Harnessing the Power of Many: Extensible Toolkit for Scalable Ensemble Applications

Vivek Balasubramanian*, Matteo Turilli *, Weiming Hu†, Matthieu Lefebvre‡, Wenjie Lei‡, Guido Cervone†, Jeroen Tromp‡ and Shantenu Jha*,

*ECE, Rutgers University  † Penn State University  ‡ Princeton University

§ Brookhaven National Laboratory

* Contributed Equally

Abstract—Many scientific problems require multiple distinct computational tasks to be executed in order to achieve a desired solution. We introduce the Ensemble Toolkit (EnTK) to address the challenges of scale, diversity and reliability they pose. We describe the design and implementation of EnTK, characterize its performance and integrate it with two distinct exemplar use cases: seismic inversion and adaptive analog ensembles. We perform nine experiments, characterizing EnTK overheads, strong and weak scalability, and the performance of two use case implementations, at scale and on production infrastructures. We show how EnTK meets the following general requirements: (i) implementing dedicated abstractions to support the description and execution of ensemble applications; (ii) support for execution on heterogeneous computing infrastructures; (iii) efficient scalability up to \( O(10^5) \) tasks; and (iv) fault tolerance. We discuss novel computational capabilities that EnTK enables and the scientific advantages arising thereof. We propose EnTK as an important and unique addition to the suite of tools in support of production scientific computing.

I. INTRODUCTION

Traditionally, advances in high-performance scientific computing have focused on the scale, performance and optimization of a workload with a large but single task, and less on workloads comprised of multiple tasks. However, many scientific problems are expressed as applications that require multiple distinct computational tasks to be executed in order to achieve a desired solution.

“Task” is used to represent processes at different scales and granularity. In this paper, a computational task is a generalized term for a stand-alone process that has well defined input, output, termination criteria, and dedicated resources. Specifically, a task is used to represent an independent simulation or data processing analysis, running on one or more nodes of a high-performance computing machine.

When the collective outcome of a set of tasks is of importance, this set is defined to be an ensemble. The individual tasks within the set might be uncoupled; if coupled, the tasks might have global (synchronous) or local (asynchronous) exchanges, and regular or irregular communication. This is in contrast to traditional parameter sweeps, or high-throughput computing (HTC) workloads, where the tasks are typically identical, uncoupled, idempotent and can be executed in any order. Ensembles may also vary in the number and types of task, tasks’ executable, and tasks’ resource requirements. Ensembles assume no tasks dependencies, and as such, are a special case of the more general concept of workflow.

The number and type of applications that can be formulated as ensembles is vast and span many scientific domains. Some applications are naturally expressed as ensembles (e.g., calculating average properties) and some can be expressed as ensembles (e.g., statistical uncertainty quantification). Some scientific problems that have traditionally been expressed as a single computational task, must be reformulated using ensembles so as to overcome limitations of single task execution [1]. For example, in biomolecular sciences, due to the end of Dennard scaling, and thus limited strong scaling of individual MD tasks, there has been a shift from running single long running tasks towards multiple shorter running tasks, as evidenced by a proliferation of ensemble-based algorithms [2, 3].

The execution of an ensemble on HPC resources present several challenges: encoding scientific problems into algorithms amenable to distributed and coordinated solution; sizing, acquiring, and managing resources; and managing the execution of the ensemble. Encoding scientific problems into ensembles requires the relations of order and inclusion, alongside a large and heterogeneous set of properties to describe tasks. Acquiring HPC resources for distributed execution requires co-existence with policies that support single, large-scale task executions. Distributing an execution over multiple partitions of resources requires tailored coordination and communication infrastructure and protocols.

In response to these challenges and requirements, the growing importance of ensemble-based applications in scientific HPC, and the absence of middleware providing scalable, extensible and general solutions, we have designed and implemented the Ensemble Toolkit (EnTK). EnTK promotes ensembles to a high-level programming abstraction, providing specific interfaces and execution models for ensemble-based applications. EnTK is engineered for scale and a diversity of computing platforms and runtime systems, and it is agnostic of the size, type and coupling of the tasks comprising the ensemble.

