To Derive or Not to Derive: I/O Libraries Take Charge of Derived Quantities Computation

Ana Gainaru, Norbert Podhorszki, Liz Dulac, Qian Gong, Scott Klasky
Oak Ridge National Laboratory
gainarua@ornl.gov

Greg Eisenhauer
Georgia Institute of Technology

Antonios Kougkas, Xian-He Sun
Illinois Institute of Technology

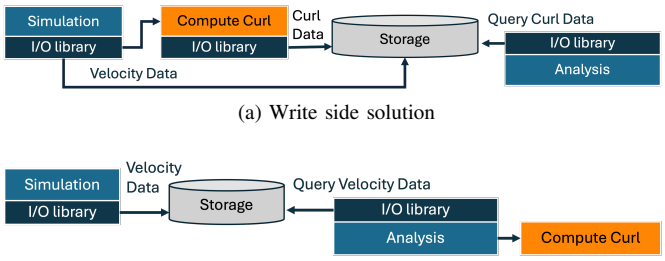
Jay Lofstead
Sandia National Laboratory

***Abstract*—The ever-increasing volume of data produced by HPC simulations necessitates scalable methods for data exploration and knowledge extraction. Scientific data analysis often involves complex queries across distributed datasets, requiring manipulation of multiple primary variables and generating derived data that needs to be handled efficiently, creating challenges for applications that need to parse many large datasets. Relying on individual applications to handle all intermediate data generally leads to redundant computations across studies and unnecessary data transfers. In this paper, we investigate the performance of different approaches where applications define derived variables as quantities of interest (QoIs) and offload the computation and transfer of these QoIs to the I/O library. This significantly reduces redundancy and optimizes data movement across the distributed storage and processing infrastructure by allowing control over when and where derived variables are computed. We present a detailed analysis of the performance-storage trade-offs associated with different solutions and showcase results for our study on two large-scale datasets created from climate and combustion simulations.**

***Index Terms*—Large-scale I/O, Derived Variables, HPC Analysis, Queries for Scientific Data, HPC Quantities of Interest**

I. Introduction

HPC simulations are rapidly outpacing our ability to store and analyze the deluge of data they produce. Scientific codes, like S3D [16] (high-fidelity simulation of turbulent combustion) or XGC [26] (global gyrokinetic particle-in-cell code), are currently generating several TB of data for each simulation step. In order to find insights in such large datasets, scientific workflows often require complex queries to be executed across these distributed datasets, involving manipulation of multiple primary variables. For example, domain scientists working with satellite images might require visualizing portions of data identified by new variables that capture change events and trends from the data and that are not directly stored by the application (e.g. query on the slope of a linear regression of a vector when only the vector data is generated and saved). This is an example of *derived data*, that is, data or quantities of interest that are not specifically the result of the principal calculation of the application, but which can be computed or extrapolated (derived) from that primary result. For the purposes of this paper derived data is interesting in that it can conceivably be generated anytime the primary data is available, from the point at which the primary data is written to the point at which derived data might be queried or consumed. While this conceptual flexibility is valuable, taking full advantage of it presents significant challenges, particularly when dealing with large and numerous datasets.



(a) Write side solution

(b) Read side solution

Fig. 1: Current solution of computing derived quantities of interest on the writer (storing primary data and QoIs) and reader side (storing primary data and computing QoIs).

Current HPC I/O solutions leverage metadata to facilitate efficient querying of scientific data and only transferring data of interest to a given study. However, I/O libraries can only rely on primary data for which metadata is being stored and cannot be used for derived data. Current methods for handling derived data are implemented at the application level either on the writer or reader side (Figure 1). In writer side solutions, users generate and store all required derived data, allowing the I/O libraries to generate metadata and thus allowing the reader to query this data directly. However, these solutions are leading to significant storage overheads. Conversely, the reader side solutions involves reading all or most primary data and generating derived data on-the-fly during analysis, like the approach implemented in Paraview [3] for visualizing derived data. Since it is usually difficult to translate from the query requirements of a derived variable to requirements on primary data, these solutions typically result in excessive data transfer.

This paper argues for the data management layer to include logic for computing derived variables and for allowing applications to offload this task. We investigate the advantages and limitations of different strategies for where the computation to take place and for how the I/O layer can use this logic to speed-up queries. The contributions of the paper are the

following:

- A novel offloading approach where the I/O library stores statistics about derived variables that can be used at query time to optimize the amount of data being transferred. This approach provides a middle ground between the two existing methods and aims to significantly reduce redundancy in data generation and optimize data movement.
- A study of the trade-offs between different solutions. For this purpose, we design a performance model for offloading derived variables and we offer a detailed performance analysis.
- Finally, we showcase the effectiveness of our approach through studies on two large-scale datasets generated by the S3D combustion simulation and the E3SM climate simulation [4].

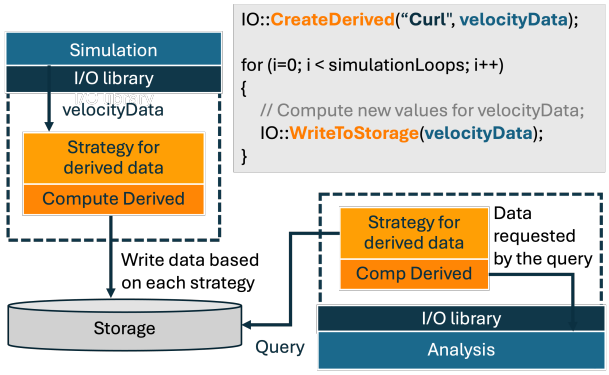


Fig. 2: Extended I/O libraries with strategies for dealing with derived variables can be used by applications to offload their handling and computation.

II. Related work

In scientific analysis, derived variables play a crucial role in unlocking deeper insights from complex simulation data. These derived quantities, obtained by mathematical transformations of primary data, allow researchers to focus on specific aspects of the simulation. For instance, in combustion simulations, calculating the magnitude of the velocity creates a derived variable that effectively identifies areas of high interest, such as regions with intense burning [16]. Similarly, satellite data often captures time series of environmental parameters. By applying a linear regression to an entire year's peak greenness data, scientists can derive a slope variable. This slope, not present in the raw data, reveals crucial trends in vegetation health over time, helping identify areas of potential environmental change [15], [17]. In climate simulations [4], [24], calculating the curl of wind velocity data, researchers can create a new variable representing atmospheric rotation, a crucial factor in understanding global weather patterns. These examples showcase how derived variables act as powerful tools for scientific visualization and analysis, guiding researchers towards a more nuanced understanding of the phenomena under study.

The current practice of utilizing derived variables for insight discovery includes writer side solutions where workflows include analysis codes running with applications computing and storing the required derived data (e.g. [8], [14]) or on the reader side typically by using visualization technology capable of computing derived variables on the fly (e.g. using Paraview [21]). We analyze the performance of these two types of solutions and expose their limitations and advantages in different situations.

