An Architecture for Interactive *In Situ* Visualization and its Transparent Implementation in OpenFPM

Aryaman Gupta[*]
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG Dresden

Pietro Incardona
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG Dresden

Ata Deniz Aydin
Center for Systems Biology Dresden
MPI-CBG Dresden

Stefan Gumhold
Technische Universität Dresden
Cluster of Excellence Physics of Life

Ulrik Günther
Center for Advanced Systems Understanding, Görlitz
MPI-CBG Dresden

Ivo F. Sbalzarini[†]
Technische Universität Dresden
Center for Systems Biology Dresden
MPI-CBG Dresden
Cluster of Excellence Physics of Life

ABSTRACT

Live *in situ* visualization of numerical simulations – interactive visualization while the simulation is running – can enable new modes of interaction, including computational steering. Designing easy-to-use distributed *in situ* architectures, with viewing latency low enough, and frame rate high enough, for interactive use, is challenging. Here, we propose a fully asynchronous, hybrid CPU–GPU *in situ* architecture that emphasizes interactivity. We also present a transparent implementation of this architecture embedded into the OpenFPM simulation framework. The benchmarks show that our architecture minimizes visual latencies, and achieves frame rates between 6 and 60 frames/second – depending on simulation data size and degree of parallelism – by changing only a few lines of an existing simulation code.

CCS CONCEPTS

• **Human-centered computing** → *Visualization*; • **Computing methodologies** → *Parallel algorithms*; *Simulation support systems.*

KEYWORDS

in situ visualization, distributed rendering, computational steering, parallel simulation software

ACM Reference Format:
Aryaman Gupta, Pietro Incardona, Ata Deniz Aydin, Stefan Gumhold, Ulrik Günther, and Ivo F. Sbalzarini. 2020. An Architecture for Interactive *In Situ* Visualization and its Transparent Implementation in OpenFPM. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV'20), November 12, 2020, Atlanta, GA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3426462.3426472

[*]argupta@mpi-cbg.de
[†]ivos@mpi-cbg.de

1 INTRODUCTION

In situ visualization, i.e., the visualization or analysis of a simulation as it runs, is becoming increasingly important as the disparity between computational throughput and file I/O grows for large-scale parallel simulations. In addition, *in situ* visualization, when performed interactively, enables new modes of interacting with a simulation, including online visual analytics [14] and computational steering [20]. Such interaction can help the scientist develop a better understanding of the simulated physics.

Live and interactive *in situ* visualization of distributed-memory parallel simulations has therefore received much interest in the recent past. A common challenge is achieving visualization frame rates that are high enough, and latency that is low enough, for interactivity.

Here, we address these challenges by proposing a fully asynchronous, hybrid CPU–GPU architecture that emphasizes interactivity in *in situ* visualization of distributed numerical simulations. We present the design choices made to optimize performance and interactivity of *in situ* visualization. We provide a transparent implementation of the proposed architecture in the open-source C++ simulation framework OpenFPM [15] for scalable particle- and mesh-based simulations across application domains. Our implementation follows the design goal of OpenFPM in that it provides easy-to-use, high-level abstractions to the user. We demonstrate how *live in situ* visualization of both particle- and mesh-based simulations can be configured in a few lines of C++ code, requiring only minimal changes to the simulation application. With visualization functionality built into OpenFPM, users do not need to download, install, or link their simulation with any third-party *in situ* tool.

We describe and benchmark our implementation for varying simulation sizes and degrees of parallelism, showing that it achieves high-enough frame rates for smooth interactive viewpoint changes and zooming. We also show that the asynchronous architecture proposed here reduces the time delay for visual interaction commands. In particular, we contribute the following:

- We present an asynchronous embedded architecture for *in situ* visualization that optimizes interactivity.
- We extend OpenFPM to transparently support interactive *in situ* visualization with minimal simulation code changes.

2 BACKGROUND

Before presenting our proposed *in situ* architecture, we review existing solutions, rationalizing our design choices.

2.1 Simulation Frameworks

Several frameworks offer high-level abstractions to more rapidly implement parallel numerical simulations. Most of these are specific to a numerical method. OpenFOAM [17], for example, is for finite-volume simulations, FEniCS [2], DUNE [5], and AMDiS [24] for finite-elements, DualSPHysics [7] for Smoothed-Particle Hydrodynamics (SPH), LAMMPS [21] for Molecular Dynamics (MD) and Dissipative Particle Dynamics (DPD), and AMReX [26] for Adaptive Mesh Refinement (AMR). In addition to method-specific simulation frameworks, more generic ones cover entire data-structure classes.

