# Adaptive Performance-Constrained In Situ Visualization of Atmospheric Simulations

Matthieu Dorier,* Robert Sisneros,† Leonardo Bautista Gomez,* Tom Peterka,* Leigh Orf,‡
Lokman Rahmani,§ Gabriel Antoniu,¶ and Luc Bougé§
*Argonne National Laboratory, Lemont, IL, USA, {mdorier,leobago,tpeterka}@anl.gov
†NCSA, UIUC, Urbana-Champaign, IL, USA, sisneros@illinois.edu
‡University of Wisconsin - Madison, Madison, WI, USA, leigh.orf@ssec.wisc.edu
§ENS Rennes, IRISA, Rennes, France {lokman.rahmani,luc.bouge}@irisa.fr
¶Inria, Rennes Bertagne Atlantique Research Centre, Rennes, France, gabriel.antoniu@inria.fr

***Abstract*—While many parallel visualization tools now provide in situ visualization capabilities, the trend has been to feed such tools with large amounts of unprocessed output data and let them render everything at the highest possible resolution. This leads to an increased run time of simulations that still have to complete within a fixed-length job allocation. In this paper, we tackle the challenge of enabling in situ visualization under performance constraints. Our approach shuffles data across processes according to its content and filters out part of it in order to feed a visualization pipeline with only a reorganized subset of the data produced by the simulation. Our framework leverages fast, generic evaluation procedures to score blocks of data, using information theory, statistics, and linear algebra. It monitors its own performance and adapts dynamically to achieve appropriate visual fidelity within predefined performance constraints. Experiments on the Blue Waters supercomputer with the CM1 simulation show that our approach enables a $5\times$ speedup with respect to the initial visualization pipeline and is able to meet performance constraints.**

## I. Introduction

Today's petascale supercomputers enable the simulation of physical phenomena with unprecedented accuracy. Large numerical simulations typically run for days on hundreds of thousands of cores, generating petabytes of data that has to be stored for offline processing. But storage systems are not scaling at the same rate as is computation. Consequently, they become a bottleneck in the workflow that goes from running a simulation to actually retrieving scientific results from it. Trying to avoid this bottleneck led to *in situ visualization*: running the visualization along with the simulation by sharing its computational and memory resources and bypassing the storage system completely [1]. Several frameworks have been proposed to enable in situ visualization. VisIt's libsim interface [2] and ParaView Catalyst (previously called "co-processing library") [3] are two examples. Middleware such as Damaris [4] and ADIOS [5] has been developed to reduce the necessary code changes in simulations and provide additional data-processing features.

While in situ visualization solves the storage bottleneck problem, the additional processing time imposed by in situ visualization can be prohibitively high and increase the run time and the performance variability of the simulation. Approaches such as Damaris [4] that hide the cost of in situ visualization in dedicated cores are required to skip some iterations of data in order to keep up with the rate at which the simulation produces them.

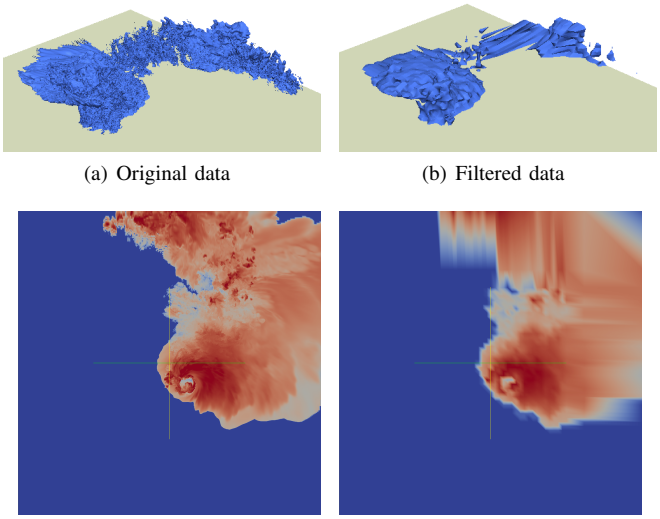
Yet, not all generated data is relevant to understanding and following the simulated physical phenomena. For example, atmospheric scientists running storm simulations are interested mainly in areas of high data variability, potentially indicating the presence of a forming tornado. The physical phenomenon of interest (e.g., the tornado) can be localized in a relatively large domain. The rest of the data in the domain corresponds to regions of the atmosphere where state variables (wind speed, temperature, etc.) present little variation. This spatial locality of the region of interest also produces load imbalance across processes when attempting to visualize it.

Based on these observations, we propose a new in situ visualization pipeline that aims to both *improve* and *control* the performance of in situ visualization. This pipeline starts by detecting regions of high data variability using a set of either generic or user-provided metrics. It then filters out blocks (regular subdivisions of a subdomain) of data that do not carry much information. Additionally, our pipeline redistributes blocks of data across processes in order to achieve better load balance. Our pipeline monitors its performance and dynamically adapts the amount of data in order to meet the simulation's run-time constraints.

Our proposed method requires domain scientists to provide appropriate metrics measuring the scientific relevance of data regions and appropriate in situ visualization scenarios. We show, however, that a set of generic metrics based on statistics, information theory, and linear algebra can highlight potentially interesting regions.

In this work, we demonstrate the benefit of our approach through experiments on the Blue Waters petascale system [6] at the National Center for Supercomputing Applications (NCSA) using the CM1 atmospheric simulation [7], with ParaView Catalyst [8] as our visualization backend. Compared with a normal pipeline that does not filter or redistribute data, we show that our pipeline enables a $4\times$ speedup of the visualization task on 64 cores and a $5\times$ speedup on 400 cores, even without reducing the amount of data. Moreover, our pipline is able to meet targeted performance constraints by reducing the amount of data supplied to the visualization task. Additionally, we evaluate each component of our pipeline individually.

The rest of this paper is organized as follows. We present the motivation for our work in Section II, along with the simulation code and visualization scenarios we consider in this study. Section III presents related work. Section IV describes



(a) Original data (b) Filtered data

(c) Original data (d) Filtered data

Fig. 1: Volume rendering (a,b) and colormap (c,d) of the reflectivity (dBZ) in the CM1 simulation when feeding the visualization pipeline with original data (a,c) and with filtered data (b,d).

our performance-constrained in situ visualization framework. We then present its evaluation in Section V. Section VI presents our conclusions and give an overview of future work.

## II. Motivation

In this section, we first present the use case driving our study. We then motivate the use of data redistribution and reduction as a means to achieve performance-constrained in situ visualization.

### A. Use case: the CM1 atmospheric model

Atmospheric simulations are good candidates for in situ visualization. They are generally compute-bound rather than memory-bound and can therefore share their resources with visualization tools [4].