EnTK adheres to the principle of design, development and integration of middleware and workflow systems by means of a building blocks approach [4, 5]. This approach advocates...
a sustainable ecosystem of software components from which tailored workflow systems can be composed, as opposed to having to fit workflows to pre-existing frameworks. The building blocks approach overcomes the limited flexibility of monolithic workflow systems, by enabling composability and extensibility, and thereby supporting the wide range of workflow requirements. As circumstantial evidence, EnTK has been used to develop several diverse domain-specific workflow systems [4].

In this paper, we describe the design, architecture and implementation of EnTK. We characterize its performance on different HPC platforms for a variety of task types, and observe invariance across platforms and tasks. We use EnTK to support the requirements of two ensemble-based scientific applications but with widely differing characteristics and requirements. We show how the use of EnTK enables both applications to utilize new computational capabilities in production mode, and discuss the scientific implications and importance of these new capabilities. As such, we propose EnTK as an important and unique addition to the suite of tools in support of production scientific computing.

II. ENSEMBLE TOOLKIT (ENTK)

The design and implementation of EnTK are iterative and driven by use cases. Use cases span several scientific domains, including Biomolecular Sciences, Material Sciences, and Earth Sciences. Users and developers collaborate to elicit requirements and rapid prototyping. EnTK is loosely specified in UML, validated against its requirements and characterized via a profiler. Jenkins and Travis are used for continuous integration and automated testing. Documentation and code are managed and made available via a Git repository [6].

A. Requirements

As seen in [1], the space of ensemble applications (hereafter simply ‘applications’) is vast, and thus there is a need for simple and uniform abstractions while avoiding single-point solutions. We elicited requirements for computing infrastructures (CIs), scale, fault-tolerance, and usability. EnTK is required to: (1) support heterogeneous CIs; (2) abstract the complexity of execution and resource management; and (3) performance independent of the type of CI.

The use cases motivating EnTK require up to $O(10^4)$ concurrent ensemble members (tasks). Most limitations of scale derive from the runtime system (RTS) so one of the requirements of EnTK is to enable the use of different middleware, depending on the requirements of applications and CIs.

A critical requirement for EnTK is to be fault-tolerant at scale, i.e., when both the probability and cost of failure increase. EnTK is required to support resubmission of failed tasks and restarting of failed RTS and components. In this way, applications can be executed with multiple attempts, without restarting completed tasks.

Usability plays an important role in the development of EnTK, as it must support diverse programming and development skills. Special attention is given to lowering the time to encode use cases into executable applications.

B. Design

The design of EnTK is based on an application model called ‘PST’ and a distributed architecture. We use the PST model to enable users to describe and reason about applications and the distributed architecture to abstract details and localize information and functionalities to components.

1) PST Model: We model applications by combining sets and lists: Lists represent the runtime dependencies of tasks or groups of tasks; sets represent absence of dependencies. We defined the following user-facing constructs:

- **Task**: an abstraction of a computational task that contains information regarding an executable, its software environment and its data dependences.
- **Stage**: a set of tasks without mutual dependences and that can be executed concurrently.
- **Pipeline**: a list of stages where any stage $i$ can be executed only after stage $i-1$ has been executed.

Figure 1 shows an application described with the PST model. The application consists of a set of pipelines, where each pipeline consists of a list of stages, and each stage consists of a set of tasks. All the pipelines can execute concurrently, all the stages of each pipeline can execute sequentially, and all the tasks of each stage can execute concurrently.

Note that the PST model can be extended to describe dependences among groups of pipelines as lists of sets of pipelines. Further, the specification of branches in the execution flow of applications does not require to alter the model: Branching events can be specified as tasks where a decision is made about the runtime flow. For example, a task could be used to decide to skip some elements of a stage, based on some partial results of the ongoing computation.

2) Architecture: EnTK sits between the user and the CI, abstracting resource management and execution management from the user. Fig. 2 shows the components (purple) and subcomponents (green) of EnTK, organized in three layers: API, Workflow Management, and Workload Management.

