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.

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"?

ISSUE #1: NEEDED: Survey of previous terms, including concurrent, co-processing, and runtime

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, like concurrent processing.

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 terminology 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.

This paper is organized as follows:
….

2. Axes

Our six identified axes are:
• Integration Type
• Proximity
• Access
• Division of Execution
• Operation Controls
• Output Type

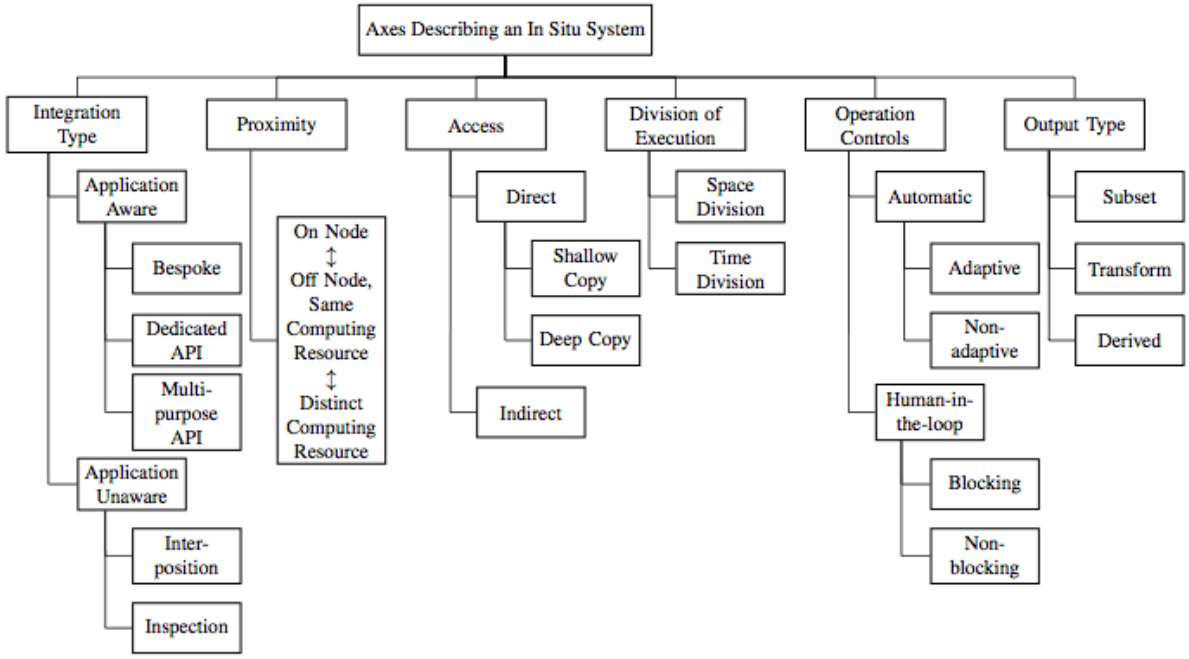

Figure 1: *The top of this diagram has our six basic axes describing in situ systems. Underneath each of the six axes are its corresponding categories and sub-categories.*

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 sub-categories 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 sub-categories of application-aware integrations cover configurations where systems are integrated into the simulation code, and data is marshalled into those frameworks via APIs. One sub-category, **Dedicated API**, describes the case where the framework 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 sub-category, **Multi-purpose API**, describes the case where the scope of the framework 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 like 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 sub-categories of application-unaware integration types. That said, the application-unaware approach is relatively new for in situ processing, and new sub-categories may need to be added as this approach evolves. **Interposition,** the first sub-category, 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 sub-category, 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 possibilities 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?).
ISSUE #2: do we want to distinguish between one retrieval (indirect access) vs many (direct access)?

When considering proximity, it is important to consider the path from where the data resides to where it should be 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 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 incurred 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(reference) 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 by making a data representation that contained other arrays, including arrays from simulation data, and then accessing data via a layer of indirection. VTK [SML96]is there a better citation?? is taking a similar approach to EAVL, although it handles variation in data layout via virtual functions.

ISSUE #3: HAZARDS vs CONCURRENCY HAZARDS
Both direct 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), in this case meaning that the simulation tries to deliver data for a new time slice before the in situ routine has finished working on the current time slice. In the direct access case, the hazard would manifest as the in situ routine working on data that could be removed or overwritten midway through execution. The result of such a hazard is difficult to predict, but would range from a memory error to incorrect results. In the indirect access case, the hazard can be handled during the data exchange. Options would include making the simulation code stall until the in situ routine finishes, or aborting the in situ routine and accepting the new data.