Current I/O libraries for HPC are optimized for large scientific simulations providing applications with the flexibility to customize the data structures and support data management, metadata management, and data sharing policies across thousands of processes. Offloading the I/O to a specialized software has the advantage of applying I/O optimizations automatically and allowing the application to utilize features of each system it is running on. This is a common practice for HPC applications that frequently offload their data access to I/O libraries such as HDF5 [25], ADIOS [13], PnetCDF [20], and MPI-IO [5]. These libraries support a rich variety of data structures and optimizations for high throughput by tuning the parameters across multiple I/O layers. In [6], the authors integrate the ADIOS2 I/O library with Hermes [18], an hierarchical buffering, which enables data placement and prefetching across the spectrum of I/O devices. They compute basic derived quantities required by I/O applications and showed the performance implications of using the memory hierarchy to buffer data and metadata on the I/O hierarchy or in metadata databases, like Empress [19]. However, as far as we know, no general purpose HPC I/O libraries specifically deal with derived data as a separate conceptual class of data, at least not beyond allowing its inclusion as an undifferentiated addition to the primary data. The MDSplus [22], a data acquisition and management software package specifically for the fusion community, is the only library that provides basic mathematical expressions to its applications. Very similar to Paraview, the library does not save data or statistic information and it is used as a read time solution. In this paper we will show advantages of treating derived data as conceptually different than primary data. We use I/O libraries to provide an implementation for three strategies that deal with derived variables and seek to understand the performance trade-offs between different strategies.

III. Approach

Our methodology is described in Figure 2. We extended an I/O library to include strategies for dealing with derived variables, allowing applications to use simple APIs to offload the task of handling derived data. A simple example is presented in the upper right side of the figure, where the application is defining a derived variable for Curl over velocity data. During the simulation, the application computes new values for velocity in every loop and gives this data to the I/O library. With our method, the I/O library decides when to compute and how to store the curl data.

I/O libraries designed for HPC (e.g. ADIOS2, HDF5) divide data into blocks to be able to efficiently handle them. For this,

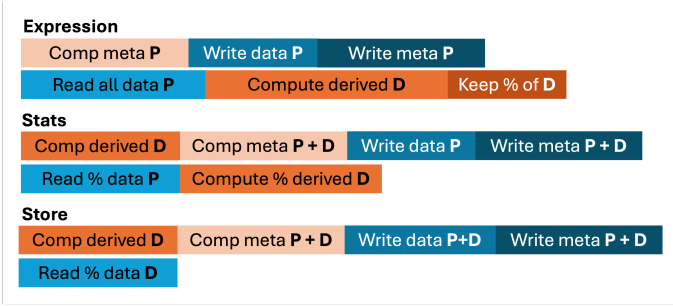


Fig. 3: Stages of three strategies of writing (top bars) and reading (bottom bars) primary and derived variables.

the libraries save metadata together with data to indicate where these blocks are saved in storage. Most libraries, compute and store basic statistics about blocks (like min and max values for the entire block) together with the data. This is useful when reading the data if applications are interested in values with certain properties (e.g. values over a threshold will be stored only in blocks with min values above the threshold). Since analysis codes are relaying more on complex queries that involve math expressions over derived variables, we argue in this paper that it is becoming important to have access to block-based statistics for these derived variables. For example, if an analysis detects cyclones in atmospheric data, it will focus on areas of high atmospheric rotation. Primary data includes velocity but not rotation. The analysis would have to compute the curl of wind velocity data in order to detect the area of interest. If the I/O layer would be aware of this quantity of interest (for e.g. through APIs like the `CreateDerived` method in Figure 2), it could save statistics or data for the curl and, thus the analysis would only transfer the blocks of interest. For this, applications need to offload their derived data handling to an I/O library. We implemented such an abstraction and used it with several derived variables.

Through this abstraction applications can: i) define derived variables on the writer side as mathematical expressions on primary data (specifying that these variables are quantities of interest for the analysis); ii) query data based on values for derived variables on the reader side. We investigate the performance of three strategies that I/O library can take by using the information provided by applications (illustrated in Figure 3):

- Writer side strategy (named **Store**): The derived variables are computed during the write operation and both statistics and data for derived variables are stored on storage. This strategy is the equivalent of write side solutions from literature where the application treats the derived data as primary data. Storage will increase based on how much data the derived function generates and the write operation performance will encounter a hit. However, the reader will be able to query the derived data and transfer only the areas of interest.
- Reader side strategy (named **Expression**): Only the math expression is being saved during the write operation and the derived variables are computed during the read operation. This strategy is the equivalent of read side solutions, like the one implemented in Paraview. There is no overhead in storage and the writer performance is unaffected. The read operation however needs to read all the primary data necessary for computing the derived variables for every query that requires derived data.
- Statistics based strategy (named **Stats**): We propose a new strategy where the derived variables are computed during the write operation and only metadata is stored (statistics about blocks of data). This information will allow readers to transfer only the data relevant to a query, but not have access to derived data directly. If the derived data is needed, the read operation will include the time to read the primary data and compute the derived variables but only for the amount of relevant data.

TABLE I: Notations used in the performance model

Notation	Description
T_W	Total write time
T_R	Total read time
$T_{C\{op\}}$	Total time for computing operation op
$T_{W\{data\}}$	Total time for writing $data$
$T_{R\{data\}}$	Total time for reading $data$
T_{meta}	Time to compute/update metadata
T_{filter}	Time to filter irrelevant data for a query
B_{op}	Bandwidth for operation op
$O_{exp}(f)$	Number of ops in computing $exp(f)$
$D_{exp}(f)$	Amount of data accessed for $exp(f)$
Q_i	Percentage of data that query i will read
S	Size of a variable
D	Storage footprint
$nVar$	Number of variables required by the derived Op

In this section, we design a performance model to understand the implications that different system characteristics and application requirements have on the performance of these three strategies. We look at the trade-offs between reader and writer performance for each scenario and at the total time to write and read between the three strategies.

This section includes three parts: 1) write side model; 2) query model when including derived variables; and 3) storage model. We start by defining the items involved in the model. Primary variables represent N dimensional data structures that are being generated and stored by scientific simulations. Derived variables are represented by a math expression applied on primary data (e.g. magnitude is a derived variable over a vector (x_i, y_i, z_i) given by equation $\sqrt{x_i^2 + y_i^2 + z_i^2}$ where x, y, z are primary variables).

We denote with $V_i(S_i)$ a list of variables, where variable i has size S_i. Our observations showed that the shape of the variables does not change the performance of our implementations, so the model only considers the total size ($S_i = d_0 * d_1 * ... * d_{numDim}$) where d_i is the size for dimension i. Experiments are done with 3D arrays since this structure is the most used by applications but we measured 2D and 4D and the performance characteristics are similar. Table I shows the notations used throughout this paper when defining our model.

