Eurographics Symposium on Parallel Graphics and Visualization (2016), pp. 1–7
W. Bethel, E. Gobbetti (Editors)

The In Situ Terminology Project

Send Hank an email when you know you want to be a co-author

Abstract

The term "in situ processing" has evolved over the last decade to mean both a specific strategy for processing data and an umbrella term for a processing paradigm. The resulting confusion makes it difficult for visualization and analysis scientists to communicate with each other and with their stakeholders. To address this problem, a group of approximately fifty experts convened with the goal of standardizing terminology. This paper summarizes their findings and proposes a new terminology for describing in situ systems. An important finding from this group was that in situ systems can be described via multiple, distinct axes: integration type, proximity, access, division of execution, operation controls, and output type. This paper discusses these axes, evaluates existing systems within the axes, and explores how currently used terms relate to the axes.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

For decades, the dominant paradigm for visualization and analysis has been "post hoc" processing. With post hoc processing, simulation codes save data to permanent storage (e.g., "spinning disk"), and visualization and analysis programs load this data after it is stored. Simulation codes typically store data iteratively, checkpointing the state of the simulation at a given time (a "time slice"), advancing for a while, saving their state again, and so on.

The alternate paradigm to post hoc processing is to process data as it is generated. This paradigm, including all of its possible instantiations, is often referred to as "in situ" processing. That said, in situ is likely not the best term to use to describe the paradigm. The phrase "in situ" comes from Latin, and translates to "on site," "in position," or "in place." When a visualization or analysis algorithm is applied to simulation data and that data is not being moved (i.e., is already in the processor's registers), then the term in situ is appropriate. But the notion of processing data as it is generated is broader than just data in registers. If simulation data is moved to distinct resources on the same cluster, for example nodes dedicated for visualization and analysis on a supercomputer, then the "in situ" description seems more dubious. On the one hand, this term can be viewed as correct, since the data is being processed "in place" as it stayed on the same computer (or supercomputer). On the other hand, if the data is moved to distinct resources, then is it still being processed "in place"?

While the term "in situ" is dominant today, early research used equally applicable terms, often in reference to specific variants: "concurrent processing" [EGH*06] to refer to processing data at the same time as the simulation is running, "co-processing" [Hai94, HB95, HE97, FMT*11] to refer to visualization routines directly coupled with simulation code, and "runtime visualization" [Ma95, IPD*07] to refer to applying visualization in place. In each case, the terms used previously were likely as suitable in describing this processing paradigm as the "in situ" term, although they did not ultimately garner the same popularity.

Despite the presence of alternate, perhaps more appropriate terms, our group ultimately decided to continue using "in situ" when describing the paradigm that processes data as it is generated, although consensus was not achieved on this point. In a vote, 70% of our participants supported continuing to use the "in situ" term, in large part because it had too much inertia to reverse course. In particular, it was noted that this term has been adopted by our stakeholders and funding agencies, and promoting an alternate term — even if more correct — could create confusion. On the other side, 30% of our participants voted that we should focus on a more appropriate term.

An important contribution of this effort is in identifying the six axes that we feel describe in situ systems. Our axes show that there are a diverse set of approaches behind the paradigm devoted to processing data as it is generated. Another important contribution is our new proposed terminol-

ogy for in situ systems. In our terminology, an in situ system is described by stating its options for each of our six axes. As a further contribution, we analyze existing systems and terms within the axes.

The paper is organized as follows:

- Section 2 defines the six axes to describe an in situ system, as identified by our group.
- Section 3 describes how to apply our axes in the context of complex workflows.
- Section 4 describes some notional in situ systems, and classifies them according to our axes.
- Section 5 looks at recent in situ systems and classifies them based on our axes.
- Finally, section 6 documents the process used for our group to organize, discuss issues, and reach consensus.

2. Axes of In Situ Systems

Our six identified axes are:

- Integration Type
- Proximity
- Access
- Division of Execution
- Operation Controls
- Output Type

The options for each axis are shown in Figure 1.

2.1. Integration Type

Integration type refers to how the in situ visualization and analysis routines are integrated into the simulation code. In the majority of implementations, the simulation code is aware of the integration and makes calls in support of data marshalling. However, it is also possible to integrate in situ routines without the simulation being aware. We use this distinction — **Application-Aware** versus **Application-Unaware** — as the top-level category describing integration type.

