An Adaptive Elasticity Policy For Staging Based In-Situ Processing

Zhe Wang,* Matthieu Dorier,† Pradeep Subedi,‡ Philip E. Davis,‡ Manish Parashar‡
*Rutgers University, jay.wang@rutgers.edu
†Argonne National Laboratory, mdorier@anl.gov
‡University of Utah, pradeep.subedi@utah.edu, philip.davis@sci.utah.edu, manish.parashar@utah.edu

2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS) | 978-1-6654-1136-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/WORKS54523.2021.00010

***Abstract*—In-situ processing alleviates the gap between computation and I/O capabilities by performing data analysis close to the data source. With simulation data varying in size and content during workflow execution, it becomes necessary for in-situ processing to support resource elasticity, i.e., 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 towards the end of the job when interesting data appears. 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 paper, we present an adaptive elasticity policy that adopts workflow runtime information collected online to predict how to trigger the addition and removal of processes in order to minimize in-situ processing overheads. We integrate the presented elasticity policy into a staging-based elastic workflow and evaluate its efficiency in multiple elasticity scenarios. The results indicate that an adaptive elasticity policy can save overhead in finding a proper resource configuration, when compared with a static policy that uses a fixed number of processes for each rescaling operation. Finally, we discuss multiple existing research opportunities of elastic in-situ processing from different aspects.**

***Index Terms*—In Situ Processing, Data Staging, Elasticity, Policy**

I. Introduction

In-situ processing addresses the gap between computation and I/O capabilities by processing data as it is generated on the same system. *Tightly coupled* and *loosely coupled* in-situ are two major models for in situ processing [7]. In a tightly coupled system, the simulation program is linked with an in-situ library. When the API for in-situ processing is called, the in-situ library transforms the data layout as needed and executes data analysis or visualization tasks (abbreviated as ana/vis), using the same computing resources as the simulation. 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, the data generated by the simulation is transferred to other remote nodes via a high-speed network [2], [10], [11] or to other processes on the same nodes via shared memory [28]. The memory used to store the transferred simulation data, whether local to the simulation nodes or remote, is sometimes called a *data staging area*. A *Staging-based scientific workflow* adopts the loosely coupled model to process the ana/vis, and we use *in-staging execution* to represent the process of executing ana/vis tasks in the data staging area.

With HPC simulations moving towards more complex workflows, in-situ workflows need to process simulation data that changes over time in a flexible way. For example, S3D simulation [6] data contains more details near flame fronts. When performing topological analysis, more computing resources are required to process the flame front region compared with other regions. Similarly, interesting phenomenons of the CM1 simulation [23], such as tornadoes, appear at the end of the simulation. We need more computing resources to do the analysis once these interesting data appear in order to finish the data processing within specific time constraints.

In-situ workflows may adopt multiple solutions to process varying simulation data and associated ana/vis discussed above. For example, the processing frequency of simulated data can be adjusted during workflow execution [19], and the number of ana/vis tasks can also be adjusted speculatively to match the data production rate and available resources [18]. In this work, we focus on solutions that adopt dynamic resource elasticity, which is identified as a key research challenges for processing dynamic workloads [13], [24]. Computing resource elasticity consists of 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. The term *expand/-contract* or *grow/shrink* is also used in related works [1], [15] with a similar meaning.

Checkpoint/restart is a classical mechanism with which to implement elasticity. It consists of stopping the current job and starting a new job with an adjusted number of nodes or processes [15], [25]. The restart operation, however, adds overhead to the workflow execution, and data redistribution can be difficult. With the support of dynamic process in the MPI standard 2.0 and associated extensions [9] or MPI-like communication libraries[1], resource elasticity can be achieved in an online manner without restarting the job and application. With the support of elasticity in the communication layer, the batch scheduler can dynamically add or remove nodes during the job execution [1], [5]. In addition, these communication

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

978-1-6654-1136-3/21/$31.00 ©2021 IEEE
DOI 10.1109/WORKS54523.2021.00010

libraries also facilitate the simulation program [20] and associated in-situ processing [13] for elasticity.

With an infrastructure supporting elasticity primitives, one research question to answer is *when and how to trigger these primitives for elasticity*. Depending on whether we know the requirements for computation resources before the workflow execution, the elastic workflow execution policy can be divided into two patterns [15]. In a *plan-driven* method, the user provides an elastic plan before workflow execution. For example, we can specify rules such that the job will be rescaled to a particular resource configuration after a specific run time or at a specific iteration. In an *event-driven* method, the trigger of elasticity operations depends on the detection of specific events, such as changes in key metrics during the workflow execution. The elasticity policy is in charge of issuing these triggering events. Once the trigger for elasticity is determined, a question remains: *how many processes/nodes should be added or removed during the rescaling operation*.

