Adaptive Elasticity Policies For Staging-Based *In Situ* Visualization

Zhe Wang, Matthieu Dorier, Pradeep Subedi, Philip E. Davis, Manish Parashar

Abstract

In situ processing aims to alleviate the growing gap between computation and I/O capabilities by performing data processing close to the data source. *In situ* processing is widely used to process data generated by multiple data sources, including observation data from edge devices or scientific observational facilities and the simulation data generated by scientific computation on a high-performance computing (HPC) platform. For a scientific workflow that is run on an HPC platform and composed of a simulation program and an *in situ* data analytics or visualization (abbreviated as ana/vis) task, there is an implicit assumption that the computing resources assigned to the workflow keep static during the workflow execution. However, with the converging trend between the HPC and cloud computing platform, running the *in situ* ana/vis task in an elastic way is promising to decrease its overhead and improve its resource utilization rate. Resource elasticity represents the ability to change resource configurations such as the number of computing nodes/processes during workflow execution. An elastic job may dynamically adjust resource configurations; it may use a few resources at the beginning and more resources toward the end of the job when interesting data appear. However, it is hard to predict *a priori* how many computing nodes/processes need to be added/removed during the workflow execution to adapt to changing workflow needs. How to efficiently guide elasticity operations, such as growing or shrinking the number of processes used for *in situ* analysis during workflow execution, is an open-ended research question. In this article, we present adaptive elasticity policies that adopt workflow runtime information collected during workflow execution to predict how to trigger the addition/removal of processes in order to minimize *in situ* processing overhead. *Taking in situ visualization tasks as an example*, we integrate the presented elasticity policies into a staging-based elastic workflow and evaluate its efficiency in multiple elasticity scenarios. Compared with the situation without elasticity or with a static elasticity policy that uses a fixed number of processes for each rescaling operation, the adaptive elasticity policy can save overhead in finding a proper resource configuration and improve resource utilization efficiency. For example, one experiment illustrates that the adaptive elasticity policy saves 41% of core-hours compared with the situation without the resource elasticity.

In Situ ProcessingScientific Visualization, Data Staging, Elasticity Policies

1 Introduction

In situ processing addresses the gap between computation and I/O capabilities by processing data as they are generated. The precise definition of *in situ* processing depends on the platform it targets. For the data generated by edge devices [1] or observational facilities [2], *in situ* processing represents executing tasks on the devices without transferring the data to remote cloud platforms [3]. This research mainly focuses on the scientific workflow composed of the simulation and data analysis or visualization (abbreviated as ana/vis) applications running on HPC platforms. The *in situ* processing in this context is mainly divided into *tightly-coupled* and *loosely-coupled* system [4]. In a tightly-coupled system, the simulation program is linked with an *in situ* library. When APIs for *in situ* processing are called, the *in situ* library transforms the data layout as needed and executes data ana/vis tasks. After that, the *in situ* library executes its operations on the same hardware, such as the CPU and memory, with the simulation application. In contrast, the simulation program in a loosely-coupled system is linked using an API for data management. Once the API is called for *in situ* processing, data blocks generated by the simulation are transferred to other remote nodes via a high-speed network [5, 6] or to other processes on the same nodes via shared memory [7, 8]. The memory used to store the transferred simulation data, whether local to the simulation nodes or remote, is often called a *data staging area*. A *staging-based scientific workflow* adopts a loosely-coupled *in situ* approach to perform ana/vis tasks. In this article, we use the term *in-staging processing* to represent the process of executing ana/vis tasks *within the data staging area*.

The data blocks that contain interesting scientific information are usually unevenly distributed across processes and iterations of simulation computation. For example, the simulation that adopts adaptive mesh refinement (AMR) uses dynamic local refinements during the computation. This operation can increase the spatial and temporal resolution of the specific interesting domain to acquire detailed information. However, it may also increase the resource cost for analyzing the data [9]. Similarly, interesting phenomena of the simulation based on Cloud Model 1 (CM1) [10], such as tornadoes, appear at the end of the simulation. To finish data processing within specific time constraints, more computing resources are required for data analytics tasks once these interesting data appear. The mismatch between the data waiting to be processed and the available computing resources may slow down *in situ* processing.

Multiple research solutions can help to relieve the mismatch between data generation and consumption. For example, the processing frequency of simulated data can be adjusted during workflow execution [11], and the number of ana/vis tasks can also be adjusted speculatively to balance the data production rate and available resources [12]. The resource elasticity of *in situ* processing indicates that the number of processes executing ana/vis tasks can be changed, such as adding or removing physical computing resources (cores and nodes [13]), during the workflow execution. This elasticity mechanism can improve resource utilization efficiency by allocating a proper amount of computing resources ac-

cording to the current workload level [9]. Resource elasticity is also identified as a key research challenge for processing dynamic workloads for *in situ* processing [14, 13, 15]. In particular, resource elasticity consists of two abstract primitives: the *resource join* operation can start a new task or schedule new computing resources, and the *resource leave* operation can release computation resources when they become idle[1].

Resource elasticity policies are well explored on the cloud computing platform [18]. With the convergence between cloud platform and HPC platform [19], either migrating classic *in situ* processing onto the cloud platform with full-fledged elasticity capabilities or executing it on the HPC platform managed by an elastic scheduler [20], efficient elasticity strategies need to be carefully designed to decrease the overhead of *in situ* processing. Assuming the underlying platform on which the *in situ* processing runs on supports the computing resource elasticity, one typical research question is how elasticity policies decide *when and how to add or remove computing resources*. Although this research mainly targets *in situ* applications running on HPC platforms, elasticity strategies used for cloud computing are still worth using for reference.

The design of resource elasticity policies is widely explored by research works in the area of cloud computing [18]. A typical resource elasticity policy needs to consider the following factors [21]: (1) the condition to trigger the elasticity operation, (2) the method to forecast the future workload, and (3) the method to compute how many computing resources need to be added or removed for rescaling actions to satisfy the future resource requirements. There are multiple existing elasticity policies for applications running on the cloud computing platform [18]. Compared with web applications running on the cloud computing platform, the different assumptions and resource constraints for *in situ* processing listed below require specialized elasticity policies.

Resource elasticity goals: The primary goal of the elasticity policy is to improve the workflow execution time within the available resource constraints. *In situ* ana/vis tasks usually run concurrently with the simulation, and these tasks may need to overlap with simulation computation as much as possible to decrease the workflow execution time. Furthermore, when the simulation execution time is much longer than the execution time of ana/vis tasks for each iteration, the elasticity policy can decrease the computing resources allocated to *in situ* ana/vis tasks to improve resource utilization.

Elasticity operations for MPI applications: *In situ* ana/vis tasks are usually implemented using the MPI and run in a parallel manner. Performance models of these applications need to consider how the data size and available resources influence *in situ* processing. Besides, the rescaling operation may cause overhead, such as time for syncing newly added processes. Efficient elasticity policies need to balance the tradeoff between the benefits and the overhead of rescaling operations.

Forecasting future workloads: Although multiple machine learning meth-

[1]The term *expand/contract* or *grow/shrink* is also used in related works [16, 17] with a similar meaning.

ods are used for training the performance model to forecast the future workloads [18, 22], these methods are difficult to be used in the *in situ* processing scenario. This is because few historical datasets are available to train the performance model for *in situ* processing running in a streaming way. If we assume little prior knowledge about how data are generated by simulation computation, an adaptive online method is required to forecast the application execution time.

How to achieve elasticity policies that satisfy the requirements above in an efficient way for *in situ* processing has not yet been fully explored. This article mainly focuses on the resource elasticity for scientific visualization[2], an important type of task that can run *in situ* [4]. We present adaptive elasticity policies to guide the execution of rescaling operations, i.e., adding or removing processes, for visualization running in the data staging service. In particular, we first present a model that can predict how the execution time of in-staging visualization changes with the number of running processes and the size of data processed by visualization tasks. Based on the prediction results of this model, the elasticity policy can decide how many processes should be added or removed. Depending on the resource constraints of the job, we further discuss two scenarios for using elasticity for *in situ visualization*. One scenario assumes the computing resources allocated to the job are fixed, and the elasticity policy can redistribute resources among applications; another scenario assumes the computing resources allocated to the job can change during the workflow execution. We integrate presented policies into a staging-based elastic workflow and evaluate its efficiency on the Cori Cray-XC40 system at the National Energy Research Scientific Computing (NERSC). Compared with a *plan-driven* elasticity operation with static rescaling plans, the results show that adaptive elasticity policies produce less overhead in searching for a resource configuration that is favorable for the *in situ* visualization. The results also illustrate the effect of elasticity policies in saving the core-hours for the *in situ* visualization.

This article builds on our previous work [23] and makes new contributions along multiple dimensions. First, we extend the model used to estimate the *in situ* visualization execution time. Previously, we had considered only how the number of processes influences the *in situ* visualization execution time; in this article, we discuss how *in situ* visualization execution time changes with both the varied simulated data size and the number of processes. Second, we systematically discuss different scenarios that adopt *in situ* elasticity policies. For example, when the total computing resources are fixed during the workflow execution, we can redistribute resources allocated to the simulation and the data staging service[3]. When the total computing resources can be changed during the workflow execution, we can rescale the resource allocated to the data staging service. Third, our previous work adopts only the synthetic data ana/vis tasks in the evaluation. In contrast, we adopt a distributed visualization that supports the capability of resources elasticity in the evaluation of this article.

[2]We mainly target the parallel rendering task when we discuss the *in situ* visualization task in this work.

[3]The simulation program and the data staging service run in a concurrent manner and share the computing resources assigned to the current job.

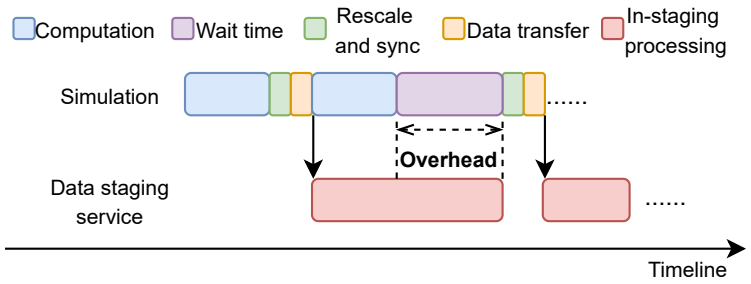


Figure 1: Different stages of in-staging processing with resource elasticity.