They also simulate their phenomena (e.g., tornadoes) on a physically large, static domain so that the region of interest has enough space to evolve without interacting with domain boundaries. The domain decomposition across processes in such simulations is regular and independent of each subdomain's content. As a result, many subdomains may contain uninteresting data.

Our study focuses on the CM1 atmospheric simulation [7]. CM1 is used for atmospheric research and models small-scale atmospheric phenomena such as thunderstorms and tornadoes. The simulated domain is a fixed 3D rectilinear grid representing part of the atmosphere. Each point in this domain is characterized by a set of field variables such as local temperature and wind speed. CM1 proceeds by iterations, alternating between a computation phase during which equations are solved and I/O phases during which data is output to storage and/or fed to an in situ visualization system.

Figure 1(a) shows the result of in situ volume rendering of the reflectivity (dBZ) field in CM1. This field corresponds to the simulated radar reflectivity. It derives from a calculation based on cloud rain, hail, and snow microphysical variables, and it can be compared with real weather radar observations. A 45 dBZ isosurface reveals a feature called the weak echo region, which is linked to the physical onset of the storm. An isosurface is a surface built from the points in space where a 3D field takes a given value (here 45 dBZ). Such an isosurface is usually computed by using the marching cubes algorithm [9].

### B. Improving performance through data redistribution

Figure 1 also shows that the region of interest is very localized. Thus, some processes have more to render than others. The overall rendering time is driven by the rendering time of the process with the highest load. Since each subdomain handled by a process can be further decomposed into multiple blocks, redistributing blocks to balance the load may improve performance.

### C. Improving performance through data reduction

In Figures 1(b) and 1(d), each original $55 \times 55 \times 38$-point block of data has been reduced to a $2 \times 2 \times 2$-point block, keeping only corner values, before being fed to the visualization pipeline. While 50 seconds were required to produce Figure 1(a) on 400 cores of the Blue Waters supercomputer, only 1 second was required to produce Figure 1(b). Even though the loss of visual quality is evident in Figure 1(b), we confirmed with atmospheric scientists that such results can still be useful for tracking the evolution of the phenomena being studied.

### D. Adapting to performance constraints

In our previous work [4], we showed that in situ visualization can largely increase the run time of a simulation when done in a time-partitioning manner (i.e., the simulation stops periodically to produce images). We also showed that in some situations the high cost of running in situ visualization algorithms in dedicated cores while the simulation keeps running forces the dedicated cores to skip some iterations and to reduce the frequency at which images are produced.

In this work, we address this problem by proposing performance-constrained in situ visualization. The main idea is that different blocks of data have different scientific value and that blocks that are not interesting can be filtered out in order to gain performance. Consequently, the in situ visualization pipeline will be continuously adapted to achieve good visual fidelity for the end user while staying close to a given visualization time, in a best-effort manner.

## III. Related work

In the following we present related work in the field of in situ visualization and, in particular, techniques that attempt to adapt the in situ visualization pipelines.

### A. Adaptive in situ visualization

More and more efforts are put into designing in situ visualization frameworks that adapt to the content of the data (for instance, its compressibility) or to the availability of resources such as local memory.

Zou et al. [10] presented an in situ visualization framework based on EVpath that takes into account the quality of information (QoI) as well as the quality of service (QoS).

Their QADMS approach applies lossy compression selectively depending on a tradeoff between QoI (defined as the ratio between compressed data size and original data size) and QoS (defined as the end-to-end latency). While they lay the foundation of data reduction for in situ visualization, our approach is different in that our data reduction method consists of removing entire blocks (keeping the corners) rather than lossy-compressing the full set of points. Since the number of points in their approach does not change, the rendering time remains the same, and only the data transfer between the simulation and a staging area is improved. Our approach improves both data redistribution and rendering time.

Malakar et al. [11] introduced an in situ visualization framework in which data is sent from the simulation to a visualization cluster at a frequency that is dynamically adapted to resource constraints. This approach tries to maximize the temporal accuracy (i.e., by maximizing the frequency of in situ visualization updates) but keeps a fixed spatial resolution. Our approach proposes to adapt the spatial resolution as well and to do it selectively on chunks of data considered relevant.

Jin et al. [12] proposed to adapt the in situ visualization process either by adapting the resolution at which the data is rendered or by changing the location of the rendering tasks (using either in situ visualization or in transit visualization). To adapt the resolution of the data, they used entropy-based down-sampling. We proposed and evaluated several other metrics, in particular those based on the use of floating-point compressors. Additionally, we investigated the impact of data redistribution on such metrics.

Closer to our work is the work by Wang et al. [13], who proposed finding *important* data in time-varying datasets by using information theory metrics and by looking at the evolution of such metrics across different time steps. Although their work provides key insights into defining the *importance* of a piece of data, their solution is not applied in situ, that is, in a context where the performance of filtering relevant data is extremely important in order to avoid any impact on the running simulation.

Nouanesengsy et al. [14] have also proposed a framework to filter out unimportant data prior to in situ visualization. Their approach is based on analysis-driven refinement, which partitions datasets according to a user-defined importance criteria. They propose a set of generic metrics to measure importance, some of which are also based on information theory. Our work can be seen as a particular application of this approach to meet the run-time constraints of in situ visualization tasks.

The capability of an in situ visualization framework to select relevant subsets of data to be stored or visualized was mentioned by Thompson et al. [15] as one of the design issues for in situ visualization.

### B. Integration in existing in situ visualization frameworks

While we have used the Catalyst library [3] to demonstrate our approach, it could be easily leveraged by other in situ visualization packages such as VisIt [16] through its libsim library [2].

In the past few years, a number of data management libraries have been proposed to ease the integration of in situ visualization into existing simulation codes. These libraries, such as ADIOS [5], offer an interesting opportunity for integrating our approach in a way that is more decoupled from the simulation code on one side and from the visualization code on the other.

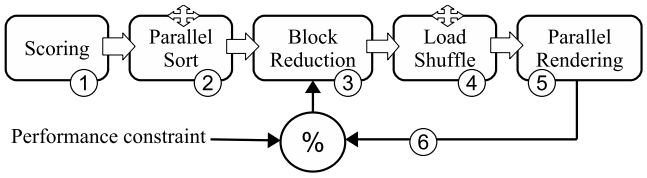



Fig. 2: Overview of our performance-constrained in situ visualization approach. Crosses represent steps that involve collective communications. The run time of the full pipeline is monitored at each iteration and used to control the percentage of blocks that have to be reduced.

