid plugins. See Section 2.3.1 to get information on how to implement such plugins.

```
1 #include "viennax/core/plugin_loader.hpp"
2
3 viennax::PluginLoader  pl("path/to/top/level/plugin/directory/");
```

Note that the plugin loader class recursively traverses the sub-folders beginning using the provided path as origin. All detected dynamic libraries, both Linux and MacOS X versions are supported, are automatically discovered and registered within ViennaX.

It is important that the instantiated object, `pl` in the provided example, is not destroyed before the framework and/or the plugins are utilized. The plugin loader automatically destroys the loaded plugin objects upon destruction, as such it has to be kept alive during the execution of ViennaX.

2.4 Configuration

ViennaX is based on run-time configuration to allow user-input to actively drive the execution of the task graph. ViennaX relies on the XML to build a flexible configuration environment. An external library is utilized, being pugixml [30], which is distributed under the MIT license. The reason for choosing pugixml is threefold. First, it is (as of Version 1.2) a header only library, rendering the inclusion into projects trivial. Second, it supports XPath, enabling convenient data access via a path-like access language. Third, the code base is very lightweight, it only consists of three source files.

Another reason for XML is the fact that a possible graphical user interface GUI can be added easily on top of this system. The GUI would have to generate ViennaX valid XML data, which could then be used to drive the framework execution.

This section provides details on how to setup a ViennaX configuration file based on the XML.

2.4.1 General

A ViennaX configuration file has to have the file ending `.xml`. The path to this file is supposed to be forwarded to the provided schedulers, as depicted in Section 2.2.

A basic configuration file, which sole purpose is to utilize a single plugin `Dummy9`, is depicted in the following.

```
1 <plugins>
2    <plugin>
3       <key>Dummy9</key>
4    </plugin>
5 </plugins>
```

The general `plugins` region contains the set of all plugins, which should be utilized during the execution (Lines 1-5). Each plugin is defined within its own region (Lines 2-4). The name of the plugin has to be mentioned within the key region (Line 3), which has to be exactly the same key as the name set by the `PLUGIN_NAME` macro of the respective plugin (see Section 2.3.1 for more details).

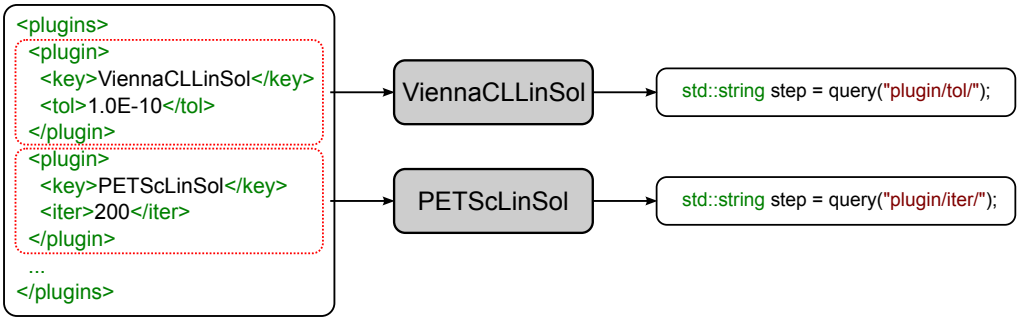

Figure 2.6: Each plugin has its own configuration region within the input configuration file. This data can be accessed from within the plugin, by querying the configuration object.

An important aspect is to forward parameters to the plugin via the configuration file, as depicted in Figure 2.6. Therefore, we can attach additional, arbitrary data fields within the respective plugin configuration region, as depicted in the following.

```
<plugins>
   <plugin>
      <key>Dummy9</key>
      <step>0.3</step>
   </plugin>
</plugins>
```

An arbitrary configuration parameter has been added to the plugin specific configuration part (Line 4). This data can be accessed from within the configuration or the execution region of the particular plugin instance via querying the configuration object. The following depicts such a data access.

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

In the following a more intricate example is shown, where a couple of plugins are utilized. This setup is based on the `basic.xml` example.

```
<plugins>
   <plugin>
      <key>Dummy0</key>
   </plugin>
   <plugin>
      <key>Dummy1</key>
   </plugin>
   <plugin>
      <key>Dummy2</key>
   </plugin>
   <plugin>
      <key>Dummy3</key>
   </plugin>
   <plugin>
      <key>Dummy4</key>
   </plugin>
</plugins>
```

Each plugin has its own dedicated region, supporting additional plugin specific, arbitrary configuration parameters.

