UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Technology

Ihar Suvorau

Real-time visualization of parallel simulations in CERN material design

Bachelor's Thesis (12 ECTS)

Curriculum Science and Technology

Supervisors:
Prof. Vahur Zadin
Assoc. Prof. Andreas Kyritsakis
Dr. Mihkel Veske

Tartu 2021

Real-time visualization of parallel simulations in CERN material design

Abstract: This work presents the implementation of the *in situ* visualization module for multiscale-multiphysics simulation code FEMOCS and demonstrates its behavior in the simulation of vacuum breakdown. The visualization module makes it possible to observe in real-time the course of the simulation in FEMOCS and makes it more straightforward to set up a new simulation or develop additional features into the code.

The first and second chapters briefly introduce the vacuum breakdown phenomenon and describe general aspects of numerical simulations. The third chapter describes the *in situ* method as a way of improving FEMOCS. The fourth and fifth chapters present the final solution and the impact of the solution on the overall running time of the simulation.

Keywords: Software development, numerical simulations, vacuum breakdown, in situ, co-processing, post-processing, visualization steering.

CERCS: T111 Imaging, image processing; T120 Systems engineering, computer technology; T150 Material technology; P170 Computer science, numerical analysis, systems, control.

Paralleelsete simulatsioonide reaalajas visualiseerimine CERNi materjali kujundamisel

Lühikokkuvõte: Käesolevas töös esitatakse *in situ* visualiseerimismoodul multiskaalsele ja -füüsikalisele simulatsioonikoodile FEMOCS ning demonstreeritakse selle toimimist vaakumläbilöögi simulatsiooni näitel. Arendatud visualiseerimismoodul võimaldab jälgida reaalajas FEMOCSi töövoogu ja lihtsustab nii korrektsete simulatsiooniparameetrite leidmist kui koodi edasist arendamist. Esimeses ja teises peatükis tutvustatakse lühidalt vaakumläbilöögi nähtust ja käsitletakse numbriliste simulatsioonidega seonduvat. Kolmandas peatükis kirjeldatakse *in situ* visualiseerimismoodulit kui FEMOCSi laiendust. Neljas ja viies peatükk kirjeldavad lõpplahenduse tehnilisi üksikasju ning mõju simulatsiooni üldisele tööajale.

Võtmesõnad: Tarkvaraarendus, numbrilised simulatsioonid, vaakumläbilöök, *in situ* andmetöötlus, andmete järeltöötlus, visualiseerimise juhtimine.

CERCS: T111 Pilditehnika; T120 Süsteemitehnoloogia, arvutitehnoloogia; T150 Meterjalitehnoloogia; P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria).

TABLE OF CONTENTS

TERMS, ABBREVIATIONS, AND NOTATION

ADIOS	The Adaptable IO System
API	Application programming interface
CERN	Conseil européen pour la recherche nucléaire
CESM	Community Earth System Model
CLIC	Compact Linear Collider
FEM	Finite element method
FEMOCS	Finite Elements on Crystal Surfaces
FLOPS	Floating-point operations per second
HPC	High-Performance Cluster
HPL	High-Performance Linpack
I/O	Input-output (computer operations)
LAMMPS	Large-scale Atomic/Molecular Massively Parallel Simulator
LHC	The Large Hadron Collider
MD	Molecular dynamics
MPI	Message Passing Interface
OS	Operating System
POSIX	The Portable Operating System Interface
URI	Uniform Resource Identifier
VTK	The Visualization Toolkit

1 INTRODUCTION

Scientific computing is an interdisciplinary field, which provides computational infrastructure, tools, and methods for studying natural phenomena. For materials science, it gives a computational framework for creating and developing new materials and studying why the existing materials fail or underperform. In some cases, computer simulations provide a more cost-effective way of conducting experiments, in others — it is the only way possible. However, there is still a fundamental gap between a simulation and a real-world system; thus, more robust and complex simulations for analysis and predictions are always needed.

Because of that, modern simulations have increased in complexity enormously. The well-known Top500 list of supercomputers of the world had an entry requirement of 1.32 petaflops on the High-Performance Linpack (HPL) benchmark in 2020 [1]; and the most performant system in the list was the Fugaku supercomputer from Japan, which did 442 petaflops on the benchmark. *Ahrens et al.* provide a hypothetical example of how a modern exascale simulation for temperature and density calculations of 1000 time steps can consume roughly 24 TB of the disk space for its imagery data [2].

As supercomputers become more accessible to researchers and the complexity of simulations increases, there is an increased demand for more interactivity in simulations. The *in situ* processing method reduces the data storage and transfer cost for simulation users and developers. Furthermore, the *in situ* method enables real-time simulation monitoring and debugging through visualization and simulation steering.

As a use case for applying the *in situ* visualization technique, we took the problem of vacuum breakdown in materials design for the Compact Linear Collider project, which causes damage to the accelerator's materials and limits the system's overall performance. A computational model of vacuum breakdown around a nanostructure has been developed in the collaboration between the University of Tartu and the University of Helsinki [3]–[5]; and the simulation code was implemented in the software called FEMOCS [4], [6], [7].

To understand how the *in situ* processing can be helpful in FEMOCS, we can approximately estimate the computational complexity of the nano-tip simulation. Each time step of the simulation equals 4×10^{-15} s, if we want to simulate the phenomenon for a nanosecond, which is a reasonable duration for vacuum breakdown to occur [5], we need $\frac{1\times10^{-9}}{4\times10^{-15}} = 250000$ time steps. If each time step takes on average about 2 s to compute on a single node with multiple cores, this gives us 500000 s or about six days of computational time to finish the simulation. During the simulation development, it is not an easy task to make everything right from a single approach; thus, multiple runs of a simulation might be needed; and if the simulation runs roughly for a week per run, and we want to test it for 2−3 times, our waiting time then is about 2−3 weeks.

This example demonstrates that using multiple computers for complex simulations and the parallelization of simulations is needed; and that receiving feedback from a simulation before the simulation finishes can critically impact the speed of the simulation development. The *in situ* visualization technique is a well-known way of improving the visualization workflow and reducing the time until the first visual feedback from the simulation by moving the visualization routines closer to simulation calculations without waiting for the simulation to finish [8], [9].

FEMOCS uses the *post-processing* method for visualization at the moment, in which visualization happens after the full stop of the simulation. This does not allow for interactivity during a simulation run; thus, this research aims to improve the visualization workflow of FEMOCS by introducing the *in situ* visualization capabilities into it and allowing real-time interactions with the simulation data as the simulation is still running. The new real-time visualization feature for FEMOCS would allow getting scientifically important insights from a simulation quicker while also removing the need to store the intermediate simulation data. Besides the sample nano-tip simulation used in this work, FEMOCS can be used for a wide range of simulation tasks, which require concurrent atomistic and continuous-space calculations. For instance, there are attempts to use it for estimating mechanical stress in nanomaterial [10] and implementing a two-temperature model for swift heavy ion simulations [11]. The *in situ* capabilities developed in this work can be adjusted for such simulations as well.

1.1 Aim

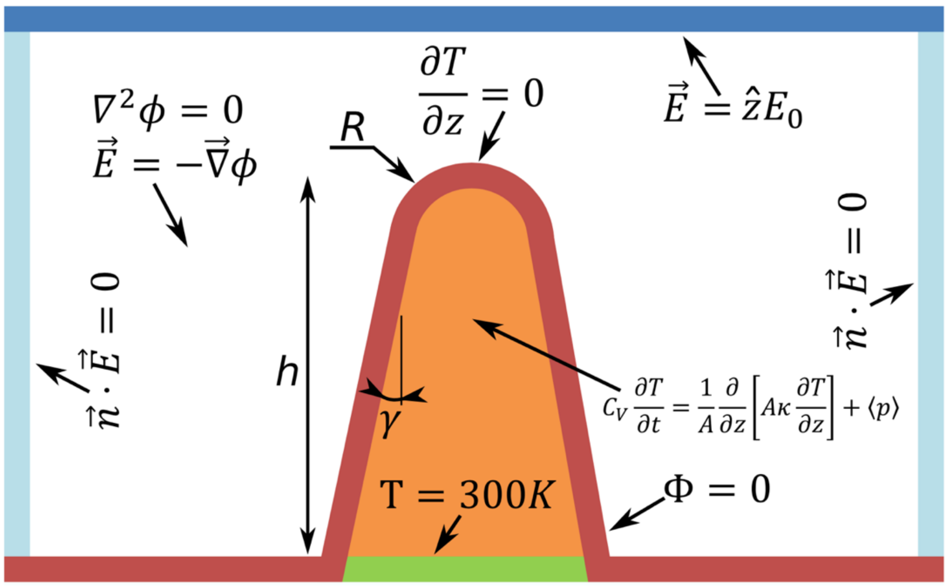
This work aimed to implement *in situ* visualization capabilities for FEMOCS to improve the visualization workflow for FEMOCS simulation developers and users. To do that, the following objectives were set in place:

- Write a ParaView Catalyst adaptor library between FEMOCS and ParaView Catalyst.
- Create an extension for LAMMPS to integrate FEMOCS with the adaptor library.
- Investigate the runtime overhead of the resulting adaptor library in comparison to the rest of the code.
- Test the adaptor library on the existing nano-tip simulation.