The API layer enables users to describe an application in terms of pipelines, stages and tasks. The Workflow Management layer retrieves information from the user about available CIs, initializes EnTK, and holds the global state of the application.
AppManager holds these descriptions and, upon initialization, ensures failures of its components and subcomponents. During execution, the Workload Management layer acquires resources via the runtime system (RTS) for the execution.

The Workflow Management layer has two components: AppManager and WFProcessor. AppManager uses the Synchronizer subcomponent to update the state of the application at runtime. WFProcessor uses the Enqueue and Dequeue subcomponents to queue and dequeue tasks from the Task Management layer. The Workload Management layer uses ExecManager and its Rmgr, Emgr, RTS Callback, and Heartbeat subcomponents to acquire resources from CIs and execute the application.

All the information about the state of the execution is recorded in the application’s description of AppManager. All state updates are transactional, making EnTK fault-tolerant towards failures of its components and subcomponents. Upon failure, each component can be re-instantiated with minimal loss of information. In addition, users may also choose to enable resubmission of failed tasks, letting the Dequeue subcomponent to reset the failed tasks to their initial state. These tasks are then identified as new tasks, ready to be managed by the Enqueue subcomponent.

Another benefit of this architecture is the isolation of the RTS into a stand-alone subsystem. This enables composability of EnTK with diverse RTS and, depending on capabilities, multiple types of CIs. Further, EnTK assumes the RTS to be a black box enabling fault-tolerance. When the RTS fails or becomes unresponsive, EnTK can tear it down and bring it back, loosing only those tasks that were in execution at the time of the RTS failure.

3) Execution Model: EnTK components and subcomponents communicate and coordinate for the execution of tasks. Users describe an application via the API, instantiate the AppManager component with information about the available CIs and then pass the application description to AppManager for execution. AppManager holds these descriptions and, upon initialization, creates all the queues, spawns the Synchronizer, and instantiates the WFProcessor and ExecManager. WFProcessor and ExecManager instantiate their own subcomponents.

Once EnTK is fully initialized, WFProcessor initiates the execution by creating a local copy of the application description from AppManager and tagging tasks for execution. Enqueue pushes these tasks to the Pending queue (Fig. 2, 1). Emgr pulls tasks from the Pending queue (Fig. 2, 2) and executes them using a RTS (Fig. 2, 3). RTS Callback pushes tasks that have completed execution to the Done queue (Fig. 2, 4). Dequeue pulls completed tasks (Fig. 2, 5) and tags them as done, failed or canceled, depending on the return code of the RTS.

Throughout the execution of the application, tasks, stages, and pipelines undergo multiple state transitions in both WFProcessor and ExecManager. Each component and subcomponent synchronizes these transitions with AppManager by pushing messages through dedicated queues (Fig. 2, 6). AppManager pulls these messages and updates the application states. AppManager then acknowledges the updates via dedicated queues (Fig. 2, 7). This messaging mechanism ensures that AppManager is always up-to-date with any state change, making it the only stateful component of EnTK.

C. Implementation

EnTK is implemented in Python, uses RabbitMQ message queuing system and the RADICAL-Pilot (RP) [7] runtime system. All EnTK components are implemented as processes, and all subcomponents as threads. AppManager is the master process spawning all the other processes. Tasks, stages, and pipelines are implemented as objects, copied among processes and threads via queues and transactions. Synchronization among processes is achieved by message-passing via queues.

EnTK relies on RabbitMQ to manage the creation of the communication infrastructure to transport the objects and messages among components. RabbitMQ provides methods to increase the durability of messages in transit and of the queues, and to acknowledge messages. Most importantly, it supports the requirement of managing at least $O(10^4)$ tasks concurrently.

RabbitMQ is a server-based system and requires to be installed before the execution of EnTK. This adds overheads but it also offers the following benefits: (1) producers and consumers do not need to be topology aware because they interact only with the server; (2) messages are stored in the server and can be recovered upon failure of EnTK components; and (3) messages can be pushed and pulled asynchronously because data can be buffered by the server upon production.