Middleware has also been proposed with the same purpose but using dedicated resources to run data management tasks. PreDatA [17] and GLEAN [18] use dedicated nodes (staging area) to run data processing tasks asynchronously. Damaris [4], [19] leverages dedicated cores in multicore nodes to achieve the same goal. The use of dedicated cores for in situ data processing and analytics can also be found in other works [20], [21].

These middleware approaches that leverage dedicated resources pose interesting questions regarding the integration of our approach. In particular, our pipeline can arguably run some parts at the simulation side (block reduction), other parts during data transfer to dedicated resources (sorting blocks), and other parts in dedicated resources (rendering).

## IV. Performance-Constrained In Situ Visualization

This section presents our approach to performance-constrained in situ visualization. We first give an overview of the approach. We then discuss each of its steps: how to give a score to blocks of data, how to reduce blocks with a low score, how we redistribute the load, and how to adapt the pipeline to meet performance constraints.

### A. Overview of our approach

In the following, we call the full 3D array produced by the simulation at a given iteration the *domain*. We call a subarray of a domain handled by one process a *subdomain*. We call a subarray of a subdomain a *block*. The number of blocks per subdomain is constant across processes. The size of all blocks is also constant.

Figure 2 illustrates our approach to performance-constrained in situ visualization. Given input data divided into blocks and distributed across processes, our pipeline consists of six steps.

1) Blocks of data are scored by using a generic or user-provided metric evaluating their relevance to either the scientific phenomenon studied or the visualization algorithm employed.
2) The scores are sorted across processes.
3) A percentage of blocks with the lowest scores is reduced.
4) A load redistribution takes place to redistribute the blocks in order to better balance the phenomenon of interest across processes.
5) The blocks are rendered through a visualization pipeline.
6) The run time of the above steps is measured, and the percentage of blocks to reduce is adapted in order for

the next iteration to be processed in a targeted amount of time.

The following subsections describe these steps in more detail.

### B. Scoring blocks of data

The first step in our approach consists of evaluating the potential relevance of each block of data, so that the least relevant blocks can later be filtered out to improve performance. Our main idea is to score how important it is to the scientific phenomena or to the visualization algorithm. While no universal metric exists for evaluating the relevance of data, we found that a set of generic metrics can still give a good idea of the importance.

In our scenario, atmospheric scientists rely on a combination of techniques to analyze their data. For example, they may render isosurfaces at different levels and use other 3D visualization scenarios, such as streamlines based on wind vectors, or 2D scenarios, such as the colormap shown in Figure 1(c). For these visualization scenarios, to give accurate results, *we are interested in keeping intact areas of high data variability*. Therefore we investigated several metrics to score blocks based on their variability.

*a) Statistics:* The *range* metric computes the difference between the minimum and maximum values in a block of data. The intuition is that a block of data that spans a large range of values might be more interesting to keep than another. However, this metric will give a low score to blocks of data that present high variations but within a small range. A second metric in this category is the *variance* of the data in a block.

*b) Interpolation:* Interpolation-based metrics measure the mean square error between the original data and a block of data rebuilt from an interpolation of a reduced set of values (its corners, for example). For 3D blocks, we use trilinear interpolation. Because many visualization algorithms use trilinear interpolation for rendering, this metric matches the error that a visualization algorithm will make when rendering blocks of data that have been reduced.

*c) Entropy:* The entropy of a block of data measures the amount of information contained in a block. The entropy is obtained by building a histogram of the values found in a block of data and by computing $E = -\sum p_i \log_2(p_i)$, where $p_i$ represents the probability of a single value in the block to fall into bin $i$ of the histogram. In order to be comparable across blocks, the same parameters (range and number of bins) should be used for the histogram across all processes. Doing so requires working with a variable that falls in a known range (this is the case for the reflectivity, which falls in the range $[-60, 80]$) and knowing this range in advance. The number of bins can be more difficult to tune, however. In our experiments we tried 32, 256, and 1,024 bins. While 256 bins seemed more appropriate (better discrimination among blocks for a good performance), no rule of thumb exist for selecting such a number. We also considered the local entropy (entropy computed at each point using a local neighborhood) as a possible metric, but this metric turned out to consume too much time relative to the duration of other components of our pipeline. We used the ITL library [22] to implement entropy-based filters.

*d) Bytewise entropy:* We implemented a lightweight entropy analyzer (LEA) to cope with the limitations of the classical way of computing the entropy. LEA considers each float (or double) as an array of 4 bytes (resp. 8 bytes). It then computes independently the entropy of the first byte of all float values, then the entropy of the second byte, and so on, returning the sum of these entropies as a score. This method does not require tuning a histogram; since each byte can take 256 values, the probability $p_i$ of a value $i$ is simply its frequency of appearance.

*e) Compressors:* We also evaluated compression algorithms as a means of scoring blocks of data. Our intuition is that the compression ratio should correlate with the amount of information contained in a block. Compressors do not require extra information such as histogram parameters. We used the FPZIP [23], ZFP [24], and LZ [25] floating-point compressors, with different tunings for each (such as different levels or lossiness/precision). FPZIP and ZFP also have knowledge of the fact that blocks are 3D arrays; thus we can expect them to take locality into account. Because of space constraints, we present the results of FPZIP only. The results obtained with ZFP and LZ are similar.

We *do not* claim that any of these filters gives an absolute answer to the question of whether a block of data is *interesting*, the notion of *interesting* being subjective and tied to both the field of study and the visualization scenario. We provide this set of filters only as a starting point, and we rely on interactions with domain scientists to find which filter is the most appropriate for the phenomenon studied.

Though we evaluated 30 filters (or variants of filters) in our experiments, we show results only for a representative subset of them: RANGE (range metric), VAR (variance), ITL (entropy), LEA (bytewise entropy), FPZIP (floating-point compression), and TRILIN (trilinear interpolation).

### C. Sorting and reducing blocks

After each block has been given a score, the sets of pairs <id, score> are globally sorted by increasing *scores* (two blocks with the same score are sorted by *id*). The resulting sorted array is broadcast back to all processes so that each process knows the scores of all blocks including those belonging to other processes.

Based on this set, the $p$ percent blocks with the lowest score are *reduced*. This reduction step consists simply of keeping the 8 corners of 3D blocks (4 corners for 2D blocks) and their coordinates. In our use case, $55 \times 55 \times 38$-point blocks are reduced to $2 \times 2 \times 2$ points. The percentage $p$ of blocks to reduce is set to 0 for the first iteration and dynamically adapted later based on performance constraints.