We identified three distinct sub-categories of application-aware integrations, although these subcategories may be viewed as points along a spectrum. The first, **Bespoke**, refers to the case where custom visualization and analysis routines are written specifically for a single simulation code, and is tailored to its needs. This is also sometimes referred to as "embedded routines." The latter two subcategories of application-aware integrations cover configurations where systems are integrated into the simulation code, and data is marshalled into those frameworks via APIs. One subcategory, **Dedicated API**, describes the case where the system is dedicated to visualization and analysis, and so the simulation code is aware that interactions with this API are for the purpose of visualization and analysis. This is the approach used by systems like VisIt/LibSim and ParaView/Catalyst. The other subcategory, **Multi-purpose API**, describes the case where the scope of the system is data, meaning that it includes visualization and analysis, but that it also might include I/O or data movement between components. This is the approach used by systems such as ADIOS. With multi-purpose API, the simulation code may or may not be aware whether the API is doing visualization and analysis tasks. We still refer to this case as Application-Aware, since the simulation code is aware of the framework's API, and does data marshalling to support the framework.

We identified two subcategories of application-unaware integration types. That said, the application-unaware approach is relatively new for in situ processing, and new subcategories may need to be added as this approach evolves. **Interposition**, the first subcategory, refers to the practice of creating a dynamically-loaded library which contains symbols known to the simulation code, and inserting this library into the place of the original library that the simulation code was expecting. For example, if a simulation code writes data using the MPI-IO library, then an interposition approach would create a new library with function names matching those of MPI-IO, would have its implementations of those functions perform in situ processing, and would swap the new library in for the MPI-IO library at runtime. **Inspection**, the second subcategory, refers to the practice of inspecting memory to infer patterns in data layout and automatically add in situ processing. Inspection-based in situ relies on system facilities used by tools such as debuggers and profilers.

There are three main considerations motivating the five categories of integration type. One is the effort to integrate the in situ routines into the simulation code (referred to here as "simulation code effort"). Another is the effort to develop the in situ system (referred to here as "in situ system effort"). The final consideration is the reusability of the in situ system across multiple simulation codes. These last two considerations are related, as increasing reusability likely increases in situ system effort. Bespoke approaches often require minimal simulation code effort (since they are tailored to the simulation code) and in situ system effort (since the approach often requires a trivial system), but its reusability is often highly limited. Dedicated API and Multi-purpose API require much more simulation code effort and in situ system effort, but often have higher reusability. The application-unaware categories may require the highest in situ system effort, but they require no simulation code effort (by definition), and the reusability possibilites are high.

2.2. Proximity

The Proximity axis characterizes the cost to access data. This cost could be in time (how fast can we access data?) or in energy (how much energy is required to access data?).

When considering proximity, it is important to consider the path from where the data resides to where it should be