ISSUES #4 and 5:
Asymmetric discussion: For direct access we discuss what goes awry, for the indirect option, we name how to solve it. Maybe we should discuss both aspects for both options.

"Options" ->This seems to be an accommodation to the long gone synchronicity axis. It seems a bit out of place compared to the other sections that methodically traverse the specific branch. Is there classification to be teased out based how a system deals with this issue? These two solutions are presented pretty negatively (stall, abort). Is there a reasonable technique or two?

2.4 Division of Execution

Division of Execution refers to how compute resources are divided between simulation and in situ routines. The two main options are:
• Space Division. A subset of the compute resources are devoted exclusively to in situ routines.
• Time Division. No compute resources are devoted exclusively to in situ routines. Some (or all) of the compute resources alternate between advancing the simulation and visualization and analysis.

Time division: talk about baton passing, and goldrush (opportunistic)

ISSUE #6: Need more here. Maybe Tom Peterka can help?

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 execution. 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, some key criteria may trigger the execution of some routines that were not executed otherwise. With the **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, the simulation will continue to advance.

Note: the first time our group discussed this axis, we came up with the categorization above. The second time we discussed this axis, new concerns were raised about user controls that specify the resources the operations should execute on. That is, while the describe above focuses on which operations are applied, there is a missing discus- sion of where they are applied ... at least with respect to the user potentially having control over them. This missing perspective still needs to be incorporated in this document in some way, possibly as a new

access. Finally, I note that the phrases "logical operation controls" and "physical operation controls" were used to capture the difference between what is in the current text (logical) and the missing perspective (physical).

ISSUE #7: REMOVE THE ABOVE OR NOT?

ISSUE #8: discuss computational steering?

ISSUE #9: examples?

2.6 Output Type

Output Type describes what operations are performed to 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**.

ISSUE #10: Subset->Filter?

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 or as with topologies queries (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 render to 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 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).

ISSUE #11: Topological queries in subset, topological analysis in filters

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 may 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 them 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.

ISSUE #12: comment from Michel Rasquin that this should be extended to deal with more complex, graph-like topologies (instead of pipelines with one input and one output)

4. Classifying Example In Situ Systems

In this section, we describe three example 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: Tightly-Coupled

The following system is classified in Table 1: A simulation code links an in situ library into their 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; it is compiled in 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

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

ISSUE #13: group suggestion is that this example be streamlined to its essence, and non-essential axes be removed

4.2. Example 2: Loosely-Coupled

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	Subset

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

4.3. Example 3: Hybrid In Situ

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 on the same node. 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.

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

ISSUE #14: need more examples

For now, let's just add a giant table. Please add a paragraph, table entry, and reference at the end for whatever system you add.

Here's a paragraph for VisIt:

VisIt provides in situ capabilities via its LibSim [#1] library. This library has a dedicated API which simulation codes use to pass data and hand off control. The data from the simulation code is often deep-copied, but the simulation's arrays are shallow-copied when their format agrees with VisIt's data model (e.g., row-major vs column major). LibSim executes algorithms using the same resources as the simulation code, and execution alternates between simulation code and in situ routines. There are two modes for directing the visualization operations to perform. With the first mode, Python scripts are set up ahead of time and employed at regular intervals. With the second mode, end users can connect to the in situ library via VisIt's GUI and adaptively direct which routines to apply. Outputs include anything that VisIt can generate. For the most part, this involves saving images, but it is possible to save out subsets of data.

ParaView [#4] provides in situ analysis and visualization with the Catalyst [#5, #8] library. Similar to LibSim, it uses a dedicated API for driving the in situ operations. It relies on the adapter design pattern, customized for each simulation code interfacing with Catalyst, to construct objects in the VTK data model from direct access to the simulation code's data structures. In the adapter, the grids are typically deep copied while for arrays the preferred access is through shallow copying. Shallow copying of arrays can be done for both array-of-structures and structure-of-arrays memory layouts. Automatic operation controls can be done through either C++ routines or Python routines. Logic can be added to these to support adaptive automatic operation controls. For Human-in-the-Loop operation controls, the analyst connects the ParaView GUI to a running simulation and provides both blocking and non-blocking functionality.

QIso [#6] is a library that is dedicated to generating and rendering isosurfaces in MPI-based simulations. It uses marching cubes on structured grids obtained directly from the simulation's arrays, renders to an OpenGL off-screen Mesa (OSMesa) buffer, and parallel-composites the result via MPI to a single image.