In this work, we present an adaptive elasticity policy to control the triggering of elasticity operations, such as adding or removing processes. The presented adaptive elasticity policy utilizes workflow runtime information to predict how in-staging execution time changes with the number of running processes and decides how many processes should be added or removed accordingly. We integrate the presented policy into a staging based elastic workflow and evaluate its efficiency in multiple elasticity scenarios. Compared with the static elastic policy, the results show that the adaptive elasticity policy takes less overhead to find a proper resource configuration that is favorable for in-situ processing. At last, we discuss its limitation and future research opportunities for implementing a full-fledged elastic in-situ workflow.

The rest of the paper is organized as follows. Section II discusses the background and motivation. In Section III, we elaborate on the details of the adaptive elasticity policy. In Section IV, we further discuss the implementation details of integrating the adaptive elasticity policy into the in-staging processing. The evaluation results are discussed in Section V, then we present existing research opportunities in Section VI. We discuss related work in Section VII, and conclude the paper in Section VIII.

II. Background and Motivation

In this section, we first introduce the workflow with elastic in-staging processing, then we discuss potential solutions to predict how available computing resources influence the in-staging execution time.

A. Workflow with elastic in-staging processing

Figure 1 illustrates key stages of a workflow with elastic in-staging execution [17]. We assume the data staging service supports resource elasticity, and the elasticity primitives can be triggered at any iteration of the simulation computation. The *Rescale and sync* stage of the simulation program checks if there are newly added or removed data staging processes and confirms that the current simulation process knows which

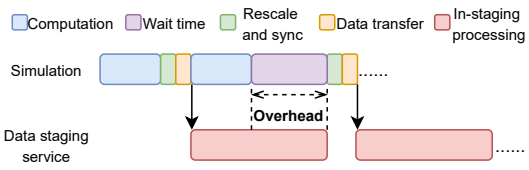


Fig. 1. In-staging processing workflow with resource elasticity.

processes of the data staging service are alive. The number of data staging processes for executing ana/vis remains constant during the processing of a single analysis task. If in-staging execution takes too long to complete, the simulation needs to wait until it is finished before the next *Rescale and sync* stage. The extra wait time increases the overhead of in-situ processing. If there is a way to adjust the in-staging execution time by dynamically using more computing resources, we may avoid this overhead. On the other hand, if the data staging service finishes faster than the simulation computation, we can remove some of the processes used for data staging service and give them back to the scheduler (if the resource scheduler supports elasticity) without introducing extra wait time. Furthermore, if the simulation program also supports elasticity, we may redistribute the computing resources used by the simulation and data staging service dynamically to decrease the workflow execution time.

B. Distribution of in-staging execution time

Although we can decide when to trigger the rescaling operation by monitoring the wait time discussed in subsection II-A, it is still hard to properly predict how many processes should be added or removed for each rescaling operation. If there is a way to predict how the in-staging execution time will change with the number of data staging processes, we can compute how many processes should be added or removed in a more accurate manner.

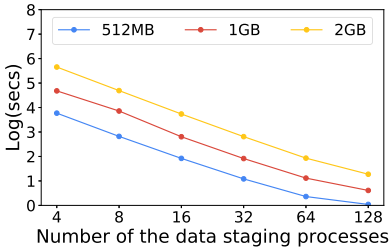


Fig. 2. An example that shows how the in-staging data processing time changes with the number of data staging processes. The log scale with base 2 is used for both x and y axis.

Figure 2 illustrates an example of how the in-staging execution time changes when varying the number of data staging processes. We use the Gray-Scott mini-application[2] as the data source and visualize its data in the staging area using Paraview Catalyst [3]. We construct multiple combinations of simulation data size and number of data staging processes; then, we execute the associated visualization pipeline and record the

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

execution time. From the results, in-staging execution time is inversely proportional to the number of processes. The speedup of the execution time keeps comparatively constant when we double the number of data staging processes. A similar distribution can be observed in other parallel computations [21].