The rest of the article is organized as follows: Section 2 discusses the background and motivation. In Section 3, we elaborate on the design of adaptive elasticity policies. In Section 4, we further discuss the implementation details of integrating the adaptive elasticity policies into the in-staging processing with visualization tasks. The evaluation results are discussed in Section 5, and then we further discuss relevant aspects that are outside the scope of the evaluation in Section 6. We discuss related work in Section 7, and conclude the article in Section 8.

2 Background and Motivation

This section introduces background information from the application layer to the infrastructure layer with a top-down approach. We first introduce typical stages of the *in situ* workflow with the elastic in-staging processing. Then we use the visualization task as an example to show how the in-staging processing time can be affected by available computing resources and the size of data to be processed. After that, we discuss a method that utilizes the concept of autonomic computing and event-driven programming to implement the adaptive elasticity policies for *in situ* processing. Lastly, we discuss how the HPC platform supports applications with a resource elasticity design.

2.1 *In situ* workflow with elastic in-staging visualization

Figure 1 illustrates key stages of an *in situ* workflow with elastic in-staging processing [24]. We assume the data staging service supports resource elasticity, and elasticity primitives may be triggered after any iteration of the simulation computation according to workflow runtime information. After each simulation computation, the *Rescale and sync* stage of the simulation program checks if it is necessary to trigger rescaling operations. If not, the number of data staging processes for executing ana/vis remains constant, and data blocks generated by the simulation are transferred to the data staging service. If the in-staging processing takes too long to complete compared with the simulation computation,

we may use a rescaling operation to add more computing resources and increase the data consumption rate. In this situation, the elasticity primitive is executed, and the number of available data staging services may change. The simulation process needs to confirm which processes of the data staging service are alive and transfer data to the staging service with a varied number of processes.

However, when one rescaling operation is in progress, the simulation cannot transfer data to the staging service before completion of the rescaling. This is because the rescaling operation may interfere with the data placement mechanism and communication channels used by the data staging service. The task executed by the data staging service may require a static number of processes [15]. For example, the parallel image compositing operation [25] usually assumes the process number does not change during the parallel execution. The same constraint can be found for other parallel visualization filters [26, 27]. In this situation, rescaling operations cannot be executed while in-staging processing tasks are in progress. This constraint introduces an overhead to the *in situ* processing with resource elasticity, and the elasticity policy needs to balance the trade-off between the benefits and overhead of rescaling operations. The main goal of the elasticity policy is to make the consumption rate match the production rate under specific resource usage constraints while minimizing the overhead caused by rescaling operations as much as possible.

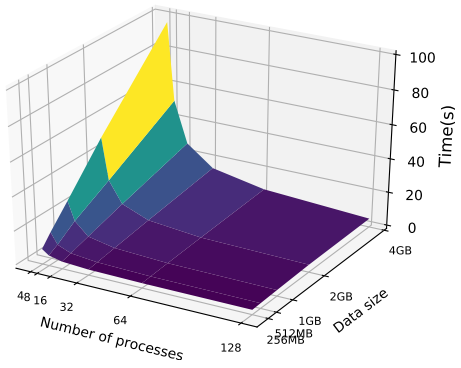
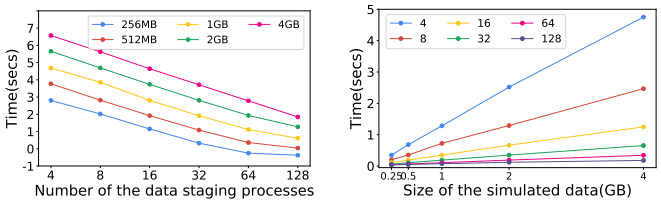


Figure 2: An example that shows how the execution time of in-staging visualization changes with the number of data staging processes and the size of the data generated by simulation.

2.2 Factors affecting in-staging visualization time

By comparing the disparity between the simulation computation and in-staging processing time shown in Figure 1, the elasticity policy can decide when to trigger the rescaling operation. In addition, it is also necessary to properly estimate how many processes should be added or removed for each rescaling operation. The execution time of particular in-staging processing might be influenced by factors such as the number of data staging processes or the size of



(a) Case 1 (b) Case 2

Figure 3: Subfigure (a) shows how the in-staging data visualization time changes with the number of the data staging process, and the log scale with base 2 is used for both x and y axes for this figure. Subfigure (b) shows how the in-staging data visualization time changes with the size of the simulated data.

the simulated data. If there is a mechanism to predict how the data processing time changes with these factors, the elasticity policy can further compute the number of processes that should be added or removed to achieve a targeted in-staging processing time.

Figure 2 illustrates an example of how the in-staging visualization time changes when varying the number of data staging processes and the data size. We use the Gray-Scott mini-application[4] as the data source, extract its isosurface, and visualize its data in the staging area using VTK and Catalyst [28, 27] in a distributed manner. We construct multiple configurations that contain different sizes of simulation data and numbers of data staging processes. We then execute the associated visualization pipeline with these configurations and record execution times. From the results, the in-staging visualization time is inversely proportional to the number of processes and positively related to the size of simulated data in different degrees. The data processing task discussed in this article is mainly the scientific visualization (parallel rendering task), a common task that can be processed in an *in situ* manner for the simulation-visualization scientific workflow. The execution time of other types of data processing tasks executed *in situ* can be influenced by various factors, even the domain-related variables, which are outside the scope of this article. For example, the execution time of particle advection [29] is affected by the seeding strategies; and the execution time of the analysis task designed to find halo regions for cosmological simulation [30] is affected by the density threshold describing the halo region.

Figure 3 provides more details on how the execution of in-staging visualization changes with associated factors. In Figure 3(a), the speed-up of the execution time remains comparatively constant when we double the number of the data staging processes. Gradually adding more processes makes the execution time decrease in proportion. For example, when there are four processes and 4GB of simulated data for each simulation iteration, adding four more processes can reduce the in-staging visualization time by approximately half. However, when there are 64 processes, adding four more processes is negligible for de-

[4]https://github.com/pnorbert/adiosvm/tree/master/Tutorial/gray-scott

creasing the in-staging visualization time, and we need up to 64 more processes to reduce the visualization time by approximately half. This observation shows that adding a particular number of processes achieves different benefits depending on the existing available data staging processes. In contrast, for the results shown in Figure 3(b), the in-staging visualization time increases in proportion to the size of the simulated data. In order to design a proper elasticity policy to trigger rescaling, we need to collect workflow runtime information and use this information to estimate how the execution time changes with a variation in the number of processes and the size of simulated data. Then we can confirm how many processes need to be added/removed in order to achieve the targeted execution time. If an *in situ* workflow with new data processing tasks using the same data source starts execution, the existing performance model may become unsuitable for these new tasks. We need to recollect the runtime information for these new data processing tasks and recompute their performance models (discussed in subsection 3.1).

2.3 *In situ* trigger for supporting resource elasticity

Managing the software system in an autonomic way can be implemented by the paradigm of autonomic computing [31, 32]. From the perspective of the system architecture, the autonomic system contains one or more *autonomic elements* that serve different self-management goals. The autonomic elements typically consist of *autonomic manager* and *managed elements*. The *autonomic manager* is in charge of monitoring and controlling the autonomic elements, which represent all kinds of resources such as storage, computing resources, or other abstracted entities that can be adjusted. Conceptually, the *autonomic manager* needs to provide the capability to monitor the *management elements*, analyze the monitored data, and execute particular actions to update the *managed elements* based on a customized plan that describes objectives in a high level, such as "minimizing the workflow execution time".

In the context of *in situ* processing, the trigger mechanism has been widely adopted to dynamically adjust the running behaviors of *in situ* processing [33, 34, 35]. These examples can be viewed as concrete implementations following the methodology of autonomic computing. In particular, triggers are usually lightweight analysis tasks that are composed of three main stages: *trigger detection*, *trigger decision*, and *trigger action*. The *trigger detection* is in charge of inspecting the content of the data generated by the simulation or workflow runtime. The *trigger decision* decides how to trigger the action based on specific user-defined principles. In the context of the elastic in-staging processing, the *trigger action* can be the execution of computing resource rescaling operations; the *trigger detection* can be the process to monitor and collect and workflow runtime information; the *trigger decision* can be the dedicated strategies to decide when and how to trigger rescaling operations based on the result of the *trigger detection*.

2.4 Executing elastic applications on the HPC platform

When the elasticity policy decides to trigger an elasticity operation, the underlying resource management infrastructure needs to respond to the associated operation. Checkpoint/restart is a classical mechanism with which to implement elasticity primitives with few resource requirements for the resource management software, and it requires only the disk that can store the data. The application needs to stop the current job and start a new job with an adjusted number of nodes or processes [36, 16]. However, saving and reloading data based on checkpoint/restart operation adds overhead to the workflow runtime. With the support of dynamic process in the MPI standard 2.0 and associated extensions [37] or MPI-like communication libraries[5], resource elasticity can be achieved in an online manner without restarting the job and application [15] for the HPC platform. With these supports of elasticity in the communication layer, components of the *in situ* processing tend to be implemented in an elastic manner during the workflow execution without rebooting. For example, the simulation [38] and associated *in situ* processing [13, 15] can be elastic to accommodate changes in requirements of data generation and consumption, and the emerging batch scheduler for the HPC platform can also dynamically add or remove nodes during job execution [17, 20] without restarting a new job.

3 Design of Elasticity Policies

The critical steps of elasticity policies are to decide when and how to trigger rescaling operations. Depending on different scenarios of elasticity, we need to specify the conditions to trigger the elasticity and the number of processes added into (or removed from) the data staging service. In this section, we first present a model in subsection 3.1 to illustrate how the *in situ* processing time is related to the data size and the number of data staging processes. Based on this model, we further discuss designs of adaptive elasticity policies for two typical scenarios in subsection 3.2. Finally, subsection 3.3 discusses how the overhead of the elasticity operation influences key parameters of the elasticity policies.

3.1 Modeling *in situ* processing execution time

We first discuss how the execution time of distributed *in situ* processing is influenced by the number of processes when the total data size is fixed. According to the results shown in Figure 3(a), there is a linear relationship between the *in situ* processing execution time and the number of processes in log scale. We use $y = a \cdot x^b$ as a fitting function[6] to describe this relationship. Specifically, y represents the execution time of the in-staging processing, and x represents the number of processes; a and b are two parameters influenced by the properties of

[5] e.g. `https://github.com/mochi-hpc/mochi-mona`

[6] This function is a typical form of the power-law distribution [39]. Although we adopt the power-law distribution as a fitting curve in this article, the model used to predict the process number can be expressed by different equations.