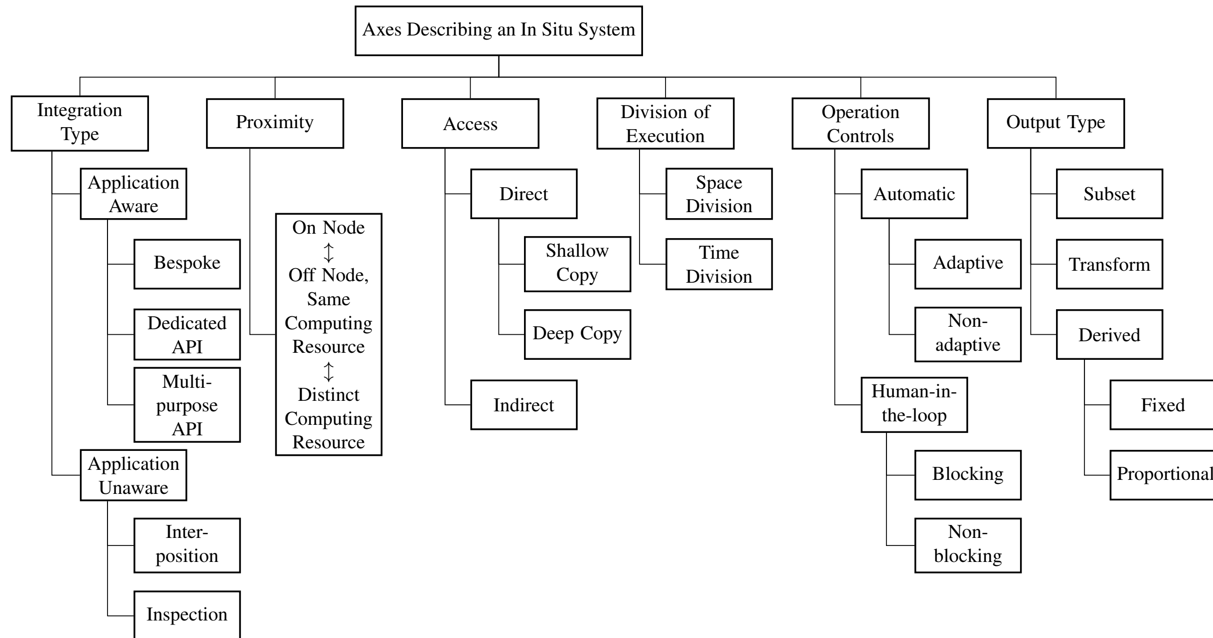

Figure 1: *The top of this diagram has our six basic axes describing in situ systems. Underneath each of the six axes are its corresponding categories and sub-categories.*

processed. That said, there are myriad possible configurations this path can take. As such, we view this axis as a continuous spectrum, not a discrete one with a fixed number of choices. This is particularly true given innovations in architecture, as any attempt to enumerate all options would likely become stale quickly.

We group options for proximity into three broad categories:

- **On Node**
- **Off Node, Same Computing Resource**
- **Distinct Computing Resource**

With **On Node** access, the memory hierarchy forms the basic model. Closest access is when visualization or analysis algorithms are applied to data that is in the memory data registers, followed by options such as L1-cache, L2-cache, L3-cache, and random-access memory. Beyond this are options such as NUMA accesses to memory on other sockets, non-volatile memory on node, and local disks. Placement for each of these options onto a spectrum requires understanding of latency and bandwidth, and may vary based on architecture, especially as hardware components improve over time (e.g., NVLink).

With **Off Node, Same Computing Resource**, there are fewer options: traveling one switch between nodes, two switches, etc. Of course, within a node, there still may be costs occurred on node, i.e., costs from pulling data from NVRAM on a node to send it over the network to another node, which then places it in an accelerator's memory. In these cases, we believe all costs incurred along the path from where the data originally resides to location where it is processed should be considered.

The last option, **Distinct Computing Resources**, strains the usage of the term "in situ." Further, it is worth noting that a recent Department of Energy workshop on workflows [DP*15] drew the line at distinct computing resources, stating that this use case should no longer be considered as in situ. On the other side of the argument are use cases where data is streamed (maybe in a reduced form) from the simulation's source to scientists in remote locations, who can then explore the data using local resources.

2.3. Access

Access refers to how the simulation makes data available to visualization and analysis routines. The main options for access are **Direct** access (where the in situ routines run in the same logical memory space as the simulation code) and **Indirect** access (where in situ routines run in a distinct logical memory space from the simulation code). Sometimes Access is conflated with Proximity, because **Direct Access** often occurs with **On Node Proximity** and **Indirect Access** often occurs with **Off Node Proximity**. However, these axes can pair oppositely. For example:

- **Direct Access** and **Off Node Proximity** pair when a simulation code exposes data via remote direct memory

access (RDMA) or a partitioned global address space (PGAS).
- **Indirect Access** and **On Node Proximity** pair when the simulation code and in situ routines ran as separate processes (to minimize integration effort) and exchanged data via files on NVRAM.

Within **Direct Access**, we distinguish between **Deep Copy** and **Shallow Copy** implementations. With **Deep Copy** implementations, in situ routines make a copy of their input data from the simulation. This is often because the routines use a fixed data structure, and this data structure does not match the simulation code, for example because the simulation stores data in column-major order and the in situ routine assumes row-major order. This approach is expedient for software development, but suboptimal in terms of resources, specifically using extra memory and taking extra time to copy data. **Shallow Copy** implementations, on the other hand, adapt their data structures to those of the simulation code. SCIRun [JPW00] did this by using a templated approach and adapting to the simulation code at compile time. EAVL [MAPS12] did this making a data representation that contained other arrays, including arrays from simulation data, and then accessing data via a layer of indirection. VTK [SML04] is taking a similar approach to EAVL, although it handles variation in data layout via virtual functions.