The reason for reducing blocks this way, rather than keeping a single point with an average value, for example, is that a reduced block should still be connected to its neighboring blocks. Keeping two points along each dimension allows us to retain the extents of a block. Keeping the values of these points allows a continuity with neighboring blocks. Visualization algorithms will also be able to rebuild more points if necessary using interpolation from these $2 \times 2 \times 2$ points. As can be seen in Figure 1(b), reduced blocks in a region of high variability come out blurry as a result of such interpolation.

More elaborate downsampling strategies could be envisioned, from adapting the number of points kept to the score of

the block, or using Most-Smale complexes [26]. This would, however, involve more computation that would decrease the benefits of our approach. Hence we leave those considerations to future work.

### D. Load redistribution (shuffling)

As a result of block reductions, the amount of data can become imbalanced across processes. Blocks with a high score (therefore not reduced) are indeed likely to be clustered in a small region handled by a reduced number of processes. This imbalance adds up to the imbalance of rendering load, defined as the time required for a piece of data to be rendered. Even if none of the blocks are reduced, the locality of the physical phenomenon and the resulting isosurface lead to some processes having more rendering load than others.

This situation may impair the performance of the final rendering step. In particular the total run time of the rendering step is driven by the run time of its slowest process, that is, the process with the highest load.

In order to gain performance, the blocks must be redistributed across processes. Since process rank 0 already broadcasts the scores of all blocks to all processes, all processes have the same full, sorted list of blocks. Upon reading this list, each process issues a series of nonblocking receives to get blocks that they need and a series of nonblocking sends to send blocks to other processes.

We implement two load redistribution strategies.

- **Random Shuffling** Each process is given the responsibility for a random set of blocks (the number of blocks per process remains constant). The redistribution of blocks is computed the same way in all processes by making sure all processes use the same seed. This strategy constitutes our baseline; it does not take the scores into account, and it does not attempt to optimize communications.
- **Round Robin** The blocks, sorted by their score, are distributed across processes in a round-robin manner. That is, process 0 takes the block with the highest score; process 1 the block with the second highest score, and so on, looping over processes until no more blocks remain to be distributed. This strategy takes the scores into account but does not attempt to optimize communications.

Our experiments show that such communications have a negligible overhead, on the order of 1 second, on the target platform (Blue Waters) compared with the rendering time, on the order of tens to hundreds of seconds.

### E. Adapting to performance constraints

The last step in our approach consists of dynamically adapting the number of blocks that are reduced based on predefined performance constraints. In our case the performance constraint is the maximum run time for the full pipeline to complete.

To implement this adaptive reduction of data, we assume that (1) for a given iteration $n$, the total run time of the pipeline is a monotonically increasing function $f_n$ of the number of nonreduced blocks and (2) for every iteration $n$, $f_{n-1}$ is a good approximation of $f_n$.

Assumption (1) is intuitive, given that all parts of the pipeline either do not depend on the number of reduced blocks (the scoring component and parallel sort) or benefit from the reduction (load redistribution and rendering).

**Algorithm 1** Computes the percentage of blocks to reduce based on the percentages used for the two previous iterations ($p_{n-1}$ and $p_n$) and the observed timings ($t_{n-1}$ and $t_n$); *target* is the required run time of the full pipeline.

```
1:  function ADAPT_PERCENT(target, t_{n-1}, p_{n-1}, t_n, p_n)
2:      if p_{n-1} = p_n then                ▷ Deal with a vertical slope
3:          if t_n > target and p_n < 100 then return p_n + 1
4:          end if
5:          if t_n < target and p_n > 0 then return p_n − 1
6:          end if
7:      end if
8:          ▷ Compute linear estimation, i.e., we find a and b
    such that t = a × p + b
9:      a ← (t_n − t_{n-1}) / (p_n − p_{n-1})
10:     b ← t_n − a × p_n
11:     if a ≥ 0 then return min(100, p_n + 1)        ▷ May
    happen because of randomness in rendering time.
12:     end if
13:     p ← (target − b) / a                  ▷ Estimate next percentage
14:     return min(100,max(p,0))              ▷ Make sure p is in
    [0,100]
15: end function
```

Assumption (2) may not always be true, especially because the performance of the rendering pipeline is inherently variable and because the rendering load varies as the physical phenomenon evolves (for example, if a cloud gets bigger, it spans more domains and requires more time to be rendered). Although we increase the percentage of reduced blocks from iteration $n-1$ to iteration $n$ (which should lead to a decrease of run time), the rendering time may increase as well because $f_{n-1}$ was not a good approximation of $f_n$. Our algorithm takes this case into account by simply increasing the percentage by 1 instead of decreasing the percentage of reduced blocks in the hope of decreasing the run time.

Algorithm 1, our solution to the above problem, starts by assuming that the rendering time $t_0$ when all blocks are reduced is $t_0 = 0$. The first output of the simulation is not reduced ($p_1 = 0$), and leads to a time $t_1$. After the first iteration, we always keep the rendering time and percentages of the two previous iterations ($t_{n-1}$, $p_{n-1}$, $t_n$, $p_n$), and compute an estimate of the rendering time as a function of the percentage. This linear approximation allows us to get the next percentage $p_{n+1}$ required to reach the *target* run time. Lines 2 to 7 prevent our algorithm from being stuck because it used the same percentage two iterations in a row. The case of Assumption (2) being broken is handled in line 10. Line 13 makes sure that the resulting value stays within $[0, 100]$.

We note that, although not studied here, the maximum percentage of reduced blocks could easily be bounded by the user.

## V. Experimental evaluation

In this section, we evaluate all the components of our pipeline individually and together. After describing the experimental setup, we divide our evaluation into several parts, each focusing on a single component of the pipeline. The last part is the overall performance gain.

### A. Description of the experiments

We demonstrate the benefit of our approach through experiments with the CM1 application on NCSA's Blue Waters petascale supercomputer [6]. We focus in particular on the reflectivity field produced by CM1. Although the colormap visualization scenario is already fast (on the order of a second to complete), rendering the isosurface can take several minutes. We therefore focus on this scenario specifically. The colormap will, however, be used to show how the scores given by different metrics map to certain regions of the data.

In our previous work we used Damaris/Viz and VisIt to enable in situ visualization in CM1. In the present work, we use ParaView Catalyst instead, since it allows us to define various batch visualization pipelines through Python scripts. We use an isosurface algorithm for volume rendering. This algorithm computes a mesh of the isosurface using a marching cubes method, then renders this mesh. The rendering time in one process therefore depends on the number of mesh elements handled by this process, which itself depends on the content of the data in this process.