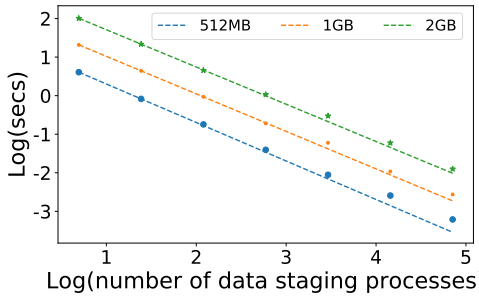


Figure 4: Comparison between actual values and predicted values. The solid dot represents the actual value, and the dashed line represents the estimated value based on curve fitting.

ana/vis tasks and processed data size, and b is usually a negative real number. We can take the logarithm of both sides of the original equation and change it to a linear equation as follows:

$$\ln y = \ln a + b \cdot \ln x \tag{1}$$

We then need only two sample points to estimate the values of a and b. In order to describe how the ana/vis processing time is influenced by the number of the data staging processes (illustrated in Figure 3(a)), we use the first two sample points as input to estimate unknown parameters in the equation $y = a \cdot x^b$. Figure 4 uses a dashed line to represent the estimated data and uses solid dots to represent actual data for the logarithmic version of the equation. As shown in the results, the estimated data calculated by the model can accurately match the actual data with different simulated data sizes. Therefore, given an execution time of in-staging processing, the associated number of processes can be expressed as

$$x = \exp(\frac{\ln y - \ln a}{b}) \tag{2}$$

Assuming the number of data staging processes is x_0, the in-staging execution time is y_0, and the targeted in-staging execution time is y_1; in order to decrease the execution time from y_0 to y_1, we can use the following equation to compute the number of processes that should be added into the data staging service:

$$p = \exp(\frac{\ln y_1 - \ln a}{b}) - \exp(\frac{\ln y_0 - \ln a}{b}), \tag{3}$$

where p represents the number of processes added to the data staging service. In addition, we can use a similar equation to compute how many processes can be removed if we need to increase the in-staging processing time.

We further loosen constraints and discuss the situation in which the *in situ* processing time is influenced by both the data size and the number of data

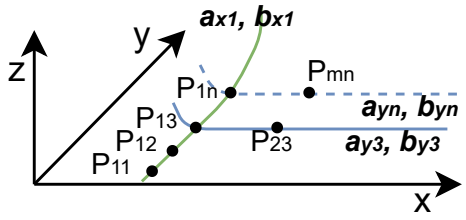


Figure 5: Illustration of the algorithm that can predict the execution time with the varied data size and the number of processes.

staging processes. Two submodels need to be considered in this situation: the first model assumes a fixed data size and a varied number of processes (which have been discussed); the second model assumes a fixed number of processes and varied data sizes. According to the results shown in Figure 3(b), we can also use the linear regression to model the relationship between the execution time and the data size when the number of processes is fixed. With these two submodels, we can conveniently predict the in-staging processing execution time when the combinations of data size and the number of processes are arbitrary.

As results shown in Figure 3(b), with a fixed number of processes, there is a linear relationship between the visualization execution time and the data size. Other types of data visualization or analysis tasks may contain more complicated patterns, and we can use nonlinear regression to describe the associated relationship between the execution time and the data size. When there are multiple available data samples, we can use the least square approach [40, 41, 42] to find the most appropriate values of parameters in the fitting function.

Figure 5 further illustrates the idea of how to estimate the *in situ* processing execution time with varying data size and number of processes. In particular, the x axis represents the number of processes, the y axis represents the size of the simulated data, and the z axis represents the execution time of *in situ* processing. Assuming there are three known sample points, i.e., $P_{11}(x_1, y_1, z_{11})$, $P_{12}(x_1, y_2, z_{12})$ and $P_{23}(x_2, y_3, z_{23})$, the goal is to develop an algorithm that can estimate the execution time at the point P_{mn} where the number of processes is x_m and the size of data is y_n. If we can predict the key parameters of the model that fit the blue dashed curve shown in Figure 5, then we can compute z_{mn} from the x_m and y_n. The blue solid curve and the blue dashed curve represent two specific cases when there is a fixed data size. According to Figure 3(a), we can observe that the curves with different simulated data sizes are parallel to each other, and we can assume these curves have the same decreasing rate within a specific range. Therefore, we can also assume the blue solid curve and the blue dashed curve in Figure 5 have the same decreasing rate after applying the natural logarithm operator, i.e., the slope value (b) in equation 1. This value can be calculated based on known points P_{13} and P_{23}. We still need another known point on the blue dashed curve to predict its key model parameters. Based on

Algorithm 1: Estimating *in situ* processing execution time with an arbitrary combination of the simulated data size and the number of processes.

Input: $P_{11}(x_1, y_1, z_{11}), P_{12}(x_1, y_2, z_{12}), P_{23}(x_2, y_2, z_{23})$

1 $b_{x1} = (z_{11} - z_{12}))/(y_1 - y_2)$;
2 $a_{x1} = z_{11} - b_{x1} \times y_1$;
3 $z_{13} = a_{x1} + b_{x1} \times y_3$;
4 $b_{y3} = (ln(z_{13}) - ln(z_{23}))/(ln(x_1) - log(x_2))$;
5 $ln(a_{y3}) = ln(z_{13}) - b_{y3} \times ln(x_1)$;
6 $z_{1n} = a_{x1} + b_{x1} \times y_n$;
7 $b_{yn} \approx b_{y3}$;
8 $ln(a_{yn}) = ln(z_{1n}) - b_{yn} \times ln(x_1)$;
9 $ln(z_{mn}) = ln(a_{yn}) + b_{yn} \times ln(x_m)$;
10 return $exp(ln(z_{mn}))$;

two known sample points P_{11} and P_{12}, we can predict the z_{1n} of P_{1n}, which is the intersection between the green solid curve and the blue dashed curve. From the P_{1n} and decreasing rate of the solid blue curve, we can further estimate the parameters of the dashed blue curve and compute the value of z_{mn} associated with P_{mn}.

Algorithm 1 illustrates the detailed procedures to compute the *in situ* processing execution time with an arbitrary combination of the data size and the number of processes. From line 1 to line 2, we compute key parameters associated with the model that fits the green line, representing how the *in situ* processing time changes with a fixed number of processes. From line 3 to line 5, we compute the slope and the intercept of the model shown in equation 1 that fits the blue solid line. From line 6 to line 8, we compute the key parameters of the model that fits the green dashed line based on the P_{1n} and the slope of the blue solid line. We finally return the value of z_{mn} according to the model that fits the blue dashed line.

Based on Algorithm 1, we can compute how many processes need to be added/removed for the rescaling operation with an arbitrary combination of the data size and the number of processes. For example, assuming the current number of processes used for *in situ* processing is x_1, if we predict that the data generated for the next iteration is y_n, and plan to complete the *in situ* processing within z_{mn}, then we can compute a_{yn} and b_{yn} and try to find a x_m such that $z_{mn} = a_{yn} \cdot x^{b_{yn}}$. The estimations of a_{yn} and b_{yn} are discussed in Algorithm 1. After that, the number of processes to add can be described as

$$p = \exp(\frac{\ln z_{mn} - \ln a_{yn}}{b_{yn}}) - x_1 \tag{4}$$

The parameters in Algorithm 1 can be computed in an offline manner before the workflow starts or in an online manner during the workflow execution [43]. For the offline parameter estimation, we execute the data processing with dif-

ferent combinations of the data size and the number of processes. Then we collect input sample data points, namely P_{11}, P_{12}, and P_{23} in Algorithm 1. The benefit of offline parameter prediction is to select the representative input data to improve the accuracy of model prediction. For example, we can use the least square approach [44] to find appropriate parameters based on multiple representative data samples (evaluated in subsection 5.1). In contrast, the benefit of online parameter estimation is to avoid the process of offline parameter computing. However, the online parameter estimation requires at least two rescaling operations to get enough data for parameter estimation. If the model is not ready, the policy can rescale the *in situ* visualization only with a fixed value, such as adding or removing 1 process for each rescaling operation. The accuracy of the model prediction can be influenced if the values in sample data sets are close to each other (shown in subsection 5.1).

3.2 Typical scenarios of computing resource elasticity

For the first scenario, we assume that the total amount of computing resources assigned to the job is fixed during the workflow execution. The elasticity policy can move computing processes between different services as needed and run different programs on these processes. For example, the policy may remove processes from the simulation and add corresponding processes to the data staging service. For the second scenario, the total amount of computing resources assigned to the job can vary during the workflow execution. The data staging service can add/remove processes according to workflow runtime information, such as the simulation computation time, data transfer time, and the size of the simulated data. Finally, we discuss how to integrate elasticity policies into the *in situ* processing for these two scenarios.

3.2.1 Fixed amount of computing resources

This scenario assumes that the total amount of computing resources is fixed, and processes assigned to the simulation or the data staging service can be redistributed dynamically during the workflow execution. To find out if there exists a more efficient scheme of resource configuration, we consider how the number of available processes influences both simulation and in-staging processing time. Specifically, we first assume the simulation computation time T_c is equal to $f_1(P_{sim})$, and the data staging processing time T_p is equal to $f_2(P_{stage})$, where P_{sim} and P_{stage} represent the number of processes used by the simulation and the data staging service, respectively. The basic form of function f is mainly determined by the property of the visualization task. One important factor influencing the function value is the number of available processes. If the elasticity policy switches p processes from the simulation program to the data staging service, we end up with $T'_c = f_1(P_{sim} - p)$ and $T'_p = f_2(P_{stage} + p)$. The elasticity policy needs to find out if there exists a p that satisfies

$$max(T'_c, T'_p) + O < max(T_c, T_p). \tag{5}$$

Algorithm 2: Adaptive strategy for rescaling with both process adding and removing.

```
1 if T_p − T_c > TH_1 then
2 |   p = estimateProcessesNum(join, T_p − T_c);
3 |   doProcessJoin(p);
4 if T_c − T_p > TH_2 then
5 |   p = estimateProcessesNum(leave, T_c − T_p);
6 |   doProcessLeave(p);
```

In this case, the time savings of adding processes to the data staging service are more significant than the overhead of removing processes from the simulation program. If there exist multiple eligible p values, we select one that can minimize the $max(T'_c, T'_p)$. Therefore, removing p processes from the simulation program and then adding them to the data staging service can decrease the in-staging processing time. The elasticity operation may also introduce extra overhead, which is represented by the term O in equation 5. The sources of associated overhead are discussed in detail in Section 3.3.