As the results illustrated in Figure 2, we need to gradually add more processes to keep execution time decrease at the same rate. For example, when 2GB of data is generated by each simulation iteration, and there are 4 processes, adding 4 more processes can reduce in-staging execution time by approximately 1 second in the log scale. However, when there are 64 processes, adding 4 more processes is trivial for decreasing the in-staging execution time, and it needs up to 64 more processes to reduce execution time by around 1 second. This fact shows that adding a particular number of processes achieves different benefits depending on existing available data staging processes. In order to design a proper elasticity policy to trigger the elasticity primitives such as process joining or leaving, we need to collect workflow run-time information and use this information to predict how the execution time changes with a variation in the number of processes. Then we can confirm how many processes need to be added/removed in order to achieve the targeted execution time. The details of designing elasticity policies are discussed in Section III.

III. Methods

In this section, we first describe procedures that control when to trigger the elasticity operations with both static and adaptive strategy in subsection III-A. Then we discuss designs of adaptive elasticity policy for two typical scenarios. For the first scenario discussed in subsection III-B, the number of processes assigned to the job can vary during the workflow execution. The data staging service can add/remove processes based on workflow runtime information. For the second scenario discussed in subsection III-C, the total computing resources assigned to the job are fixed before the workflow execution, and we can switch the process between different programs during the workflow execution. For example, we may remove some processes from the simulation and add corresponding processes to execute the data staging service to optimize resource utilization.

A. Static vs Adaptive elasticity strategy

The main goal of the elasticity policy is to decide when and how to trigger elasticity primitives such as adding or removing processes. 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) data staging or simulation programs. Assuming the trigger condition is specified, one critical step of the aforementioned elasticity policy is to estimate the value of k, which is the number of processes added or removed during the rescaling operation. When the *Static* strategy is adopted, we set key parameters manually and add or remove a fixed number of processes, such as 1. With the *Adaptive* strategy, we first check the status

Algorithm 1: Adaptive strategy for rescaling with both process adding and removing

```
if T_w > TH_1 then
    k = estimateProcessesNum(join, T_p − T_c);
    doProcessJoin(k);
if T_c − T_p > TH_2 then
    k = estimateProcessesNum(leave, T_c − T_p);
    doProcessLeave(k);
```

of the model that expresses *the relationship between the in-staging ana/vis execution (or simulation computation) time and the number of processes* (discussed in subsection III-B). If the parameters used by the model are not initialized, we can predict key parameters based on historical data in an online manner. If there is not enough historical data, such as the first iteration of in-staging processing, the policy still returns a fixed number as the rescaling number of processes. If key model parameters have existed, we use the current model to calculate how many processes need to be added or removed to achieve an expected in-staging execution time[3]. Besides, we may adopt more constraints, such as the total number of available processes we can use or the minimal amount of memory for in-staging execution to decide the final number of processes for elasticity.

B. Rescaling the data staging service

Based on the discussion of elastic workflow in subsection II-A, we may add processes to the data staging service whenever possible to decrease wait time. Algorithm 1 illustrates when and how to trigger the elasticity in this case. This algorithm can be executed at the *rescale and sync* stage of the elastic workflow shown in Figure 1. In particular, when the time to wait for the in-staging execution to finish (T_w) is longer than a threshold (TH_1), we add k processes into the data staging service; conversely, we remove k processes if the difference between the latest computation time (T_p) and in-staging execution time (T_c) is longer than a threshold (TH_2).

One critical step in Algorithm 1 is to compute the number of processes added or removed for each rescaling operation. For the example shown in Figure 2, the in-staging execution time is inversely proportional to the number of data staging processes. In order to express this relationship in a more general form, we can use $y = a \cdot x^b$ as a fitting function[4]. 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 ana/vis tasks and processed data size, and b is usually a negative real number. For the convenience of computing the parameters used in the model, we can take the logarithm of both sides of the original equation and change it to a linear equation as follows: $\ln y = \ln a + b \cdot \ln x$. We then need only two data points to estimate the values of a

[3]This value can be specified by the trigger conditions.

[4]This is one typical form of power-law distribution [8]. Although we adopt the power-law distribution as a fitting curve in this work, the model for anticipating the process number can be expressed by different equations.