Examples of application-agnostic simulation frameworks include Liszt [8] for stencil codes, and FDPS [16] and OpenFPM [15] for particle-mesh codes. OpenFPM is open source (http://openfpm.mpi-cbg.de) and actively developed. It provides high-level abstractions, including domain decomposition, dynamic load balancing, and distributed particle and mesh data structures to implement simulations across application domains, including SPH, MD, DPD, AMR, and stencil codes. Despite this universality, simulations written using OpenFPM have been shown to be as, or more, efficient than simulations using more specific libraries, and OpenFPM can also be used for simulations for which there exists no dedicated library, such as Vortex Methods [6]. Therefore, we chose to implement the present *in situ* architecture in OpenFPM.

2.2 *In Situ* Architectures and Libraries

Bauer et al. [4] provide an extensive review of existing libraries for *in situ* visualization, as well as the diverse architectures they are based on. *In situ* visualization can run in close proximity with the simulation, on the same compute nodes, or it can be executed on a different set of compute nodes, called *in transit* processing. On-node proximity between simulation and visualization reduces data transfer, but in-transit processing can achieve better resource utilization [18]. Architectures utilizing on-node proximity are further distinguished into those where visualization and simulation run synchronously, time-partitioning their access to shared resources, and those running asynchronously.

Several libraries, including ParaView Catalyst [3], VisIt Libsim [25] and ISAAC [19], run visualization synchronously and on the same nodes as the simulation, while Catalyst and Libsim can also perform asynchronous in-transit processing, e.g., enabled by the libIs library [23]. ADIOS [27] can be used for *in situ* visualization with either in-transit or on-node proximity, running either synchronously or asynchronously. ExaViz [10] and Damaris/Viz [9] perform asynchronous *in situ* on the same node as the simulation, with visualization running on dedicated CPU cores, while ExaViz also performs in-transit processing. All of these libraries can enable *live in situ* visualization. The Cinema framework [1], on the other hand, enables *post hoc* interactive visualization using a database of images generated *in situ* over a range of visualization parameters.

In situ libraries can also be distinguished between those that are embedded into a simulation system, and are therefore more lightweight, and those that are general-purpose and enable better code reuse. Since general-purpose libraries require a bridge to be implemented between the simulation and visualization data structures, we choose an embedded design, which allows users of OpenFPM to activate *in situ* visualization by adjusting only a few lines of the simulation code.

3 *IN SITU* ARCHITECTURE

We propose an on-node, asynchronous, embedded architecture that emphasizes interactivity in *live in situ* visualization and computational steering of distributed-memory parallel simulations.

We choose on-node processing to minimize communication overhead, which can be significant in large distributed simulations. Simulation data computed on a particular node is therefore rendered on that node itself. While previous work has achieved interactive frame rates using in-transit processing [23], the present on-node architecture enhances interactivity in computational steering. This is because any change in simulation data is reflected in the visualization at the earliest, without having to first transmit the new simulation data for visualization.

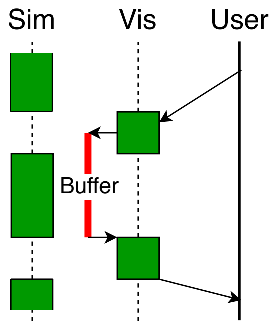
To further improve interactivity, we choose fully asynchronous execution of simulation and visualization, allowing them both to proceed at their own frequencies, never having to wait for the other. A simulation time step often takes a few seconds or longer. In a synchronous *in situ* architecture, any change to visualization parameters would only take effect after completing the next simulation time step.

Figure 1 compares the timing and message flow in synchronous vs. asynchronous *in situ* architectures. In both cases, the worst-case delay in visual feedback occurs when a visual interaction command, e.g. a viewpoint change, arrives just after a visualization time step has begun, too late to be incorporated. The visual feedback of the interaction would then be visible only after the completion of the on-going and the next visualization time step. In the fully synchronous case, the simulation time step comes in between. The maximum delay in visual feedback in the synchronous case is therefore just below $t_{\text{sim}} + 2t_{\text{vis}}$, while in the asynchronous case it is less than $2t_{\text{vis}}$. While visualization time steps can be longer in the asynchronous case due to resource sharing, our benchmarks in Section 5 show that asynchronous execution still has less latency. In both cases, network latency is additional.

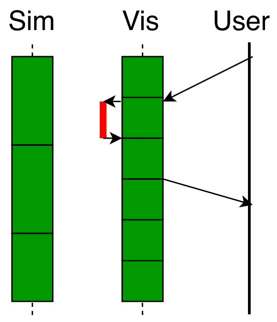
While other frameworks, e.g. Damaris/Viz [9], also use on-node, asynchronous architectures, we aim to enhance interactivity by leveraging the heterogeneous CPU–GPU architecture increasingly found in high-performance computers. Within each compute node, the simulation runs on the CPU, while rendering takes place asynchronously on the GPU. Not only does the GPU accelerate rendering, further reducing visualization latency, it also helps minimize competition for resources. To control GPU rendering and data transfer, a small number of cores on the CPU are dedicated to visualization while the others perform simulation.