3.2.2 Varied amount of computing resources

In this scenario, we assume the total amount of computation resources assigned to the data staging service can be adjusted during the workflow execution. Algorithm 2 illustrates when and how to execute the elasticity operation in this case. In particular, when the gap between the simulation computation time (T_c) and in-staging processing time (T_p) is longer than a threshold (TH_1), we add p processes to the data staging service. The model discussed in subsection 3.1 can estimate how many processes need to be added or removed (value of p) to achieve the targeted in-staging processing time with a particular size of data. Similarly, we can remove p processes if the difference between the latest computation time and the in-staging processing time is longer than a threshold (TH_2). In addition, other constraints may also influence the number of processes for rescaling. For example, the number of newly added processes cannot exceed the total number of available processes; the minimum number of computing nodes should also contain enough memory for loading and processing the data. The threshold values used in this scenario are also related to the overhead of the rescaling operation, which is discussed in detail in Section 3.3.

3.2.3 Integrating elasticity policies into *in situ* processing

The main operations of integrating the elasticity policies into the workflow with the simulation computation and *in situ* processing can be described as follows. If the model prediction is executed in an offline manner, we first compute the parameters of the performance model discussed in subsection 3.1 based on sample data points before the workflow starts. At the beginning of each simu-

lation iteration during the workflow execution, we make the elasticity decision and compute how many processes can be added/removed for the simulation and the data staging service depending on the scenarios described in subsection 3.2.1 or subsection 3.2.2. If the policy decides to execute the rescaling operation, we wait for the simulation rescaling to finish and then update the communicator used by the simulation computation based on all alive simulation processes. After the simulation computation, we wait for the ana/vis to finish and record the latest metrics used by model computation[7]. The data redistribution and migration are also important issues to guarantee the correctness of malleable simulation computation [38]; however, these issues depend on specific use cases and are beyond the scope of the present work[8].

Finally, we wait for the data staging rescaling operations to finish and update records for all alive data staging processes. After the simulation process updates the endpoints of the alive data staging processes, we call the data transfer RPC to put data into the data staging service, and then call the execution RPC to trigger the associated ana/vis tasks. Implementation details of the rescaling operation are discussed in Section 3.3.

3.3 Overhead of rescaling operations

Although the rescaling operation aims to decrease the gap between the data generation rate and the consumption rate, it also introduces extra overhead. The overhead may come from both the elasticity mechanism itself and the constraints from applications for rescaling operations, such as initializing the data processing pipeline. Therefore, we need to guarantee that the benefits of rescaling exceed its overhead when executing the rescaling operation. Assuming it takes O seconds to execute one rescaling operation, and the operation can decrease $N \times W$ wait time, where N represents the remaining number of iterations of the *in situ* processing, and W represents the wait time saved by applying the rescaling operation. It is necessary to apply the rescaling operation when there is $O < N \times W$; namely $W > \frac{O}{N}$. Both the increase of the rescaling overhead and the decrease of the remaining number of data processing iterations can decrease the benefit of the rescaling operation. The term $\frac{O}{N}$ formulates the amortized overhead of each resource rescaling operation, and it needs to be considered by the condition of policies for deciding the resource elasticity. The overhead value listed in equation 5 and the threshold value used in Algorithm 2 are both determined by the $\frac{O}{N}$. In our evaluation of these policies (discussed in Section 5.3 and Section 5.4), we test the overhead before the workflow starts and set this value manually when using these policies. Our future work will explore how to compute the overhead value in an adaptive way.

[7] This information is useful if we need to compute the model parameters during the workflow execution in an online manner.

[8] The mini-simulation used in the evaluation of this article requires only updating its communicator to support elasticity.

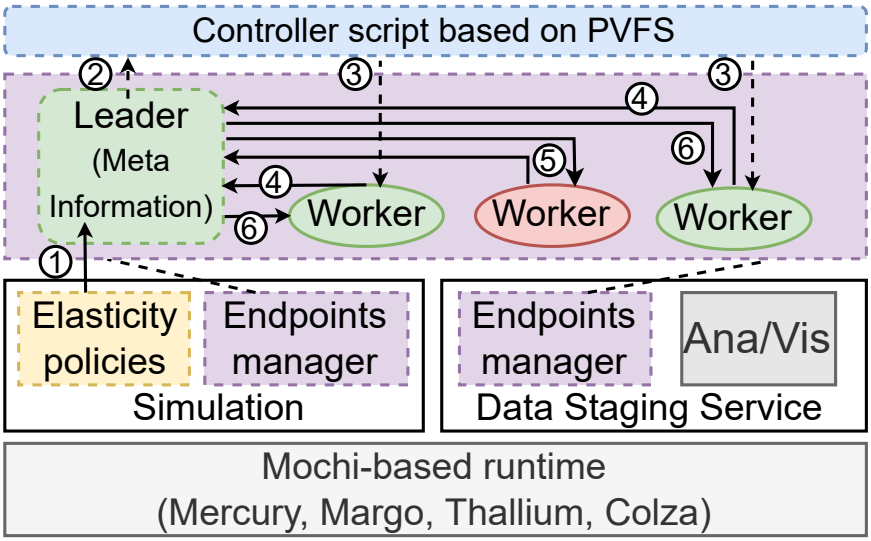


Figure 6: The architecture for implementing the in-staging processing with the elasticity policies. Step 1: The elasticity policy computes the number of processes that need to be added/removed. The policy then updates the meta information in the leader process. Step 2: The leader process writes a file on to the parallel virtual file system (PVFS) as a signal. Step 3: The controller script monitors the signal and starts new processes as workers. Step 4: New started workers register their addresses to the leader process. Step 5: When the elasticity policy decides to remove one process, it updates the meta information and sends the RPC to a dedicated worker (red color); the corresponding worker also sends the RPC to the leader process to remove its address before exiting. Step 6: The leader process sends updates of the address list to all alive worker processes, and every worker process will update its communicator based on the latest address list.

4 Implementation overview

Figure 6 illustrates the architecture of the data staging service with elasticity policies. Major components adopted by the simulation (clients) and the data staging service (servers) are shown in the figure. In particular, the data staging service is built using the *Mochi* suite of HPC libraries [45]: *Margo* and *Thallium* provide remote procedure calls (RPC) service, using *Mercury* for networking and Argobots for user-level thread management. *Colza*[9] can execute customized ana/vis pipeline in a data staging service. We use the Paraview Catalyst [27] to execute distributed visualization tasks based on *Colza* in this work. The user-defined ana/vis can be executed in user-level threads provided by *Argobots* and using the collective communication primitives provided by *Colza*. Although

[9]https://github.com/mochi-hpc/mochi-colza

the implementation is built using *Mochi data services*, the elasticity policies discussed in Section 3 are not limited to the *Mochi data services*; they can be adapted to other data staging services that support the RPC service, and can also recreate the communication group when there are new added/removed processes after the rescaling operation.

The newly added components for elasticity in this work are circled with dashed lines. In particular, the main function of *Elasticity policies* is to collect the workflow run-time information and decide when and how to execute the elasticity primitives based on the policy discussed in Section 3. The *Endpoints manager* records all alive processes and updates the communicator based on the latest endpoint list. Once the policy decides to add/remove the process, it will send RPC to update the expected number of processes recorded by the *Endpoints manager*. A leader-worker strategy is adopted to manage the consistency of alive processes. For example, the added/removed processes send an RPC to the leader process (the process with rank 0) to register/deregister their endpoints. The leader process broadcasts the updated endpoint list to all worker processes, and the worker process will update its communicator to achieve a consistent view for all alive processes. The *Endpoints manager* can be integrated with both the simulation program and the data staging service to support elasticity. Step 4 to Step 6 illustrated in Figure 6 are mainly implemented by *Endpoints manager*.

With the components illustrated in Figure 6, we can construct the elasticity trigger discussed in subsection 2.3 to control when and how to execute the elasticity primitives. In particular, we first need to register the elastic ana/vis tasks to the *Colza* runtime [15]. The elasticity policy is in charge of detecting the workflow runtime information (*trigger detection*) and determining when and how (*trigger decision*) to execute the elasticity primitives (*trigger action*) based on the method discussed in Section 3. Once the policy decides to rescale the data staging service, such as adding a new process, an event is issued. For example, the event can be represented by a dedicated configuration file, and the component in charge of executing the elasticity operation can be implemented by bash code to detect the existence of the file. When the file is detected, it triggers rescaling operations, such as using `srun` to start new processes.

5 Evaluation

The evaluation is divided into four subsections. In particular, subsection 5.1 evaluates the accuracy of the model for predicting the exaction time of visualization tasks. Subsection 5.2 presents the performance of the elasticity primitives for the data staging service and discusses sources of rescaling overhead. In subsection 5.3, we evaluate the efficiency of the elasticity policy when there is a fixed total amount of computing resources during the workflow execution, and the processes can be dynamically switched between the simulation and the data staging service. Furthermore, in subsection 5.4, we evaluate how the elasticity policy works in the scenario in which the total computing resources can change

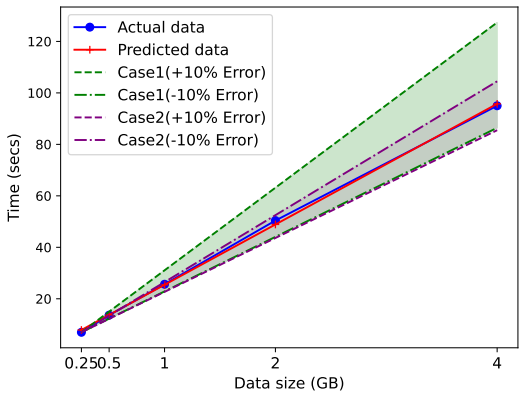


Figure 7: The solid blue line illustrates the actual execution time of the visualization task including all collected data samples. The solid red line represents the predicted execution time of visualization tasks, and it uses all data samples to compute model parameters. Case 1 represents the predicted execution time using data samples where the data sizes are 0.25GB and 0.5GB. Case 2 represents the predicted data computed by the model based on data samples where the data sizes are 0.25GB and 4GB. The "+10%(-10%) Error" means 10% more (less) error than the actual data for the second data sample (0.5GB in Case 1 and 4GB in Case 2) used for the model prediction.