2 RATIONALE

2.1 Background

The Compact Linear Collider (CLIC) [12]–[14] is a linear electron-positron accelerator under development at CERN. CLIC is one of the most promising projects for further particle physics experiments, especially Higgs-boson, top-quarks, physics beyond Standard Model, and Dark Matter. In order to achieve high energies in linear accelerators like CLIC, high accelerating gradients and thus extremely high electric fields are necessary, since particles need to be accelerated in short distances; however, such electric fields cause vacuum arcs, also known as vacuum breakdowns [15], [16]. Vacuum arcs are electrical discharges between metal electrodes, occurring through vacuum forming plasma effect and plasma formation [17]. This becomes possible when a high electric field is applied to the electrode material, causing it to switch its state from solid to plasma, which breaks down vacuum insulation due to plasma's expansive and conductive properties. This phenomenon found its application in, e.g., vacuum arc deposition of thin films [17]. However, it is an undesirable effect in CLIC as it distorts the accelerated particle beam and inflicts damage to the accelerator equipment's surface, thus limiting how high an electric field can be applied and down-



$\nabla^2\phi = 0$
$\vec{E} = -\vec{\nabla}\phi$
R
$\frac{\partial T}{\partial z} = 0$
$\vec{E} = \hat{z}E_0$
$\vec{n}\cdot\vec{E} = 0$
h
γ
$\vec{n}\cdot\vec{E} = 0$
$C_V\frac{\partial T}{\partial t} = \frac{1}{A}\frac{\partial}{\partial z}\left[A\kappa\frac{\partial T}{\partial z}\right] + \langle p\rangle$
$\mathrm{T} = 300K$
$\Phi = 0$

Figure 1: Schematic of the partial differential equations and their corresponding boundary conditions used for a nano-tip modeling in the thermal runaway study (Kyritsakis et al., 2018)

grades the overall performance of the project [18]. Vacuum breakdown is a known and observed phenomenon, but its mechanism is not well understood yet. For this reason, extensive research has been initiated to study the plasma formation during vacuum breakdown.

As a result of one such study, a multi-physics model of vacuum breakdown has been developed, which concurrently simulates the electric field and temperature distribution on a nano-tip and its changing shape by dynamically recalculating the mesh around the tip and positions of atoms [3], [5], [19], [20]. Figure 1 schematically illustrates the nano-tip with differential equations and boundary conditions used to model this physical phenomenon. The specifics of the simulation model and physics behind it are out of the scope of this work; however, to implement the model, a C++ software called FEMOCS was developed, which combines molecular dynamics (MD) with the finite element method (FEM) in a single framework [4]. FEMOCS itself and the visualization approach for the results of its simulations is under investigation in this thesis.

The unique feature of FEMOCS is that it extends an atomistic simulation by importing atoms' locations of a nanostructure, building a mesh from the imported atoms, and executing continuous-space calculations around this dynamically generated mesh at each simulation step and further returning the solution back to the atomistic simulation [4]. Figure 2

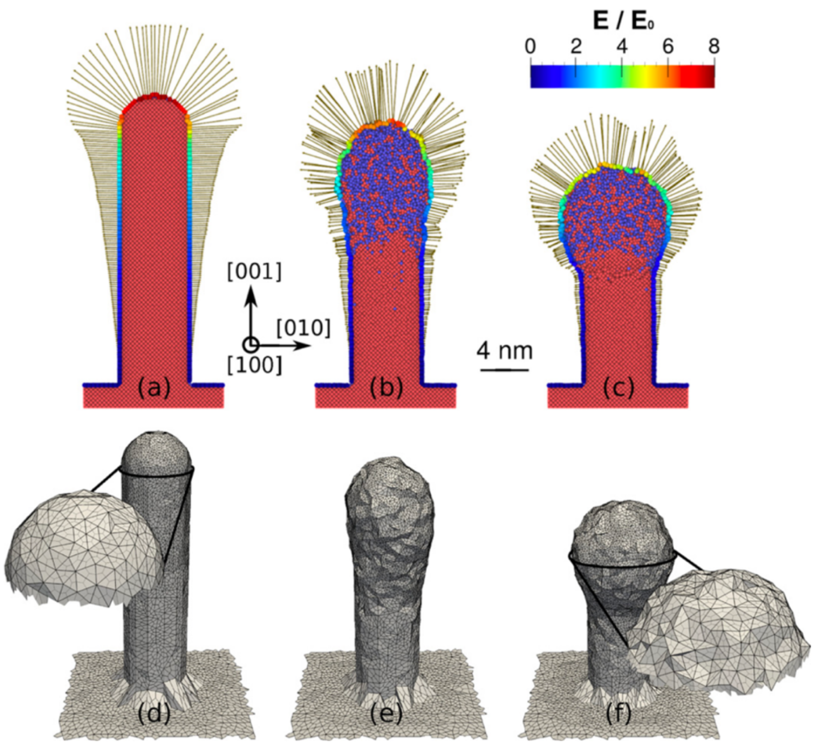


Figure 2: (a)–(c): atomistic simulation of the nano-tip; (d)−(f): continuous-space simulation represented as the surface of the mesh at the same time steps (Veske et al., 2018)

shows two sets of images: on (a)–(c) we observe the result of the atomistic simulation with the electric field calculations; and on (d)–(f) we see the dynamically generated mesh at the corresponding time steps, which is used for the calculation of the heat distribution in the system.

Currently, the FEMOCS visualization workflow consists of three general steps (**Error! Reference source not found.**): pre-processing during which an input describing an initial state of the model is provided; simulation, which is a computation of the next state of the model for multiple iterations; post-processing which is any further analysis and visualization of results saved during the simulation. This is a straightforward and standard workflow in scientific visualization, but it has notable drawbacks for large scale long-running and parallel simulations, such as the data transfer bottleneck between the simulation and post-processing steps, disk I/O operations, and in some cases, long waiting time between the start of the simulation and the final post-processing step, when a user gets graphical visualization of results [21]. To tackle such problems, the so-called *in situ* or on-the-fly processing approach has recently started being introduced in simulation software [22], [23].

In situ visualization removes the data transfer bottleneck and disk I/O operations problem by eliminating the need to write output data to persistent storage before processing, as visualization and data analysis can be done while the simulation is still running [8]. This

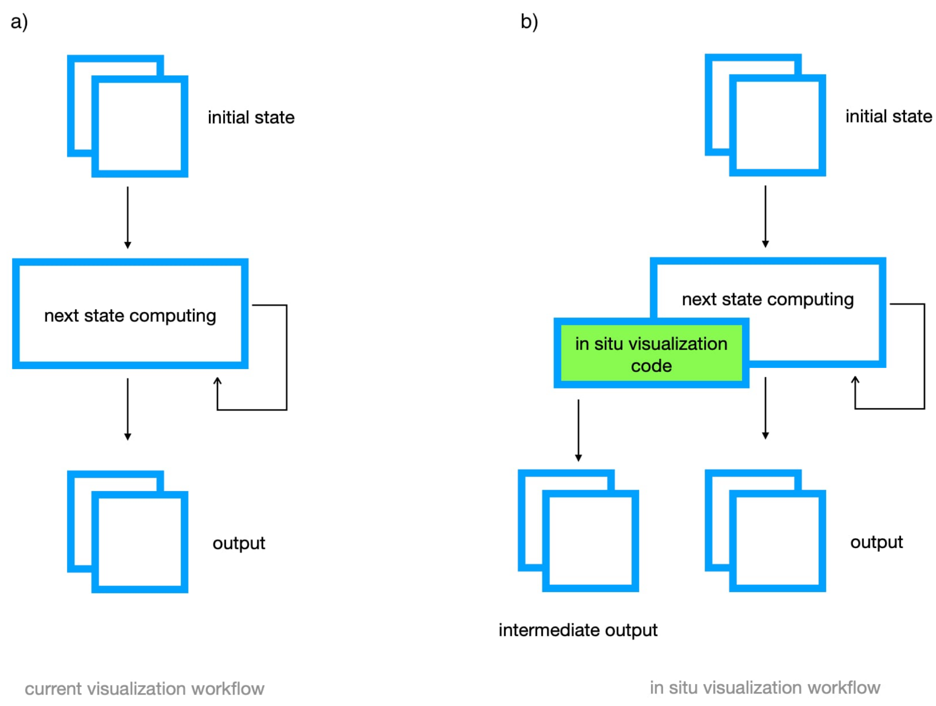


Figure 3: Comparison of the common visualization workflow for FEMOCS with the *in situ* approach

approach also significantly shortens the waiting time between the start and the actual visualization, as one gets results almost immediately. Additionally, if there is still a need to save analytical and graphical results on the disk, on-the-fly visualization reduces data as visualization techniques usually tend to compress data [8].

2.2 Motivation

As was mentioned before, the current FEMOCS' visualization workflow uses a post-processing approach for visualization, and it does not allow real-time simulation debugging and visualization steering. A user usually waits until a simulation is over to visualize it, then inspects it and, if the simulation needs additional adjustments, updates some initial simulation parameters and runs the simulation all over again. If the simulation requires significant computational power, it can run for days and weeks, making the simulation development and debugging a long and inconvenient process. Besides that, the post-processing approach introduces a need to write data to the disk and store intermediate simulation data, which increases the runtime and requires additional resources. Again, it can require terabytes of additional disk space in computationally intensive simulations, which increases the overall simulation cost.