3.1 Simulation Data Handling

In order to reduce memory usage on the CPU, communication of data from the simulation processes to the visualization process within a compute node takes place via shared memory. In order to

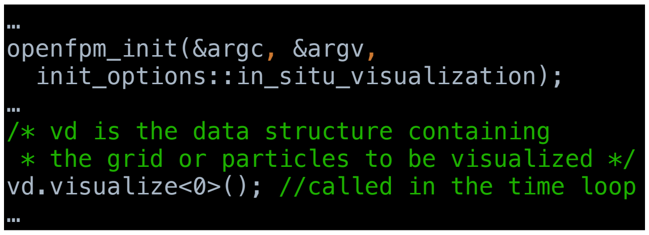


(a) Synchronous execution



(b) Asynchronous execution

Figure 1: Sequence diagram comparing latency for visual interactions (e.g., camera pose change) between synchronous (a) and asynchronous (b) *in situ* architectures. Time increases linearly downward. Interaction messages are buffered (shown in red) until the beginning of the next visualization time step.



```
…
openfpm_init(&argc, &argv,
  init_options::in_situ_visualization);
…
/* vd is the data structure containing
 * the grid or particles to be visualized */
vd.visualize<0>(); //called in the time loop
…
```

Figure 2: A code snippet showing the lines of code required to configure *in situ* visualization in an OpenFPM simulation.

reduce the data sent to the GPU, simulation data (typically double-precision floating point) are converted to 16-bit unsigned integers. Any data structure selected by the user for visualization is transparently converted by OpenFPM to unsigned integers at the end of each simulation time step. The unsigned int buffer is allocated on a block of memory that can be shared with the visualization process, instead of on normal heap memory.

The use of shared memory allows the visualization process to asynchronously read from the unsigned int buffer at any time and with zero copy. Since the visualization process only *reads* data from the shared buffer, but never writes to it, no synchronization is required between the simulation process and the visualization process. This enables fully asynchronous execution as shown in Figure 1b. Since the unsigned int buffer is updated only at the end of a simulation time step, visualization artifacts that may result from this lack of synchronization are short-lived (around a hundred milliseconds in the benchmarks presented here) and hardly visually perceived, since numerical stability requires the simulation to only change slightly between time steps. Changes and re-allocations in the OpenFPM data structure, e.g. due to dynamic load balancing, are transparently handled using a double-buffer protocol.

4 IMPLEMENTATION IN OPENFPM

We describe the software implementation of the architecture from Section 3 and describe how a simulation application uses it.

4.1 Software Architecture

All visualization and simulation processes are part of the same MPI job. This means that an OpenFPM simulation with *in situ* visualization enabled can be launched with a single `mpirun` command. If *in situ* visualization has been activated by the user, the process with MPI rank 0 is converted to a master process responsible for receiving incoming messages on a TCP socket, e.g., for computational steering and visual interaction. Of the remaining processes, the lowest-ranked process on each compute node is converted to a visualization process. All other processes proceed with the simulation, as normal.

The master process and all visualization processes are grouped in an MPI communicator, called the *visualization communicator*, where they can send messages for changes in visualization parameters. Similarly, the master process is placed in the *steering communicator* together with all simulation processes, for communicating computational steering messages. All visualization processes are also part of the *rendering communicator*, where they can communicate for the purpose of compositing images. All simulation processes are part of the *compute communicator*, used for communications of distributed solvers and dynamic load balancing.

These communicators are only created if the user activates *in situ* visualization. Without it, all processes are launched as simulation processes. This fulfills one of our design goals: The *in situ* infrastructure has minimal impact on an OpenFPM simulation when not in use. The impact when not in use is limited to an increased resident set size (RSS) and a few additional conditional branches that are traversed once when the application starts. Existing OpenFPM simulation code does not require any changes when not using *in situ* visualization.

4.2 Distributed Rendering

For rendering images of simulation data, the rendering processes launch an application written using the open-source rendering library *scenery* [12]. Scenery uses the high-performance, low-level Vulkan API instead of OpenGL, as still predominantly used in most *in situ* tools, to better leverage the power of modern GPUs. Scenery also provides hardware-accelerated H.264/HEVC video encoding, used for streaming the final rendered images off the compute cluster to a potentially remote display client.