Currently, EnTK uses RP as the runtime system. RP is a pilot system that enables late binding of tasks to resources. RP submits resource placeholders to one or more CIs and, once acquired the resources, RP executes a software called ‘Agent’ on them. Agent pulls tasks from the application and schedules them on the acquired resources. In this way, users can execute multiple tasks both concurrently and sequentially, depending on the amount of available resources and their walltime. Compared to other pilot systems, RP supports MPI-based and GPU-based tasks; offers extensive profiling capabilities; and, uses SAGA to support executions on HPC, HTC, and Cloud CIs.

III. USE CASES

We describe the two use cases of this paper, focusing on their computational and functional requirements.
Adjoint Source Creation (1) and adjoint (3) are by far the most computationally expensive part. The optimization (5) takes about 10,000 core-hours, in which smoothing the kernels is the most time-consuming part. The data processing (4) takes about 1000 core-hours in each iteration but requiring the data to undergo smoothing the kernels obtained for each earthquake are them summed to produce a gradient to be used (5) the optimization and the update routines. This process is repeated until convergence.

Fig. 3. Simplified seismic tomography workflow. This iterative process consists in 5 steps. (1) For each earthquake, synthetic data are generated, (2) both synthetic and real data are processed and compared to create the so called adjoint sources that are then (3) used to initiate adjoint simulations. (4) The synthetic and real data are processed and compared to create the so called adjoint sources that are then (3) used to initiate adjoint simulations. (4) The kernels obtained for each earthquake are them summed to produce a gradient to be used (5) the optimization and the update routines. This process is repeated until convergence.

A. Seismic Inversion

A number of seismic algorithms have been developed to improve the understanding of the Earth’s interior. In what follows, the focus is on the most powerful of these techniques: the inversion of full-waveform, wide-bandwidth seismic data [8]. Scaling full waveform inversion to larger models [9] and the assimilation of data from thousands of earthquakes is challenging, not only by the amount of computational resources it requires, but also by the quantity of human labor needed to perform such studies. These challenges mandate a more automated approach such as the one implemented by the EnTK.

A brief overview of the scientific workflow involved to perform seismic tomography is useful to understand where the challenges lie. For clarity purpose, we restrict the discussion to global tomography, where the models are approximations of the entire Earth’s structure and where data are seismograms recording earthquake generated waves. A high level view of this workflow is shown in Figure 3. Seismic data (or seismograms) are recorded as time series of a physical quantity, such as displacement, velocity, acceleration or pressure. The overall goal is to iteratively minimize differences between a set of observed data and corresponding synthetic data through a pre-defined misfit function. As the adjoint-based optimization procedure is carried on and the data misfit decreases, the model get closer to reality.

The computational characteristics of each part of the workflow are better explained by looking at the specificities of state-of-the-art production runs. As of today, and at the global scale, routine production work consists in assimilating data from about a thousand earthquakes. The simulations (both forward (1) and adjoint (3)) are by far the most computationally expensive part and each of them is ran on 384 GPUs for a total of 10 millions core-hours per iteration. The data processing part (2) is relatively inexpensive, utilizing about 48000 core-hours in each iteration but requiring the data to undergo a significant number of steps. The post-processing (4) takes about 10000 core-hours, in which smoothing the kernels is the most time-consuming part. The optimization (5) takes about 1 million core-hours, since forward simulations of a subset of events are launched to determine the step length.

Each part of the workflow has been relying on a Python-based proto-workflow management system using abstractions provided by the RADICAL-SAGA package. However, scaling to higher resolutions and assimilating data from a complete database of 6,000 recorded earthquakes requires a paradigm shift and the adoption of a more automated approach to ensure reliability, minimize errors at the user level and lower the overall time to solution.