A. Write performance model

Writing a variable V_i includes at most four main stages (top bars for each strategy in Figure 3):

1) computing the values of derived variable
2) computing statistics for variables
3) updating the metadata structure to include information about each written variable and statistic
4) move the data, stats and metadata to consumers (either by writing it to storage or streaming it directly to consumer applications)

In this section we will define and model the performance of each stage. The following equation covers all the costs enumerated above for one variable:

$$T_W = T_{C\{derOp\}} + T_{C\{stats\}} + T_{meta} + T_{W\{data\}} + T_{W\{stats\}} + T_{W\{meta\}} \quad (1)$$

a) Computing the values of the variable: Derived variables are not directly provided by user applications so they need to be computed on the fly based on the the values of other variables. The cost to compute is usually given by roofline models (like the ones in [11]) that take into consideration the complexity of the derived expression, the FLOPS of the architecture used to compute the operations and the memory bandwidth. The first term of Eq. 1) includes this cost for computing derived variables ($T_{C\{derOp\}}$. For example, the addition derived expression requires $nVar * S$ ops where $nVar$ is the number of variables being added and S the total size of the variable (for addition all variables have the same size $S_{i=0:nVar} = S$). The speed of the computation is dependent on the memory bandwidth (e.g. between host and device $B_{mem\{GPU\}}$) to load the data needed by the derived expression and the peak FLOPS of the architecture.

$$T_{C\{exp\}} = min(\frac{D_{exp}(nVar, S)}{B_{mem\{arch\}}}, \frac{O_{exp}(nVar, S)}{FLOPS\{arch\}})$$

The time to compute the derived variable is given either by the time it takes to load the operands or the time to compute the operation. The first term divides the amount of data required by an operation (e.g. $D_{exp}(nVar, S) = nVar * S + S$ for loading $nVar * S$ and storing S for the add expression) by the memory bandwidth of the platform. The second term is the ratio between the complexity of the operation which defines the amount of operations required to compute the variable over the FLOPS of the platform. For primary variables this cost is 0 ($T_{C\{derOp\}} = 0$).

b) Computing statistics:: Statistics need to be computed for variables in order to be able to query parts of data without having to read the entire dataset. For example, having min/max for blocks of data, allows readers to only bring blocks that guarantee to have values over (or below) a given threshold, thus reducing the data that needs to be transferred to a consumer (either from storage or from a producer). The same as for computing the derived variables, computing the statistics depends the number of operations required to compute the statistics and on the bandwidth of the computation unit. The number of operations required to compute the statistics of one variable depends on how many statistics are computed (for example, the stats computed by ADIOS2 are min and max for the data on each rank) and on the size of the variable. For example, the number of operations required to compute stats for ADIOS2 would be $2 * log(S)$ where S is the size of the variable since reductions require $log(S)$ operations.

$$T_{C\{stats\}} = min(\frac{S}{B_{mem\{arch\}}}, \frac{log(S)}{FLOPS\{arch\}})$$

We use a simple roofline model to profile the computational performance of different stages since we want to understand the most important factors affecting the performance trade-off between the three strategies and not to predict the exact performance of each strategy. In the future we plan to extend our analysis to more complex performance models, like the ones in CARM [7], MaRM [23].

c) Updating the metadata:: Most modern I/O libraries require to compute and store metadata to be able to locate data and stats related to data during reading. The cost to update the metadata is usually fixed per variable, or per statistic so it only depends on the number of variables and amount of stats computed for each variable. We measured this time and it is negligible compared to the other term, so for the rest of the paper we consider this cost 0.

d) Moving the data:: Writing to storage or streaming the data directly to consumers requires moving data, stats and metadata. The time for writing depends on the amount of data that needs to be transferred, the transferring bandwidth/latency and the strategy used by the I/O library. The first three terms of Eq. 1 typically do not require for ranks to synchronize or exchange any information. Computing the derived variables, statistics and updating the metadata are typically done per rank since synchronization would incur prohibitive costs. Data transfers usually required to aggregate the metadata, stats and/or data to a number of designated writers depending on the strategy each I/O library implements. This costs can be significant so it needs to be included in the model.

$$T_{W\{d\}} = \frac{S_d}{B_{write}} + T_{additional}$$

The cost includes an additional overhead for preparing the data to be transferred $T_{additional}$ that is depended on the I/O algorithm. The data, metadata and stats have different sizes (Section III-B). In addition, each might have different I/O algorithms and thus will have different costs. We model the ADIOS2 library since our implementations rely on the writing algorithms in ADIOS2. For data, each rank is separately writing their local data ($B_{write} = min(peakB_{IO}, peakB_{network})$ and there is no overhead ($T_{additional} = 0$). For stats and metadata, the information is aggregated on rank 0 and then transferred to storage. The aggregation step is using MPI Gather with a delay of $T_{additional} = log(ranks)$. Since the

amount of stats and metadata data is usually small (MBs), the transferred time is usually limited by latency.

e) Performance model: We use Eq. 1 to model all variables, both derived and normal. The following table summarizes the different way the equation is used by each case.

For primary variables: $T_{C\{derOp\}} = 0$
For derived variables:

- Expression strategy: Nothing needs to be computed or written except metadata: $T_W = T_{meta} + T_{W\{meta\}}$
- Stats strategy: The data for the derived variable is not written: $T_{W\{data\}} = 0$
- Store strategy: All terms of the equation will be used.

B. Storage model

The storage size for a primary variable is given by the aggregated local size for each rank for each variable with its corresponding stats, in addition to the size of the metadata. The total storage footprint D_i of primary variable i is represented by:

$$D_i = (S_{statsV_i} + S_{dataV_i} + S_{metaV_i}) * numRanks \quad (2)$$

For derived variables, if only the expression is saved, only the metadata needs to be updated to include the quantity of interest information. The size of the metadata holding the expression is small (in the range of a few bytes) but it increases linearly with the number of ranks ($D_i = k_1 * ranks$ for this case).

If a derived variable requires saving stats but not data, $S_{dataV_i} = 0$ in Eq. 2. Typically the statistics size does not increase linearly with the size of the variables used to compute the derived value. The data is usually divided in equal number of blocks and stats are computed per block (for example, the default behavior in ADIOS2 is to use one block per rank and thus compute min and max values for all the data in one rank). This means that the size of the storage space taken by derived variables in this scenario has a complexity of $O(blocks)$. The metadata size increases for this variable since it needs to hold information about the block stats. For ADIOS2 $D_i = k_2 * ranks$, with $k1 < k2$ are constants and we estimate them based on metadata file sizes of several runs.

If both data and metadata are saved for a derived variable, all terms in Eq. 2 are greater than zero and the metadata size needs to include information about where data blocks are being stored. Typically the size of the derived data increases with the size of the variables used to compute the derived value (for example, for adding existing variables, the derived variable `ADD` can be used and will have the same size as the input variables). We will look at different derived variables in the following sections.