Both **Direct Access** and **Indirect Access** must worry about hazards, although they manifest themselves in different ways. In both cases, the type of hazard is a Write-After-Read (WAR): the simulation tries to deliver a new data item, typically a new time slice, before the in situ routine has finished working on its current data item. Whether **Direct Access** or **Indirect Access**, this hazard will lead to unpredictable program behavior, including likely corrupting the resulting output. Further, the unsynchronized data write may result in a program failure for the in situ system, the simulation code, or both. Thus, in both cases, data access has to be synchronized. For the **Direct Access** case, this can be achieved using standard synchronization methods, e.g., mutexes. For the **Indirect Access** case, this can be achieved via a communication protocol between simulation code and in situ system. There are several options for dealing with a simulation producing data faster than the in situ system can handle. These include: stalling the simulation until new data can be taken on; buffering raw data, which will, however, drive up memory consumption; and aborting the in situ routine and restarting it on the new data.

2.4. Division of Execution

Division of Execution refers to how and when compute resources are divided between simulation and in situ routines. The two categories within Division of Execution are:

- **Space Division**. The physical resources (space) are divided between simulation and in situ routines throughout the execution (in time) of the in situ system. That is, a subset of the compute resources is exclusively dedicated to in situ routines.
- **Time Division**. Some (or all) of the compute resources alternate between advancing the simulation and visualization and analysis. That is, no compute resources are exclusive to in situ routines.

The resources required by in situ routines are generally less than those needed by the simulation, often by a significant amount. Regardless, division of execution between simulation code and in situ system is a critical issue for the efficacy of the overall system — allocating insufficient resources or insufficient duration to an in situ system can slow down the simulation code. That said, it is sometimes difficult to assess the necessary computational resources and duration for an in situ system to complete its tasks, since factors such as algorithm scalability, computational bottlenecks, and sensitivities to data layouts have large impacts on performance. Fortunately, the division need not be fixed, as the simulation can choose to adapt resource usage with many combinations of integration type, proximity, and access.

Each division strategy has potential benefits and pitfalls, with manifestations varying across in situ configurations. **Space Division** facilitates both the efficient execution of the simulation as well as the appropriation of an ideal set of resources to in situ routines. However, variations in the scales and runtimes of the routines could lead to underutilized or oversubscribed subsets of resources. Managing this synchronization as well as possibly necessary data transfers may require significant additional infrastructure. **Time Division** requires substantially less (or no) synchronization and data transfer efforts. However, while in situ routines are frequently I/O bound in many instances, optimal efficiency is contingent upon data partitioning. For instance, the parallel scaling of visualization algorithms relies on infrastructure which can be sensitive to the size and shape of data domains, for example ghost data generation. The domain decomposition native to a simulation is sometimes unfavorable for analysis and visualization, an issue more easily addressable with post processing or **Space Division**. Inefficiencies arising in **Time Division** are particularly costly as they correspond directly to periods where the simulation is needlessly unable to progress.

2.5. Operation Controls

Operation Controls describes the mechanism for selecting which operations are executed during run-time. We identify two major categories within operation controls — **Automatic** and **Human-in-the-Loop** — both of which have sub-categories.

With **Automatic** Operation Controls, users select which operations to perform in advance of the calculation, and there is no human-in-the-loop during the simulation's ex-

ecution. Within this category, we have identified two sub-categories. With the **Adaptive** sub-category, the in situ routines can adapt which operations are performed as the simulation executes. As an example, a key criteria may trigger the execution of some routines that were not executed otherwise. With the **Automatic: Non-adaptive** sub-category, the in situ routines are static.

With **Human-in-the-Loop** Operation Controls, stakeholders modify which visualization and analysis routines are executed in situ. With the **Blocking** sub-category, the simulation can pause when waiting for guidance from a stakeholder. With the **Non-blocking** sub-category, simulation will not pause to wait for input from a stakeholder.

2.6. Output Type

Output Type describes what operations are performed on the simulation data before it is output (meaning either stored or sent to another in situ sub-system). We identify three major categories for output type: **Subset**, **Transform**, and **Derived**.

Subset refers to operations where a subset of the data is selected, and the rest is discarded. Examples include subsampling (i.e., coarse versions of the data), focusing on regions of interest, or extracting portions with a certain property, as with query-driven visualization [SSWB05] or as with topologies queries [HLH*16](i.e., N largest connected components).

Transform refers to operations that are performed on each element of the data. Our notion of **Transform** does not include reduction, meaning that we expect the data sets created by the transformation process are the same order as the input data. Wavelet transformations would be an example of a transform that may be applied in situ.