From a workflow management point of view, several challenges are provided by this seismic tomography workflow. First, it interleaves relatively large simulations with smaller data-processing tasks, or each task requiring respectively leadership-scale systems and moderately sized clusters. Second, it is coupled with a data management problem, since some data needs to be temporarily or permanently saved, or moved around.

Ensemble-based applications are particularly well-suited to encode the seismic tomography workflow. In EnTK, both the simulation and analysis phases can be described as stages of a pipeline, avoiding the need for a dedicated MPI application to execute multiple simulation concurrently. EnTK also allows the execution of the ensemble of simulations with a varying degree of concurrency and sequentiality, without requiring specific coding. EnTK offers automation and fault-tolerance avoiding the overheads experienced by the users with full-fledged workflow systems.

B. High Resolution Meteorological Probabilistic Forecasts

We used a new methodology called Analog Ensemble (AnEn) [10] to generate high-resolution, probabilistic forecasts for specific parameters like temperature or cloud cover. We developed a specific implementation of AnEn to generate high-resolution depictions of environmental variables, using relationships between current and past forecasts from the Weather Research and Forecast model (WRF) data. To generate an analog ensemble for a given time t and a given location l, our implementation finds the most similar historical forecasts within the search space based on a similarity metric. The observations associated with the most similar past forecasts are used as analogs. These observations can be calibrated using spatio-temporal coefficients if predictable errors are known (e.g., the degradation of a solar panel yields lower wattage as a function of time).

We drove the development and assessed the suitability of EnTK for analog computation by testing the AnEn method with dataset including forecast predictions for 13 variables (e.g., wind speed, precipitation, pressure, etc.) for the years 2015 and 2016. Data for both analysis and forecasts are from the North American Mesoscale Forecast System (NAM) maintained at the National Center of Atmospheric Research (NCAR).

One of the challenges of this use case is that analogs have to be computed for a large number of geographical locations, potentially including millions of specific locations. However, in principle it is possible to exploit some unique physical characteristics of the predictand variable (e.g., temperature), and
The workflow can be divided into 4 steps: initialization; preprocess-
tation; iterative computation; and post-processing. The initializa-
tion step specifies the search space and the test space, and sets
up starting parameters for the AnEn. The preprocessing step
generates preparatory data that will be used in the subsequent
steps. The larger amount of computation occurs in the iterative
computation step: (1) Compute the unstructured grid to rep-
resent the locations where analogs need to be computed; (2)
Distribute the analog computation to the available computing
resources when a larger search space is required in order to
improve prediction accuracy. This is paramount when not
many good analogs are found in the previous search space.

One of the main difficulties with this approach is that the algo-

rithm is iterative, and at each iteration it performs a variable
number of operations. EnTK addresses the resource manage-
ment challenges arising from such variations: by using an
iterative approach, the algorithm can generate higher predic-
tions for places where there is higher variability, using an
unstructured grid. EnTK also empowers us to achieve a higher
level of automation where the algorithm can allocate more
computing resources when a larger search space is required in
order to improve prediction accuracy. This is paramount when
not many good analogs are found in the previous search space.

Figure 4 shows the workflow of the AUA algorithm. The
workflow can be divided into 4 steps: initialization; preproces-
sing; iterative computation; and post-processing. The initializa-
tion step specifies the search space and the test space, and sets
up starting parameters for the AnEn. The preprocessing step
generates preparatory data that will be used in the subsequent
steps. The larger amount of computation occurs in the iterative
computation step: (1) Compute the unstructured grid to rep-
resent the locations where analogs need to be computed; (2)
Distribute the analog computation to the available computing
resources; (3) Aggregate the analogs computed at different lo-
cations; and (4) evaluate the predictions, and determine if the
termination condition is met. The termination condition could
be either the exhaustion of the computational resources avail-
able, or an overall prediction error below a specified threshold.
Finally, a post-processing task interpolates the analogs at each
location to generate the forecast solution.

We use the unstructured grid methodology based on the De-
launay triangulation method [11] to drive the algorithm. The
algorithm starts by generating a number of random triangles
to partition the domain. At each iteration, one point is selected
within each triangle, and it is assigned a value equal to the aver-
age of the triangle’s vertices. This estimated value is compared
to the analysis (true) value. If the error is above a threshold,
this point will be used as a new vertex in the next iteration.
EnTK can dynamically dedicate resources as needed.