The main focus of this work is to introduce real-time *visualization* capabilities to FEMOCS. To improve the simulation data distribution workflow significantly, it is advised to look closer at ADIOS, a library designed for extreme-scale I/O operations with a straightforward API and different transport possibilities, such as POSIX IO, MPI-IO, DataTap asynchronous IO [24].

3 IN SITU VISUALIZATION

3.1 In situ method

In situ processing or co-processing, in comparison to post-processing, is not a new technique, and it is known since as far as 1964 [8]. The idea behind the *in situ* method is to move the processing step closer to calculations without waiting until the simulation's end. Co-processing introduces computational interactivity to an application, computational monitoring, and simulation steering [8]. The *in situ* method can generate graphical output during a running simulation, providing a way to receive immediate feedback [9]. Often, scientific problems require a researcher to look at a problem, at the simulation data and its visualization from different angles, literally, if we speak about a 3D-scene, or, figuratively, when we speak about different types of visualization, such as 2D or 3D graphics, series plots, histograms, tables. One might use one type of simulation data representation, e.g., a 3D scene with an unstructured grid but multiple fields, like the temperature and electric field. A field itself can be represented as scalar or vector data. The *visualization steering* technique provides a way to change the graphical representation of the simulation data on-the-fly while the simulation is running [22]. Thus the in situ *visualization* is a part of the *in situ* processing method, which can be used besides visualization for real-time simulation data analysis, data reduction, feature extraction, index generation for search systems, and other purposes [22].

In situ visualization has become more prominent recently because of the ever-increasing scale of modern scientific simulations [2]. Extreme-scale simulations benefit significantly from *in situ* visualization because of the data transfer reduction and quicker visual feedback from a simulation, improving the user experience and getting insights from the data quicker, which is the main idea behind scientific visualization [25].

The term *in situ* might be ambiguous because there are multiple similar terms: "concurrent processing", meaning data processing as a simulation is running; "co-processing" refers to the tight coupling between the simulation and visualization code; "runtime visualization" means visualization in place — to bring clarity *The In Situ Terminology Project* has been initiated [26]. As a result, "six axes" or six criteria have been introduced to describe an *in situ* system better: the integration type, proximity, access, division of execution (synchronization), operation controls, output type [8], [26] — these criteria are used further to describe the used *in situ* system.

3.2 ParaView Catalyst

In situ visualization can be achieved simply by placing the visualization code after each simulation iteration. However, to achieve the ability to choose *interactively* what and how to visualize the data, to get a more flexible and reliable system, a proper and robust framework is needed. This is a reason for not implementing a custom solution but to use an existing *in situ* visualization software. ParaView Catalyst (Catalyst) has been chosen for implementing *in situ* visualization capabilities for FEMOCS in this work. ParaView on its own is a well-known and widely used software for scientific visualization, and Catalyst allows a smooth integration of the *in situ* code with the visualization tool. Catalyst's unique feature is the ParaView *Live* session, which allows on-the-fly debugging of the simulation, analysis, and visualization steering[8]. For a visualization pipeline, Catalyst accepts its description in C++ and Python. For this work, the visualization pipeline was described in Python as this language is widely used in the academic community.

To describe Catalyst, we use six criteria provided by *The In Situ Terminology Project* [8], [26]. Catalyst is a dedicated API, and it uses the application-aware style of integration, meaning the simulation code directly calls Catalyst API. The visualization code is located on the same node as the simulation code, it is the closest proximity possible, and it allows the direct memory access to data for visualization routines without requiring a deep copy of the data; thus, a shallow copy of the data can be made for better performance [26]. However, the visualization itself happens on the node or nodes where the ParaView server is running. Because Catalyst runs synchronously to the simulation, its overhead might be significant, depending on its integration with the simulation code. *Karimabadi et al.* [27] show the Catalyst's overhead of 20−30% on running time. However, the runtime overhead appeared to be much less in this work and is presented in the *Timing analysis* chapter.

Regarding the operation controls, Catalyst allows both automatic and human-in-the-loop ways of controlling the visualization. Automatic control is allowed by a visualization pipeline script, which defines what data to consider and how to visualize it, while human-in-the-loop control can be achieved through a ParaView Live session that enables interactive visualization control and simulation pausing. Finally, as an output, Catalyst provides a wide range of possibilities: it supports an ability to extract a subset of data, allows *in situ* transformation, data derivation, e.g., polygonal data geometry, and the extraction of explorable images [8].

ParaView and ParaView Catalyst use the Visualization Toolkit (VTK) under the hood, and it is vital to introduce VTK because all underlying data structures, which were used in the intermediate code between the simulation and ParaView in this work, used the VTK library. VTK is written in the C++ language; it is a cross-platform library, which uses CMake [28] for the building process, and provides foundational data structures and algorithms as well as utility tools for building scientific visualization software [29]. VTK has a rich history of 27 years of work and development; it is widely used in different projects, such as ParaView, VisIt, Avogadro2, tomviz, and others; and it provides Python, Tcl/Tk, and Java wrappers to extend the usage and adoption of the toolkit [29].

ParaView Catalyst allows to extract and visualize simulation data while it is being computed, and the ParaView *Live* session feature makes it possible to have an on-the-fly preview of what is happening in the simulation at the moment, as well as to pause the simulation at a particular step and inspect the data, visualization parameters or to change a visualization technique [30]. However, as the complexity and scale of simulations increase, the size of already visualized data in the form of images can take up many terabytes of disk space, so it becomes cumbersome to manage the imagery using only an OS file system and the already mentioned problem with data transfer is still there, even so in this case it is related not to the simulation data, but to the visualization of the simulation data. For that reason, an image database can be a better approach to managing the visualization output. There is a practice to make image collections and archives for further processing and distribution, as an example, authors of [2] mention the CESM (Community Earth System Model), an archive of datasets from "Earth's past, present, and future climate states" simulations [31], but even this can be improved. An image-based approach for extreme-scale *in situ* visualization has been developed [2] and is implemented in ParaView *Cinema* [32]. This approach allows the collection of simulation images together with its metadata, and each image has a URI [2], which makes it straightforward to locate and display an image wherever it is located; Cinema makes it possible to create an interactive exploration database with metadata searching and content querying using GUI [2].

3.3 Drawbacks and limitations

Having the general description and benefits of the *in situ* visualization technique, it is worth mentioning possible drawbacks and challenges. The work by Kwan-Liu Ma [22] points to the scientific community's reluctance to adopt the *in situ* visualization method because of the expensive supercomputer time; thus, researchers choose to move the visualization part to other less expensive computers. Integrating the real-time visualization into a simulation might also require a significant effort [22]. Even though companies and institutions behind *in situ* software do their best to simplify a programming interface and documentation, the task of integrating the *in situ* visualization into a simulation is still not trivial. It requires expertise in programming languages, building and debugging tools, code management, and version control systems.

Relating to the Catalyst specific challenges, as ParaView uses VTK for the data representation and processing, it might be unclear at times what VTK data structure is better suited for the simulation data, thus understanding of VTK internals is essential in order to map the simulation data to VTK efficiently [33]. Besides knowing VTK, a simulation developer should also be familiar with the ParaView API (*vtkCPProcessor*, *vtkCPPipeline*, *vtkCPInputDataDescription,* and *vtkCPDataDescription* classes*)*, therefore introducing a dependency into the simulation code for a particular version of ParaView [33]. To stress this more, a simulation that incorporates the *in situ* visualization Catalyst API depends on the same version of ParaView: if one uses ParaView Catalyst API v.5.8.2, as was done in this work, a user can use only the ParaView v5.8.* server and client for a visualization task. To build an adaptor library for converting the simulation data into the data readable by ParaView, a simulation developer needs to compile the ParaView SDK from the source, further increasing the development complexity [33]. However, these issues have been addressed in the newer version of ParaView v5.9 [33].

In conclusion, given the benefits and drawbacks of the *in situ* visualization method, it is still a viable and attractive solution for implementing on-the-fly visualization capabilities, especially for extreme-scale simulations. It requires some software development and management expertise and might take significant time to implement an adaptor library between a simulation and ParaView Catalyst. However, *in situ* frameworks' developers continue to improve their software reducing unnecessary dependencies and making the simulation development process more straightforward.

4 SOFTWARE DEVELOPMENT

To implement *in situ* visualization capabilities for FEMOCS, the C++ library was created to make an adaptor between data structures used in FEMOCS and the ParaView Catalyst programming interface. Catalyst uses VTK-based data structures for the simulation data storage and operations. It also uses ParaView API to pass the simulation data to the ParaView server, which serves data to the ParaView client. An important feature of FEMOCS is that it uses LAMMPS [34] for the MD simulation; therefore, we needed to create a LAMMPS extension to integrate the adaptor library with FEMOCS into a single workflow.

ParaView has a notion of *the visualization pipeline* [35], which is how a user implements visualization of a particular piece of data. A visualization pipeline consists of data sources, filters, data modifiers, and 3D-scene-related parameters, such as camera position, lighting. The pipeline defines what and how would be visualized at the end. Catalyst allows providing the visualization pipeline in C++ or as a separate Python script.

4.1 Development environment

During this work, two different computers were used:

- For the software writing, management, and containerization: macOS 11 with 2.6 GHz 6-Core Intel® Core i7 CPU and 16 GB 2667 MHz DDR4 memory; Wi-Fi AirPort Extreme 802.11ac.
- For the software building, testing, and benchmarking: Ubuntu 20.04 with 3.00 GHz 10-Core Intel® Xeon® E5-2690 v2 CPU and 264 GB 1866 MHz DDR3, gcc 9.3.0; Ethernet Intel Corporation 82574L Gigabit Network Connection.