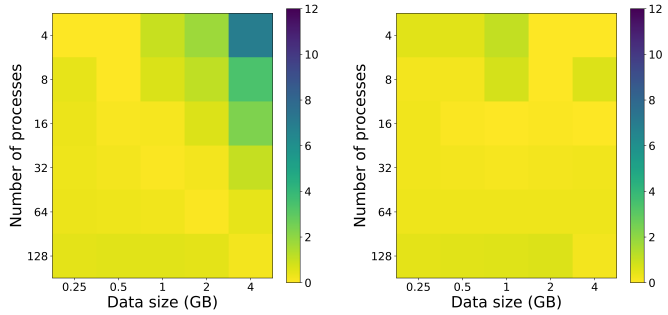
during the workflow execution. The corresponding code[10] is publicly available.

All experiments in this evaluation were performed on the Cori supercomputer at NERSC, a Cray XC40 system with a peak performance of about 30 petaflops. The partition used in this evaluation (Haswell) contains 2,388 nodes, with 128 GB DDR4 2133 MHz memory on each node. Each node is equipped with two sockets, and each socket contains a 2.3 GHz 16-core Intel Xeon Processor E5-2698 v3, which supports two hyper-threads. Cori employs the "Dragonfly" topology for the interconnection network with more than 45 TB/s global peak bisection bandwidth. The type of RDMA used in the Cori system is uGNI [46], which adopts Dynamic RDMA Credentials (DRC) service [47] to transfer data between different programs.

5.1 Accuracy for predicting the visualization execution time

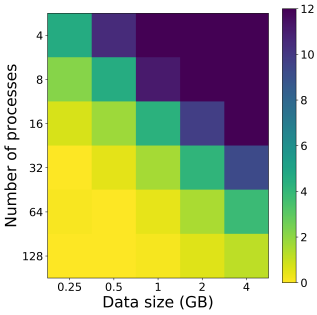
Firstly, we evaluated how the choice of sample data points influences the model estimation error. We use the results shown in Figure 3 as the data source for evaluation. Figure 8 shows the errors between the actual values and estimated values in multiple cases. In particular, when we adopt the data points at the left side of Figure 2, such as Case 1 and Case 2, the errors are trivial (the

[10] `https://git.io/JOUCc`

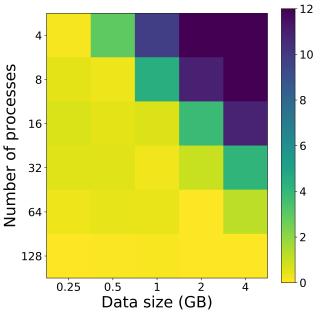


(a) Case 1

(b) Case 2



(c) Case 3



(d) Case 4

Figure 8: The error of the model prediction with different sample data. Case 1, Case 2, Case 3, and Case 4 adopt sample points at the lower left corner, upper left corner, lower right corner, and upper right corner of the data shown in Figure 2, respectively. The error is the difference between the predicted value and actual value, and the unit of the error is second.

light color represents the small error); however, if we adopt the data points at the right side of Figure 2, such as Case 3 and Case 4, the errors become significant, especially when there is a long execution time. The reason for these errors is the execution time does not strictly follow a simple fitting function with a large number of data staging processes. For example, the decrease rate of execution time changes when there are 128 data staging processes in Figure 3(a). In the evaluation for policies (discussed in Section 5.3 and Section 5.4), we start the workflow from a comparatively small data size and the number of processes for the samples in order to avoid the large error.

In actual practice, we also found that when the y values (size of the data) of two sample points are very close to each other, the error and noise of the actual data processing time influence the accuracy of the model prediction. The error of model prediction can be decreased if we use two sample data points that have an obvious distinction in the y axis. It is necessary to use the same number of

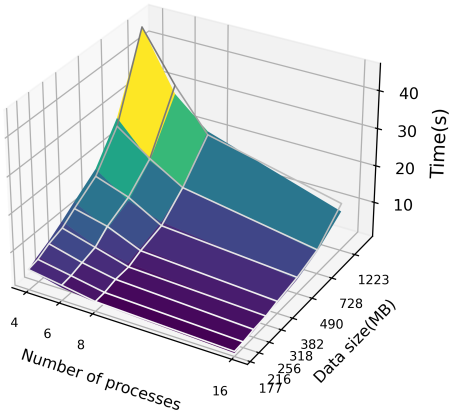


Figure 9: The gray frame illustrates actual visualization execution times, and colored areas present the predicted visualization execution times. We use the data generated by the deep water simulation as the data source [48].

data staging processes to process different sizes of the data before the workflow starts to acquire this prior information. We use the data samples shown in Figure 3(b) as an example to compute the relationship between the visualization task execution time and the data size when there is a fixed number of processes (four processes). As illustrated in Figure 7, Case 1 uses two data samples with a slight difference in the data size (0.25GB and 0.5GB). In contrast, Case 2 uses two data samples with a significant difference in the data size (0.25GB and 4GB). For both Case 1 and Case 2, when we add 10% errors for the second data sample (0.5GB in Case 1 and 4GB in Case 2) used for the model computation, the results of Case 1 show more variations. Therefore, the error of the data sample in Case 1 has a more severe impact for model accuracy than the Case 2.

In addition, we also use data sets generated by the deep water simulation [48] as the input to validate the performance of the model. The size of the data sets increases from 177MB to 1223MB, and we use a different number of processes to visualize these data sets. Figure 5.1 shows the results between the prediction value and the actual value for executing the visualization task with a different number of processes and data sizes. We use the following combination of the number of processes and the data size (the unit is MB) as the input data for computing the model parameters: $(4, 177)$, $(4, 1223)$, and $(6, 256)$. These sample data points contain different regions of the tested data, which show good overall prediction results, and the predicted value can match the actual value with small errors shown in Figure 5.1. Compared with the actual execution time, the percentage of error for predicted execution time is 9.8%, 6.1%, 9.5% , and 12.4% for 4, 6, 8, and 16 processes, respectively.

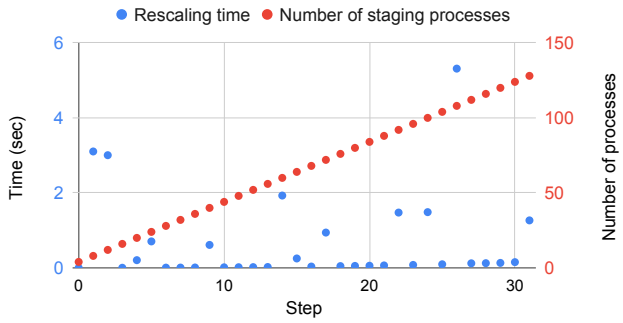


Figure 10: The overhead of adding processes to the data staging service.

5.2 Performance of rescaling the data staging service

In this experiment, we use the Mandelbulb mini-simulation[11] as the data source, and the simulated data blocks are processed by synthetic ana/vis, which is a sleep function that changes the sleep time according to the number of processes based on the power law discussed in subsection 3.1. Figure 10 illustrates details of the rescaling performance. We gradually add 4 data staging processes at each iteration and increase the number of processes from 4 to 128 within 30 steps. For most rescaling operations, synchronizing the data staging service when there is a newly added process can finish within 2 seconds. If we assume the alternative hypothesis is the situation in which the mean time of the rescaling time is less than 1.5 second, the associated p-value of the statistical hypothesis test is 0.0002, and the result is statistically significant.

The rescaling operation can be beneficial to the workflow execution only if the benefit of the rescaling outweighs the overhead. The overhead for starting new processes comes from several aspects in our evaluation: (1) building the model and deciding when or how to trigger rescaling operations; (2) sending the signal to trigger the addition of a new process; (3) updating the communicator in staging service when there are added/removed processes; and (4) initializing the data ana/vis tasks. In our experiment, the overhead of (1) and (3) is trivial (less than 1 second). However, sending a signal and triggering new processes may take a long time (up to 6 seconds in this experiment) when the performance of the parallel file system and batch scheduler is relatively slow. This is because the trigger of `srun` command depends on the detection of the configuration file served as a triggering signal. A heavy workload on the system may increase the delay of the file detection and process triggering. In our evaluation, this overhead[12] varies from several seconds to tens of seconds. The overhead of initializing the ana/vis task mainly depends on different use case scenarios of ana/vis tasks. For example, a distributed visualization task used

[11] `https://github.com/mdorier/MandelbulbCatalystExample`

[12] The time period from the moment of executing the configuration file write operation to the moment that the batch scripts detect the dedicated file.

in section 5.4 may take several seconds to load necessary libraries during each rescaling operation.

5.3 Efficiency of dynamic resource redistribution

In this experiment, we discuss the evaluation results for the elasticity policy presented in subsection 3.2.1. We evaluate the performance of the elasticity policy using a synthetic ana/vis task. In particular, we assume both the simulation program and the staging service are elastic, and the total computation resources are fixed during the workflow execution. We redistribute the computation resources assigned to the simulation and data staging service to find a proper resource configuration dynamically based on the adaptive elasticity policy discussed in subsection 3.2.1. In particular, if the benefit of adding one process to the data staging service outweighs the overhead of removing one process from the simulation program, we can decrease the simulation process and then start the staging service on the process removed from the simulation program. In this way, we may decrease the execution time of the staging-based workflow without adding new computing resources to the job.

We update the mini-simulation used in experiment 5.2 and make it elastic based on the endpoints manager described in Section 4. In particular, we manually add sleep time during the simulation computation to emulate the increase of the simulation computation time. We aim to evaluate how the number of visualization processes changes with the fluctuation of the simulation computation time. During the *in situ* processing, when a particular process leaves the simulation program, the endpoints manager updates its list for all alive processes and recreates the collective communicator based on existing processes to guarantee the correctness of the simulation computation. In addition, a dedicated configuration file is written to the parallel file system when one simulation process leaves. The job script keeps monitoring the file system and starts a new data staging process when the corresponding configuration file is detected.

Figure 11(a) shows different stages of *in situ* processing without using the elasticity strategy. Figure 11(b) and Figure 11(c) show the results of using the static policy and adaptive policy, respectively. The static policy adopted a fixed number of data staging processes for each rescaling service. The adaptive elasticity policy uses the model discussed in subsection 3.1 to estimate the number of data staging services. Both the static and adaptive elasticity policies can decrease the wait time of the simulation program compared with the results shown in Figure 11(a) without using the elasticity mechanism.