2.4.2 Conversion

ViennaUtils provides a conversion mechanism for convenient utilization of the query results in numerical expressions.

```
#include "viennautils/convert.hpp"
TargetType target = viennautils::convert<TargetType>()(source);
```

This conversion utility can be directly coupled with the query result, for example:

```
double step = viennautils::convert<double>()(query_value("plugin/step"));
```

2.4.3 Mathematical Evaluations

The Lua programming language is used to evaluate mathematical string-typed expressions during run-time. For instance, the following plugin configuration contains such an expression in the `step` parameter.

```
<plugins>
   <plugin>
      <key>Dummy9</key>
      <step>0.3*2</step>
   </plugin>
</plugins>
```

This parameter can be accessed and automatically evaluated by the following.

```
double result = eval_mathstr<double>(query_value("plugin/step"));
// result equals 0.6
```

2.4.4 Cloning

In case several instances of one plugin have to be created, ViennaX provides a convenience layer to ease this step: the clone mechanism.

In the following the clone functionality is introduced, which is also utilized in the `clone.xml` example.

```
<plugins>
   <plugin>
      <key>Dummy9</key>
      <clones>3</clones>
   </plugin>
</plugins>
```

Note the `clones` region in Line 4. Merely setting the number of clones results in the generation of exactly this amount of instances.

2.5 Tools

ViennaX provides tools to support software developers to successfully implement applications.

2.5.1 Graph Dump

ViennaX dumps the graph to an output file (`graph.dot`). The data is written in the dot/-graphviz format and can be visualized with the graphviz tool [31]. For example, the following command converts the dot file into a `ps` file.

```
$> cd /your-ViennaX-path/build/examples/
$> dot -Tps graph.dot -o graph.ps
```

Visualizing the graph can greatly improve the debugging experience.

The DTPM scheduler dumps the association of each plugin with a certain MPI process. It is vital to check this, as bad partitioning results, for instance, in unnecessary communication. Use performance analysis 2.5.6 and plugin weights 2.3.4 to tackle this.

2.5.2 Logger

For logging tasks ViennaX provides three functions of different debugging level, which are implemented in the `viennax/utils/debug.hpp` source file.

```
inline void error   (std::string const& msg, std::ostream& str = std::cout);
inline void warning(std::string const& msg, std::ostream& str = std::cout);
inline void info    (std::string const& msg, std::ostream& str = std::cout);
```

All three functions have the same parameter list, string object `msg` acting as the actual data to be printed to the stream `str` (default stream is `std::cout`).

Error output is written in red (to the terminal), warning in yellow, and info in green.

2.5.3 Timer

A convenient execution timer is provided, which measures the elapsed time in seconds. The implementation resides in an auxiliary library, called ViennaUtils (Section 1.2.7). The library is located in `external` and has to be added to the include path (Our CMake configuration already deals with that). However, the header file, relative to the ViennaUtils inclusion folder, is located here: `viennautils/timer.hpp`

```
viennautils::Timer timer;
timer.start();
// do something intensive
std::cout << " exec-time: " << timer.get() << " s" << std::endl;
```

All example plugins utilize the timer.

2.5.4 Type Dump

Another ViennaUtils (Section 1.2.7) functionality is provided by a type name print function, located in the source file: `viennautils/dumptype.hpp`. This tool can be used to print a string representation of an arbitrary type to the terminal or an arbitrary stream. In the following the available function signatures are provided.

```
template<typename T>
void dumptype(std::ostream& stream = std::cout);

template<typename T>
void dumptype(T & t);

template<typename T>
void dumptype(T const& t);
```

This functionality enables to do the following:

```
double val;
viennautils::dumptype(val);        // prints: "double"
viennautils::dumptype<double>();   // prints: "double"
```

This functionality might get very handy, when intricate datatypes are employed.

2.5.5 Dynamic Multi-Dimensional Array

A Boost serializable dynamic array datastructure implementation [32] is provided, enabling, for instance, to associate an array object to a socket, where the dimension is not known in advance. The required header file is located in `utils/multi_array.hpp`.