The C++ language was a requirement from the Catalyst's and FEMOCS sides, as the libraries are written in C++ and provide the corresponding API.

Python 3 was used for the ParaView visualization pipeline script. It is possible to use C++ for the task as well, but Python appeared to be a more desirable way because the language itself requires less programming expertise, and the ParaView GUI provides a way to automatically export the visualization pipeline Python script.

The latest versions of *make* (4.2.1), *cmake* (3.16.3), *gcc-9* available for Ubuntu 20.04 at the moment of writing this work, the winter–spring of 2021, were used. There was no need for specialized debugging software, but to inspect the intermediate states of the software print statements and logs from FEMOCS and LAMMPS were used. For code management and version control, the *git* software [git version 2.25.1 and git version 2.21.0 (Apple Git-122.2)] together with the GitHub and GitLab services were used [36]–[38].

To document all dependencies and the process of compilation and linking of dependencies, the containerization software named *Docker* (version 20.10.5) was used with the release notes available at [39]. The Dockerfile-script has a straightforward syntax, easy to read by a person unfamiliar with Docker, and this tool introduces transparency into the most tedious part of the work: figuring out the dependencies and steps needed to build the target software — which took a significant amount of time and effort. For that reason, it became evident that a scripted and, better, automated approach to building is necessary to simplify further work. *Appendices I−III* provide the relevant Docker scripts.

4.2 Adaptor Library

The adaptor library is a middleman between FEMOCS and ParaView using VTK data structures. It is also used in the LAMMPS extension. In the beginning, we need to get familiar with the programming interfaces of these libraries.

ParaView Catalyst is the *in situ* infrastructure, which provides a set of tools for enabling *ad hoc* processing for a simulation. Catalyst consists of the *CoProcessing* C++ library [40], [41], *coprocessing* Python module [42], and the *Live Insitu* C++ module [43]. The C++ *CoProcessing* module was used directly to pass simulation data to ParaView. The Python *coprocessing* module was used in a visualization pipeline script composition, and the *Live Insitu* C++ module was not used directly, as this functionality is implemented in ParaView GUI and can be called through the toolbar's Catalyst menu commands.

The C++ *CoProcessing* module

The *CoProcessing* C++ module of version 5.8.* was used, and KitWare Inc. provides the documentation at the ParaView C++ Reference website [40]. The *vtkCPProcessor* class provided means to initialize a Catalyst session, finalize it (to free the used resources) and register a visualization pipeline Python script by providing a path to the script's location. An instance of *vtkCPProcessor* is responsible for determining if the co-processing must be done at this time step given the current time step and time, as well as it is responsible for actually calling the co-processing routine on the ParaView side. The *vtkCPPythonScriptPipeline* class was used during the initialization step to register a visualization pipeline given a location of an input Python script. The *vtkCPDataDescription* class was used in the co-processing step, and it allowed to encapsulate the simulation data and provide the current time step and time of the simulation for ParaView.

The Python *coprocessing* module

The Python *coprocessing* module of version 5.8.* was used, and the documentation is provided at the ParaView Python Reference website [42]. The Python module, in general, is a wrapper around the C++ version and provides bindings to the C++ API through the Python language making the ParaView library accessible for a broader audience. The Python *coprocessing* module was used to compose the visualization pipeline script, which should be provided during the initialization step of the adaptor library.

The *paraview.coprocessing.CoProcessor* class provides all the functionality for co-processing routines. It allows specifying data sources (in terms of VTK, it is the source of the data that does not have any input data and only has an output [35]), which come from the adaptor library and are included in the instance of *vtkCPDataDescription*. The *coprocessing.CoProcessor* class gives API for writing data to the disk if needed; it defines the update frequency for ParaView and enables the Live feature of ParaView for the real-time simulation visualization.

The *paraview.simple* Python module provides more general means of setting up a 3D scene, adding lighting, a camera, and its position, making slices, or adding filters to the pipeline. However, this module was not used in this work as it is optional, and all needed manipulation can be done through ParaView's graphical interface.

The FEMOCS interface

From FEMOCS, the adaptor library needs the simulation data: a mesh of the nano-tip and the field data for atoms, the temperature, and the electric field. Developers of FEMOCS made it straightforward to extract this data with one overloaded method call (i.e., the same method with different signatures) on the *femocs::FEMOCS* class' instance:

- *int export_data(double* data, const int n_points, const string& data_type)*, which was used for exporting the scalar and vector field data;
- *int export_data(const int** data, const string& data_type)* was used for exporting atoms' locations in 3D space;
- *int export_data(const double** data, const string& data_type)* was used for exporting mesh nodes.

The Adaptor library interface

The adaptor library is the core of the solution for the aim of this work. The architecture of the solution (Figure 4) is based on the work by *Fabian et al.* [41], in which authors propose to write an adaptor between the simulation code and ParaView using the ParaView CoProcessing library. The solution presented in this works similar to the one proposed by *Fabian et al.* [41] with the addition of one function: *void ImportAtoms(double **atoms, int numAtoms)*. This function was introduced because the current architecture of the LAMMPS extension for FEMOCS did not allow to import atoms at the same step, where *CoProcess(...)* is called; thus, we needed to import atoms' locations at a different place in the LAMMPS extension. More about the LAMMPS extension is covered in the *LAMMPS extension* section. The whole interface of the adaptor library, the C++ header file, is listed in *Appendix IV*. The logic behind the library can be described with four steps in the order of calling it from the LAMMPS extension:

1. Initialization of the library, where we register a location of the visualization pipeline Python script, initiate vtkCPProcessor, and some internal variables.
2. Importing of the atoms' locations, where locations are provided by LAMMPS.

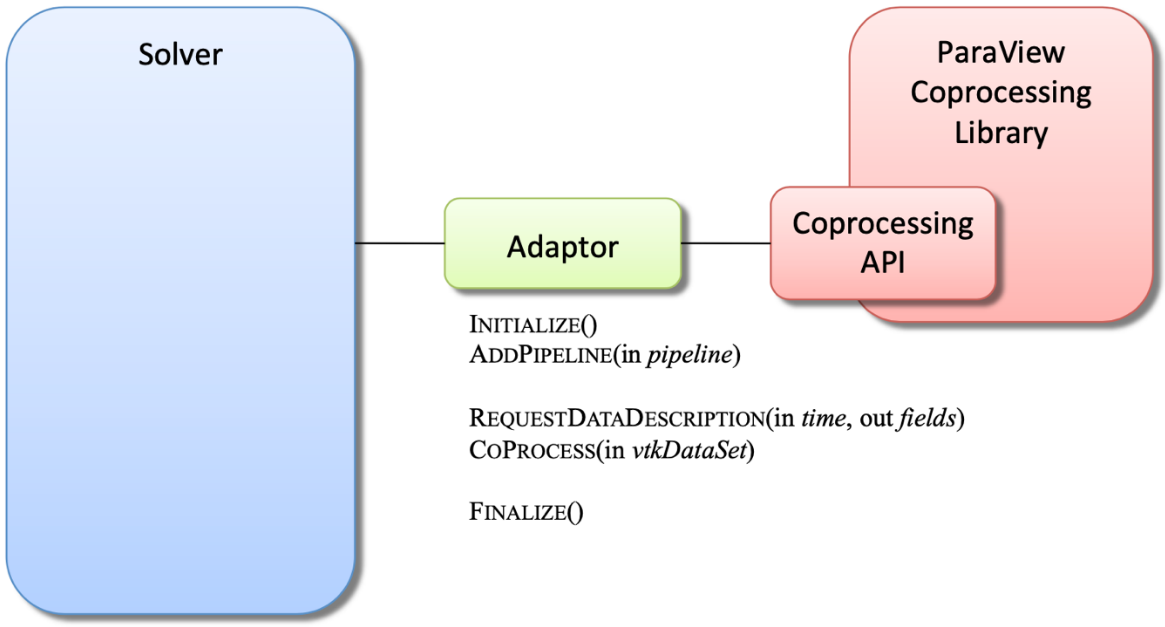

Solver
ParaView Coprocessing Library
Adaptor
Coprocessing API
INITIALIZE()
ADDPIPELINE(in *pipeline*)
REQUESTDATADESCRIPTION(in *time*, out *fields*)
COPROCESS(in *vtkDataSet*)
FINALIZE()

Figure 4: The architecture of the solution to the integration of the ParaView CoProcessing library into the simulation code (Fabian et al., 2011)

3. Co-processing, which includes the export of the mesh and field data, and the call of the underlying ParaView's co-processing routine. In this work, two unstructured grids were used, one is with the mesh, and the other contained the atomistic data, the field data, the temperature, and electric field normals and vectors — the *VTK* section describes how this data was converted from FEMOCS into VTK data structure for ParaView's use.
4. Finalization, which is freeing up of all used resources. Because of the use of the new feature of VTK, the *vtkNew* template [44], most of the cleaning up is done automatically.

Figure 5 shows the data flow from a program to program and the place of the adaptor and other pieces of the solution in a single diagram. LAMMPS is the driver of the simulation and calls the FEMOCS/Catalyst extension, which calls in its turn the FEMOCS and adaptor libraries. The adaptor then passes the data to ParaView through Catalyst. The user of the simulation interacts with ParaView through the ParaView client with GUI.