The experiment results shown in Figure 11(b) adopt a static elasticity policy to switch the process between the simulation and data staging service. The elasticity policy monitors the wait time of each in-staging processing, if the wait time is larger than a threshold value[13], we remove one simulation process and then start a data staging process. With more computing resources added to

[13] We use the strategy discussed in subsection 3.3 to decide the threshold value. For example, if the estimated overhead is 1 second and there are 10 iterations for *in situ* processing, the threshold value is 0.1.

the data staging service, the static elasticity policy decreases the wait time to zero. In particular, the number of data staging processes increases from 2 to 6 gradually from step 1 to step 5 and decreases the wait time to zero. When there is an increase in the simulation computation time, the data staging service then decreases gradually from 6 to 2.

Compared with the decrease in wait time, the increase of the simulation computation time is insignificant even if we decrease the number of simulation processes. The simulation computation is insensitive to the variation of the number of processes according to the resource configuration used in the experiment. Specifically, the simulation program computes 256 data blocks. When there are 37 processes, 7 blocks are updated by each process at most; when there are 32 processes, 8 blocks are updated by each process at most. This difference is trivial for the simulation computation time.

Compared with the case with a static elasticity policy shown in Figure 11(b), the adaptive elasticity policy illustrated in Figure 11(c) shows a better performance in achieving a proper resource configuration for both the simulation and the data staging service. The elasticity policy decides to remove five processes at the second step and start new staging processes accordingly; it decreases the overhead caused by rescaling operations based on the adaptive elasticity policy. One overhead of adaptive strategy is to compute the parameters adopted by the model for ana/vis tasks before the workflow starts.

5.4 Efficiency of rescaling the data staging service

This experiment evaluates the elasticity policy with the varied resources (discussed in subsection 3.2.2) during the workflow execution. We use a distributed data visualization pipeline to show how the elasticity policy can complete *in situ* visualization in a timely manner.

In particular, we use the deep water asteroid impact simulation [48] as the data source, which aims to study the effects of how the asteroids impact the deep water oceans. Figure 12 illustrates different iterations of the deep water asteroid impact simulation with the same Paraview Catalyst [27]. The results show that the data amount increases gradually with the progression of the simulation computation. This experiment adopts the data generated by the first 25 simulation iterations, and the size of the simulated data increases gradually from 161MB to 1583MB. The execution time of the data visualization can also increase with the rise of the data generated by the simulation program. If we proportionally increase the computation resource used for the data visualization, we may properly overlap the visualization with the simulation computation and decrease the workflow execution time.

This experiment uses a proxy simulation program to load the file generated by the real deep water impact simulation. The proxy simulation program then feeds the data into the elastic data staging service to execute the data visualization pipeline. The details of the deep water impact simulation are beyond the scope of this work, and we use only the data generated by the simulation to evaluate the elastic visualization pipeline in this experiment. The data generated

by the original simulation run on 512 processes, and every process generates one VTU file [28] for each iteration. The proxy simulation is run by the MPI program; it distributes VTU files evenly among every process and then sends them to the associated data staging process. The distributed data visualization pipeline is executed by the data staging service. It replaces the MPI communicator with the *Mochi runtime* that can support the process elasticity [15].

We mainly compare three typical resource elasticity strategies in this experiment. Specifically, the *No elasticity policy* adopts a fixed number of data staging processes during the workflow. The *Static elasticity policy* specifies the rescaling plan before workflow starts, and it adds a fixed number of new processes periodically in this experiment. The *Adaptive elasticity policy* computes the number of added processes based on the policy discussed in subsection 3.2.2. One key step of this policy is to compute how many processes need to be added at the next step. In order to use the model discussed in subsection 3.1, we need to provide the estimated execution time and the estimated data size of the data visualization for the next iteration. The estimated visualization execution time is computed based on the latest wait time of the simulation program. The adaptive elasticity policies aim to decrease the wait time and overlap with the simulation computation as much as possible. The estimated data size is computed using a variable k times the current size of data. For example, when k equals two, the policy will compute how many data staging processes can process the data that is twice as large as the current size of data within the expected execution time of data visualization.

Figure 13(a) shows how the number of the data staging processes changes during the workflow progression, and Figure 13(b) illustrates the associated execution time of data visualization at each step. In particular, the execution time of *No elasticity policy* increases gradually with the increase of the data size because it adopts a fixed computing resource to process a gradually increasing amount of data. The *Static elasticity policy* gradually adds new processes[14]. However, the frequent rescaling operations increase the overhead of rescaling operations (discussed in subsection 5.2). The *Adaptive elasticity policy* decreases the number of rescaling operations, and it tries to estimate the size of the data after several steps and then computes how many processes should be added. The granularity of rescaling depends on the value of variable k. For example, the policy decides to add 20 processes at step 9 when the $k = 3$; in contrast, the policy with $k = 2$ decides to add a comparatively small number of processes gradually. By accumulating the *in situ* processing time spent on every iteration, the adaptive elasticity with $k = 3$ saves around 64% of the execution time compared with the case without an elasticity strategy.

Figure 14 compares the core-hours spent on elasticity strategies. We compute the core-hours by using the number of cores assigned to the program times the execution time of the associated program. Although adaptive elasticity policies adopt more core-hours for the data staging service, they decrease the wait time of the simulation program and save the total core-hours consumed by

[14]The policy decides to add two processes every two iterations in this experiment.

the workflow. For example, compared with the *No elasticity policy*, the *Adaptive elasticity policy* with $k = 3$ saves around 41% of the core-hours for the evaluated workflow.

6 Discussion

The evaluation in this article did not adopt the batch scheduler that supports elasticity because of the limitation of the accessible platform. Instead, we reserve enough nodes in advance but use several of them to show the efficiency for elasticity for the proof of concept. The presented policies need to be integrated with a job scheduler that supports the resource elasticity [17], which can resize the job during its execution. In this way, we can achieve the real benefits of resource elasticity. Furthermore, the signal represented by the configuration file may be influenced by the performance of the parallel file system, which causes the extra overhead of process triggering. A more efficient trigger procedure should be explored in a future study.

In the evaluation of subsection 5.3, we use the synthetic simulation that does not require a data distribution during the rescaling operation. One research opportunity is to try to adapt the existing simulation to support the elastic operation. Furthermore, the data staging service used in this article for the proof of concept is only minimally capable of managing in-staging data. It is interesting to integrate the elasticity policy and elastic ana/vis with the state-of-the-art data staging services such as DataSpaces [6] and Damaris [49] to compare their efficiency for elastic *in situ* processing.

For the adaptive elasticity policy discussed in this article, we assume both the execution time of in-staging processing and the simulation computation time vary gradually and follow the model discussed in subsection 3.1. However, it is possible that in-staging processing is more complicated, such as the single process with unbalanced execution time shown in subsection 5.4. The model estimation of execution time can also be updated to adapt to various data ana/vis tasks. This article mainly discusses the performance model and the resource elasticity policy for data rendering tasks. Future research needs to extend the current model to support other visualization tasks, such as streamline or iso-contour tasks. Moreover, if there is workflow runtime information, such as the performance of the `srun` operation, the adaptive policy can make a more accurate decision about whether resource rescaling is necessary.

For the evaluated cases with a fixed data size, such as the results shown in Figure 11, the static and adaptive elasticity policies show a similar performance. The adaptive elasticity policies need at least three iterations to collect sample data points and estimate key parameters of the model; within these several iterations, the static method can also achieve a proper configuration. For the case with a variation of both data size and process number, such as the results shown in Figure 13, the adaptive elasticity shows better performance by estimating the future simulated data size.

Our experiments evaluate only the scenario when the simulation and data

processing runs by the same job. The situation in which the simulation and program run at different jobs and transfer the data by file-system are beyond the scope of this study. In this case, even if the simulation does not need extra wait time since it dumps data to the file system directly, the elasticity can also help to find a sweet point between computing resource utilization and time constraints of data processing.

Although the presented adaptive elasticity policies try to decide the elasticity operation in an online manner, prior information is still needed to improve the accuracy of the elasticity policies. In particular, the threshold value discussed in subsection 3.2.2 is influenced by the overhead of the rescaling operation, which is collected according to the test run before the actual workflow execution. The properly chosen sample data points for model prediction are more accurate than using all the data collected during the workflow execution for model prediction. For example, if the P_{11} and P_{12} discussed in subsection 3.1 have the closed y values, the error of sample data can decrease the accuracy of model prediction.

Research works	Type	Platform	Goal	Main approach
Tong et al. [9, 24]	Policy (RT)	HPC	Minimizing the *in situ* processing time	Adding a fixed number o for each elasticity operat
Gari et al. [22]	Policy (RT)	Cloud	Minimizing the cost and execution time for *post hoc* scientific workflows	Using reinforcement lear the model to guide the e
Monge et al. [50]	Policy (RT)	Cloud	Minimizing the cost and the execution time of simulation	Using evolutive algorithm the multiobject optimiza
Duan et al. [51]	Policy (RT)	HPC	Recovering from node failure for data staging service	Adding new nodes into t when there is a node fail
Shu et al. [52]	Policy (NRT)	HPC	Minimizing the *in situ* processing time	Selecting proper workflo based on the pretrained
Kress et al. [12]	Policy (NRT)	HPC	Decreasing the cost of *in situ* processing	Using cost model to deci the resource configuratio
Dorier et al. [15]	Framework	HPC	Supporting the elasticity for *in situ* visualizations	Using an elastic commun online rescale for scientif
Fox et al. [16]	Framework	HPC	Supporting the elasticity for *post-hoc* scientific workflows	Using checkpoint-restart a new job to implement
Chadha et al. [20]	Framework	HPC	Extending SLURM to support the elasticity	Using elastic MPI to sup

Table 1: Comparison between different aspects of related works for deciding resources used for scientific workflows. The "RT" and "NRT" represents the "real-time" and "non-real-time", respectively.

7 Related Work

This section discusses related works in detail and how our work differs from theirs. In particular, we use Table 7 to illustrate different aspects of related works for deciding the computing resources used for scientific workflows. In particular, each work is viewed from aspects of the type (Policy or Framework), the platform (HPC or Cloud), the goal, and the approach.