The following depicts a basic utilization.

```
1 typedef MultiArray<Numeric>     MultiArrayT;
2 MultiArrayT::dimensions_type    dim;
3 dim.push_back(3);
4 dim.push_back(4);
5 MultiArrayT                     multiarray(dim);
6 multiarray(make_vector(2,3))  = Numeric(3.5);
```

The array holds elements of type `Numeric` (Line 1). A dimension object is created, enabling to non-intrusively set the extents of the data structure (Lines 2). A two-dimensional array is created, where the first dimension can hold three and the second dimension can hold four elements (Lines 3,4). The datastructure is accessed, by forwarding the dimension object (Line 5). Elements are accessed via a unified, generic access method (Line 6).

If the dimension of the array is static, compile-time approaches, like Boost Array [33], should be applied to increase run-time performance.

2.5.6 Performance Analysis

To identify bottlenecks in ViennaX simulations, it is vital to use trace-based analysis tools to investigate the execution performance in great detail. For this reason, we use the free open source tool visualization tool Paraver [17] which is capable of analyzing traces from, for instance, the Extrae library [17].

Although both packages have to be available on the target system, we provide example scripts to ease the utilization experience. The scripts are available in the following subdirectory.

```
1  $> cd profiling/
```

For the following example, we assume that the environment variable `EXTRAE_HOME` is set to the installation directory, containing the `include` and `lib` folder of the Extrae library.

First, the trace data of a MPI application has to be collected. The Extrae library provides different approaches to do this, in the following we use the one based on linking against Extrae's `libmpitrace.so` library. The example is based on the mesh adaptation example, provided with ViennaX. To collect the trace data, the execution of the ViennaX application has to be altered, by using the trace script provided by ViennaX.

```
1  $> cd build/examples/
2  $> EXTRAE_CONFIG_FILE=../../profiling/extrae.xml ../../profiling/trace.sh \
3  $> mpirun -np 4 ./vxdtpm ../../examples/inputs/parallel.xml \
4  $> plugins-mpi/
```

No recompilation is required to generate the trace data with Extrae!

Note that in Line 2 the configuration file and trace script is utilized, which is shipped with ViennaX, beginning with Version 1.1.1. The trace file generation is successful, when the following line is printed to the terminal

```
1  $> mpi2prv: Congratulations! mpi_ping.prv has been generated.
```

The second step is to visualize the generated trace data. For this task we can utilize the Paraver tool, which has been downloaded in binary form from the developer webpage. After opening Paraver, the trace can be loaded by `File-Load Trace` and then point to the `*.prv` file in the build directory. Afterwards, generate a new timeline window by pressing the corresponding button in the top left corner. A new window opens, which looks for example like the following.

Figure 2.7: A total timeline is depicted. Blue denotes processing, pink/yellow communication. No reasonable communication and especially no waiting processes can be identified.

If a good part of the overall runtime is spent on waiting, the task distribution is bad. In case of the partitioned MPI scheduler, try to identify shortcomings by investigating the dumped task graph (Section 2.5.1), and tackle them by assigning weights on the vertices (Section 2.3.4).

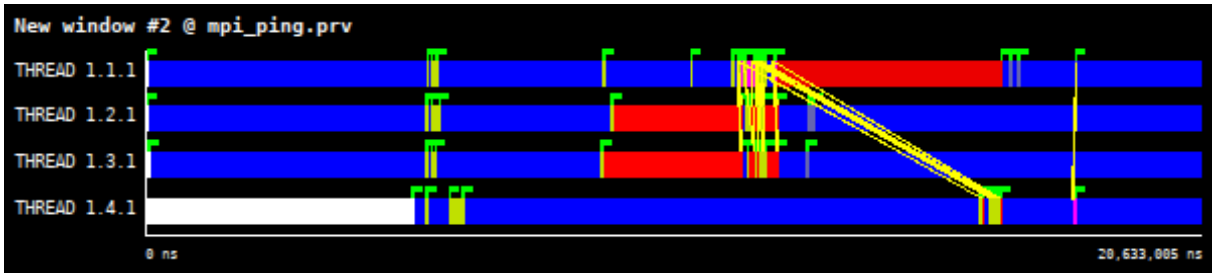

Figure 2.8: A zoomed view of the startup region of the previous total view is provided. White denotes non-started up process, red denotes a waiting process. Obviously during the startup phase, some preprocessing plugins introduce some waiting-time.