For scalability, we perform sort-last compositing of images using the direct-send algorithm [11], which has previously been used for *in situ* visualization of simulations scaling to 216,000 cores [13]. Rendering and compositing of volume data is performed using Vulkan compute shaders, while rendering of particles uses fragment shaders. The rendering application, provided embedded with OpenFPM, is also available stand-alone as open source (https://github.com/scenerygraphics/scenery-insitu).

4.3 Usage Example

Using our implementation, *in situ* visualization can be configured in OpenFPM with just a couple of lines of code, illustrated in the snippet in Figure 2. When initializing OpenFPM, the user needs to specify that they would like to activate *in situ* visualization. Then, in the time loop of the simulation, the user calls the `.visualize()` method of any data structure to be visualized, specifying the field to be rendered as a C++ template parameter. For the Gray-Scott and Vortex-in-cell simulations in Figure 3, only 2 of the 116 and 531 lines of code, respectively, of the OpenFPM simulation application needed to be changed. The type of rendering, particle or volume, is determined automatically based on the data structure. The `.visualize()` method also accepts optional parameters, e.g., to specify whether to visualize the magnitude or a component of a vector field. All optional parameters, along with transfer function and camera pose, can also be interactively changed at run-time from the provided display client.

Example codes are provided in the OpenFPM repository to show how *in situ* visualization can be activated and configured in real-world applications.

5 BENCHMARKS

We perform benchmarks to test the performance of our implementation, the validity of the asynchronous architecture, and its impact on the simulation. In all cases, images of resolution $1200x1200$ pixels are generated and frame-rates are averaged over a realistic interactive session, with a script triggering camera movements from a remote display client. Unless otherwise stated, the Gray-Scott simulation of Figure 3a is used, distributed across 12 nodes, generating volume data of size 1500^3.

Figure 4 shows the rendering frame rate, i.e., the number of images generated *in situ* per second, versus the number of 20-core nodes used, for two different data sizes.

The frame rates are sufficiently high for interactive visualization and remain steady during a visualization session. We also explore the effect of increasing the number of CPU cores available to the visualization process on each node. Launching fewer simulation processes per node (see inset legend) improves the performance of visualization, which is consistent with previous observations [22]. While OpenFPM simulation processes are single-threaded, the visualization process is multi-threaded. The user can choose the number of cores to use for visualization by choosing the total number of processes per node when launching the MPI job.

We further observe that frame rates decrease with increasing number of nodes. The reason for this is revealed by profiling the visualization process. We find that the slow-down is caused by increased network communication. Profiling is performed using the Gray-Scott simulation with grid resolution 1500^3. We run the simulation on 4, 8, and 12 nodes, with both 15 and 18 simulation processes per node. In every case, we find that at least 97% of the time required for image compositing is spent in network communication. The absolute wall-clock times for network communication are also found to increase for higher numbers of nodes, resulting in reduced rendering frame rates. The present strong-scaling benchmark, where a fixed data size is distributed onto an increasing number of compute nodes, leads to increased communication between the simulation processes, which further adds to the network overhead. Alongside compositing, rendering the simulation data on any node takes place asynchronously on the GPU, controlled by another thread of the visualization process on the CPU. This results in the overall visualization frame rates being limited by the network communication required by the direct-send [11] compositing algorithm.

Type	t_{vis}/s	t_{sim}/s	Max delay/s
Asynchronous	0.065	0.891	0.130
Synchronous	0.020	0.566	0.606

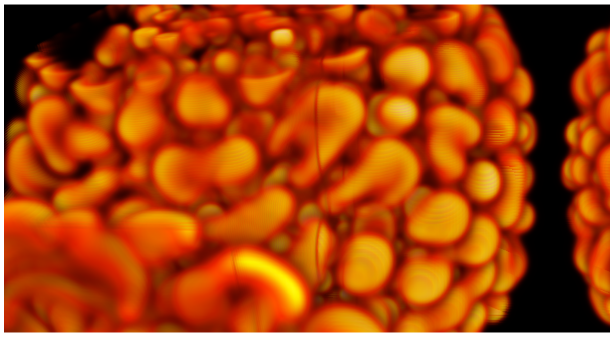
Table 1: Wall-clock times in seconds and maximum visualization delay/latency.

Number of nodes	Simulation all 20 cores, no vis.	Simulation 18 cores + vis.	Simulation 18 cores, no vis.
12	114 s	137 s	111 s

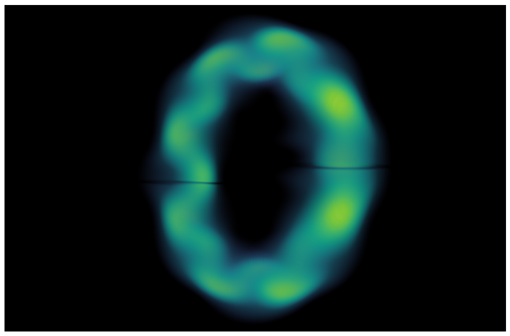
Table 2: Overall wall-clock time to complete 200 simulation time steps with and without *in situ* visualization.