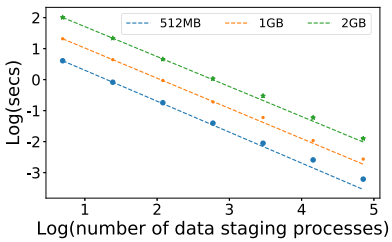


Fig. 3. Example of the model prediction based on power-law distribution. The solid dot represents the actual value, the dash line is drawn based on estimated model.

and b. Based on this model, given an execution time of in-staging processing, the associated number of processes can be expressed as:

$$x = \exp(\frac{\ln y - \ln a}{b}) \tag{1}$$

Assuming the current number of processes is x_0, the current execution time is y_0, and the targeted execution time is y_1. If we try to predict how many processes need to be added into the staging service to decrease the execution time from y_0 to y_1, we need to calculate:

$$k = \exp(\frac{\ln y_1 - \ln a}{b}) - \exp(\frac{\ln y_0 - \ln a}{b}) \tag{2}$$

The value of k is the number of processes that we will add to the data staging service. Similarly, we can compute how many processes need to be removed from the data staging service if we need to increase the in-staging execution time to a specific value.

Figure 3 shows an example of how it works to anticipate the parameters used in the model discussed above. Assuming we try to predict a model that matches the ana/vis execution time shown in Figure 2. We need at least two data points as input to estimate unknown parameters in power-law distribution $y = a{\cdot}x^b$. Figure 3 shows the estimated data using a dashed line and actual data using solid dots for the equation with log operators, namely $\ln y = \ln a + b{\cdot}\ln x$. As shown in results, the estimated data calculated by the model can well match the actual data with different simulated data sizes. With the online prediction of key parameters for the model, we can conveniently compute how many processes need to be added (removed) to decrease (increase) the data staging execution time to a particular value.

C. Rescaling data staging service and simulation

Instead of adding more resources during the execution of the job, in this scenario, we assume the total computing resources are fixed, and processes assigned to the simulation or 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 execution time. Assuming the simulation computation time T_c is equal to $f_1(P_{sim})$ and the data staging execution time T_p is equal to $f_2(P_{stage})$, where P_{sim} and P_{stage} represents the number of processes used by simulation computation and data staging service, respectively, if the elasticity policy switches k processes from simulation program to the data staging service, we end up with $T'_c = f_1(P_{sim} - k)$ and $T'_p = f_2(P_{stage} + k)$. The elasticity policy needs to find out if there exists a value k that satisfies $max(T'_c, T'_p) < max(T_c, T_p)$. If there exist multiple eligible values, we select one that can minimize the $max(T'_c, T'_p)$. Therefore, removing k processes from the simulation then adding them to the data staging service can decrease the in-staging processing time.

Algorithm 2: Adaptive strategy for rescaling with both the simulation and the data staging service.

```
1  while step < totalstep do
2      Computing the k value;
3      Exiting k simulation processes;
4      Starting k data staging processes;
5      Waiting for the simulation rescaling to finish;
6      Simulation computation;
7      Waiting for previous ana/vis to finish;
8      Updating models based on key metrics;
9      Waiting for the data staging rescaling to finish;
10     Calling the data transfer RPC;
11     Calling the execution RPC;
12     Syncing new step value;
13 end
```

We list key operations in the simulation program that support the elastic resource redistribution in Algorithm 2. From line 2 to line 4, we make the elasticity decision based on the aforementioned adaptive strategy and correspondingly decrease (increase) simulation (data staging service) processes. In particular, we may try to remove the current process from the simulation program and send a signal to an elasticity trigger to start a new staging process. From line 5 to line 8, 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 update the models to compute the simulation (ana/vis) execution time based on the latest metrics. In this work, we assume the distribution of execution time does not change during the workflow execution; however, we may also update key model parameters if the error between the estimated value and the actual value is larger than a threshold. The data redistribution and migration are also key issues to guarantee the correctness of malleable simulation computation [20]; however, these issues depend on specific use cases and are out of the scope of current work[5]. From line 9 to line 12, we wait for the data staging rescaling operations to finish and update records for all alive data staging processes. Besides, the simulation process updates the endpoints record of alive data staging processes. Next, we call the data transfer RPC to put data into the data staging service, then call the

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

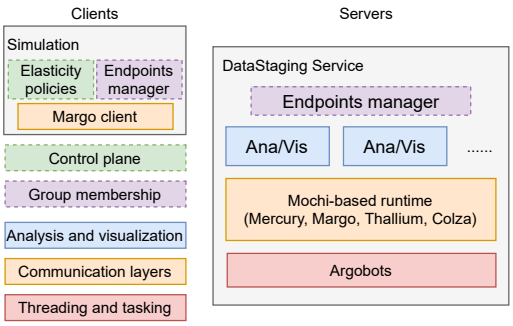


Fig. 4. The architecture for implementing the in-staging processing with the elasticity policies.