C. Read performance model

The consumer application typically has a list of queries with each query reading Q_i percentage of the data (i.e. $S_i * Q_i$ total size per rank for variable V_i). If all data needs to be read, the request is equivalent in our model for a query with $Q_i = 1$. Querying a variable V_i includes the following stages:

1) Read the metadata and stats of the variable
2) Read the data that fits the query
3) Compute the derived variable (if there is no data stored for the variable)
4) Filter out values that do not fit the variable.

In a similar way to the write performance model, we define and model the performance of each stage in the querying process. The following equation covers all the costs enumerated above for variable V_i:

$$T_R = T_{R\{meta\}} + T_{R\{stats\}} + \sum_{k=0}^{nVar_i} T_{R\{data\}}(S_k * Q_i) + T_{C\{derOp\}} + T_{filter} \quad (3)$$

a) Read the metadata, stats and data: All read times follow the same logic as the write equation, by taking into consideration the amount of data to be read, the read bandwidth of the system and properties of the reading algorithm used by the I/O library. Same as for the write cost, the read cost includes the additional cost $T_{additional}$ given by the I/O algorithm used for dealing with the data.

$$T_{R\{b\}} = \frac{S_b}{B_{read}} + T_{additional}$$

Metadata needs to be read to be able to access stats and data. Typically, one rank reads the entire metadata then scatter to the other ranks. The stats and data are read only for the chunks of data belonging to the local rank. Once stats have been read, they can be used to identify the blocks of data that fit the query requirements. If the derived variable is storing data in addition to the stats and expression, from the read perspective it becomes a primary variable. In this case, reading the data includes fetching from storage only the blocks of data identified by the stats for the given variable.

For derived variables that only contain the expression, the primary data needed for computing the derived variable are identified based on the expression. Data for all variable block need to be read ($Q_i = 1$). Is stats are also available, the block of data needed to compute the derived variables are identified and data is being fetched only for those blocks, for all the variables needed to compute the derived variable ($Q_i < 1$).

b) Compute the derived variable: If data is not available, the derived variables need to be computed once all the needed primary variables are read. Similar to the write side, the cost for computing the derived variable depends on the number of variables required by the expression ($nVar$) the number of blocks returned by a query and the derived expression as well as on characteristics of the computational unit used to compute the derived values (FLOPS and $B_mem\{arch\}$). Similar to reading the data, if only the expression is stored for a derived variable, the values for the entire dataset needs to be computed ($Q_i = 1$).

c) Filter the data: If stats are stored for a given variable, there is no need for filtering since only the blocks that fit the query would be fetched from storage. If only the expression is provided for a derived variable, after computing the values, the query engine will identify the blocks that fit the query and will filter everything else. Typically, this step can be merged with the computation of the derived values, in which case the filtering cost is completely negligible. For the rest of the paper we consider $T_{filter} = 0$.

d) Performance model: We will use Eq. 3 to model all variables, both derived and primaryThe following table summarizes the different way the equation is used by each case.

For primary variables, the stats and data are saved to storage, so the cost required by the query consists of: 1) reading the metadata and stats to decide which blocks to read and 2) reading only the block that fits the query. In Eq. 2:

- $nVar_i = 1$ and $T_{C\{derOp\}} = 0$

Derived variables that have data stored are identical to normal variables. For the other derived variables, all terms of the equation will be used and variables required to compute the derived values will need to be read.

- Stats strategy: use stats to chose blocks ($Q_i < 1$)
- Expression strategy: need to read all data ($Q_i = 1$)

D. Main factors influencing the performance

We measured the performance of each step of our implementation on the Frontier and Perlmutter systems.Frontier [1] is an HPC system at OLCF consisting of over 9,408 compute nodes, each with an AMD EPYC "Trento" CPUs (56 usable cores), eight compute dies of AMD MI250X, and 512 GB of DDR4 memory. Perlmutter [2] is a system at NERSC consisting of 3,072 CPU-only and 1,792 GPU-accelerated nodes. It uses AMD EPYC "Milan" CPUs and Nvidia A100 GPUs and 256 GB of DDR4 memory. We ran small tests to capture the network bandwidths, and the compute performance numbers for both Frontier and Perlmutter and used the numbers to instantiate the model.

We implemented the three strategies for writing and querying derived variables using ADIOS2 as our underlying I/O implementation. We use ADIOS2 with the default file engine [10] and with an aggregation strategy where all processes are concurrently writing to the PFS in a similar manner as our model (`EveryoneWritesSerial`). Our implementation is on top of ADIOS2 but we kept our algorithms and analysis generic and could be used with HDF5 or NetCDF or any other I/O library. Our experiments show that the time to compute the stats and metadata is negligible compared to computing the derived values, the time to gather and write/read the stats/metadata is negligible compared to writing/reading data and the time to filter out values that do not fit a query is negligible compared to reading. If we remove these terms from the equations in the previous section, we get the following simplified equations for the total time (that includes writing and reading Q percentage of a derived variable) for one derived variable D and k primary variables P:

$$T_{\{Store\}} = T_{C\{D\}} + k * T_{W\{P\}} + T_{W\{D\}} + Q_i * T_{R\{D\}}$$

$$T_{\{Exp\}} = k * T_{W\{P\}} + k * T_{R\{P\}} + T_{C\{D\}}$$

$$T_{\{Stats\}} = (1 + Q_i) * T_{C\{D\}} + k * T_{W\{P\}} + Q_i * k * T_{R\{P\}}$$

The time for the `Store` strategy includes computing the derived variables and writing the primary and derived data on the writer side, and reading the percentage of derived data on the reader side. The `Expression` strategy includes the time to write the primary data, the time to read all primary data and compute the derived variables. The `Stats` strategy computes the derived data twice (on the write and on the percentage of read data) and includes the time to write all the primary data and read the primary data that fits the query. Removing the terms that exist in all three scenarios leaves us with the following main costs:

$$T_{\{Store\}} = T_{W\{D\}} + Q_i * T_{R\{D\}}$$

$$T_{\{Exp\}} = k * T_{R\{P\}}$$

$$T_{\{Stats\}} = Q_i * T_{C\{D\}} + Q_i * k * T_{R\{P\}}$$

The `Store` strategy is mainly influenced by the write bandwidth (and the storage cost). The `Expression` performance will heavily depend on the read bandwidth and on how many primary variables are needed to compute the derived one. The `Stats` strategy is mainly influenced by how fast derived variables can be computed and how much data is returned by the query. To better understand these trade-off, we make experiments in the next section to quantify just how much the following factors influence the performance of each strategy:

- The computing performance and the complexity of the derived expression
- The write to read bandwidth ratio
- The query characteristics

IV. Performance analysis

We analyze in this section the trade-offs given by different system and application characteristics. Based on the model in the previous section, we have identify five main factors that influences the performance of writers and readers namely: computing capacity, the derived kernel complexity, query size, the number of readers accessing the data and the network bandwidth (write to read bandwidth performance). We analyze each in the following subsections and highlight limitations and advantages of our three strategies.