The visualization pipeline

The visualization pipeline Python script is a way to provide information on what and when should be extracted and saved from the simulation, as well as on which portion of the data

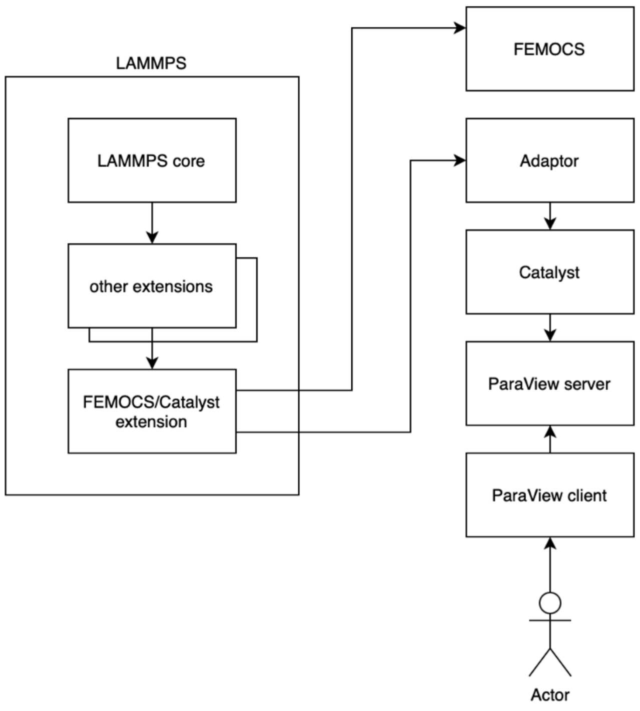


Figure 5: The dataflow between programs involved in the *in situ* solution for FEMOCS

the visualization algorithms should be applied to [41]. There are two ways how a user can compose the pipeline script: manually by writing the script using the ParaView *paraview.simple* and *paraview.coprocessing* modules for Python [42], [45]; the other method is through a ParaView's plugin, which allows using a sample of the simulation data in order to set up a scene, position a camera, apply the slicing or filtering of the data, and export these operations to a Python script [41]. However, during the work on the adaptor library, the automatic method through the plugin behaved unreliably because the resulted scripts had not worked straight away without modifications — the scripts required a significant rewrite; otherwise, the Python interpreter threw critical errors, and it was not able to proceed. Thus, the first manual approach in the pipeline script composition was used.

Building

The compilation and linking of the adaptor library, i.e., the building phase, is executed mainly using CMake. In the CMake configuration file, *CMakeLists.txt*, all the dependencies are specified together with compilation flags and two building targets: the library itself and the demo application for testing and debugging purposes. Besides that, the Make system was used to simplify the calling of CMake because the command-line interface command in Bash's shell for CMake is quite long.

The source code of the adaptor library was open-sourced and published on GitHub at [46] and DockerHub at [47]. The use of GitHub greatly simplifies the collaborative development process and provides access to the work results to everybody with the internet connection.

4.3 VTK

ParaView uses The Visualization Toolkit (VTK) for scientific data processing [29]; therefore, VTK's data types are essential for the adaptor library, as they serve as a bridge between the simulation and Catalyst.

VTK has a wide range of different data types described in detail by *Schroeder et al.* [35]. The simulation data in this work was organized around two datasets, and both are represented by an unstructured grid. The unstructured grid data type is the most general type of dataset, it is less computationally efficient than a structured grid, but it gives the freedom to represent any topology and geometry that may arise during the simulation [35].

An unstructured grid for FEM calculations consisted of mesh nodes with XYZ coordinates and cells represented by quadrangles. An unstructured grid for the MD results included atoms' XYZ coordinates, and cells were represented as vertices. Besides that, the MD dataset had the field data: an array of scalars for the temperature at each atom, an array for the electric field normals, and an array of vectors of the electric field.

4.4 LAMMPS extension

LAMMPS [34] is the driver of the simulation. It starts the simulation given a configuration file the description of the experiment, it does molecular dynamics calculations and calls FEMOCS as an extension (a "fix" in terms of LAMMPS), i.e., as an external program, which given the atomistic data, generates a mesh and calculates the temperature and electric field distribution around the nano-tip. After that, FEMOCS passes the data back to LAMMPS to finish an iteration and continue to the next time step.

To integrate the real-time visualization into FEMOCS, the current LAMMPS extension for FEMOCS has been modified to call not only FEMOCS API but the adaptor's library co-processing API as well. As LAMMPS controls the execution of the whole simulation, it is an obvious choice of the adaptor library incorporation.

To write an extension for LAMMPS, a developer needs to create a new class inheriting from the public *Fix* class and implement the needed logic in this class [48]. *Setmask* is the only required method that has to be implemented to satisfy the *Fix* interface. The rest depends on a particular use case.

To integrate the adaptor library into the existing FEMOCS extension for LAMMPS [49], there was no need to change much, but to add additional calls for the adaptor library API:

- During the fix class initialization, the adaptor library is initialized as well.
- During the *post_force()* step, the adaptor library is used to import the atoms' data.
- During the *end_of_step()* step, the adaptor library is called to co-process the simulation data.
- During the class destruction, the adaptor library instance is also destructed.

5 TESTING AND BENCHMARKING

5.1 Testing

The testing of the entire system — LAMMPS, FEMOCS, the adaptor library, a ParaView server and client, a Catalyst server (which is part of the ParaView server) — was conducted using both of the development machines mentioned in the *Development environment* section. The Ubuntu machine ran the simulation (LAMMPS, FEMOCS, the adaptor library), a ParaView server, a ParaView GUI client. The macOS machine established a network connection through the University of Tartu VPN to the Ubuntu computer and connected through an SSH tunnel to the :3389 port using the Parallels Client application for a remote desktop connection. Then the ParaView client connected to the ParaView server and the Catalyst server.

The test simulation configuration and the visualization pipeline script are available online at [50]. The *in.lmp* file contains the description of the simulation for LAMMPS, and on line 109 (the only modified line for this work) the *femocs/catalyst* LAMMPS extension is called providing the path to *in.fem*, FEMOCS-specific configuration, and the visualization pipeline script named *paraview_pipeline.py*.

From the *in.fem* file, we see that the simulation uses Cu atoms for the atomistic modeling and the time step of 4.0 fs. The simulation length was 200 time steps.

5.2 Timing analysis

To get an insight into the impact of the adaptor library on the rest of the simulation code, the following time measuring procedure has been conducted. Using the *chrono* module of the standard C++ library, we measured the running time of the whole simulation and the *CoProcess(...)* function of the adaptor library, which was executed 200 times, one time per each time step. For each of the *CoProcess(...)* execution, a duration was saved into a C++ vector of values, and at the end of the simulation, the measurements were saved to the disk in the CSV format. For further processing using Python, we calculated the total duration of *CoProcess(...)*, and given the total runtime for the whole code, the average runtime of the rest of the code per time step was calculated. The results are plotted in Figure 6. We are not interested in duration differences between time steps for the adaptor library or the rest of the simulation; for that a more robust benchmarking should be done. We are interested more in the overall picture about the impact of the adaptor, and from the plot, we see that, on average, the difference for each timestep is 10^3. Given the total time of the whole simulation (623 s)

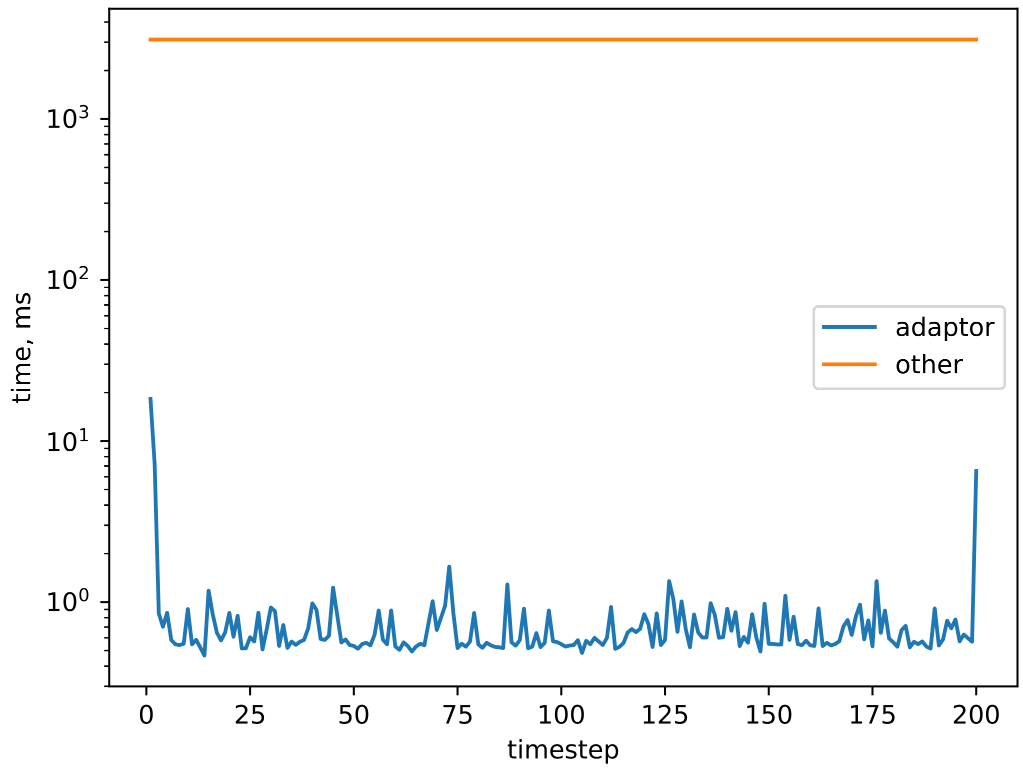


Figure 6: Measuring durations of *CoProcess(...)* of the adaptor library and comparing it to the rest of the simulation code

and the total runtime of the *CoProcess(...)* function (163 ms), and neglecting the impact of initialization and ImportAtoms(…) (which copies only a single pointer), the impact of the adaptor library is 0.03%. This means that the simulation was running for 99.97% of the time, while the adaptor library with *in situ* visualization routines ran for about 0.03% of the total time.