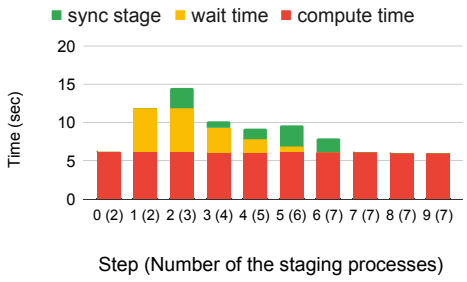
execution RPC to trigger associated ana/vis tasks. At last, we increase the step value and update it for each existing simulation process.

IV. Implementation Overview

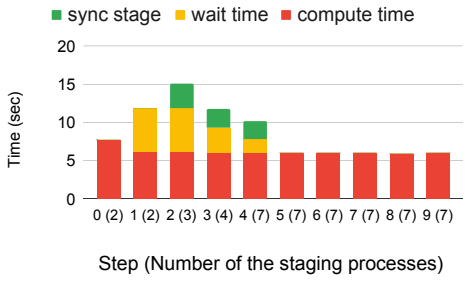
Figure 4 illustrates the architecture for implementing the staging-based workflow with elasticity policies. Major components adopted by the simulation (clients) and the data staging service (servers) are shown in the figure. In particular, the data staging service is built on Mochi data services [26]: Margo and Thallium provide the service of remote procedure call (RPC), which is built on Mercury for transferring data by RDMA, and Argobots for user-level thread management; Colza[6] can execute customized ana/vis pipeline in data staging service. The user-defined ana/vis can be executed based on the user-level thread provided by Argobots and the collective communication primitives provided by Colza. Although the implementation is built on Mochi-based runtime, the elasticity policies discussed in section III are not limited to the Mochi data services; they can also be adapted to other data staging services that support elasticity primitives.

The newly added components for elasticity in this paper are circled by the dashed line. In particular, *Endpoints manager* records all alive processes and updates the communicator based on the latest endpoint list. A Leader-worker strategy is adopted to manage the consistency of alive processes. In particular, the added/removed processes send an RPC to the leader process (the process with rank 0) to register/deregister their endpoints; then, the leader process broadcasts the updated endpoint list to all worker processes to update their communicator. The *Endpoints manager* can be integrated with both the simulation program and the data staging service to support elasticity. The main function of the *Elasticity policies* is to collect the workflow runtime information and decide when and how to execute the elasticity primitives based on the policy discussed in section III. When it decides to add a new process, a signal is sent to the elasticity trigger to start a new process. The signal can be a dedicated configuration file, and the elasticity trigger can be implemented by bash code, which calls `srun` to start new processes.

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



(a) Static elasticity policy



(b) Adaptive elasticity policy

Fig. 5. Comparison between the static and the adaptive elasticity policy.

V. Evaluation

The evaluation in this paper is divided into three categories. In subsection V-A, we evaluate how the elasticity policy can dynamically find a resource configuration that minimizes the wait time for in-staging execution. Furthermore, in subsection V-B, we evaluate the efficiency of the elasticity policy that dynamically switches the process between simulation and data staging service. At last, we use more realistic ana/vis to show the efficiency of elasticity policy in section V-C. All experiments in this evaluation were performed on the Cori supercomputer's Haswell partition [22]. **The corresponding code[7] is publicly available.**

A. Efficiency of adding data staging processes

In this experiment, we aim to evaluate the adaptive elasticity policy discussed in subsection III-A. We use the Mandelbulb mini-simulation[8] as the data source in this experiment. The simulated data is processed by synthetic ana/vis, which is a sleep function that changes sleep time according to the number of processes based on the power-law distribution.

Figure 5(a) illustrates key metrics during the workflow execution with the static elasticity policy. In this case, the elasticity policy checks the wait time (illustrated in Figure 1) at the beginning of each iteration; if it is larger than 0.1 seconds, we add one new process into the data staging service.[9] With the number of processes for data staging services increasing from 2 to 7, the wait time gradually decreases to zero. After

[7]https://git.io/J0UCc
[8]https://github.com/mdorier/MandelbulbCatalystExample
[9]We assume one newly added process runs on one node in this experiment.