Chapter 3

Versioning

Each release of ViennaX carries a three-fold version number, given by

```
ViennaX X.Y.Z.
```

For users migrating from an older release of ViennaX to a newer one, the following guidelines apply:

- `X` is the *major version number*, starting with `1`. A change in the major version number is not necessarily API-compatible with any versions of ViennaX carrying a different major version number. In particular, end users of ViennaX have to expect considerable code changes, when changing between different major versions of ViennaX.
- `Y` denotes the *minor version number*, restarting with zero whenever the major version number changes. The minor version number is incremented whenever significant functionality is added to ViennaX. The API of an older release of ViennaX with smaller minor version number (but same major version number) is *essentially* compatible to the new version, hence end users of ViennaX usually do not have to alter their application code. There may be small adjustments in the public API, which will be extensively documented in the change logs and require at most very little changes in the application code.
- `Z` is the *revision number* or *patch level*. If either the major or the minor version number changes, the revision number is reset to zero. The public APIs of releases of ViennaX, which only differ in their revision number, are compatible. Typically, the revision number is increased whenever bug-fixes are applied, performance and/or memory footprint is improved, or some extra, not overly significant functionality is added.

Always try to use the latest version of ViennaX before submitting bug reports!

Chapter 4

Change Logs

4.1 Overview

Version 1.2.0

- *(new)* Added new distributed data parallel scheduler
- *(new)* Introduced new scheduler naming scheme
- *(new)* Run-time evaluation of mathematical string-typed expressions
- *(new)* Plugin interface now follows three-step init/exec/finalize model
- *(new)* Improved fallback handling for DTPM partitioner
- *(new)* Added dynamic multi array datastructure facility
- *(new)* Added support for Fortran plugins
- *(new)* Improved debugging facilities
- *(new)* Added new plugin convenience macros
- *(new)* Introduced `link_source`
- *(Change)* Configuration ID is now computed automatically (Section 4.2.1)
- *(change)* ViennaFactory has been merged into ViennaX (Section 4.2.2)
- *(change)* Removed old 'centralized' scheduler (Section 4.2.3)

Version 1.1.1

- *(new)* Plugin computation/communication-level support
- *(new)* Improved graph dumping
- *(new)* Execution scripts for Paraver/Extrae performance analysis tools
- *(patch)* Partitioned MPI scheduler can now handle non-contiguous graph partitions
- *(patch)* Simplified/improved build environment
- *(change)* The METIS library is now shipped (Section 4.2.4)

Version 1.1.0

- *(new)* Added new MPI scheduler based on task graph partitioning
- *(new)* Examples: Jacobi, FEM Stress, Parallel Mesh Adaptation
- *(new)* Advanced regression test system
- *(new)* Dynamic plugin libraries are built in mpi and non-mpi versions
- *(change)* Changed signature of configuration and execution plugin functions (Section 4.2.5)
- *(change)* Removed 'context' feature (Section 4.2.6)
- *(change)* Decoupled and simplified configuration queries (Section 4.2.7)
- *(patch)* Additional sanity checks with respect to plugin loading
- *(patch)* Code cleanup / removal of obsolete code
- *(patch)* Improved plugin-related cmake build system
- *(patch)* Updated manual

Version 1.0.1

- *(patch)* Improved user manual
- *(patch)* Improved CMake configuration

Version 1.0.0

First release

4.2 Changes

4.2.1 Configuration ID Obsolete

Beginning with Version 1.2.0, the requirement to provided each plugin region in the input configuration file with a unique integer-based ID has been lifted. ViennaX automatically handles the plugins internally.

4.2.2 Merged ViennaFactory

The standalone ViennaFactory library has been merged into the ViennaX project. This is an internal change, thus the user is not affected.

4.2.3 Removed Centralized Scheduler

This scheduler was removed due to fundamental design issues.

4.2.4 METIS Shipped

As the METIS library is now shipped with ViennaX, the METIS related CMake parameters are obsolete. No user interaction is required.

4.2.5 Updated Plugin Function Signatures

The user-level plugin member functions `configure()` and `execute()` have been updated with respect to their signature.