To avoid running CM1's computational part for every experiment, and because interesting phenomena start to appear only after a few thousand iterations, we use a dataset already generated by atmospheric scientists. This method does not limit the generality of our approach, since the real CM1 would normally alternate between computation and visualization phases. The dataset used consists of 572 iterations of data (starting after approximately 5,000 iterations of the simulation), each a $2200 \times 2200 \times 380$ array of 32-bit floating-point values representing the reflectivity on each point of a 3D rectilinear grid. It was generated from a 3-day run of CM1 on Blue Waters. We reloaded this dataset using the Block I/O Library (BIL) [27] into an in situ visualization kernel of CM1 that feeds it to a Catalyst pipeline.

We use 10 iterations, equally spaced in time, to evaluate our approach, except when evaluating the self-adaptation mechanism, in which case we use 30 iterations. We run our experiments on 64 cores (4 nodes) and 400 cores (25 nodes). In both cases, the data is initially read and distributed across processes the same way CM1 would have generated it at these scales.

### B. Score metrics: performance and relevance

We compared our block-scoring metrics in several ways. First, we measured how fast these metrics score blocks. Table I presents the computation time on 64 and 400 cores, with 16,000 blocks of $55 \times 55 \times 38$ floating-point values. These times must be put in perspective with the rendering time. For example, on 64 cores it takes about 160 seconds to render all the blocks without reducing any of them. Using the TRILIN function adds 14.3 seconds to this run time, which, in our opinion, is not acceptable for a function that aims only at guiding a later selection of blocks. We therefore prefer a scoring function such as LEA or VAR, which take only 2.03 and 1.41 seconds, respectively.

The second aspect of the metrics that has to be studied is how they rank blocks compared with one another. Since our approach consists of selecting a percentage of blocks with the highest score, two metrics may not select the same blocks.

In Figure 3, each graph compares a pair of metrics. Each point on a graph represents a block. The abscissa of the point represents the rank of the block when blocks are sorted according to the first metric. Its ordinate represents the rank of the block when blocks are sorted according to the second metric.

TABLE I: Computation time required for different metrics.

| Metric | Time on 64 Cores (sec) | Time on 400 Cores (sec) |
|---|---|---|
| LEA | 2.03 | 0.32 |
| FPZIP | 8.85 | 1.42 |
| ITL | 13.30 | 1.97 |
| RANGE | 7.03 | 1.12 |
| VAR | 1.41 | 0.23 |
| TRILIN | 14.30 | 2.28 |

From these figures we can clearly see a set of blocks that all metrics "agree" are not variable enough to be considered relevant. The scores of these blocks is the minimum score that the metrics can give; therefore they are sorted by *id* rather than by score, leading to the same order according to all metrics. For blocks that present more variability, the metrics tend to disagree on the ordering. This is an expected result because each metric evaluates a *different aspect* of variability. Some relations between metrics can still be highlighted, such as the fact that a large entropy with ITL seems to imply a large variance, while the opposite may not be true, and the fact that the trilinear interpolation score seems to correlate well with the variance, which may come from the fact that in both cases a mean square error with respect to a reference value (for the variance) or function (for trilinear interpolation) is computed.

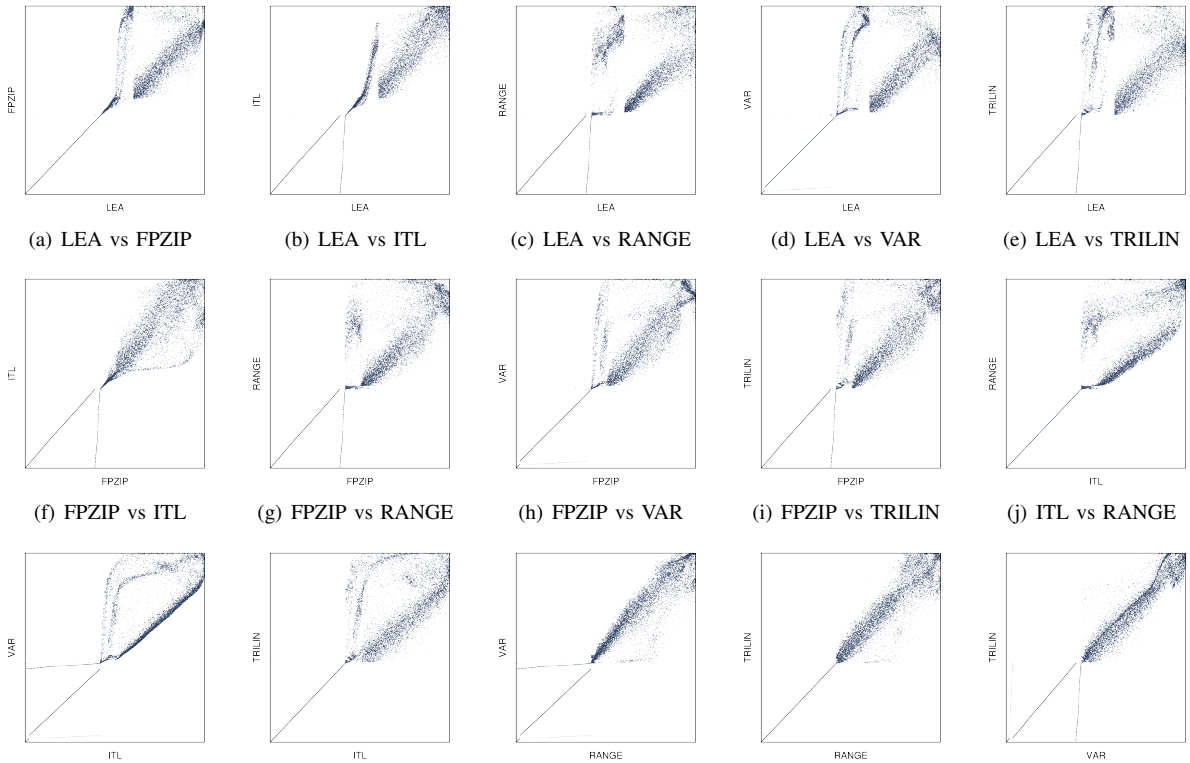
To guide the user in choosing metrics, we display an image (such as the colormap presented in Section II) and show how each block part of the image is scored. This kind of 2D visualization is easy to compute and fast; it can also be done offline with samples of data from previous runs of the simulation.

Figure 4 shows *score maps*, that is, colormaps of the domain where colors represent scores of blocks, and compares them with the original reflectivity field. It shows that some metrics such as VAR or TRILIN give a higher score to regions with larger overall variability (e.g., contours of the phenomenon) while others such as ITL or FPZIP also give a high score to blocks inside the phenomenon itself. Note that the longer blocks on the borders of the domain are due to the simulation grid, which is rectilinear. These blocks have the same number of points as any other.