IV. EXPERIMENTS

We perform experiments to characterize the overheads in
EnTK and its weak and strong scaling performance. We then
measure the overheads of EnTK when executing, at scale, the
implementation of the two use cases described in §III.

We use four applications in our experiments: Sleep, Grom-
acs [12], Specfem [13], and Canalogs [14]. Sleep and Grom-
acs enable to control the duration of task execution and
compare EnTK overheads across task executes. Gromacs and
the NTL9 protein offer a reference benchmark for EnTK
weak and strong scalability, while Specfem and Canalogs are
required by the two use cases of this paper.

We perform our experiments on four CIs: XSEDE SuperMIC,
Stampede, Comet and ORNL Titan. We use four resources
for the characterization of EnTK overheads; Titan for the
characterization of the scalability; and Titan and SuperMIC
for the use case implementations.

A. Characterization of EnTK Performance

We use a prototype of EnTK to benchmark its performance,
providing a reference hardware configuration to support execu-
tion of up to $O(10^6)$ tasks. We then perform five experi-
ments to characterize the overheads of EnTK.

1) Performance of EnTK Prototype: We created a proto-
type of EnTK to instantiate multiple producers and consumers
of tasks. Each producer pushes tasks into RabbitMQ queues
and each consumer pulls tasks from these queues, passing
them to a empty RTS module. The prototype abstracts RTS
implementations and most of the architectural details described
in §II-B but implements the most computationally expensive
functionality of EnTK to manage tasks.

We benchmarked producers and consumers time, total ex-
ecution time, base memory consumption when the components
are instantiated, and peak memory consumption during the
execution. We benchmarked configurations with $10^6$ tasks and
a different number of producers, consumers, and queues.

Fig. 5 shows that tuning of the prototype can reduce the pro-
cessing time linearly, at the cost of increased memory usage.
Eight producers and consumers require 107 seconds to pro-
cess $10^6$ tasks, with a peak memory consumption of 3126MB.
Uneven distributions of producers and consumers resulted in
different execution times and memory consumption.

The execution model of EnTK can be tuned on the basis of
this benchmark, workload requirements, and hardware ca-
pabilities. This benchmark shows that the performance of the
core functionality of EnTK depends on the number of concur-
rently processed tasks. This has relevant implications for the
understanding of EnTK overheads and scalability.

Fig. 4. Adaptive unstructured analog algorithm workflow.

make assumptions that can drastically decrease the amount of
calculations needed while maintaining the prediction accuracy.

In this particular application, the simplest scenario is to com-
pute analogs at each grid point. This solution yields the highest
possible resolution, but it is very slow and most importantly,
often unnecessary. In fact, for some output variables, such as
temperature, the highest resolution of the analogs is required
only at specific regions, where drastic gradient changes occur.
Therefore, we implemented a dynamic iterative search process,
named the Adaptive Unstructured Analog (AUA) algorithm,
which generates analogs at specific points, and interpolates the
analog using an unstructured grid.

One of the main difficulties with this approach is that the al-
gorithm is iterative, and at each iteration it performs a variable
number of operations. EnTK addresses the resource manage-
ment challenges arising from such variations: by using an
iterative approach, the algorithm can generate higher predic-
tions for places where there is higher variability, using an
unstructured grid. EnTK also empowers us to achieve a higher
level of automation where the algorithm can allocate more
computing resources when a larger search space is required in
order to improve prediction accuracy. This is paramount when
not many good analogs are found in the previous search space.