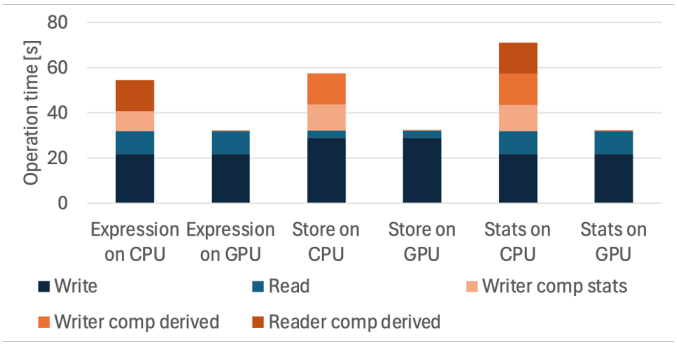


Fig. 4: Time spent in each Writer and Reader building block for one node of Perlmutter when: writing 3 primary variables and one derived variable, each of 11.7GB; and reading the entire dataset

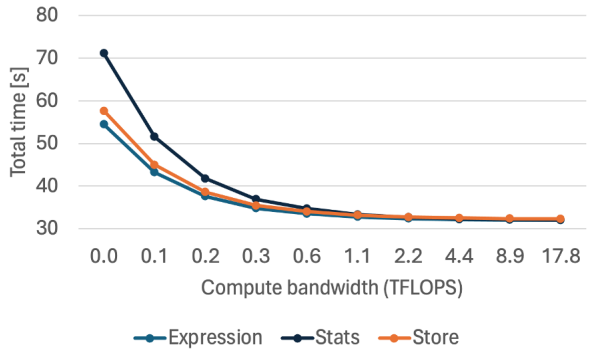


Fig. 5: Total time for writing 3 primary variables and one derived variable, each of 11.7GB, and reading the entire dataset when simulating different compute capabilities on Perlmutter

A. *Compute characteristics*

a) Computing performance of the processing unit: The computing performance influences the length of the dark orange bars in Figure 3 and thus the total execution time for the writer and reader. As seen in the previous section, how fast derived variables can be computed can make the *Stats* strategy very expensive (since it needs to compute the derived variable twice). In this sub-section, we assume the analysis is reading the entire dataset.

Figure 4 shows the breakdown of time spent in each stage for both the writer and the reader for Perlmutter when either the CPU or the GPU are used as the compute platform. In this example 3 primary variables are written and magnitude is the derived variable computed by the I/O strategy. The compute time for both the derived variable and stats is negligible compared to the I/O time when the GPU is used. In this case, the performance difference between the strategies is given by the time to write more statistics for `Stats` and the time to write stats and the derived data for `Store`. The `Store` strategy has a 1.33x increase in storage since the derived variable has the same size as the primary data. The `Stats` strategy has similar size with `Expression` with the stats and metadata difference typically accounting for only KB (MB at scale) compared to GB for data.

TABLE II: Performance of derives kernels for 1.6 GB data size

Kernel	Frontier CPU	Perlmutter CPU	Frontier GPU	Perlmutter GPU
Add	0.27 s	2.37 s	4.33 ms	3.74 ms
Magnitude	0.79 s	4.49 s	4.61 ms	3.75 ms
Curl	7.2 s	42.17 s	53.11 ms	43.53 ms

Figure 5 shows the predicted execution time for Perlmutter for our previous example when controlling the FLOPS of the compute platform. The left side of the figure (<0.1 TFLOP) represent the Perlmutter CPU case and the right side (>9 TFLOPS) the GPU case presented in Figure 4. We are interested in seeing what is the point where the total execution time of the three strategies become equivalent. Starting with as early as 0.3 TFLOPS the execution time between the strategies is within 10% of each other, with `Store` requiring more storage. While our example reads the entire dataset, analysis can use the statistics to query and read only the parts of the data that are needed for a study, decreasing the reader time for `Stats` and `Store`.

> The `Stats` strategy has similar performance with the other strategies when fast computing platforms are used (GPUs or CPUs with hundreds of GFLOPS performance) without paying the storage cost of `Store` and allowing queries to optimize the read process.

b) Derived kernel complexity.: Different derived functions will have different complexities and will require accessing more or less primary variables, changing the compute time and the trade-off between the three strategies. We test in this paper two derived variables: magnitude used by S3D and curl used by the E3SM applications and we implement `Add` as a baseline since it has the simplest implementation. The complexity of all the derived variables used in this paper are O(N), however, the number of total floating point operations differs significantly between the functions. Table II shows the execution time for different derived kernels on Frontier and Perlmutter. Longer compute times for derived variables influence the moment when the writer using a `Stats` strategy becomes as fast as the others. Updating the Figure 5 for curl for example shows that `Stats` is within 10% of `Expression` only for compute platforms of over 1 TFLOPS (due to lack of space we omit the figure).

> Heavyweight kernels wanting to use the `Stats` strategy will have to pay the price of a less performant write operation. If the analysis code typically reads the entire data set, the `Expression` strategy will be a better choice.

More complex derived variables that require data from multiple steps (like derivatives) or that require stencil type of computations will increase the cost of computing the derived variable and will offset the advantage of using fast processing units like the GPU. Offloading the derived variable

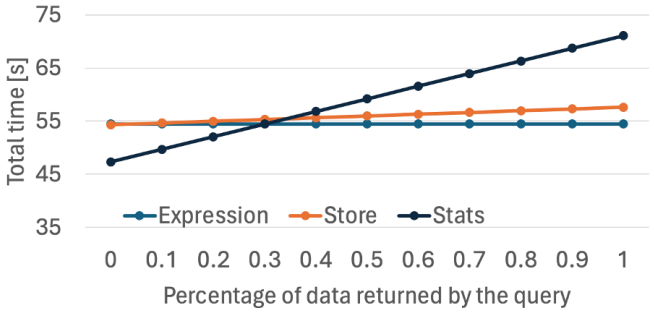


Fig. 6: Total execution time on Perlmutter for the three strategies when only a percentage of the data is analyzed by the reader.

Fig. 7: Read time on Perlmutter for the three strategies when only a percentage of the data is analyzed by readers.

computation to I/O libraries allows the I/O layer to choose the best strategy for each derived expression.