Even though this is far from the vigorous benchmarking and it was executed only once, this test gives us an insight that the impact of the adaptor library on the rest of the code is insignificant; thus, we can propose to use the library for long-running simulations.

6 DISCUSSION

6.1 Conclusion

This work presents the current visualization workflow for FEMOCS and an extended one, using the nano-tip simulation of vacuum breakdown as an example. The extended workflow allowed real-time monitoring and steering of the visualization; it reduced time to the first visual feedback, eliminated the need to store intermediate simulation data, and showed very little overhead in the test run.

The solution was implemented as *the C++ adaptor library* that was integrated into the LAMMPS extension for FEMOCS as well, allowing a user to choose between using only FEMOCS or FEMOCS with ParaView Catalyst. The implementation of the solution was based on the already developed architecture by *Fabian et al.* [41] and on the ParaView Co-Processing library and its dependencies. The little overhead demonstrated in the *Timing analysis* chapter suggests that the resulted solution can be used without being concerned with slowing down the simulation calculations.

Besides the nano-tip simulation, FEMOCS itself can be used in any simulation, which requires atomistic and continuous-space calculations with automatic mesh recalculations at each iteration. A new simulation would benefit from the adaptor library as well; however, some adjustments to the adaptor might still be needed, depending on the requirements of the new simulation: one might add or remove the field data, export different types of data from FEMOCS, or a different cell topology.

6.2 Future work

The current approach to starting all the needed services and programs for the system uses mostly the command-line interface of Linux. This can be improved by providing a script, which can encapsulate multiple commands into one. However, a more intuitive approach can attract more potential users to FEMOCS and its newly designed Catalyst adaptor — a web UI can greatly simplify the system control. ParaViewWeb [51] provides the JavaScript API that can be used to provide visualization control through a web browser instead of the ParaView GUI client.

The adaptor library itself requires more rigorous benchmarking to fully understand the performance implications of the library in different conditions. The memory consumption analysis has not been done yet for the adaptor. Running the adaptor in HPC can potentially reveal some additional challenges. Extensive saving of intermediate visualization to the disk may introduce more considerable performance overhead. As was mentioned in *Rationale*, for further improvements related to disk I/O operations, the well developed ADIOS framework can be used.

One more way of improving the visualization workflow has not been touched in this work. The resulting new *in situ* visualization workflow allows real-time monitoring, but if a user wants to save the intermediate imagery and have the ability to navigate in time between graphics easily, it is possible only through the Linux file system at the moment. However, there is a more convenient and enabling solution exists — ParaView Cinema. Cinema is an exploratory distributed database that allows the storing of visual data and metadata [32]. Having metadata of each visualization may allow a more interactive and comfortable way of navigating through a large number of images.

Besides that, ParaView developers starting from version 5.9 have changed the Catalyst API significantly, which was announced in January 2021 [52]. The new Catalyst API was redesigned to loosen the currently tightly coupled *in situ* visualization code with simulation code. Therefore, for newer versions of ParaView, the solution presented in this work needs to be revisited as well.

7 SUMMARY

As the main goal of this work was to implement *in situ* visualization capabilities for FEMOCS, we can conclude that the goal was successfully achieved. The resulted adaptor library allows on-the-fly visualization monitoring, debugging and steering, and improves the visualization workflow of a FEMOCS user in general by giving more options to visualization and eliminating some drawbacks of the post-processing approach to the visualization.

The solution in this work is presented as the C++ adaptor library and the LAMMPS extension, which integrates the adaptor into the current FEMOCS approach. The testing of the full system was conducted on the nano-tip simulation of vacuum breakdown, and the adaptor performance impact on the rest of the simulation is shown to be insignificant.

REFERENCES

[1] "November 2020 | TOP500." https://top500.org/lists/top500/2020/11/ (accessed May 11, 2021).

[2] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An Image-Based Approach to Extreme Scale in Situ Visualization and Analysis," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, Nov. 2014, pp. 424–434. doi: 10.1109/SC.2014.40.

[3] A. Kyritsakis, M. Veske, K. Eimre, V. Zadin, and F. Djurabekova, "Thermal runaway of metal nano-tips during intense electron emission," *J. Phys. D: Appl. Phys.*, vol. 51, no. 22, p. 225203, Jun. 2018, doi: 10.1088/1361-6463/aac03b.

[4] M. Veske, A. Kyritsakis, K. Eimre, V. Zadin, A. Aabloo, and F. Djurabekova, "Dynamic coupling of a finite element solver to large-scale atomistic simulations," *Journal of Computational Physics*, vol. 367, pp. 279–294, Aug. 2018, doi: 10.1016/j.jcp.2018.04.031.

[5] M. Veske, A. Kyritsakis, F. Djurabekova, K. N. Sjobak, A. Aabloo, and V. Zadin, "Dynamic coupling between particle-in-cell and atomistic simulations," *Phys. Rev. E*, vol. 101, no. 5, p. 053307, May 2020, doi: 10.1103/PhysRevE.101.053307.

[6] M. Veske, A. Kyritsakis, F. Djurabekova, K. N. Sjobak, A. Aabloo, and V. Zadin, "Dynamic coupling between particle-in-cell and atomistic simulations," *Phys. Rev. E*, vol. 101, no. 5, p. 053307, May 2020, doi: 10.1103/PhysRevE.101.053307.

[7] M. Veske, *FEMOCS — Finite Elements on Crystal Surfaces*. 2021. Accessed: May 18, 2021. [Online]. Available: https://github.com/veskem/femocs

[8] A. C. Bauer *et al.*, "In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms," *Computer Graphics Forum*, vol. 35, no. 3, pp. 577–597, Jun. 2016, doi: 10.1111/cgf.12930.

[9] K.-L. Ma, "RUNTIME VOLUME VISUALIZATION FOR PARALLEL CFD," *NASA Langley Research Center*, p. 20, 1995.

[10] V. Zadin *et al.*, "Simulations of surface stress effects in nanoscale single crystals," *Modelling Simul. Mater. Sci. Eng.*, vol. 26, no. 3, p. 035006, Feb. 2018, doi: 10.1088/1361-651X/aaa928.

[11] J. Liu, H. Vázquez Muíños, K. Nordlund, and F. Djurabekova, "Molecular dynamics simulation of the effects of swift heavy ion irradiation on multilayer graphene and diamond-like carbon," *Applied Surface Science*, vol. 527, p. 146495, Oct. 2020, doi: 10.1016/j.apsusc.2020.146495.

[14] CERN, "CERN Yellow Reports: Monographs, Vol 2 (2018): The Compact Linear e+e− Collider (CLIC) : 2018 Summary Report," p. 58.39 MB, Jan. 2018, doi: 10.23731/CYRM-2018-002.

[17] R. L. Boxman, D. M. Sanders, and P. J. Martin, Eds., *Handbook of vacuum arc science and technology: fundamentals and applications*. Park Ridge, N.J., U.S.A: Noyes Publications, 1995.

[18] T. K. Charles and others, "The Compact Linear Collider (CLIC) - 2018 Summary Report," *CERN Yellow Rep. Monogr.*, vol. 1802, pp. 1–98, 2018, doi: 10.23731/CYRM-2018-002.

[19] F. Djurabekova, S. Parviainen, A. Pohjonen, and K. Nordlund, "Atomistic modeling of metal surfaces under electric fields: Direct coupling of electric fields to a molecular dynamics algorithm," *Physical Review E*, vol. 83, no. 2, p. 026704, 2011, doi: 10.1103/PhysRevE.83.026704.

[20] M. Veske, A. Kyritsakis, K. Eimre, V. Zadin, A. Aabloo, and F. Djurabekova, "Dynamic coupling of a finite element solver to large-scale atomistic simulations," *Journal of Computational Physics*, vol. 367, pp. 279–294, 2018, doi: 10.1016/j.jcp.2018.04.031.

[21] U. Ayachit *et al.*, "ParaView Catalyst: Enabling In Situ Data Analysis and Visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, Austin TX USA, Nov. 2015, pp. 25–29. doi: 10.1145/2828612.2828624.

[22] Kwan-Liu Ma, "In Situ Visualization at Extreme Scale: Challenges and Opportunities," *IEEE Comput. Grap. Appl.*, vol. 29, no. 6, pp. 14–19, Nov. 2009, doi: 10.1109/MCG.2009.120.

[23] Hongfeng Yu, Chaoli Wang, R. W. Grout, J. H. Chen, and Kwan-Liu Ma, "In Situ Visualization for Large-Scale Combustion Simulations," *IEEE Comput. Grap. Appl.*, vol. 30, no. 3, pp. 45–57, May 2010, doi: 10.1109/MCG.2010.55.