Figure 4 shows the workflow of the AUA algorithm. The
workflow can be divided into 4 steps: initialization; preproces-
sing; iterative computation; and post-processing. The initializa-
tion step specifies the search space and the test space, and sets
up starting parameters for the AnEn. The preprocessing step
generates preparatory data that will be used in the subsequent
steps. The larger amount of computation occurs in the iterative
computation step: (1) Compute the unstructured grid to rep-
resent the locations where analogs need to be computed; (2)
Distribute the analog computation to the available computing
resources; (3) Aggregate the analogs computed at different lo-
cations; and (4) evaluate the predictions, and determine if the
termination condition is met. The termination condition could
be either the exhaustion of the computational resources avail-
able, or an overall prediction error below a specified threshold.
Finally, a post-processing task interpolates the analogs at each
location to generate the forecast solution.

We use the unstructured grid methodology based on the De-
launay triangulation method [11] to drive the algorithm. The
2) Overheads: Execution of an application using EnTK involves 92 events, including validation of the application and of the resource descriptions, submission of resource request, creation of tasks, and translation of tasks to and from the RTS. We designed five experiments to characterize the overhead added by EnTK to the time taken to execute an application, excluding the time taken by the resources to become available and by the RTS (RP) to retrieve and schedule the tasks.

We characterize EnTK overhead against five parameters that are likely to vary among applications: Task executable, duration and required cores; CI on which the application is executed; and structure of the application, i.e., the way in which tasks are grouped into stages and stages into pipelines. Accordingly, we performed five experiments where we measured the overhead of EnTK for each one of these parameters. We decomposed the total runtime into several time components, measuring, among these, the following overheads:

- **EnTK Setup Overhead**: Time taken to setup the messaging infrastructure, instantiate all the components and subcomponents, and validate the application and resource descriptions.
- **EnTK Management Overhead**: Time taken to process the application, translate tasks from and to RTS-specific objects, and communicate pipelines, stages, tasks and control messages.
- **EnTK Tear-Down Overhead**: Time taken to cancel all EnTK components and subcomponents, and shutdown the messaging infrastructure.
- **RTS Execution Overhead**: Time taken by the RTS to submit and manage the execution of the tasks.
- **RTS Tear-Down Overhead**: Time taken by the RTS to cancel its components and to shutdown.
- **Task Execution Time**: Time taken by the task executables to run on the CI.

**Experiment 1: Task Executable**: We run two applications on the XSEDE SuperMIC cluster, each with 1 pipeline, 1 stage per pipeline, 16 tasks per stage. The first application runs the Gromacs `mdrun` task executable; the second the `sleep` executable. The execution time of each task is set to \( \approx 250 \) seconds. We measure Task Execution Time and all the EnTK and RTS overheads defined above.

Fig. 5 shows that EnTK Setup Overhead is negligible for both task executables but that EnTK Tear-Down Overhead differs. We attribute this difference to the Python termination of processes and threads, independent from task executables. We observe Python taking variable amount of time to cancel processes and threads, with peaks of 80 seconds per termination.

EnTK Management Overhead has an almost identical value for the two task executables. This is consistent with a core element of EnTK design: decoupling task management from task execution. EnTK manages tasks until they are passed on to the RTS, playing no role in their actual execution. This contributes to the portability of EnTK across diverse scientific domains and CIs.

`mdrun` and `sleep` are task executables. Applications consist of 1 pipeline, 1 stage per pipeline and 16 tasks per stage, executed on XSEDE SuperMIC.
sleep executes for 100 seconds. We measure EnTK Management Overhead and Task Execution Time.

Fig. 8 shows that, as per experiment design, Task Execution Time is 100 seconds on all four CIs. EnTK Management Overhead is ≈10 seconds on SuperMIC, Stampede and Comet whereas it is ≈2 seconds on Titan. We attribute this difference to the host on which EnTK was executed: a virtual machine hosted at TACC for the applications executed on the XSEDE clusters, an ORNL login node for those executed on Titan. This is consistent with the performance characterization in Fig. 5 and with EnTK design: Application management is performed on the user host while task execution on the target CI. EnTK avoids running system software on HPC login nodes when policies do not allow it.