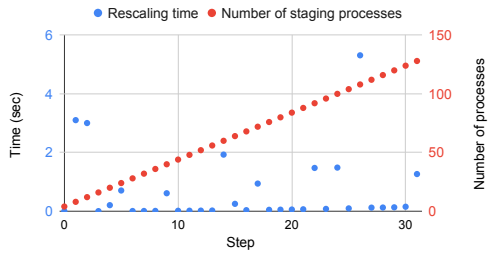


Fig. 6. The overhead of adding processes into the data staging service.

that, the simulation computation overlaps with the in-staging processing, and we do not further add data staging processes for the subsequent workflow executions.

Figure 5(b) shows the results of adopting the adaptive elasticity policy discussed in subsection III-B. For the step 1 and 2, the wait time is larger than 0.1 seconds, and there is not enough data to build the model discussed in subsection III-B; therefore, we add one new process for these two steps. After the third step, the adaptive elasticity policy confirms that adding three new staging processes can decrease the in-staging execution time to a proper level, in which there is full overlap between the simulation computation and in-staging execution. Compared with the static elasticity strategy shown in Figure 5(a), the adaptive elasticity strategy takes fewer iterations to find a proper configuration to decrease the wait time to zero. In this way, we can avoid extra overhead of resource rescaling.

The overhead of starting new processes comes from three aspects: (1) building the model and deciding when or how to trigger elasticity; (2) sending the signal to trigger an addition of new process; (3) updating the communicator in staging service when there are added/removed processes. In our experiment, the overhead of (1) and (3) are trivial, which are less than 1 second; however, the signal sending and new process triggering may take a long time when the performance of the parallel file system and batch scheduler is poor. This is because the trigger of `srun` command depends on the detection of the configuration file served as a triggering signal. The high workload on the system may increase the delay of the file detecting and process triggering. In our evaluation, this overhead[10] can vary from several seconds (illustrated in Figure 5) to tens of seconds. Figure 6 shows more details on the rescaling performance. We gradually add 4 data staging processes at each iteration and increase the number of processes from 2 to 128 within 30 steps. Most of the rescaling operations (the sync stage time) can finish within 2 seconds. Around $2/3$ rescaling operations finish in less than 0.5 seconds, which is negligible compared with the decrease of wait time enabled by elasticity strategies shown in Figure 5(a) and Figure 5(b).

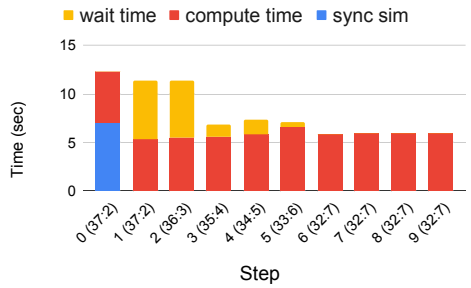
[10]The time period from the moment of executing the configuration file write operation to the moment that the batch scripts detect the dedicated file.

B. *Efficiency of the dynamic resource redistribution*

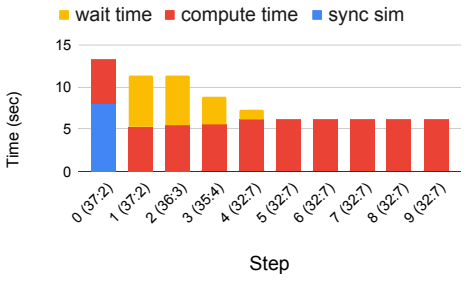
In the previous experiment, we assume the computing resources assigned to the simulation program are fixed, but the data staging service is elastic. Therefore, the computation resources assigned to the job can also be elastic during the workflow execution. In this experiment, we assume both the simulation 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 adaptive elasticity strategy discussed in the subsection III-C. For example, if the benefits of adding one process to the data staging service exceed 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 further decrease the execution path of the staging-based workflow without adding new computing resources to the job.

We update the mini-simulation used in experiment V-A and make it elastic based on the endpoints manager described in implementation details in section IV. When one 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 simulation computation. Besides, a dedicated configuration file is written to the parallel file system when one simulation process leaves. The batch script keeps monitoring the file system and starts a new data staging process when there is the detection of the corresponding configuration file.