The work most closely related to our study is the research by Tong et al. [9]. They present a resource adaptation policy to rescale the computation resources used for data staging service and to improve resource utilization efficiency on the HPC platform. They estimate the minimum number of data staging processes that contain enough memory resources and then add more computation resources to make the in-staging processing overlap with the simulation computation. A follow-up paper by Tong et al. [24] extends the rescaling operation and manages it by autonomic computing. However, they do not explain how to properly determine the number of processes to join/leave the computation group and the overhead of the elasticity operation. Once the condition of triggering the elasticity operation is satisfied, they add a fixed number of resources into the data staging service. In addition, the decision policy in their work did not discuss how the overhead of the rescaling operation influences the efficiency of the resource rescaling operations.

Elasticity is also an important aspect for the cloud computing. Ghanbari et al. [53] summarize typical approaches to support elasticity in the context of cloud computing. Zahedi et al. [54] discuss how to use the Amdahl utility function to decide the process allocation. Our work focuses on the elasticity policy from the application perspective in the context of the *in situ* ana/vis tasks for scientific workflows on an HPC platform.

Shashidharan et al. [55] present a framework that can run the geo-simulation on elastic resources at runtime. Monge et al. [50] and Yannibelli et al. [56] formulate the rescaling policy on the cloud platform as a multiobjective optimization problem. Evolutive algorithms are adopted by the autoscaler to minimize the makespan, monetary cost, and probability of failures of simulation execution. Our work focuses on the elasticity of *in situ* visualizations, and discusses how to overlap the visualization execution with the simulation computation by resource rescaling on HPC platforms with homogeneous computing nodes.

In the context of fault-tolerance for *in situ* processing, Duan et al. [51] use elasticity as a mechanism to support data failure detection and recovery for data staging processes. The User Level Failure Mitigation (ULFM) [57] presents MPI extensions to detect communicator failure, along with solutions for recovery from the failure. The trigger of the elasticity primitives depends on the error detection mechanism for these works; however, our work mainly focuses on rescaling data staging resources to achieve more efficient resource utilization.

Shu et al. [52] use the machine learning method and auto-tuner to build a surrogate model for finding the proper resource configuration for the *in situ* workflow. However, their approach needs to run offline to train the perfor-

mance model and find configurations before the workflow starts. Our method is more lightweight and is able to find a proper resource configuration during the workflow execution and adjust the configurations in an online manner.

Kress et al. [12] present cost models for *in situ* processing, and they evaluate the tightly-coupled and loosely-coupled *in situ* processing for multiple visualization tasks. The results of their study show the possibility of staging-based *in situ* processing being cost-effective over tightly-coupled *in situ* processing. With this work as one motivation, our work further explores how to rescale the resources used for staging-based *in situ* processing and improve its efficiency.

Dorier et al. [15] present a data staging framework that supports the elastic *in situ* visualization task. They show how to update the widely used Paraview Catalyst *in situ* visualization framework to execute the visualization in an elastic manner. However, their work mainly focuses on the design consideration of adapting infrastructure that executes the visualization in an elastic way, and they do not discuss the elasticity policies about when and how to trigger these elasticity operations.

Fox et al. [16] and Chadha et al. [20] focus on how to provide elasticity from the job scheduler's perspective. They discuss the mechanism to implement the elasticity primitives such as resource addition or removal on the HPC platform. Our work focuses on the policy to trigger these primitives, which is complementary to their works.

8 Conclusions and Future Work

In this article, we explored an approach that uses elasticity management to optimize the scientific workflow with *in situ* visualization on the HPC platform in real time. The goal of the elasticity policy is to decrease the gap between the data generation and data producing rate to improve computing resource utilization. The presented elasticity policies utilize the workflow runtime information to decide when to apply the elasticity operation, and how many rescaled processes for the simulation and the data staging process are required during the workflow execution. This paper makes the following contributions:

- Modeling the execution time of the *in situ* visualization task and showing how it is influenced by data size and the number of processes. This simple and efficient model provides the foundation for elasticity policy to decide the number of rescaled processes.
- Presenting elasticity policies that decide when and how to trigger resource rescaling operations under different scenarios.
- Integrating presented elasticity policies with the *in situ* simulation-visualization workflow based on Mochi data management services and VTK.
- Evaluating *in situ* processing with the elasticity policy on the Cori supercomputer. The evaluation results show that the adaptive elasticity policy

can efficiently find a proper resource configuration, decrease the overhead caused by rescaling operations, and improve computing resource utilization efficiency.

Our future work includes (1) improving the accuracy of the performance model and extending it to support more types of *in situ* visualization tasks and other processing tasks in addition to visualization, and (2) integrating presented elasticity policies into full-fledged *in situ* processing that contains an elastic simulation, ana/vis pipelines, and an elastic batch scheduler.

Acknowledgements

The research presented in this work is based upon work by the RAPIDS2 Institute supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the Advanced Computing (SciDAC) program under Award Number DE-SC0023130. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was also supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract No.DE-AC02-06CH11357.

References

[1] M. Hirsch, C. Mateos, A. Zunino, Augmenting computing capabilities at the edge by jointly exploiting mobile devices: A survey, Future Generation Computer Systems 88 (2018) 644–662.

[2] T. Peterka, D. Bard, J. C. Bennett, E. W. Bethel, R. A. Oldfield, L. Pouchard, C. Sweeney, M. Wolf, Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources, The International Journal of High Performance Computing Applications 1094342020913628.

[3] J. Xu, L. Chen, S. Ren, Online learning for offloading and autoscaling in energy harvesting mobile edge computing, IEEE Transactions on Cognitive Communications and Networking 3 (3) (2017) 361–373.

[4] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier, et al., A terminology for in situ visualization and analysis systems, The International Journal of High Performance Computing Applications (2020) 1094342020935991.

[5] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, F. Zheng, Datastager: scalable data staging services for petascale applications, Cluster Computing 13 (3) (2010) 277–290.

[6] C. Docan, M. Parashar, S. Klasky, Dataspaces: an interaction and coordination framework for coupled simulation workflows, Cluster Computing 15 (2) (2012) 163–181.

[7] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, M. Parashar, Stacker: an autonomic data movement engine for extreme-scale data staging-based in-situ workflows, in: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2018, pp. 920–930.

[8] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, L. Orf, Damaris: Addressing performance variability in data management for post-petascale simulations, ACM Transactions on Parallel Computing (TOPC) 3 (3) (2016) 15.

[9] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, H. Abbasi, Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows, in: SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013, pp. 1–12.

[10] L. Orf, A violently tornadic supercell thunderstorm simulation spanning a quarter-trillion grid volumes: Computational challenges, i/o framework, and visualizations of tornadogenesis, Atmosphere 10 (10).

[11] P. Malakar, V. Vishwanath, C. Knight, T. Munson, M. E. Papka, Optimal execution of co-analysis for large-scale molecular dynamics simulations, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 702–715.

[12] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, D. Pugmire, Opportunities for cost savings with a in-transit visualization, in: P. Sadayappan, B. L. Chamberlain, G. Juckeland, H. Ltaief (Eds.), High Performance Computing, Springer International Publishing, Cham, 2020, pp. 146–165.

[13] M. Dorier, O. Yildiz, T. Peterka, R. Ross, The challenges of elastic in situ analysis and visualization, in: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 23–28.

[14] T. Peterka, D. Bard, J. Bennett, E. Bethel, R. Oldfield, L. Pouchard, C. Sweeney, M. Wolf, ASCR workshop on in situ data management: Enabling scientific discovery from diverse data sources, Tech. rep., USDOE Office of Science (SC)(United States) (2019).

[15] M. Dorier, Z. Wang, U. Ayachit, S. Snyder, R. Ross, M. Parashar, Colza: Enabling elastic in situ visualization for high-performance computing simulations, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2022, pp. 538–548.

[16] W. Fox, D. Ghoshal, A. Souza, G. P. Rodrigo, L. Ramakrishnan, E-hpc: a library for elastic resource management in hpc environments, in: Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science, 2017, pp. 1–11.

[17] Ibm knowledge center - ibm spectrum lsf, `https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=administration-resizable-jobs`.

[18] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, P. Merle, Elasticity in cloud computing: state of the art and research challenges, IEEE Transactions on Services Computing 11 (2) (2017) 430–447.

[19] M. A. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. Cunha, R. Buyya, Hpc cloud for scientific and business applications: taxonomy, vision, and research challenges, ACM Computing Surveys (CSUR) 51 (1) (2018) 1–29.

[20] M. Chadha, J. John, M. Gerndt, Extending slurm for dynamic resource-aware adaptive batch scheduling, in: 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC), IEEE, 2020, pp. 223–232.

[21] N. Roy, A. Dubey, A. Gokhale, Efficient autoscaling in the cloud using predictive models for workload forecasting, in: 2011 IEEE 4th International Conference on Cloud Computing, IEEE, 2011, pp. 500–507.

[22] Y. Gari, D. A. Monge, C. Mateos, A q-learning approach for the autoscaling of scientific workflows in the cloud, Future Generation Computer Systems 127 (2022) 168–180.

[23] Z. Wang, M. Dorier, P. Subedi, P. E. Davis, M. Parashar, An adaptive elasticity policy for staging based in-situ processing, in: 2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS), IEEE, 2021, pp. 33–41.

[24] T. Jin, F. Zhang, Q. Sun, M. Romanus, H. Bui, M. Parashar, Towards autonomic data management for staging-based coupled scientific workflows, Journal of Parallel and Distributed Computing 146 (2020) 35–51.

[25] K. Moreland, B. Wylie, C. Pavlakos, Sort-last parallel rendering for viewing extremely large data sets on tile displays, in: Proceedings IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics (Cat. No. 01EX520), IEEE, 2001, pp. 85–154.

[26] W. J. Schroeder, L. S. Avila, W. Hoffman, Visualizing with vtk: a tutorial, IEEE Computer graphics and applications 20 (5) (2000) 20–27.

[27] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, J. Mauldin, Paraview catalyst: Enabling in situ data analysis and visualization, in: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, 2015, pp. 25–29.

[28] W. Schroeder, K. Martin, B. Lorensen, Vtk textbook (2006).

[29] R. Binyahib, D. Pugmire, A. Yenpure, H. Childs, Parallel particle advection bake-off for scientific visualization workloads, in: 2020 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2020, pp. 381–391.

[30] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, M. Day, In situ and in-transit analysis of cosmological simulations, Computational Astrophysics and Cosmology 3 (1) (2016) 1–18.

[31] M. Parashar, S. Hariri, Autonomic computing: An overview, in: J.-P. Banâtre, P. Fradet, J.-L. Giavitto, O. Michel (Eds.), Unconventional Programming Paradigms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 257–269.

[32] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50.