In a second benchmark, we assess the visualization latency. As illustrated in Figure 1, a synchronous architecture has a maximum delay in visual feedback of $2t_{\text{vis}} + t_{\text{sim}}$, whereas asynchronous execution has a maximum delay of $2t_{\text{vis}}$, where t_{vis} and t_{sim} are the visualization and simulation time step duration, respectively. It is not obvious, which number is smaller, as the time steps can be longer in asynchronous execution due to resource sharing. While for simulations with time steps in the tens of seconds it is clear that asynchronous execution provides faster response, we test latency for a simulation with sub-second time step. Table 1 shows the results, suggesting that our asynchronous architecture has significantly lower latency even for fast simulation time steps.

Lastly, we benchmark the impact *in situ* visualization has on the overall run-time of an OpenFPM simulation. We run the same Gray-Scott simulation in three scenarios: (1) 20 simulation processes per node, occupying all 20 cores of a node; (2) 18 simulation processes and 1 multi-threaded visualization process per node; (3) 18 simulation processes per node, but no visualization process. In each scenario, we measure the time to complete 200 simulation time steps. The results in Table 2 indicate that *in situ* visualization has an impact on the performance of the simulation. This impact is not due to network communication, as the time for communicating ghost data between simulation processes is not found to increase significantly when enabling *in situ* visualization. As mentioned in Section 3.1, each simulation process is responsible for scaling and converting its data to an unsigned integer buffer for visualization. Profiling the code shows that this scaling is the reason for the increased overall runtime of the simulation when *in situ* visualization is enabled. For a 1500^3 grid size distributed across 12 nodes, conversion and scaling to the unsigned int buffer incurred a constant overhead of 0.12 s per time step.



(a) Gray-Scott reaction-diffusion simulation



(b) Vortex-in-cell simulation

Figure 3: Example *in situ* rendering outputs of OpenFPM simulations using the provided remote display client. (a) A Gray-Scott reaction-diffusion simulation using central finite differences on a regular Cartesian mesh. The concentration in 3D space is visualized as color. (b) A hybrid particle-mesh simulation of the incompressible Navier-Stokes equations in vorticity formulation, initialized as a vortex-ring. The magnitude of the vorticity vector field is visualized. Both simulations were distributed across 8 compute nodes with 20 cores per node. The thin black lines indicate locations of compute-node boundaries.

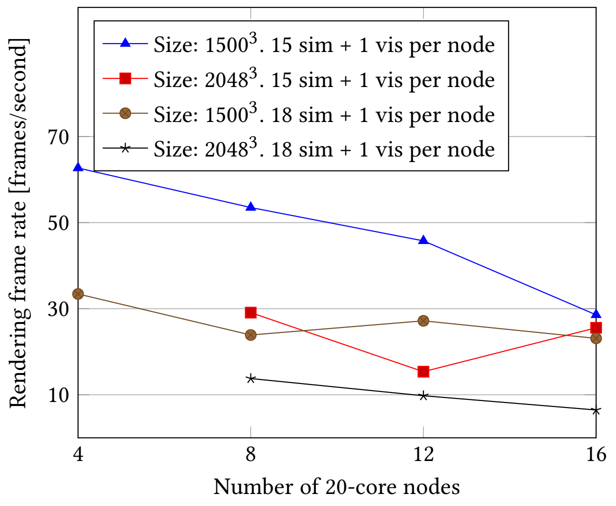


Figure 4: Rendering frame rate vs. number of nodes.

6 CONCLUSIONS

We have presented a fully asynchronous, hybrid CPU–GPU *in situ* visualization architecture for distributed parallel simulations, as well as its transparent embedded implementation in OpenFPM [15]. This enables OpenFPM-based simulations to use remote, live *in situ* visualization by changing just a few lines of code. When not in use, the *in situ* capability has no code side-effects and minimal overhead on the performance of a simulation. Using the present *in situ* visualization framework also does not depend on the availability or installation of additional software dependencies. For these three reasons, we call our architecture "transparent".

The design choices of the present transparent *in situ* architecture emphasize interactivity. Throughout the architecture, care was taken to minimize the latency from an interaction command to the visual feedback, and to maximize rendering frame rate. Therefore, we chose a zero-copy shared-memory layout and leveraged hybrid CPU–GPU hardware using the Vulkan graphics API.