[24] J. Lofstead, S. Klasky, and K. Schwan, "Flexible IO and Integration for Scientific Codes Through The Adaptable IO System (ADIOS)," p. 10.

[25] B. H. McCormick, “Visualization in scientific computing,” *SIGBIO Newsl.*, vol. 10, no. 1, pp. 15–21, Mar. 1988, doi: 10.1145/43965.43966.

[26] W. Bethel and E. Gobbetti, “The In Situ Terminology Project,” presented at the Eurographics Symposium on Parallel Graphics and Visualization, 2016.

[27] H. Karimabadi, B. Loring, P. O’Leary, A. Majumdar, M. Tatineni, and B. Geveci, “In-situ visualization for global hybrid simulations,” in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, New York, NY, USA, Jul. 2013, pp. 1–8. doi: 10.1145/2484762.2484822.

[28] K. Martin, B. Hoffman, and A. Cedilnik, *Mastering CMake: a cross-platform build system ; covers installing and running CMake ; details converting existing build processes to CMake ; create powerful cross-platform build scripts*, 5. ed. Clifton Park, NY: Kitware, 2010.

[29] M. D. Hanwell, K. M. Martin, A. Chaudhary, and L. S. Avila, “The Visualization Toolkit (VTK): Rewriting the rendering code for modern graphics cards,” *SoftwareX*, vol. 1–2, pp. 9–12, Sep. 2015, doi: 10.1016/j.softx.2015.04.001.

[30] A. C. Bauer, B. Geveci, and W. Schroeder, *ParaView Catalyst User’s Guide*. Kitware Inc., 2018.

[31] CESM, “The Community Earth System Model,” *The National Science Foundation*, 2020. https://www.cesm.ucar.edu/models/ (accessed Apr. 22, 2021).

[32] S. McKenzie, S. Jourdain, and Z. Mullen, “ParaView Cinema: An Image-Based Approach to Extreme-Scale Data Analysis - Kitware Blog,” Sep. 10, 2014. https://blog.kitware.com/paraview-cinema-an-image-based-approach-to-extreme-scale-data-analysis/ (accessed Apr. 22, 2021).

[33] “ParaView Catalyst 5.9.* Doxygen Documentation,” Jan. 27, 2021. https://kitware.github.io/paraview-docs/latest/cxx/ParaViewCatalyst.html (accessed Apr. 29, 2021).

[34] S. Plimpton, “Fast Parallel Algorithms for Short-Range Molecular Dynamics,” *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, Mar. 1995, doi: 10.1006/jcph.1995.1039.

[35] W. Schroeder, K. Martin, and B. Lorensen, *The visualization toolkit: an object-oriented approach to 3D graphics ; [visualize data in 3D - medical, engineering or scientific ; build your own applications with C++, Tcl, Java or Python ; includes source code for VTK (supports Unix, Windows and Mac)*, 4. ed. Clifton Park, NY: Kitware, Inc, 2018.

[36] "Git," 2021. https://git-scm.com/ (accessed May 04, 2021).

[37] "Build software better, together," *GitHub*. https://github.com (accessed May 04, 2021).

[38] "GitLab: Iterate faster, innovate together," *GitLab*, 2021. https://about.gitlab.com/ (accessed May 04, 2021).

[39] "Docker Engine 20.10.5 release notes," *Docker Documentation*, Apr. 30, 2021. https://docs.docker.com/engine/release-notes/#20105 (accessed Apr. 30, 2021).

[40] "ParaView: ParaView CoProcessing," Aug. 05, 2020. https://kitware.github.io/paraview-docs/v5.8.1/cxx/group__CoProcessing.html (accessed May 04, 2021).

[41] N. Fabian *et al.*, "The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library," in *2011 IEEE Symposium on Large Data Analysis and Visualization*, Providence, RI, USA, Oct. 2011, pp. 89–96. doi: 10.1109/LDAV.2011.6092322.

[42] "coprocessing Module — ParaView/Python 5.8.1 documentation," 2013. https://kitware.github.io/paraview-docs/v5.8.1/python/paraview.coprocessing.html (accessed May 04, 2021).

[43] "ParaView: Live Insitu," Aug. 05, 2020. https://kitware.github.io/paraview-docs/v5.8.1/cxx/group__LiveInsitu.html (accessed May 04, 2021).

[44] "VTK: vtkNew< T > Class Template Reference," Apr. 24, 2021. https://vtk.org/doc/nightly/html/classvtkNew.html#details (accessed May 05, 2021).

[45] "simple Module — ParaView/Python 5.8.1 documentation," 2013. https://kitware.github.io/paraview-docs/v5.8.1/python/paraview.simple.html (accessed May 05, 2021).

[46] I. Suvorau, *The ParaView Catalyst adaptor library for FEMOCS*. 2021. Accessed: May 13, 2021. [Online]. Available: https://github.com/iharsuvorau/catalyzing-femocs

[47] I. Suvorau, "Docker container for FEMOCS with the ParaView Catalyst adaptor," 2021. https://hub.docker.com/r/nokal/femocs-catalyst (accessed May 13, 2021).

[48] "3.7. Fix styles — LAMMPS documentation." https://lammps.sandia.gov/doc/Modify_fix.html (accessed May 08, 2021).

[49] M. Veske, *FEMOCS extension for LAMMPS*. 2020. Accessed: May 13, 2021. [Online]. Available: https://github.com/veskem/lammps/tree/femocs/src/USER-FEMOCS

[50] "Test configuration for the FEMOCS nano-tip simulation with the Python visualization pipeline script for ParaView Catalyst," *GitHub*, 2021. https://github.com/iharsuvorau/lammps/tree/visual-femocs/examples/USER/femocs-catalyst (accessed May 13, 2021).

[51] "ParaViewWeb," *ParaViewWeb*, 2020. https://kitware.github.io/paraviewweb/index.html (accessed May 10, 2021).

[52] "ParaView: ParaView Catalyst v5.9," Jan. 27, 2021. https://kitware.github.io/paraview-docs/latest/cxx/ParaViewCatalyst.html (accessed May 10, 2021).

APPENDICES

I. The Docker script for FEMOCS

```
FROM ubuntu:20.04

# tzdata update problem workaround
# https://grigorkh.medium.com/fix-tzdata-hangs-docker-image-build-cdb52cc3360d
ENV TZ=Europe/Tallinn
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone

# required software
RUN apt-get update && \
   apt install -y gcc-9 g++-9 gfortran git make cmake build-essential libboost-
all-dev libblas-dev liblapack-dev libtbb-dev libz-dev libmpfr-dev petsc-dev lib-
metis5 wget libarpack2 gsl-bin libnetcdf-c++4-1

# getting FEMOCS
WORKDIR /home
RUN git clone --recursive https://github.com/veskem/femocs.git

# providing compiled dealii to speedup the building
ADD dealii /home/femocs/dealii

ENV DEAL_II_DIR=/home/femocs/dealii

# creating missing symlinks
RUN ln -s /usr/bin/python3.8 /usr/bin/python
RUN ln -s /usr/lib/x86_64-linux-gnu/libmetis.so.5 /usr/lib/x86_64-linux-gnu/lib-
metis.so
RUN ln -s /usr/lib/x86_64-linux-gnu/libarpack.so.2 /usr/lib/x86_64-linux-gnu/li-
barpack.so
RUN ln -s /usr/lib/x86_64-linux-gnu/libgsl.so.23 /usr/lib/x86_64-linux-gnu/lib-
gsl.so
RUN ln -s /usr/lib/x86_64-linux-gnu/libgslcblas.so.0 /usr/lib/x86_64-linux-
gnu/libgslcblas.so
RUN ln -s /usr/lib/x86_64-linux-gnu/libnetcdf.so.15 /usr/lib/x86_64-linux-
gnu/libnetcdf.so
RUN ln -s /usr/lib/x86_64-linux-gnu/libnetcdf_c++4.so.1 /usr/lib/x86_64-linux-
gnu/libnetcdf_c++.so

# updating submodules
WORKDIR /home/femocs
# RUN git submodule update --init --recursive
# recent bug workaround
# WORKDIR /home/femocs/GETELEC
# RUN git checkout 7eb7fe57ef94845187f0eac894b592df54c19236

# building
WORKDIR /home/femocs
RUN make install-ubuntu
RUN make lib

CMD ["/bin/bash"]
```