Figure 7(a) shows the results of adopting 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 it is larger than 0.1 seconds, we remove one simulation process then start another data staging process. With more computing resources adding to the data staging service, the static elasticity policy decrease the wait time to zero. Compared with the decrease of the wait time, the increase of the simulation computation time is not obvious even if we decrease the simulation number of processes. This is because that the simulation computation is insensitive to the variation of the number of processes based on the current resource configuration 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 static elasticity policy shown in Figure 7(a), the adaptive elasticity policy illustrated in Figure 7(b) shows a better performance to achieve a proper resource configuration for both simulation and data staging service. The elasticity policy decides to remove three processes at the third step and start new staging processes accordingly; it decreases the overhead caused by rescaling new processes based on the adaptive elasticity policy.

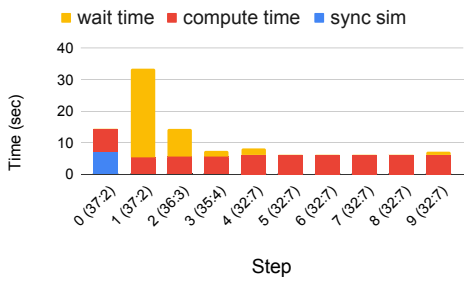


(a) Static elasticity policy

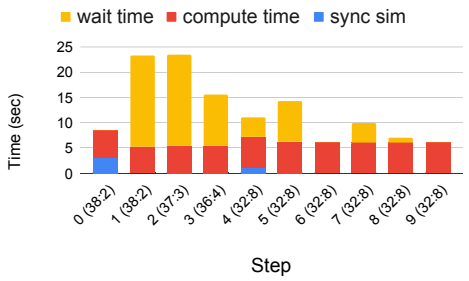


(b) Adaptive elasticity policy

Fig. 7. The performance of changing process from simulation program to the data staging service with different elasticity policies. The number in the parenthesis of the horizontal label represents the number of the simulation processes and the number of the staging processes, respectively.



(a) Case 1



(b) Case 2

Fig. 8. The performance of the adaptive elasticity policy that adjusts the resource configuration with different cases of in-staging executions. The number in the parenthesis of the horizontal label represents the number of the simulation processes and the number of the staging processes, respectively.

C. *Towards actual ana/vis*

In the previous two experiments, we use a sleep function that follows the power-law distribution as a synthetic ana/vis. In this evaluation, we update the tasks executed in the data staging service and make it closer to the actual ana/vis. In particular, we convert the simulated data into the VTK format [27] and write it out to the VTK image files. This commonly used in-staging processing can be further integrated with complicated data filters or other downstream visualization tools. In this experiment, the total computing resource assigned to the job is also fixed during the workflow execution. We reuse the adaptive elasticity policy evaluated in section V-B to redistribute the process between the simulation and data staging service. In order to construct different types of in-staging executions for evaluation, we run the aforementioned VTK data ana/vis task with different iterations. For Case 1 shown in Figure 8(a), we execute it for 10 times, and for Case 2 shown in Figure 8(b), we execute it for 20 times.

For the results shown in Figure 8(a), the policy works as expected, and it finds a proper resource configuration between simulation and data staging service at the fourth step.[11] Then the in-staging execution can overlap with the simulation computation. However, for the results of Case 2 shown in Figure 8(b), when the wait time becomes trivial at the 4^{th} step, it then becomes large at the 5^{th} step. This is because that the in-staging execution becomes unbalanced at this step. Most of the in-staging execution finish at around the 5 seconds, and some particular process needs dozens of seconds. The wait time depends on the longest in-staging execution in this case. These situations break up assumptions for methods discussed in section III-B where the total execution time follows the power-law distribution. After checking the log of data staging service in details, most of the in-staging execution follows the power-law distribution except several special processes. We will update the elasticity policy to support this scenario in future work. One potential solution is to dynamically change the available data staging processes when transfer and execute the ana/vis tasks. If the in-staging execution did not finish within a specific threshold value, the simulation program will put data to other data staging processes for the next iteration.

[11] We set number of processes that can be switched between the simulation and data staging service as 6 in this experiment.

VI. Research Opportunities

In the evaluation of this paper, we use the synthetic ana/vis that does not require a collective operation. One research opportunity is to try to adapt the existing in-situ ana/vis pipeline to support the elastic in-staging processing. For example, the exiting visualization pipeline, such as Paraview Catalyst [3] is built on the static communicator that assumes process size does not change during the workflow execution. It is interesting to explore how to update these ana/vis to support elasticity and integrate it with the current elasticity policy for in-situ workflow.