We have benchmarked the performance of our framework, and have shown that it minimizes visualization latency, and can achieve frame rates in excess of 15 frames/second even for large data sizes, if the visualization is provided with sufficient resources.

However, this speed comes at a cost to the simulation, as simulation data needs to be converted to unsigned integers to reduce the data to be sent to the GPU. While optimizing the implementation could help reduce overhead, our results inform about the trade-offs inherent in designing interactive *in situ* visualization.

In the future, we could spin off the *in situ* application, currently tightly-integrated into OpenFPM, so it can also be used with other simulation frameworks. We will also explore methods to reduce the amount of rendering data sent over the interconnect, which proved to be a bottleneck in our benchmarks.

Taken together, we believe that the presented *in situ* architecture and its transparent implementation add to the user experience of OpenFPM-based simulations. Moreover, they provide insights into best practices when designing *in situ* architectures, especially for interactive applications like computational steering.

ACKNOWLEDGMENTS

We thank Prof. Dr. Raimund Dachselt, TU Dresden, for inspiring discussions. We thank Oscar Gonzalez and the Scientific Computing Facility of MPI-CBG for providing access to the *furiosa* HPC system. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC-2068 – 390729961 – Cluster of Excellence "Physics of Life" of TU Dresden. P.I. was supported by a grant from the German Federal Ministry of Science and Education (BMBF), grant number 031L0160. U.G. was funded by the Center for Advanced Systems Understanding (CASUS), financed by Germany's Federal Ministry of Education and Research (BMBF) and by the Saxon Ministry for Science, Culture and Tourism (SMWK) with tax funds on the basis of the budget approved by the Saxon State Parliament.

REFERENCES

[1] James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H. Rogers, and Mark Petersen. 2014. An Image-Based Approach to Extreme Scale in situ Visualization and Analysis. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 424–434.

[2] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS project version 1.5. *Archive of Numerical Software* 3, 100 (2015).

[3] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. 2015. 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) *(ISAV2015)*. Association for Computing Machinery, New York, NY, USA, 25–29. https://doi.org/10.1145/2828612.2828624

[4] Andrew C. Bauer, Hasan Abbasi, James Ahrens, Hank Childs, Berk Geveci, Scott Klasky, Kenneth Moreland, Patrick O'Leary, Venkatram Vishwanath, Brad Whitlock, and E. W. Bethel. 2016. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. *Computer Graphics Forum* 35, 3, 577–597. https://doi.org/10.1111/cgf.12930

[5] Markus Blatt, Ansgar Burchardt, Andreas Dedner, Christian Engwer, Jorrit Fahlke, Bernd Flemisch, Christoph Gersbacher, Carsten Gräser, Felix Gruber, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Tobias Malkmus, Steffen Müthin, Martin Nolte, Marian Piatkowski, and Oliver Sander. 2016. The Distributed and Unified Numerics Environment, version 2.4. *Archive of Numerical Software* 4, 100 (2016), 13–29.

[6] Georges-Henri Cottet and Petros D. Koumoutsakos. 2000. *Vortex Methods: Theory and Practice*. Vol. 8. Cambridge university press Cambridge.

[7] Alejandro J. C. Crespo, José M. Domínguez, Benedict D. Rogers, Moncho Gómez-Gesteira, S. Longshaw, R. Canelas, Renato Vacondio, Anxo Barreiro, and O. García-Feal. 2015. DualSPHysics: Open-Source Parallel CFD Solver Based on Smoothed Particle Hydrodynamics (SPH). *Computer Physics Communications* 187 (2015), 204 – 216. https://doi.org/10.1016/j.cpc.2014.10.004

[8] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. 2011. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2063384.2063396

[9] Matthieu Dorier, Robert Sisneros, Tom Peterka, Gabriel Antoniu, and Dave Semeraro. 2013. Damaris/Viz: A Nonintrusive, Adaptable and User-Friendly in situ Visualization Framework. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 67–75.

[10] Matthieu Dreher, Jessica Prevoteau-Jonquet, Mikael Trellet, Marc Piuzzi, Marc Baaden, Bruno Raffin, Nicolas Férey, Sophie Robert, and Sébastien Limet. 2014. ExaViz: A Flexible Framework to Analyse, Steer and Interact with Molecular Dynamics Simulations. *Faraday discussions* 169 (2014), 119–142.

[11] Stefan Eilemann and Renato Pajarola. 2007. Direct Send Compositing for Parallel Sort-Last Rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*. 29–36. https://doi.org/10.5167/uzh-47723

[12] Ulrik Günther, Tobias Pietzsch, Aryaman Gupta, Kyle I. S. Harrington, Pavel Tomancak, Stefan Gumhold, and Ivo F. Sbalzarini. 2019. scenery: Flexible Virtual Reality Visualization on the Java VM. In *2019 IEEE Visualization Conference (VIS)*. IEEE, 1–5.