Derived refers to operations that generate new data of a different nature than the input. Within the **Derived** category, we consider two sub-types: **Fixed** and **Proportional**. Products of **Fixed** operations are independent of the input size. Examples include statistical summarizations and rendered images (when the image size is fixed). Products of **Proportional** operations vary based on input size. Examples include isosurfaces, indexing, intermediate visualization representations, and topological analysis. Some operations can be used in either **Fixed** or **Proportional** approaches. For example, Lagrangian basis flow extraction [ACG*14] can output a fixed size (and potentially miss information about the vector field) or a proportional size (and thus be more likely to capture information about the vector field).

Finally, the value for Output Type for an in situ routine can be more than a single entry. For example, wavelet compression can be accomplished by first doing a wavelet transform, and then discarding the least important coefficients. This would be categorized as **Transform | Subset**, which indicated that the data is transformed before being reduced by a subset operation. Finally, some large, dedicated in situ systems offer many simultaneous output types, and may need multiple descriptions to describe those outputs.

3. In Situ Workflows

In situ systems sometimes operate in a form where there are multiple, distinct sub-systems, which operate in a workflow-like fashion. That is, sub-system “A" will transform data and transport it to sub-system “B", sub-system “B" will transform data and transport it to sub-system “C", and so on. Of course, the flow of data does not need to be sequential from “A" to “B" to “C", but instead can flow in arbitrary ways, including forming cycles, acting as a source for multiple sub-systems, accepting input from multiple sources, etc. In some cases, “A", “B", etc., are the same program, but this program was invoked in a way that causes it to function differently. In other cases, the sub-systems are distinct programs, but those programs come from the same source code repository, and are branded under the same product name. In still other cases, the sub-systems are truly distinct pieces of software.

For our categorization, we classify each sub-system in the workflow separately. The classification of a workflow with (for example) three sub-systems would be a 6x3 matrix. That said, many workflows contain sub-systems that do not relate to visualization and analysis; when classifying an in situ system, we recommend only including sub-systems that do visualization and analysis operations in a categorization.

4. Classifying Example In Situ Systems

In this section, we describe three notional systems, and classify them according to our axes. Also, note that the terms used in the subsection headings (tightly-coupled, loosely-coupled, hybrid in situ) can have multiple interpretations, but we believe the examples specified fall within most accepted definitions.

4.1. Example 1: Notional Tightly-Coupled System

The following system is classified in Table 1: A simulation code links an in situ library into its code. When the simulation code calls a function in the in situ API, it both specifies the operations to perform and sends data to operate on. The simulation code's usage of the API is static; the simulation code compiles against the API, and the same function is called at a regular interval. When the simulation code invokes the in situ function, the in situ library immediately executes its operations on the same hardware, first transforming it to its own data model, then applying the specified operations, and finally creating images that are saved to disk. The function then returns and the simulation code resumes execution.

Integration Type	Dedicated API
Proximity	On Node
Access	Direct: Deep Copy
Division of Execution	Time Division
Operation Controls	Automatic: Non-adaptive
Output Type	Derived: Fixed

Table 1: *Classification of the in situ system in Example 1.*

4.2. Example 2: Notional Loosely-Coupled System

The following system is classified in Table 2: A simulation code links in an API for data management. When the simulation code calls functions in the in situ API, it believes it is doing I/O operations. However, the in situ library instead sends data to remote nodes. These remote nodes are dedicated to visualization and analysis. A user is running a visualization and analysis tool on the remote nodes, interacting with the data as it comes over the network. When a new time slice comes over the network, the data the user was looking at is flushed and replaced with the new data.

Integration Type	Multi-purpose API
Proximity	Off Node
Access	Indirect
Division of Execution	Space Division
Operation Controls	Human-in-the-loop: Non-blocking
Output Type	Derived: Fixed

Table 2: *Classification of the in situ system in Example 2.*

4.3. Example 3: Notional Hybrid In Situ System

The following system is classified in Table 3: The sole purpose of this system is to render isosurfaces. In this system, the isovalues desired result in a sparse isosurface (i.e., few triangles compared to the number of cells), so, when data is produced, an isosurfacing routine is immediately applied. This routine was written specifically to work on data from this simulation code. The resulting triangles are sent over the network to dedicated visualization nodes, using a data transfer library. There, separate visualization software renders the data. Since the location of the isosurface varies, the software evaluates the data set and determines the best camera angles to capture the data. It saves the resulting images to disk.