### C. Performance benefit of load redistribution

We then confirmed that redistributing the blocks to divide the cost of the physical phenomena benefits the rendering performance. To do so, we ran our pipeline without load redistribution, with random load redistribution, and with load redistribution in a round-robin fashion according to different metrics. Figure 5 shows the rendering time in these experiments. The communication time is 1.2 seconds on 64 cores and 0.6 seconds on 400 cores, both for the random shuffling strategy and the round-robin policy.

These results show that simply by redistributing the load, we can achieve a $5\times$ speedup on 400 cores and a $4\times$ speedup on 64 cores. It also shows that there is no benefit in taking the scores into account; randomly redistributing blocks already achieves a good statistical load balancing because of the relatively small size of the phenomena of interest compared

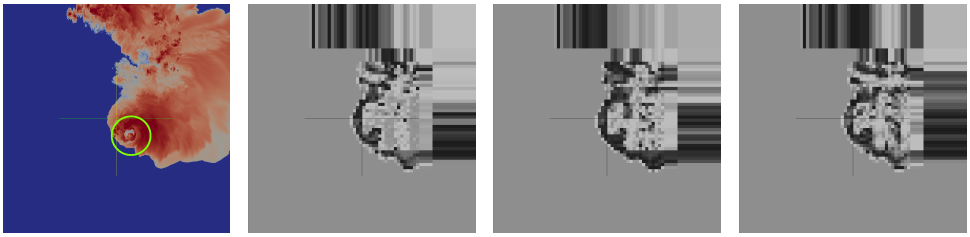



(a) LEA vs FPZIP (b) LEA vs ITL (c) LEA vs RANGE (d) LEA vs VAR (e) LEA vs TRILIN

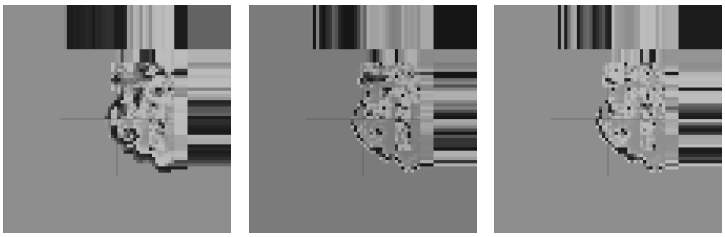
(f) FPZIP vs ITL (g) FPZIP vs RANGE (h) FPZIP vs VAR (i) FPZIP vs TRILIN (j) ITL vs RANGE

(k) ITL vs VAR (l) ITL vs TRILIN (m) RANGE vs VAR (n) RANGE vs TRILIN (o) VAR vs TRILIN

Fig. 3: Comparison of block orderings produced by various metrics. Each graph compares two metrics. Each point represents a block. The abscissa of the point represents the rank of the block when the blocks are sorted in ascending order according to the first metric (least important blocks on the left). The ordinate of the point is the rank of the block when sorted in ascending according to the second metric (least important block at the bottom).

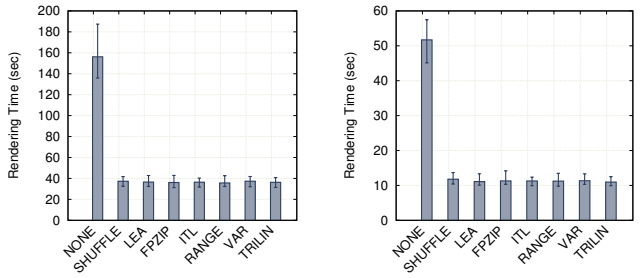


(a) Original dBZ field (b) LEA scoremap (c) FPZIP scoremap (d) ITL scoremap



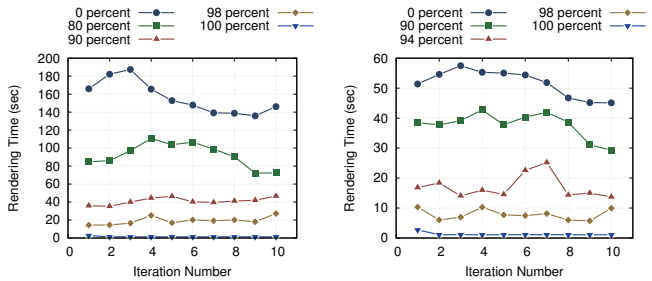
(e) RANGE scoremap (f) VAR scoremap (g) TRILIN scoremap

Fig. 4: Scoremaps: greyscale colormap of the domain according to different scores – darker regions indicate higher scores.




(a) 64 cores (b) 400 cores

Fig. 5: Run time of the rendering pipeline when none of the blocks are reduced, but load redistribution is enabled based on scores provided by different metrics. NONE represent the case where the load has not been redistributed. SHUFFLE corresponds to random shuffling. All others correspond to a round-robin distribution according to scores.




(a) 64 cores (b) 400 cores

Fig. 6: Run time of the rendering pipeline for 10 iterations, with different percentages of blocks being reduced.

with the size of the full domain. Note that while the scores are not relevant for load redistribution, they are however still relevant for knowing which blocks to reduce.

Section V-E studies the interaction of the load redistribution component and the block reduction component.

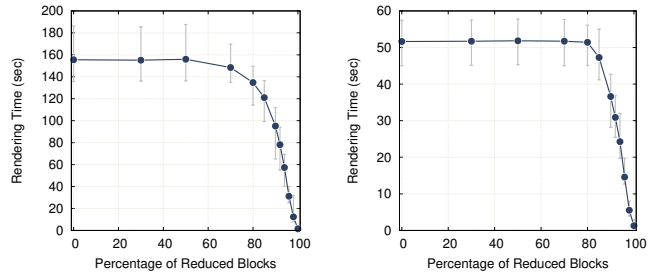
### D. Performance benefit of block reduction

In the next series of experiments we evaluate how much performance is gained by reducing a certain percentage of the blocks, and we show how the run time evolves as a function of the percentage of blocks being reduced. We used the VAR metric (other metrics yield similar results) to score blocks of data.

Figure 6 shows the run time of 10 iterations for different percentages of blocks being reduced. When none of the blocks are reduced, the run time is 160 seconds on 64 cores and 50 seconds on 400 cores. When all the blocks are reduced, this run time goes down to 1 second in both cases. This defines the margins within which we can adapt the performance by changing the number of blocks being reduced when load redistribution is not involved.