[13] Mark Howison, E. Wes Bethel, and Hank Childs. 2011. Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems. *IEEE Transactions on Visualization and Computer Graphics* 18, 1 (2011), 17–29.

[14] Seif Ibrahim, Thomas Stitt, Matthew Larsen, and Cyrus Harrison. 2019. Interactive in Situ Visualization and Analysis Using Ascent and Jupyter. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (Denver, Colorado, USA) *(ISAV '19)*. Association for Computing Machinery, New York, NY, USA, 44–48. https://doi.org/10.1145/3364228.3364232

[15] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F. Sbalzarini. 2019. OpenFPM: A Scalable Open Framework for Particle and Particle-Mesh Codes on Parallel Computers. *Computer Physics Communications* 241 (2019), 155–177.

[16] Masaki Iwasawa, Ataru Tanikawa, Natsuki Hosono, Keigo Nitadori, Takayuki Muranushi, and Junichiro Makino. 2016. Implementation and Performance of FDPS: A Framework for Developing Parallel Particle Simulation Codes. *Publications of the Astronomical Society of Japan* 68, 4 (06 2016). https://doi.org/10.1093/pasj/psw053 arXiv:https://academic.oup.com/pasj/article-pdf/68/4/54/6847738/psw053.pdf 54.

[17] Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. 2007. OpenFOAM: A C++ library for Complex Physics Simulations. In *International workshop on coupled methods in numerical dynamics*, Vol. 1000. IUC Dubrovnik Croatia, 1–20.

[18] James Kress, Matthew Larsen, Jong Choi, Mark Kim, Matthew Wolf, Norbert Podhorszki, Scott Klasky, Hank Childs, and David Pugmire. 2019. Comparing the Efficiency of in situ Visualization Paradigms at Scale. In *International Conference on High Performance Computing*. Springer, 99–117.

[19] Alexander Matthes, Axel Huebl, René Widera, Sebastian Grottel, Stefan Gumhold, and Michael Bussmann. 2016. In situ, Steerable, Hardware-Independent and Data-Structure Agnostic Visualization with ISAAC. *Supercomputing Frontiers and Innovations* 3, 4 (2016). https://superfri.org/superfri/article/view/114

[20] Steven G. Parker and Christopher R. Johnson. 1995. SCIRun: A Scientific Programming Environment for Computational Steering. In *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 52–52.

[21] Steve Plimpton. 1995. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of computational physics* 117, 1 (1995), 1–19.

[22] Tobias Rau, Patrick Gralka, Oliver Fernandes, Guido Reina, Steffen Frey, and Thomas Ertl. 2019. The Impact of Work Distribution on in Situ Visualization: A Case Study. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (Denver, Colorado, USA) *(ISAV '19)*. Association for Computing Machinery, New York, NY, USA, 17–22. https://doi.org/10.1145/3364228.3364233

[23] Will Usher, Silvio Rizzi, Ingo Wald, Jefferson Amstutz, Joseph Insley, Venkatram Vishwanath, Nicola Ferrier, Michael E. Papka, and Valerio Pascucci. 2018. LibIS: A Lightweight Library for Flexible in Transit Visualization. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (Dallas, Texas, USA) *(ISAV '18)*. Association for Computing Machinery, New York, NY, USA, 33–38. https://doi.org/10.1145/3281464.3281466

[24] Simon Vey and Axel Voigt. 2007. AMDiS: Adaptive Multidimensional Simulations. *Computing and Visualization in Science* 10, 1 (2007), 57–67.

[25] Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. 2011. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Eurographics Symposium on Parallel Graphics and Visualization*, Torsten Kuhlen, Renato Pajarola, and Kun Zhou (Eds.). The Eurographics Association. https://doi.org/10.2312/EGPGV/EGPGV11/101-109

[26] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max P. Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. 2019. AMReX: A Framework for Block-Structured Adaptive Mesh Refinement. *Journal of Open Source Software* 4, 37 (2019), 1370–1370.

[27] Fang Zheng, Hongbo Zou, Greg Eisenhauer, Karsten Schwan, Matthew Wolf, Jai Dayal, Tuan-Anh Nguyen, Jianting Cao, Hasan Abbasi, Scott Klasky, Norbert Podhorszki, and Hongfeng Yu. 2013. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 320–331.

REPRODUCIBILITY APPENDIX

For reference, we provide information regarding the open-source software artifacts developed in this work, and further details regarding the experimental setup.

A SOFTWARE ARTIFACTS

All software created through the course of this work is open-source and maintained in public repositories.