II. The Docker script for the adaptor library

```
FROM nokal/femocs:84b58f3

# required software
RUN apt-get update && \
   apt install -y git cmake build-essential libgl1-mesa-dev libxt-dev python3-dev
python3-numpy libopenmpi-dev libtbb-dev libnetcdf-c++4

# 1) buildling ParaView on your own

# # getting ParaView and switching to a specific version
# WORKDIR /home
# RUN git clone --recursive https://gitlab.kitware.com/paraview/paraview.git
# RUN cd paraview && git checkout v5.8.1 && git submodule update --init --recur-
sive

# # building ParaView
# WORKDIR /home/paraview_build
# RUN cmake -D PARAVIEW_BUILD_EDITION=CATALYST -D PARAVIEW_USE_PYTHON=ON -D PARA-
VIEW_USE_MPI=ON -D VTK_SMP_IMPLEMENTATION_TYPE=TBB -D VTK_PYTHON_OP-
TIONAL_LINK=OFF -D CMAKE_BUILD_TYPE=Release ../paraview
# RUN make -j 6



# or 2) providing compiled ParaView

# getting ParaView source and switching to a specific version
WORKDIR /home
RUN git clone --recursive https://gitlab.kitware.com/paraview/paraview.git
RUN cd paraview && git checkout v5.8.1 && git submodule update --init --recursive

# providing compiled ParaView build
ADD paraview_build /home/paraview_build
WORKDIR /home/paraview_build

# cleaning previous cmake cache
RUN rm CMakeCache.txt

# regenerating cmake configs
RUN cmake -D PARAVIEW_BUILD_EDITION=CATALYST -D PARAVIEW_USE_PYTHON=ON -D PARA-
VIEW_USE_MPI=ON -D VTK_SMP_IMPLEMENTATION_TYPE=TBB -D VTK_PYTHON_OP-
TIONAL_LINK=OFF -D CMAKE_BUILD_TYPE=Release ../paraview

# getting Catalyst Adaptor source
WORKDIR /home
```

```
RUN git clone https://github.com/iharsuvorau/catalyzing-femocs.git

# building
WORKDIR /home/catalyzing-femocs
RUN make FEMOCS_DIR=/home/femocs PARAVIEW_DIR=/home/paraview_build

# specifying shared libraries
ENV LD_LIBRARY_PATH=/home/femocs/lib:/home/paraview_build/lib:/home/catalyzing-
femocs/build

WORKDIR /home

CMD ["/bin/bash"]
```

III. The Docker script for LAMMPS

```
FROM nokal/femocs-84b58f3_catalyst_pv-5.8.1:latest

RUN apt-get update && apt install -y libpng-dev

WORKDIR /home
RUN git clone --recursive https://github.com/iharsuvorau/lammps.git

ENV LD_LIBRARY_PATH=/home/femocs/lib:/home/paraview_build/lib:/home/catalyzing-
femocs/build

WORKDIR /home/lammps
RUN git checkout visual-femocs

WORKDIR /home/lammps/src/USER-FEMOCS-CATALYST
RUN make \
FEMOCS_DIR=/home/femocs \
DEAL_II_DIR=/home/femocs/dealii/lib/cmake/deal.II \
DEAL_II_INCLUDE=/home/femocs/dealii/include \
FEMOCS_CATALYST_DIR=/home/catalyzing-femocs \
PARAVIEW_DIR=/home/paraview_build

WORKDIR /home

CMD ["/bin/bash"]
```

IV. The adaptor library interface: CatalystAdaptor.h

```
#pragma once
#include "Femocs.h"

namespace CatalystAdaptor {
    void Initialize(const char *path, const char *meshCellType);

    void Finalize();

    void ImportAtoms(double **atoms, int numAtoms);

    void CoProcess(femocs::Femocs &project,
                   const double time,
                   const unsigned int timeStep,
                   const bool lastTimeStep);
}
```

V. The adaptor library implementation: CatalystAdaptor.cpp

```
#include <cstdlib>
#include <iterator>
#include <vtkCPDataDescription.h>
#include <vtkCPInputDataDescription.h>
#include <vtkCPProcessor.h>
#include <vtkCPPythonScriptPipeline.h>
#include <vtkCellData.h>
#include <vtkCellType.h>
#include <vtkCellArray.h>
#include <vtkDoubleArray.h>
#include <vtkFloatArray.h>
#include <vtkNew.h>
#include <vtkPointData.h>
#include <vtkPoints.h>
#include <vtkUnstructuredGrid.h>
#include <vtkVertex.h>
#include "CatalystAdaptor.h"
#include "Femocs.h"

namespace CatalystAdaptor {
    vtkCPProcessor *processor = NULL;

    // FEM-related globals
    const char *meshCellType;
    VTKCellType vtkMeshCellType;
    int numNodesPerCell;

    // MD-related globals
    double **atoms;
    int numAtoms;

    void Initialize(const char *path, const char *cellType) {
        std::cout << "CatalystAdaptor::Initialize has been called" << std::endl;

        if (processor == NULL) {
            processor = vtkCPProcessor::New();
            processor->Initialize();
        } else {
            processor->RemoveAllPipelines();
        }

        // registering python visualization script
        vtkNew <vtkCPPythonScriptPipeline> pipeline;
        pipeline->Initialize(path);
        processor->AddPipeline(pipeline.GetPointer());

        // parsing mesh-related arguments
        meshCellType = cellType;
        // https://vtk.org/doc/nightly/html/vtkCellType_8h.html
        if (strcmp(meshCellType, "quadrangles") == 0) {
            numNodesPerCell = 4;
            vtkMeshCellType = VTK_QUAD;
        } else if (strcmp(meshCellType, "hexahedra") == 0) {
            numNodesPerCell = 8;
            vtkMeshCellType = VTK_HEXAHEDRON;
        } else {
            std::cout << "cell type is undefined" << std::endl;
            exit(1);
        }
    }

    void Finalize() {
        std::cout << "CatalystAdaptor::Finalize has been called" << std::endl;
        if (processor) {
            processor->Delete();
            processor = NULL;
        }
    }

    void ImportAtoms(double **arr, int num) {
        std::cout << "CatalystAdaptor::ImportAtoms has been called" << std::endl;
        atoms = arr;
        numAtoms = num;
    }

    void CoProcess(femocs::Femocs &project, const double time, const unsigned int
timeStep, const bool lastTimeStep) {
        // creating data description
        vtkNew <vtkCPDataDescription> dataDescription;
        dataDescription->AddInput("MD");
        dataDescription->AddInput("FEM");
        dataDescription->SetTimeData(time, timeStep);
        if (lastTimeStep) {
            dataDescription->ForceOutputOn();
        }

        // determining if co-processing should be done
        if (processor->RequestDataDescription(dataDescription.GetPointer()) == 0)
{
            return;
        }
        std::cout << "CatalystAdaptor::CoProcess has been called" << std::endl;


        // Generating FEM grid using mesh data

        // preparing mesh points
        const double *nodes = NULL;
        const int numNodes = project.export_data(&nodes, "nodes");
        vtkNew <vtkPoints> meshPoints;
        meshPoints->Allocate(numNodes);
        for (int i = 0; i < numNodes; i++) {
            meshPoints->InsertNextPoint(nodes[i * 3], nodes[i * 3 + 1], nodes[i *
3 + 2]);
        }

        // preparing mesh cells
        const int *cells = NULL;
        const int numCells = project.export_data(&cells, meshCellType);
        vtkIdType meshCellIDs[numCells][numNodesPerCell];
        for (int i = 0; i < numCells; i++) {
            for (int j = 0; j < numNodesPerCell; j++) {
                meshCellIDs[i][j] = cells[numNodesPerCell * i + j];
            }
        }

        // making mesh grid
        vtkNew <vtkUnstructuredGrid> meshGrid;
        meshGrid->SetPoints(meshPoints);
        for (int i = 0; i < numCells; i++)
            meshGrid->InsertNextCell(vtkMeshCellType, numNodesPerCell,
meshCellIDs[i]);


        // Generating MD grid using atomistic data

        // preparing atoms' points and cells
        vtkNew <vtkPoints> atomPoints;
        atomPoints->Allocate(numAtoms);
        vtkNew <vtkCellArray> atomCells;
        for (int i = 0; i < numAtoms; i++) {
            atomPoints->InsertPoint(i, (*atoms)[i * 3], (*atoms)[i * 3 + 1],
(*atoms)[i * 3 + 2]);
            atomCells->InsertNextCell(1);
            atomCells->InsertCellPoint(i);
        }
        // extracting field data for atoms

        double temperatureData[numAtoms] = {0};
        project.export_data(temperatureData, numAtoms, "temperature");
        vtkNew <vtkDoubleArray> temperature;
        temperature->SetName("temperature");
        temperature->SetArray(temperatureData, numAtoms, 1);

        double elfieldNormData[numAtoms] = {0};
        project.export_data(elfieldNormData, numAtoms, "elfield_norm");
        vtkNew <vtkDoubleArray> elfieldNorm;
        elfieldNorm->SetName("electric field normals");
        elfieldNorm->SetArray(elfieldNormData, numAtoms, 1);

        double elfieldData[numAtoms * 3] = {0};
        project.export_data(elfieldData, numAtoms * 3, "elfield");
        vtkNew <vtkDoubleArray> elfield;
        elfield->SetName("electic field");
        elfield->SetNumberOfComponents(3);
        elfield->SetNumberOfTuples(numAtoms);
        for (int i = 0; i < numAtoms * 3; i++) {
            elfieldData[i] *= -1;
        }
        elfield->SetArray(elfieldData, numAtoms * 3, 1);

        // making atomistic grid
        vtkNew <vtkUnstructuredGrid> atomsGrid;
        atomsGrid->SetPoints(atomPoints);
        atomsGrid->SetCells(VTK_VERTEX, atomCells);
        atomsGrid->GetPointData()->AddArray(temperature);
        atomsGrid->GetPointData()->AddArray(elfieldNorm);
        atomsGrid->GetPointData()->SetVectors(elfield);



        // Passing data to Catalyst

        dataDescription->GetInputDescriptionByName("MD")->SetGrid(atomsGrid);
        dataDescription->GetInputDescriptionByName("FEM")->SetGrid(meshGrid);
        processor->CoProcess(dataDescription.GetPointer());
    }
}
```