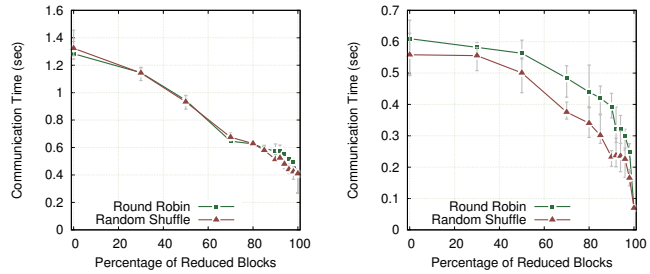
This figure also shows that the rendering time is not the same from one iteration to another. As will be shown later, this variability will affect our adaptation algorithm.

Figure 7 presents the same results as a function of the percentage of reduced blocks, with error bars representing minimum and maximum across the 10 iterations. We observe




(a) 64 cores (b) 400 cores

Fig. 7: Run time of the rendering pipeline (average, minimum, and maximum across 10 iterations) as a function of the percentage of blocks being reduced.




(a) 64 cores (b) 400 cores

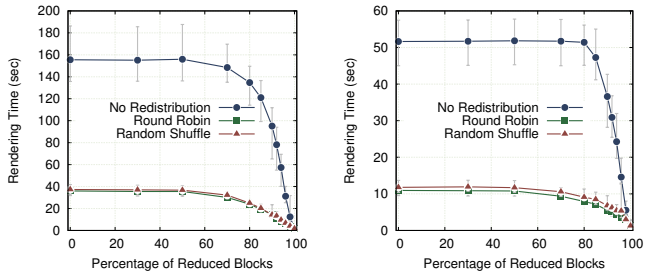
Fig. 8: Run time of the redistribution component (average, minimum, and maximum across 10 iterations) as a function of the percentage of blocks being reduced.

that the performance improvement is not proportional to the percentage of reduced blocks. Instead, a majority of the blocks need to be reduced before we start observing performance improvements. The first reason is that the selection of blocks to be reduced is based on their score, yet blocks with a high score are not evenly distributed across processes. Hence a few processes are likely to have a large number of high-scored blocks and will not see their load being reduced until the percentage is high enough that we start selecting their blocks, too. The second reason is that with the rendering algorithm used here (isosurface volume rendering), many blocks are transparent and therefore take a negligible time to render.

### E. Combined reduction and load redistribution

Data reduction has a potential impact on the time to perform load redistribution. Indeed, since data is reduced before being redistributed, reducing more blocks means exchanging less data. Although this redistribution time is negligible compared with the rendering time, we show in Figure 8 how it evolves as a function of the number of blocks being reduced. For this set of experiments we used the LEA metric. As expected, the communication time decreases as we increase the percentage of reduced blocks, as a result of a lower amount of data to be exchanged.

Load redistribution combined with data reduction has an effect on the rendering performance. This effect is shown in Figure 9. We can see that load redistribution not only improves performance but also reduces the variability of the rendering tasks.




(a) 64 cores (b) 400 cores

Fig. 9: Run time of the rendering component (average, minimum and maximum across 10 iterations) as a function of the percentage of blocks being reduced, when load redistribution is enabled or disabled.

Additionally, Figure 9 shows that the round-robin and random policies lead to the same performance of the rendering task; that is, a score-guided redistribution achieves a load balancing equivalent to the statistical load balancing.

### *F. Dynamic adaptation*

We first evaluate our dynamic adaptation technique without the redistribution component. We then add the redistribution component to show the resulting performance of the entire pipeline.

*1) Adaptation without redistribution:* In this set of experiments, we set a target run time of 120, 60, and 20 seconds per iteration on 64 cores and 30, 15, and 7 seconds on 400 cores. Figures 10(a) and 10(b) present the resulting run time for 30 iterations. They show that our approach can successfully adapt the percentage of reduced blocks in order to reach a target run time per iteration. Figures 10(c) and 10(d) show that the percentages have stabilized after a few iterations. The variability observed in the run time comes from the inherent variability of the visualization task.

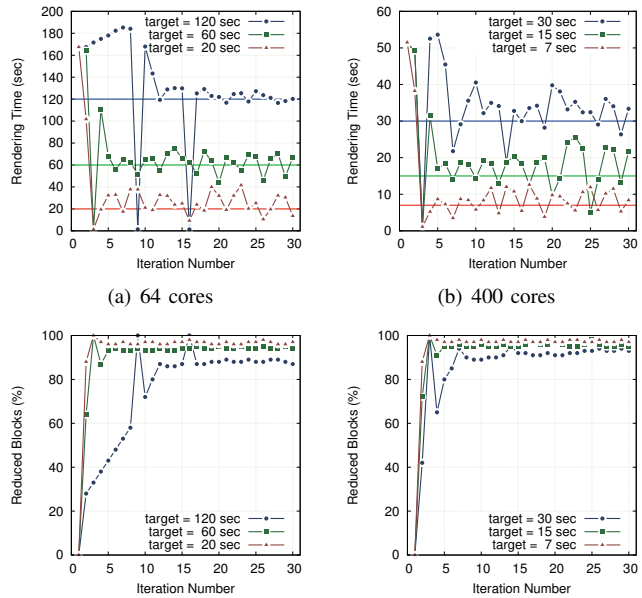
*2) Adaptation with redistribution enabled:* We evaluate the full pipeline, including load redistribution, with dynamic adaptation. Figure 11 presents the resulting run time for 30 iterations. Here the target run time is 25 and 10 seconds per iteration on 64 cores and 7 and 3 seconds per iteration on 400 cores. We used the same scale for the $y$ axis as in Figure 10 so that Figures 10 and 11 can be compared. These results show that our pipeline not only improves performance but also can meet performance constraints despite the variability of the rendering task.

*3) Feedback from scientists:* Informal discussions with atmospheric scientists indicated that they were particularly interested in the vortex region at the center of the domain (this region is circled in green in Figure 4). When they were shown the scoremaps in Figure 4 for feedback, their interest turned to the VAR and TRILIN metrics, which give a high score to this region while giving a low score to its surrounding.

Additionally, they confirmed that the produced visual results were satisfactory for their purpose of tracking the evolution of the phenomena while the simulation runs.