Finally, note that if the components of this system had non-sequential flow (i.e., splitting output, cycles, etc.), then the matrix format (6x2 in this case) would need to be adapted to better capture the flow, likely as a graph with each node containing a component's six axis options.

Integration Type	Bespoke	Multi-purpose API
Proximity	On Node	Off Node
Access	Direct: Shallow Copy	Indirect
Division of Execution	Time Division	Space Division
Operation Controls	Automatic: Non-adaptive	Automatic: Adaptive
Output Type	Derived: Proportional	Derived: Fixed

Table 3: *Classification of the in situ system in Example 3.*

5. Classifying Existing In Situ Systems

There is a lot of material in the Google Docs. I will be copying that content over in the coming days.

6. Process for In Situ Terminology Project

Two paragraphs about the process for getting here.

7. Conclusion

Paragraph 1: Summarize what we did.

Paragraph 2: Talk about the future: this should be a living document, updated as necessary. Example: fixed memory footprint. Not a thing now, but might be in the future. Another example: handling of hazards. This is a mix of looking backwards and forwards, but there is farther forward still.

[CMY*12]

References

[ACG*14] AGRANOVSKY A., CAMP D., GARTH C., BETHEL E. W., JOY K. I., CHILDS H.: Improved Post Hoc Flow Analysis Via Lagrangian Representations. In *Proceedings of the IEEE Symposium on Large Data Visualization and Analysis (LDAV)* (Paris, France, Nov. 2014), pp. 67–75. 5

[CMY*12] CHILDS H., MA K.-L., YU H., WHITLOCK B., MEREDITH J., FAVRE J., KLASKY S., PODHORSZKI N., SCHWAN K., WOLF M., PARASHAR M., ZHANG F.: In Situ Processing. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. CRC Press/Francis–Taylor Group, Oct. 2012, pp. 171–198. 6

[DP*15] DEELMAN E., PETERKA T., ET AL.: *The Future of Scientific Workflows*. Tech. rep., Report of the DOE NFNS/CS Scientific Workflows Workshop, April 2015. 3

[EGH*06] ELLSWORTH D., GREEN B., HENZE C., MORAN P., SANDSTROM T.: Concurrent visualization in a production supercomputing environment. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 997–1004. 1

[FMT*11] FABIAN N., MORELAND K., THOMPSON D., BAUER A. C., MARION P., GEVECI B., RASQUIN M., JANSEN K. E.: The paraview coprocessing library: A scalable, general purpose

in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on* (2011), IEEE, pp. 89–96. 1

[Hai94] HAIMES R.: pv3: A distributed system for large-scale unsteady cfd visualization. 1

[HB95] HAIMES R., BARTH T.: Application of the pv3 co-processing visualization environment to 3-d unstructured mesh calculations on the ibm sp2 parallel computer. In *Proc. CAS Workshop* (1995). 1

[HE97] HAIMES R., EDWARDS D. E.: Visualization in a parallel processing environment. In *Proceedings of the 35th AIAA Aerospace Sciences Meeting, number AIAA Paper* (1997), pp. 97–0348. 1

[HLH*16] HEINE C., LEITTE H., HLAWITSCHKA M., IURICICH F., DE FLORIANI L., SCHEUERMANN G., HAGEN H., GARTH C.: A survey of topology-based methods in visualization. In *Computer Graphics Forum* (2016), vol. 35, Wiley Online Library, pp. 643–667. 5

[IPD*07] INSLEY J. A., PAPKA M. E., DONG S., KARNIADAKIS G., KARONIS N. T.: Runtime visualization of the human arterial tree. *IEEE Transactions on Visualization and Computer Graphics 13*, 4 (2007), 810–821. 1

[JPW00] JOHNSON C., PARKER S., WEINSTEIN D.: Large-scale computational science applications using the SCIRun problem solving environment. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing* (2000). URL: http://www.sci.utah.edu/publications/crj00/super00_final.pdf. 4

[Ma95] MA K.-L.: *Runtime Volume Visualization for Parallel CFD.* Tech. rep., DTIC Document, 1995. 1

[MAPS12] MEREDITH J. S., AHERN S., PUGMIRE D., SISNEROS R.: EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), The Eurographics Association, pp. 21–30. 4

[SML04] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*, fourth ed. Kitware, Inc., 2004. ISBN 1-930934-19-X. 4

[SSWB05] STOCKINGER K., SHALF J., WU K., BETHEL E. W.: Query-driven visualization of large data sets. In *IEEE Visualization* (2005), vol. 5, p. 22. 5