B. Reader characteristics

a) Query size.: The amount of data that needs to be accessed by a reader will influence the overall execution time of the entire ensemble. Figure 6 explores the performance implications of querying for parts of interest in the dataset. The figure shows the total execution time when the reader analyzes only a percentage of the entire dataset on Perlmutter. We assume the study requires the derived data, so for the `Stats` strategy, the reader needs to read a percentage of the primary data and compute the derived data for the given subset. In this example, queries that parse less than 30% of data on the CPU will benefit from using the `Stats` strategy. Otherwise, the other two strategies are equivalent. The `Expression` strategy (light blue line) is horizontal since it needs to read the entire dataset regardless of the query (there are no stats for derived variables). The `Stats` (dark blue line) and the `Store` (orange line) strategies move down with better compute capabilities or with lightweight kernels (on the GPU both dark blue and orange lines are completely below the `Expression` strategy performance).

Typically studies do now know the amount of data the queries will bring. Having statistics stored for derived variables allows the flexibility of querying without bringing the entire datasets.

b) Number of readers accessing data.: When the number of readers accessing the same data increases, the read time becomes much more important. We isolate from Figure 6 only the read time of each strategy and plot it in Figure 7. The `Store` strategy is the lower limit since all it does on the read side is to access the percentage of required derived data. The `Expression` strategy presents the upper limit since it has no information about the derived variables and needs to read all the primary data and compute the derived variables in order to choose the desired subset. The two limits become closer together as the compute capability of the platform increases, but the shape remains the same. The time difference between each strategy is decreased from the writer time with every new reader. In our example, for queries that access 50% of the data in average, the `Store` strategy will save 2 seconds at each read compared to `Stats` and 8 compared to `Expression`. The time difference between the write side using the CPU for each strategy is less than 10s. Thus, with only two readers, the `Store` strategy gets much better performance.

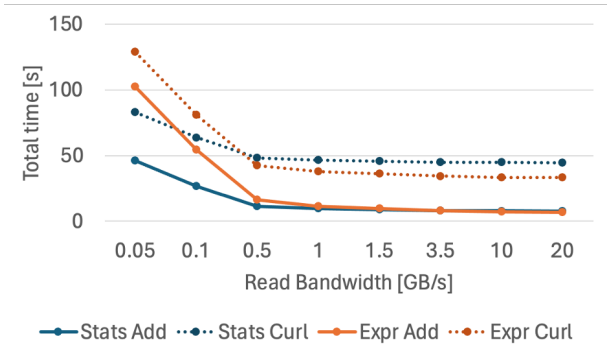


Fig. 8: Total time to write and read two types of derived variables (Add and Curl) when simulating different reading bandwidth

The `Store` strategy might be worth the cost in storage when multiple readers are analyzing the same dataset (e.g. hyperparameter search on derived data).

c) Network bandwidth.: A code writing data, typically stores the data on local storage. However, the reader code can be executed remote for wide area network access (e.g. if the analysis is executed on the scientist's laptop). Figure 8 shows the total time to write and read for different reading bandwidths when the 40% of the derived data is required for analysis. The figure looks at two kernels, lightweight `Add` and heavyweight `Curl`. The performance of the `Stats` strategy is better for limited bandwidth since it reduces the data transfer (in our e.g., less than 0.5 GB/s read bandwidth is the turning point). A decrease in kernel complexity or an increase in compute capability influences this turning point by moving it towards a higher bandwidth (e.g. on the GPU on Frontier the `Stats` strategy is a better choice for read bandwidths lower than 3 GB/s even if the entire derived dataset is read).

The percentage of data transferred by a query as well as the number of readers requiring the data decreases the ratio

between the write time and the read time and influences the results in Figure 8 in the same way as an increase in compute capacity. For example, a query that requires only 10% of the derived dataset will show that the `Stats` strategy is a better choice for read bandwidths lower than 2.3 GB/s.

The `Expression` strategy is not efficient for remote access due to its high cost and can only be used for analysis on local clusters with the data being accessed.

Overall, this section shows that the performance difference between the strategies can be significant depending on the system and application characteristics. Current solutions (write or read side dealing with derived variables) can be used out of the box in some situations but could have an exorbitant price in others. In the next section we investigate the performance of two HPC applications in different scenarios.

V. Application performance

The performance model presented in the previous sections highlights the importance of understanding what is the ratio between the time to access data and the time to compute derived variables for a platform and a given derived expression. If the derived function does not produce much data or if the derived data is accessed extensively, the `Store` strategy will show the best performance (and derived variables should be treated in fact as primary data). However, if this is not the case, careful attention should be placed on what is the limiting factor for the analysis. If compute is cheap (e.g. GPUs can be used to compute the derived variables) or the data is analyzed remotely, dealing with derived variables on the read side (e.g. Paraview) will have a low performance. Our hybrid solution, the `Stats` strategy, offers a flexible alternative that shows similar (or better) performance with `Store` without having a high overhead on storage. The `Expression` strategy shows better results for heavyweight kernels for which compute is expensive and analysis parse entire datasets of local data (e.g. in-situ visualization should use this strategy when plotting derived variables). In this section, we look at two use cases and measure their performance on Frontier for in-situ and remote visualization to test the validity of the findings in this paper.

A. S3D

The S3D simulation [16] is a direct numerical simulation of turbulent combustion that solves compressible reacting Navier-Stokes total energy and species continuity equations using high-order finite-difference methods. S3D generates 1.5 TB of data in each step through 24 primary variables. Particles are stored in 3D arrays of $1280x1280x1280$ size and their velocity is stored using 3 separate variables, each requiring 64 GB of storage space. We ran on 900 ranks on Frontier, in sequence, a writer and a reader generating and querying S3D data, each rank accessing around 75 MB of data for each variable. We ran the experiments for 4 steps and we report the average values. Scientists analyzing S3D data are interested in querying the magnitude of the velocity to identify areas of intense burning. Figure 9 shows a typical analysis workflow, with scientists plotting 2D slices of the temperature data, identifying regions

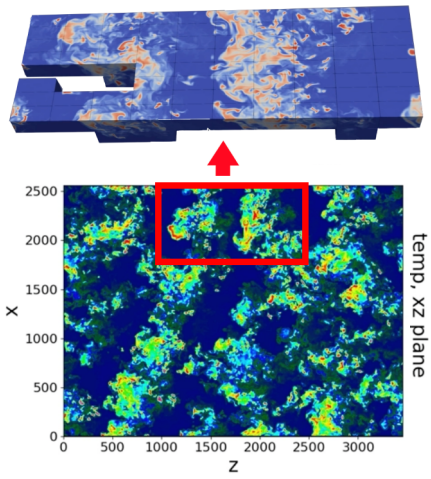


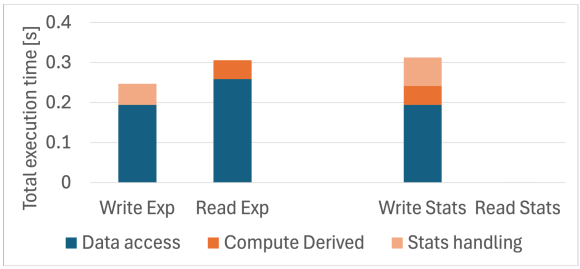
Fig. 9: Visualization of the S3D data: from 2D slices of the temperature, identify areas of interest (the red square) and creating queries for 3D blocks based on derived variables

of interest and querying for 3D blocks of temperature data in the region of interest where the magnitude is below a given threshold.