- The OpenFPM framework [15] for particle-mesh simulations, which we implement our *in situ* architecture into, is available at: https://git.mpi-cbg.de/openfpm/openfpm$_\mathrm{p}$data. The code for the *in situ* architecture is currently in the `insitu_visualization` branch, and will soon be merged into the `master` branch.
- The distributed rendering application that OpenFPM uses for *in situ* visualization is available at https://github.com/scenerygraphics/scenery-insitu.
- The application for the display client, which enables remote interactive visualization of the simulation is available at https://git.mpi-cbg.de/mosaic/insituclient.
- Both the distributed rendering application and the display client are based on the *scenery* [12] visualization framework, which is open source and available at https://github.com/scenerygraphics/scenery. Some additions were made to *scenery* through the course of this work. Those are currently

in the `parallel-rendering` branch of the repository, and will be merged into the `master` branch in the future.

B EXPERIMENTAL SETUP

The benchmarks described in Section 5 were carried out on the *furiosa* high-performance computer at the MPI-CBG Dresden. The GPU partition of the cluster was used, which houses 23 Intel Broadwell-EP nodes that host 2 Nvidia GeForce GTX 1080 GPUs each. The GPUs are Pascal-generation, GP104-400-A1 chips with 8 GiB of DRAM. Each node holds 2 CPU sockets, each of which hosts an Intel Xeon E5-2698v4 2.20GHz CPU with 20 cores and a total of 512 GiB of RAM. The cluster runs CentOs 7.0, and the batch-system scheduler is Slurm, version 18.08.5-2.

Compute nodes in the cluster are connected using a 4-lane FDR InfiniBand network (Fourteen Data Rate, at 14 Gb/s per lane). It has a bandwidth of 56 Gbps and a latency of 0.7 microseconds.

We compiled OpenFPM and ran our experiments using gcc 6.2.0 and OpenMPI 4.0.0. The distributed rendering application, `scenery-insitu`, was built using AdoptOpenJDK 11.0.7.

In the experiments described in Section 5, for every compute node used, all tasks were launched on the same CPU socket. The number of processes launched per node was therefore limited to 20 (the number of cores in the socket), and all processes were bound to the CPU socket. Nodes were requested with exclusive access.

For the measurement of rendering frame rates (Figure 4), a total of either 16 or 19 processes were launched on each compute node, depending on the testing instance as mentioned in the legend inset in the figure. Processes were not bound to CPU-core, but to socket, allowing the multi-threaded visualization process to make use of multiple CPU cores within the socket, as available. On each node, one process performed visualization, while all others performed simulation, except on one node, where one process performed TCP communication with a remote client instead of simulation. To illustrate with an example, if 12 compute nodes were used with 16 processes each, 11 of the nodes would run 1 visualization process and 15 simulation process, while the 12th node would run 1 visualization process, 14 simulation processes, and 1 master process performing TCP communication.

The benchmarks were carried out in interactive sessions, with a script triggering camera viewpoint changes from a remote display client. The script looped between 4 viewpoints inside and around the data, with an interval of between 2 and 5 seconds between successive viewpoint changes. To ensure that the data occupied a large part of the camera viewport, different values of `pixel-to-world-ratio` in *scenery* 's Volume Manager were selected for the different data sizes; for grid resolution of 1500^3, the value was 0.0036, while for 2048^3 it was 0.002. The code for the display client, which also triggered the camera viewpoint changes used in the benchmarks, can be found at https://git.mpi-cbg.de/mosaic/insituclient/-/blob/afe500ccb9f3bc4e5a89f35d08ff88f31db2c0e3/src/test/kotlin/graphics/scenery/insituclient/client.kt while the initialization set up used in the Gray-Scott simulation for benchmarking can be found at https://git.mpi-cbg.de/openfpm/openfpm$_p$data/-/blob/7a5dd37a43d39bb66b10f3a32490a24b5c4ca4ac/example/Grid/3$_g$ray$_s$cott$_3$d/main$_m$odified.cpp.

We explain the measurement of simulation time step t_{sim} and visualization time step t_{vis}, as reported in Table 1. In case of the synchronous execution we measured t_{sim} with *in situ* visualization turned off and the simulation running on all cores within one socket of each node. Visualization time, t_{vis}, was measured by running the simulation for a single time step, and launching the visualization thereafter, allowing it access to the complete set of resources. In case of asynchronous execution, t_{sim} and t_{vis} were measured with the simulation instrumented with *in situ* visualization and running using our asynchronous architecture.

When measuring the overhead of *in situ* visualization on the simulation (Table 2), eleven of the 12 nodes ran a total of 19 processes (18 simulation + 1 visualization), while the 12th node ran 20, with the master process additional to the simulation and visualization processes.