The old version:

```
1 void configure(std::size_t pid, std::string const& configstr) { ... }
2 bool execute(std::size_t pid, std::size_t call)  { ... }
```

The new version:

```
1 void configure() { ... }
2 bool execute(std::size_t call)  { ... }
```

Therefore it is not required that the configuration object of the plugin has to be loaded manually, thus the following is not required anymore, in fact it is not supported.

```
1 load(configstr);
```

4.2.6 Removed Contexts

In the initial version of ViennaX a mechanism namned context has been introduced. Due to the fact, that the socket names can be set during run-time, the unique identifiers can be set by the configuration data. This fact renders the context mechanism obsolete, thus it has been removed.

Configuration files which still contain contexts can still be utilized, but the context is ignored by ViennaX.

4.2.7 Updated Configuration Queries

The query against the configuration object has been simplified and decoupled from the XML/XPath syntax.

The old version:

```
1 std::string query_result = config().query("plugin/file/text()"));
```

The new version:

```
1 std::string query_result = query_value("plugin/file"));
```

or

```
1 std::string query_result = query_value("plugin/file/"));
```

Note that as with Version 1.0.0 in both functions, `configure()` and `execute()`, the query mechanism is available without any preprocessing (Section 4.2.5).

Chapter 5

License

Bibliography

[1] D. Keyes, L. C. McInnes *et al.*, "Multiphysics simulation," Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-321, 2011.

[2] P. Wesseling, *Principles of Computational Fluid Dynamics*. Springer, 2001.

[3] "Elmer." [Online]. Available: http://www.csc.fi/english/pages/elmer/

[4] F. Magoules, *Mesh Partitioning Techniques and Domain Decomposition Methods*. Saxe-Coburg Publications, 2008.

[5] "ViennaX." [Online]. Available: http://viennax.sourceforge.net/

[6] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[7] J. Dongarra and A. van der Steen, "High-performance computing systems," *Acta Numerica*, vol. 21, pp. 379–474, 2012.

[8] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[9] A. Dios, R. Asenjo *et al.*, "High-level template for the task-based parallel wavefront pattern," in *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011, pp. 1–10.

[10] A. Miczo, *Digital Logic Testing and Simulation*. Wiley & Sons, 2003.

[11] "GCC: The GNU Compiler Collection." [Online]. Available: http://gcc.gnu.org/

[12] "Clang: A C Language family frontend for LLVM." [Online]. Available: http://clang.llvm.org/

[13] "The Boost MPI Library." [Online]. Available: http://www.boost.org/libs/mpi/

[14] "Open MPI." [Online]. Available: http://www.open-mpi.org/

[15] "MPICH2." [Online]. Available: http://www.mcs.anl.gov/research/projects/mpich2/

[16] "LAM/MPI." [Online]. Available: http://www.lam-mpi.org/

[17] "Paraver/Extrae." [Online]. Available: http://www.bsc.es/ssl/apps/performanceTools/

[18] "METIS." [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/metis/

[19] G. Karypis and V. Kumar, "A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[20] "Xcode Developer Tools." [Online]. Available: http://developer.apple.com/technologies/tools/xcode.html

[21] "Fink." [Online]. Available: http://www.finkproject.org/

[22] "DarwinPorts." [Online]. Available: http://darwinports.com/

[23] "MacPorts." [Online]. Available: http://www.macports.org/

[24] "ViennaCL." [Online]. Available: http://viennacl.sourceforge.net/

[25] "The Trilinos Project." [Online]. Available: http://trilinos.sandia.gov/

[26] D. E. Bernholdt, B. A. Allan *et al.*, "A component architecture for high-performance scientific computing," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[27] "The Boost Units Library." [Online]. Available: http://www.boost.org/libs/units/

[28] "The Boost Serialization Library." [Online]. Available: http://www.boost.org/libs/serialization/

[29] D. Kharrat and S. Quadri, "Self-Registering Plug-ins: An Architecture for Extensible Software," in *Canadian Conference on Electrical and Computer Engineering*, 2005, pp. 1324 –1327.

[30] "pugixml." [Online]. Available: http://pugixml.org/

[31] "Graphviz." [Online]. Available: http://www.graphviz.org/

[32] J. Weinbub, K. Rupp, and S. Selberherr, "A generic multi-dimensional run-time data structure for high-performance scientific computing," in *Proceedings of the World Congress on Engineering (WCE)*, 2012, pp. 1076–1081.

[33] "The Boost Array Library." [Online]. Available: http://www.boost.org/libs/array/