## VI. Conclusion

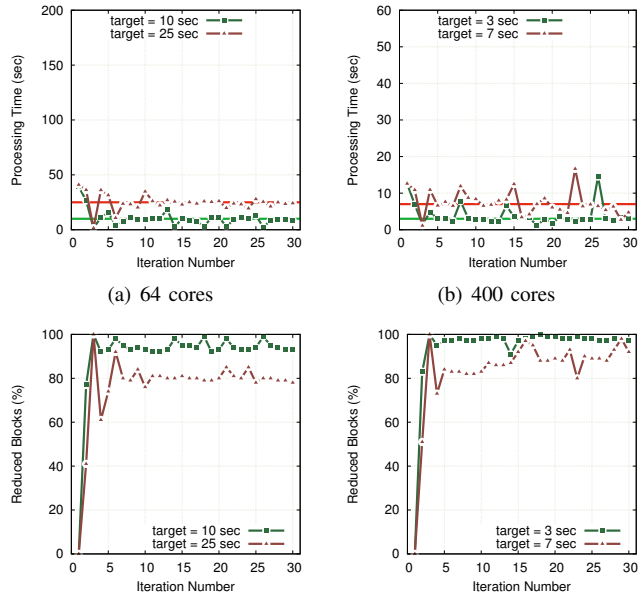
While in situ visualization enables faster insight into a running simulation, it can increase the simulation's run time




(a) 64 cores (b) 400 cores

(c) 64 cores (d) 400 cores

Fig. 10: Rendering time (a+b) and percentage of blocks reduced (c+d) on 64 and 400 cores when trying to converge toward a specified run time. Load redistribution is not activated here.




(a) 64 cores (b) 400 cores

(c) 64 cores (d) 400 cores

Fig. 11: Full pipeline (including load redistribution) completion time (a+b) and percentage of blocks reduced (c+d) on 64 and 400 cores when trying to converge toward a specified run time.

and increase its variability. Needed, therefore, are ways to improve the performance of in situ visualization, as well as to make its task fit in a given performance budget, even at the cost of reduced visual accuracy.

In this paper, we have addressed the challenge of improving in situ visualization performance in the context of a climate simulation. We realized that the strong locality of the phenomenon of interest limits the performance of a normal rendering pipeline. Hence, we proposed redistributing blocks of data and reducing a percentage of them based on their content. Additionally, we proposed adapting the percentage so that our pipeline adheres to performance constraints. We have shown that our pipeline can speed the visualization time by $5\times$ on 400 cores without affecting the visual results and that it can effectively meet given performance constraints provided that data reduction is allowed.

We plan to investigate whether more elaborate redistribution algorithms are necessary in order to achieve the same results at larger scale and on platforms with lower network performance. We will also investigate multivariate scores and other visualization scenarios tied to other field variables of CM1 and other simulations.

## Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357. This work is also supported by DOE with agreement No. DE-DC000122495, program manager Lucy Nowell.

## References

[1] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. OLeary, V. Vishwanath, B. Whitlock *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," in *18th EG/VGTC Conference on Visualization (EuroVis)*. The Eurographics Association and John Wiley & Sons Ltd., 2016.

[2] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System," in *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*. Eurographics Association, 2011.

[3] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen, "The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library," in *LDAV, IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2011.

[4] M. Dorier, R. Sisneros, Roberto, T. Peterka, G. Antoniu, and B. Semeraro, Dave, "Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework," in *LDAV - IEEE Symposium on Large-Scale Data Analysis and Visualization*, Atlanta, GA, USA, Oct. 2013. [Online]. Available: http://hal.inria.fr/hal-00859603

[5] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and Integration for Scientific Codes through the Adaptable IO System (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, ser. CLADE '08. New York, NY, USA: ACM, 2008, pp. 15–24. [Online]. Available: http://doi.acm.org/10.1145/1383529.1383533

[6] NCSA, "Blue Waters project," http://www.ncsa.illinois.edu/BlueWaters/.

[7] G. H. Bryan and J. M. Fritsch, "A Benchmark Simulation for Moist Nonhydrostatic Numerical Models," *Monthly Weather Review*, vol. 130, no. 12, pp. 2917–2928, 2002.

[8] A. C. Bauer, B. Geveci, and W. Schroeder, "The ParaView Catalyst Users Guide v2.0. Kitware, Inc." http://www.paraview.org/files/catalyst/docs/ParaViewCatalystUsersGuide-v2.pdf, 2015.

[9] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," in *ACM Siggraph Computer Graphics*, vol. 21, no. 4. ACM, 1987, pp. 163–169.

[10] H. Zou, F. Zheng, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, Q. Liu, N. Podhorszki, and S. Klasky, "Quality-Aware Data Management for Large Scale Scientific Applications," in *SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, Nov. 2012, pp. 816–820.

[11] P. Malakar, V. Natarajan, and S. S. Vadhiyar, "An Adaptive Framework for Simulation and Online Remote Visualization of Critical Climate Applications in Resource-constrained Environments," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.10

[12] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi, "Using Cross-layer Adaptations for Dynamic Data Management in Large Scale Coupled Scientific Workflows," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 74:1–74:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503301

[13] C. Wang, H. Yu, and K.-L. Ma, "Importance-Driven Time-Varying Data Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1547–1554, Nov. 2008.

[14] B. Nouanesengsy, J. Woodring, J. Patchett, K. Myers, and J. Ahrens, "ADR Visualization: A Generalized Framework for Ranking Large-scale Scientific Data using Analysis-Driven Refinement," in *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, Nov 2014, pp. 43–50.

[15] D. Thompson, N. Fabian, K. Moreland, and L. Ice, "Design Issues for Performing In Situ Analysis of Simulation Data," Technical Report SAND2009-2014, Sandia National Laboratories, Tech. Rep., 2009.

[16] LLNL, "VisIt, https://wci.llnl.gov/codes/visit/."

[17] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - Preparatory Data Analytics on Peta-Scale Machines," in *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, April 2010, pp. 1–12.

[18] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-Aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409

[19] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *IEEE International Conference on Cluster Computing (CLUSTER)*, Sept. 2012, pp. 155 –163.

[20] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.28

[21] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform," *Parallel and Distributed Processing Symposium, International*, pp. 1352–1363, 2012.

[22] A. Chaudhuri, T.-Y. Lee, B. Zhou, C. Wang, T. Xu, H.-W. Shen, T. Peterka, and Y.-J. Chiang, "Scalable Computation of Distributions from Large Scale Data Sets," in *Proceedings of the 2012 IEEE Large Data Analysis and Visualization Symposium LDAV'12*, Seattle, WA, 2012.

[23] P. Lindstrom and M. Isenburg, "Fast and Efficient Compression of Floating-Point Data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, Sept 2006.

[24] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[25] L. Gomez and F. Cappello, "Improving Floating Point Compression through Binary Masks," in *IEEE International Conference on Big Data*, Oct. 2013, pp. 326–331.

[26] N. Shivashankar and V. Natarajan, "Parallel Computation of 3D Morse-Smale Complexes," *Comput. Graph. Forum*, vol. 31, no. 3pt1, pp. 965–974, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1111/j.1467-8659.2012.03089.x

[27] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross, "Visualization Viewpoint: Towards a General I/O Layer for Parallel Visualization Applications," *IEEE Computer Graphics and Applications*, vol. 31, no. 6, 2011.