The data staging service used in this work for the proof of concept only provides the minimal capability of in-staging data management. It is interesting to integrate the elasticity

policy and elastic ana/vis with the state-of-the-art data staging services such as DataSpaces [10] and Damaris [12] to compare their efficiency for elastic in-situ processing.

Because of the limitation of the evaluation platform in this work, we did not adopt the actual batch scheduler that supports elasticity. Instead, we reserve enough nodes in advance but use several of them to show the efficiency for elasticity for the proof of concept. It is interesting to integrate the presented elasticity policy with the latest job scheduler [1] that supports the resource elasticity. Furthermore, it deserves to explore a more efficient process trigger procedure. The signal represented by the configuration file is influenced by the performance of the parallel file system that causes an extra overhead of process triggering.

For the adaptive elasticity policy discussed in this paper, we assume both the execution time of in-staging execution and the simulation computation time follow the power-law distribution. However, it is possible that in-staging execution is more complicated, such as the single process with unbalanced execution time shown in subsection V-C. The model prediction of execution time can also be updated to adapt to various scenarios. Besides, if there is workflow runtime information, such as the performance of the `srun` operation, the adaptive policy can make a more accurate decision to decide if the resource rescaling is necessary.

VII. Related Work

Tong et al. [17] present a resource adaptation policy to rescale the computation resources used for data staging service and to improve the resource utilization efficiency. However, they did not explain details of how to properly determine the number of processes to join/leave the computation group.

Fox et al. [15], and Chadha et al. [5] focus on how to provide elasticity from the job scheduler's perspective. They discuss how to update the existing scheduler to support elasticity primitives. Our work focuses on the policy to trigger these primitive, which is complementary to their works.

In the context of the fault-tolerance, Duan et al. [14] use elasticity as a mechanism to support the data failure detection and recovery for data staging processes. The User Level Failure Mitigation (ULFM) [4] presents MPI extensions to detect communicator failure, along with solutions to 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.

Elasticity is also an important aspect of cloud computing. Ghanbari et al. [16] summarizes typical approaches to support elasticity in the context of cloud computing. Zahedi et al. [29] discusses how to use the Amdahl utility function to decide the process allocation. Our work focus on the elasticity policy from the application perspective in the context of the in-situ processing for scientific workflow.

VIII. Conclusion and future work

We presented the design and evaluation of the adaptive elastic policies that can trigger elasticity primitives online for staging-based in-situ processing. The evaluation shows that the adaptive elasticity policy can efficiently find a proper resource configuration and decrease the overhead caused by rescaling. In future work, we will integrate the presented elasticity policy with full-fledged in-situ processing that contains the elastic ana/vis pipeline and uses an elastic batch scheduler.

Acknowledgement

The research presented in this paper was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program. 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 DE-AC02-06CH11357.

References

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

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

[3] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and 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*, pages 25–29, 2015.

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

[5] M. Chadha, J. John, and 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)*, pages 223–232. IEEE, 2020.

[6] J. H. Chen, A. Choudhary, B. De Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W.-K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[7] 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*, page 1094342020935991, 2020.

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

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

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

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

[12] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and 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)*, pages 67–75. IEEE, 2013.

[13] M. Dorier, O. Yildiz, T. Peterka, and 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, page 23–28, New York, NY, USA, 2019. Association for Computing Machinery.

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

[15] W. Fox, D. Ghoshal, A. Souza, G. P. Rodrigo, and 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*, pages 1–11, 2017.

[16] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai. Exploring alternative approaches to implement an elasticity policy. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 716–723, 2011.

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

[18] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire. Opportunities for cost savings with in-transit visualization. In P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, editors, *High Performance Computing*, pages 146–165, Cham, 2020. Springer International Publishing.

[19] P. Malakar, V. Vishwanath, C. Knight, T. Munson, and 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*, pages 702–715, 2016.

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

[21] K. Moreland and R. Oldfield. Formal metrics for large-scale parallel performance. In *International Conference on High Performance Computing*, pages 488–496. Springer, 2015.

[22] NERSC. The Cori System. https://docs.nersc.gov/systems/cori/.

[23] 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), 2019.

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

[25] A. Raveendran, T. Bicer, and 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*, pages 940–947. IEEE, 2011.

[26] 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, and Z. Qing. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35(1):121 – 144, Jan 2020. 10.1007/s11390-020-9802-0.

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

[28] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and 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*, pages 920–930. IEEE, 2018.

[29] S. M. Zahedi, Q. Llull, and 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)*, pages 1–14, 2018.