[33] M. Salloum, J. C. Bennett, A. Pinar, A. Bhagatwala, J. H. Chen, Enabling adaptive scientific workflows via trigger detection, in: Proceedings of the first workshop on in situ infrastructures for enabling extreme-scale analysis and visualization, 2015, pp. 41–45.

[34] J. C. Bennett, A. Bhagatwala, J. H. Chen, A. Pinar, M. Salloum, C. Seshadhri, Trigger detection for adaptive scientific workflows using percentile sampling, SIAM Journal on Scientific Computing 38 (5) (2016) S240–S263.

[35] M. Larsen, A. Woods, N. Marsaglia, A. Biswas, S. Dutta, C. Harrison, H. Childs, A flexible system for in situ triggers, in: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV '18, ACM, New York, NY, USA, 2018, pp. 1–6.

[36] A. Raveendran, T. Bicer, G. Agrawal, A framework for elastic execution of existing mpi programs, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, IEEE, 2011, pp. 940–947.

[37] I. Comprés, A. Mo-Hellenbrand, M. Gerndt, H.-J. Bungartz, Infrastructure and api extensions for elastic execution of mpi applications, in: Proceedings of the 23rd European MPI Users' Group Meeting, 2016, pp. 82–97.

[38] A. Mo-Hellenbrand, I. Comprés, O. Meister, H.-J. Bungartz, M. Gerndt, M. Bader, A large-scale malleable tsunami simulation realized on an elastic mpi infrastructure, in: Proceedings of the Computing Frontiers Conference, 2017, pp. 271–274.

[39] A. Clauset, C. R. Shalizi, M. E. Newman, Power-law distributions in empirical data, SIAM review 51 (4) (2009) 661–703.

[40] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, SciPy 1.0 Contributors, SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python, Nature Methods 17 (2020) 261–272.

[41] D. C.-L. Fong, M. Saunders, Lsmr: An iterative algorithm for sparse least-squares problems, SIAM Journal on Scientific Computing 33 (5) (2011) 2950–2971.

[42] V. Bruder, M. Larsen, T. Ertl, H. Childs, S. Frey, A hybrid in situ approach for cost efficient image database generation, IEEE Transactions on Visualization and Computer Graphics (2022) 1–1.

[43] How online parameter estimation differs from offline estimation, `https://www.mathworks.com/help/ident/ug/how-online-estimation-differs-from-offline-estimation.html`.

[44] M. Newville, T. Stensitzki, D. B. Allen, M. Rawlik, A. Ingargiola, A. Nelson, Lmfit: Non-linear least-square minimization and curve-fitting for python, Astrophysics Source Code Library (2016) ascl–1606.

[45] R. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. Gutierrez, R. Latham, B. Robey, D. Robinson, B. Settlemyer, G. Shipman, S. Snyder, J. Soumagne, Z. Qing, Mochi: Composing data services for high-performance computing environments, Journal of Computer Science and Technology 35 (1) (2020) 121 – 144, 10.1007/s11390-020-9802-0.

[46] B. Alverson, E. Froese, L. Kaplan, D. Roweth, Cray xc series network.

[47] J. Shimek, J. Swaro, M. Saint Paul, Dynamic rdma credentials.

[48] R. Imahorn, I. B. Rojo, T. Günther, Visualization and analysis of deep water asteroid impacts, in: 2018 IEEE Scientific Visualization Conference (SciVis), IEEE, 2018, pp. 85–96.

[49] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, D. Semeraro, Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework, in: 2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV), IEEE, 2013, pp. 67–75.

[50] D. A. Monge, E. Pacini, C. Mateos, C. G. Garino, Meta-heuristic based autoscaling of cloud-based parameter sweep experiments with unreliable virtual machines instances, Computers & Electrical Engineering 69 (2018) 364–377.

[51] S. Duan, P. Subedi, P. Davis, K. Teranishi, H. Kolla, M. Gamell, M. Parashar, CoREC: Scalable and resilient in-memory data staging for in-situ workflows, ACM Transactions on Parallel Computing (TOPC) 7 (2) (2020) 1–29.

[52] T. Shu, Y. Guo, J. Wozniak, X. Ding, I. Foster, T. Kurc, Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–15.

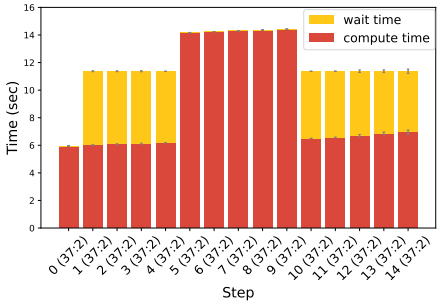
[53] H. Ghanbari, B. Simmons, M. Litoiu, G. Iszlai, Exploring alternative approaches to implement an elasticity policy, in: 2011 IEEE 4th International Conference on Cloud Computing, 2011, pp. 716–723.

[54] S. M. Zahedi, Q. Llull, B. C. Lee, Amdahl's law in the datacenter era: A market for fair processor allocation, in: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2018, pp. 1–14.

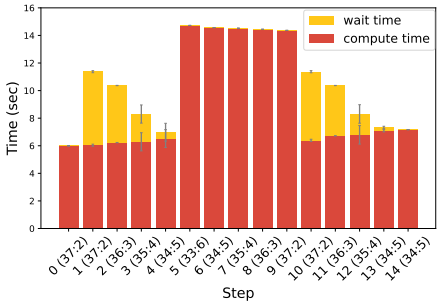
[55] A. Shashidharan, R. R. Vatsavai, R. K. Meentemeyer, Futures-dpe: towards dynamic provisioning and execution of geosimulations in hpc environments, in: Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, 2018, pp. 464–467.

[56] V. Yannibelli, E. Pacini, D. Monge, C. Mateos, G. Rodriguez, A comparative analysis of nsga-ii and nsga-iii for autoscaling parameter sweep experiments in the cloud, Scientific Programming 2020.

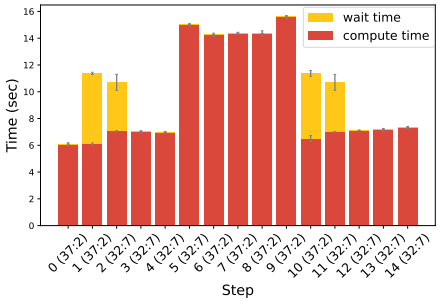
[57] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, J. Dongarra, Post-failure recovery of mpi communication capability: Design and rationale, The International Journal of High Performance Computing Applications 27 (3) (2013) 244–254.



(a) No elasticity

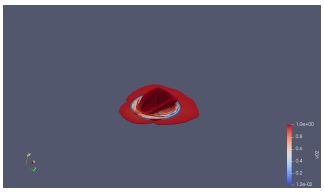


(b) Static elasticity policy

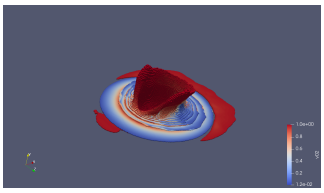


(c) Adaptive elasticity policy

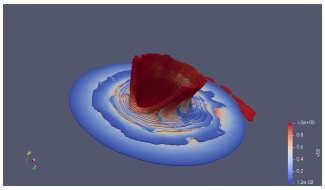
Figure 11: The experiment results of using different elasticity strategies to move processes between simulation program and the data staging service. The number in the parenthesis of the horizontal label represents the number of the simulation processes and the number of the staging processes, respectively. The error bar represents the standard deviation value for three runs.



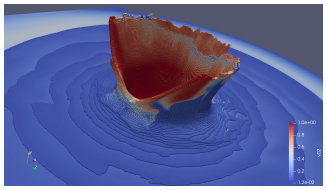
(a) Iteration 10



(b) Iteration 15

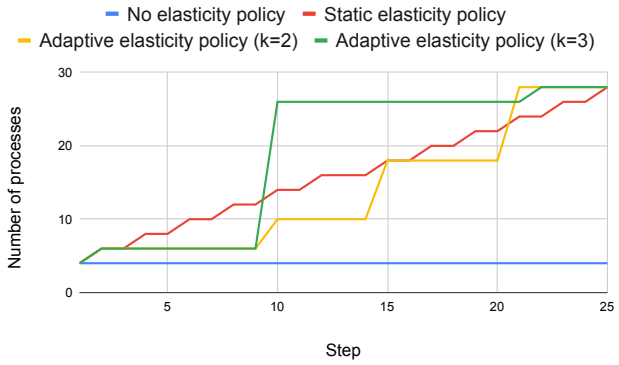


(c) Iteration 20

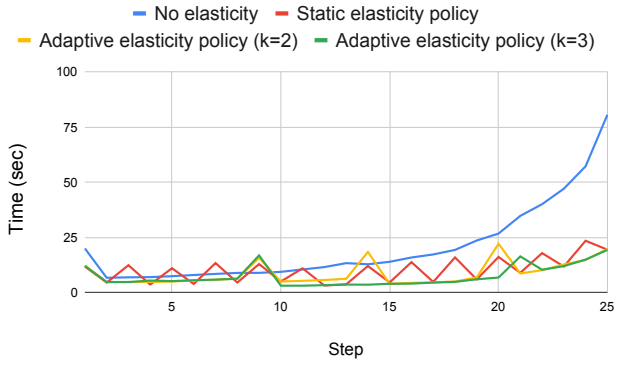


(d) Iteration 25

Figure 12: Visualization of the deep water asteroid impact simulation.



(a) Number of data staging processes.



(b) Execution time of distributed visualization.

Figure 13: Subfigure(a) illustrates how the number of processes changes in different steps. Subfigure(b) shows the *in situ* visualization execution time at each step with different resource elasticity strategies.

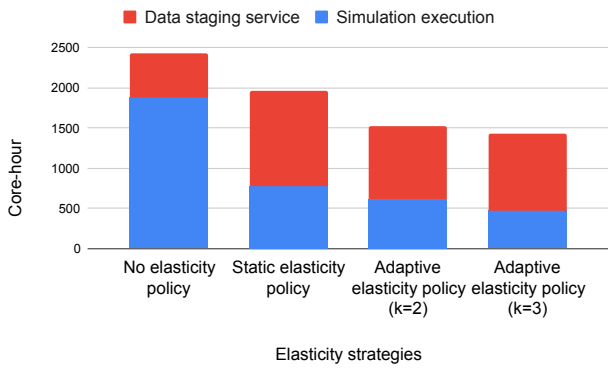


Figure 14: Accumulated core-hours for different elasticity strategies.