Name	Ref	Integration Type	Proximity	Access	Division Execution	Operation Controls	Output Type
VisIt	#1	Dedicated API	On Node	Direct: Shallow Copy (when possible), Deep Copy (when not)	Time Division	Automatic: Non-adaptive or Human-in-the-loop: Non-blocking	Mostly Derived: Fixed
Catalyst	#4, #5, #8	Dedicated API	On Node	Direct: Shallow Copy (when possible), Deep Copy (when not)	Time Division	Automatic: Adaptive, or Human-in-the-loop: Non-block or blocking	Subset, Derived: Fixed, Derived: Proportional
QIso	#6	Bespoke	On Node	Direct: Shallow Copy	Time Division	Automatic: Non-adaptive	Derived: Proportional
Damaris	#7, #8	Dedicated API	On Node or dedicated	Direct (Shallow copy or	Time Division or Space	Automatic (non-adaptive),	Backend-dependent

			nodes	deep copy) or Indirect	Division	or Human-in-the-loop (non-blocking or blocking)	

Please add citations to systems you are familiar with!!

6. Process for In Situ Terminology Project

Maybe a paragraph of two about the process for getting here.

ISSUE #15: need text

7. Conclusion

ISSUE #16: need text

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.

References:

#1:
@inproceedings{DBLP:conf/egpgv/WhitlockFM11,
  author   = {Brad Whitlock and
               Jean M. Favre and
               Jeremy S. Meredith},
  title   = {Parallel In Situ Coupling of Simulation with a Fully Featured Visualization
               System},
  booktitle = {Eurographics Symposium on Parallel Graphics and Visualization, {EGPGV}
               2011, Llandudno, Wales, UK, 2011. Proceedings},
  pages   = {101--109},
  year   = {2011},
}

#2
```
@inproceedings{SCI:SCIRun,
  author =        "Chris Johnson, Steve Parker and D. Weinstein",
  year =          "2000",
  title =         "Large-scale computational science applications using the SCIRun problem
solving     environment",
  booktitle = {ACM/IEEE conference on Supercomputing. Proceedings},
}
```

#3:
```
@inproceedings{EAVL:conf/egpgv/Meredith12,
  author    = {Jeremy S. Meredith, Sean Ahern, Dave Pugmire, Robert Sisneros},
  title     = {EAVL: the extreme-scale analysis and visualization library},
  booktitle = {Eurographics Symposium on Parallel Graphics and Visualization, {EGPGV}, 2012.
Proceedings},
  pages     = {21-30},
  year      = {2012},
}
```

#4:
```
@book{PVGuide,
  author = {Ayachit, Utkarsh},
  title = {The ParaView Guide: A Parallel Visualization Application},
  publisher = {Kitware},
  year = {2015},
  isbn = {978-1930934306}
}
```

#5:
```
@techreport{Catalyst:techreport/sandia/2013_1122,
  author = {David Rogers, Kenneth Moreland, Ron Oldfield, and Nathan Fabian},
  title = {Data Co-Processing for Extreme Scale Analysis Level II ASC Milestone (4745)},
  institution  = {Sandia National Laboratories},
  year         = 2013,
  number       = 1122,
  month        = 11
}
```

#6:
```
@inproceedings{QIso:conf/isav/Ziegeler15,
  author    = {S. Ziegeler,  C. Atkins, A. Bauer, L. Pettey},
  title     = {In Situ Analysis as a Parallel I/O Problem},
```

```
  booktitle = {SC'15 Workshop: First Annual Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization },
  year      = {2015}
}

#7: @inproceedings{dorier2012damaris,
title = {{Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O}},
author = {Dorier, Matthieu and Antoniu, Gabriel and Cappello, Franck and Snir, Marc and Orf, Leigh G.},
booktitle = {IEEE International Conference on Cluster Computing (CLUSTER)},
pages = {155--163},
year = {2012},
url = {https://hal.inria.fr/hal-00715252},
organization = {IEEE},
}
#8: @inproceedings{dorier2013damarisviz,
title = {{Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework}},
author = {Dorier, Matthieu and Sisneros, Roberto R. and Peterka, Tom and Antoniu, Gabriel and Semeraro, Dave B.},
url = {https://hal.inria.fr/hal-00859603},
booktitle = {{IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)}},
address = {Atlanta, United States},
year = {2013},
month = oct,
pdf = {https://hal.inria.fr/hal-00859603/file/paper.pdf},
hal_id = {hal-00859603},
hal_version = {v1},
}

#8: @book(catalystUG,
  author = {Andrew C. Bauer and Berk Geveci and Will Schroeder},
  title = {{The ParaView Catalyst User's Guide v2.0}},
  publisher = {{Kitware, Inc.}},
  year={2015}
)
```