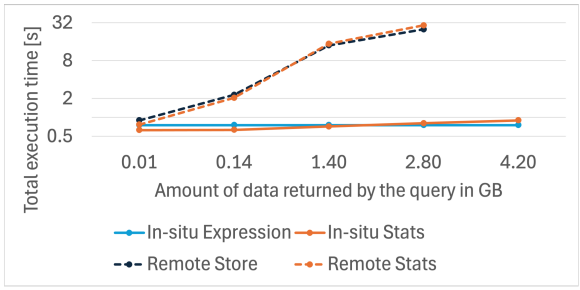
By simply adding one line in the S3D simulation code to define the `magnitude` derived variable in the underlying I/O library, we were able to query using magnitude as if it is a primary variable, regardless of the I/O and compute strategy used underneath. The `magnitude` derived variable has a size equal to the number of particles for which velocity contains values. Thus, the `Store` strategy adds 64 GB of data for each simulated step. The statistics and metadata, when 900 ranks are writing data, is using 12 MB and the difference between the strategies represents less than 10% of the metadata size, negligible compared to the data size. We made two types of experiments: 1) in-situ analysis running the query code on Frontier; and 2) remote analysis running the query and visualization codes on a local laptop and accessing the remote data on Frontier.

Due to the huge storage cost of the `Store` strategy, we only use the `Stats` and `Expression` strategies for in-situ analysis. For remote analysis, the `Expression` strategy would require storing around 256 GB of primary data on the remote site to compute the derived variable which is prohibitively expensive. For this reason, we only ran the `Stats` and `Store` strategies for this scenario.

Figure 10 shows the I/O performance results for our runs on Frontier for the workflow presented in Figure 9. For in-situ, both the writer and reader are running on 900 nodes and the reader is parsing the entire dataset. For remote analysis, the scientists are visualizing the data on their laptop so, since all data will not fit into memory, the reader queries for a subset of the data. The scientists are interested in visualizing the `temperature` where magnitude is below a threshold. This means the magnitude value is not needed on the reader side, only the stats.The `Stats` strategy stores statistics



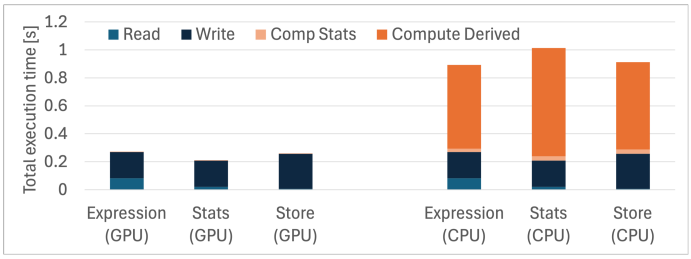
(a) Execution time on Frontier for writing S3D data and reading in-situ a small portion of the data (around 1.5%)



(b) In-situ and remote analysis of S3D data when the study includes querying multiple areas of interest

Fig. 10: Execution time of writing one step of S3D data and reading one or multiple areas of interest

(a) Total time as the analysis is querying more data



(b) Breakdown of the cost for 1GB of data

Fig. 11: Total execution time on Frontier for writing and reading magnitude of the curl over the wind velocity E3SM data.

about `magnitude`, so the reader transfers only the desired blocks for temperature (which makes the read performance negligible compared to the rest as seen in Figure 10a) while the `Expression` strategy will need to bring the velocity data (which is 3x the size of temperature data) to compute magnitude in order to know which temperature blocks to bring.

For remote access, the reader is using one process (the laptop) and based on our experience, typically needs to read between 1-2 GB of temperature data for each study that looks at one area of interest. The read time dominates the total time which makes the two strategies equivalent. The `Store` strategy, however, still needs to store 64 GB of derived data on Frontier. Figure 10b shows how the cost of Stats increases as more areas of interest are investigated in the analysis.

The performance is typical of S3D since the scientists are plotting primary data (temperature) and only query the derived data (magnitude). In this case, the `Stats` strategy is similar in size to `Expression` for in-situ and in performance to `Store` for remote access. It performs better than the other strategies for both total time (up to 15% better than `Expression` for in-situ) and total storage (25% better than `Store` strategy). However, the balance will change when we need to compute the derived variable on the reader side (next section will investigate this).

B. E3SM

The earth system models (ESMs) [4] provide state of the art simulations of the global climate. They include general circulation and thermodynamic models for ocean and atmosphere, and models for land, sea ice, and land ice processes. In the simulations made on Frontier, E3SM outputs model data at the 6-hourly interval generating around 24 GB data through 9 primary variables on 96 ranks. There are several analysis codes that can analyze the E3SM data. We focus on the tropical cyclone track code as our analysis test cases, where scientists detected from data outputted at an hourly interval, cyclone location and movement. One typical derived variable needed by the study is the curl of wind velocity data or the magnitude of the curl, to detect the atmospheric rotation. The study is done on the same cluster, either in situ or sequential within the same workflow. In our analysis, we use 10 ranks for the analysis where the data is queried for areas of high atmospheric rotation returning in average 12% of the total generated blocks of data.

Figure 11 presents the total execution time of the 3 strategies for the described test cases. The size of the curl variable is around 4 GB and 3GB for magnitude increasing the total dataset size to 28 GB or 27 GB for the `Store` strategy. The metadata and statistics for 96 ranks is around 1 MB. Since the query is usually returning a small subset of the data, the read time for the `Stats` and `Store` strategy is negligible. The write time for the `Store` strategy is around 30% higher than the other strategies. As more data is read by the queries and as derived variables become more complex (e.g. magnitude of curl compared to simply curl), the `Stats` strategy will have an increase cost. If the GPU is used, the `Stats` strategy performs 25% faster than the other strategies.

Our experiments were ran up to 900 processes (17 nodes) on Frontier, which is still a relatively small size. Scalability is important not only when computing the application performance but also for the I/O libraries performance since typically

the aggregation scheme for data/stats/metadata is to apply a gather on all nodes. The MPI gather performance can become important and can change the balance between the strategies. We plan to investigate more complex performance models in the future to profile the scalability of each strategy.

VI. Conclusions

In this paper, we investigated the benefits of offloading the computation of derived variables to an I/O layer and the performance trade-offs of different strategies of when and where is best to compute this data. Each strategy needs to be well understood to get the best performance when querying. We argue in this paper that offloading this task to the I/O library would allow for the best strategy to be chosen in each scenario since all the information necessary to choose between the trade-off is present at this level. Offloading the derived computation also allows for hybrid strategies that can balance between the other two strategies (especially if the derived computation can be triggered in transit or on storage compute units, completely hiding their computation). There are two typical analysis modes in HPC, either sequential by analyzing datasets stored to storage or concurrent by having the reader run in parallel to the writer. This paper focuses on the first. We plan to extend this paper using data staging models [9], [12] to study the impact of derived variables on the performane of staging strategies.

References

[1] Frontier user guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html. Accessed: 2024-04-28.

[2] Perlmutter architecture. https://docs.nersc.gov/systems/perlmutter/architecture. Accessed: 2024-04-28.

[3] Utkarsh Ayachit, Andrew C. Bauer, Ben Boeckel, Berk Geveci, Kenneth Moreland, Patrick O'Leary, and Tom Osika. Catalyst revised: Rethinking the paraview in situ analysis and visualization api. In Heike Jagode, Hartwig Anzt, Hatem Ltaief, and Piotr Luszczek, editors, *High Performance Computing*, pages 484–494, Cham, 2021. Springer International Publishing.

[4] S. M. Burrows, M. Maltrud, X. Yang, Q. Zhu, N. Jeffery, X. Shi, D. Ricciuto, S. Wang, G. Bisht, J. Tang, J. Wolfe, B. E. Harrop, B. Singh, and Brent. The doe e3sm v1.1 biogeochemistry configuration: Description and simulated ecosystem-climate responses to historical changes in forcing. *Journal of Advances in Modeling Earth Systems*, 12(9), 2020.

[5] Rajkumar Buyya, Toni Cortes, and Hai Jin. *Overview of the MPIIO Parallel I/O Interface*, pages 476–487. 2002.

[6] J. Cernuda, L. Logan, A. Gainaru, J. Lofstead, A. Kougkas, and X.-H. Sun. Hades: A context-aware active storage framework for accelerating large-scale data analysis. In *The 24th IEEE/ACM international Symposium on Cluster, Cloud and Internet Computing*, 2024.

[7] Afonso Coutinho, Diogo Marques, Leonel Sousa, and Aleksandar Ilic. Sparse-aware carm: Rooflining locality of sparse computations. In Demetris Zeinalipour, Dora Blanco Heras, George Pallis, Herodotos Herodotou, Demetris Trihinas, Daniel Balouek, Patrick Diehl, Terry Cojean, Karl Fürlinger, Maja Hanne Kirkeby, Matteo Nardelli, and Pierangelo Di Sanzo, editors, *Euro-Par 2023: Parallel Processing Workshops*, pages 97–109, Cham, 2024. Springer Nature Switzerland.

[8] Paulo De Marco, Júnior and Caroline Corrêa Nóbrega. Evaluating collinearity effects on species distribution models: An approach based on virtual species simulation. *PLoS One*, 13(9), September 2018.

[9] Shaohua Duan, Pradeep Subedi, Keita Teranishi, Philip Davis, Hemanth Kolla, Marc Gamell, and Manish Parashar. Scalable data resilience for in-memory data staging. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 105–115, 2018.

[10] Greg Eisenhauer, Norbert Podhorszki, Ana Gainaru, Scott Klasky, Junmin Gu, Vicente Bolea, Liz Dulac, Dmitry Ganyushin, and William Godoy. Hpc i/o innovations in the exascale era. In *Internatinal Journal of High Performance Applications*, pages 1–13, 2024.

[11] Iksoo Eo, Woojong Han, and Yoomi Park. Roofline model and profiling of hpc benchmarks. In *2022 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, 2022.

[12] Ana Gainaru, Wan Lipeng, Ruonan Wang, Eric Suchyta, Jieyang Chen, James Kress, Dave Pugmire, Norbert Podhorszki, and Scott Klasky. Understanding the impact of data staging for coupled scientific workflows. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):878–890, 2022.

[13] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, and Kshitij Mehta et al. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[14] Zakia Hammouch, Mehmet Yavuz, and Necati Özdemir. Numerical solutions and synchronization of a variable-order fractional chaotic system. *Mathematical Modelling and Numerical Simulation with Applications*, 1(1):11–23, 2021.

[15] Jennifer N. Hird, Guillermo Castilla, Greg J. McDermid, and Inacio T. Bueno. A simple transformation for visualizing non-seasonal landscape change from dense time series of satellite data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 9(8):3372–3383, 2016.

[16] Hong G Im, Arnaud Trouve, Christopher J Rutland, and Jacqueline H Chen. Terascale high-fidelity simulations of turbulent combustion with detailed chemistry. 8 2012.

[17] D. Kao, J.L. Dungan, and A. Pang. Visualizing 2d probability distributions from eos satellite image-derived data sets: a case study. In *Proceedings Visualization, 2001. VIS '01.*, pages 457–589, 2001.

[18] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. Hermes: a heterogeneous-aware multi-tiered distributed i/o buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, page 219–230, 2018.

[19] Margaret Lawson, Craig Ulmer, Shyamali Mukherjee, Gary Templet, Jay Lofstead, Scott Levy, Patrick Widener, and Todd Kordenbrock. Empress: extensible metadata provider for extreme-scale scientific simulations. In *The International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, page 19–24. ACM, 2017.

[20] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, page 39. ACM, 2003.

[21] Drishti Maharjan and Peter Zaspel. Toward data-driven filters in paraview. *Journal of Flow Visualization and Image Processing*, 29(3):55–72, 2022.

[22] G. Manduchi, T. Fredian, and J. Stillerman. A new object-oriented interface to mdsplus. *Fusion Engineering and Design*, 85(3):564–567, 2010. Proceedings of the 7th IAEA Technical Meeting on Control, Data Acquisition, and Remote Participation for Fusion Research.

[23] Diogo Marques, Aleksandar Ilic, and Leonel Sousa. Mansard roofline model: Reinforcing the accuracy of the roofs. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 6(2), October 2021.

[24] J. Jake Nichol, Matthew G. Peterson, Kara J. Peterson, G. Matthew Fricke, and Melanie E. Moses. Machine learning feature analysis illuminates disparity between e3sm climate models and observed climate change. *Journal of Computational and Applied Mathematics*, 395:113451, 2021.

[25] Kenneth A. Philbrick, Alexander D. Weston, Zeynettin Akkus, Timothy L. Kline, Panagiotis Korfiatis, Tomas Sakinis, Petro Kostandy, Arunnit Boonrod, Atefeh Zeinoddini, Naoki Takahashi, and Bradley J. Erickson. Ril-contour: a medical imaging dataset annotation tool for and with deep learning. *Journal of Digital Imaging*, 32(4):571–581, Aug 2019.

[26] Eric Suchyta, Jong Youl Choi, Seung-Hoe Ku, David Pugmire, Ana Gainaru, Kevin Huck, Ralph Kube, Aaron Scheinberg, Frédéric Suter, Choongseock Chang, Todd Munson, Norbert Podhorszki, and Scott Klasky. Hybrid analysis of fusion data for online understanding of complex science on extreme scale computers. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 218–229, 2022.