**Experiment 4: Application Structure**: In experiments 1–3, we used the same structure for all the applications, i.e., 1 pipeline, 1 stage per pipeline, 16 tasks per stage. In this experiment, we run three applications on XSEDE SuperMIC with: (1) 16 pipelines, 1 stage per pipeline, 1 task per stage; (2) 1 pipeline, 16 stages per pipeline, 1 task per stage; and (3) 1 pipeline, 1 stage per pipeline, 16 tasks per stage. The task executable sleep executes for 100 seconds in all the applications. We use application #3 as baseline for comparison with the previous three experiments, and we measure EnTK Management Overhead and Task Execution Time.

Fig. 9 shows that with 16 pipelines (first) and 16 tasks (last), all the tasks execute concurrently and hence Task Execution Time is ≈100 seconds. However, in the case of 16 stages, tasks execute sequentially, resulting in Task Execution Time of ≈1600 seconds. Although Task Execution Time varies depending on the structure of the application, EnTK Management Overhead is stable at ≈10 seconds. This is consistent with the results of the previous experiments and the decoupling between task management and execution.

**Experiment 5: Cores per task**: In all the previous experiments, we execute tasks on one core but scientific applications often require also multi-core tasks. In this experiment, we run five applications on XSEDE SuperMIC, each with 1 pipeline, 1 stage per pipeline, 16 tasks per stage and the Gromacs mdrun task executable. We use 1, 2, 4, 8 and 16 cores per task and we set task duration at ≈250 seconds on 1 core. We request (task*cores_per_task) cores to run all the tasks concurrently as in the previous experiments. We measure EnTK Management Overhead and Task Execution Time.

Fig. 10 shows that EnTK Management Overhead is ≈10 seconds, independent from the number of cores used per task. As expected, Task Execution Time decreases with the number of cores due to parallelization of execution. This is consistent with the results of the previous experiments and the decoupling between task management and execution.

Experiments 1–5 confirm that EnTK Management Overhead does not vary with any of the five parameters of task execution we measured. This validates our design and implementation against our requirements for a toolkit that can be used in various scientific domains and with different task executables across heterogeneous CIs.
In absolute terms, EnTK Management Overhead is always around 10 seconds, becoming relevant only when considering corner cases in which applications have a small amount of 1 second-long tasks to execute. Experiment 3 shows that these corner cases could be handled by running EnTK on a host with a suitable amount of resources.

B. Scalability

We perform two experiments to characterize weak and strong scalability of EnTK. As with Experiment 1–5, we measure and compare EnTK Management Overhead and Task Execution Time. Weak scaling relates these measures to the amount of concurrency used to execute the application’s tasks; Strong scaling to the amount of serialization. Weak and strong scalability are a measure of EnTK’s performance, not of the RTS or CI. Experiments 1–5 shows that measuring EnTK overheads isolates EnTK performance.

1) Strong scalability: The strong scaling experiment measures EnTK Management Overhead and Task Execution Time maintaining the number of cores constant and increasing the number of tasks executed. We run four applications on Titan, each with 1 pipeline, 1 stage per pipeline, and 512, 1024, 2048, or 4096 tasks per stage. Each task executable is Gromacs mdrun, configured to use 1 core and execute for \( \approx 600 \) seconds. The total amount of resources acquired is 512 cores.

We designed this experiment based on the requirements of the two use cases of this paper. We used: Titan and up to 4096 tasks as the use cases require between 1000 and 4096 tasks; 1-core tasks because the number of cores per tasks does not affect EnTK Management Overhead as we show in Experiment 5; and task duration of \( \approx 600 \) seconds as RP is optimized to execute tasks of at least 600 seconds, when managing the execution of more than 4000 tasks.

Fig. 11 shows that Task Execution Time increases linearly with the number of tasks. The similarity in the duration of each task execution and the execution concurrency of 512 tasks explain this linear increase. EnTK Management Overhead remains constant at \( \approx 2 \) seconds as at every point in time, EnTK does not manage more than 512 concurrent tasks.

Experiment 3 and EnTK design suggest that our experimental design did not use enough resources on the Titan’s login node to show an increase in EnTK Management Overhead. The walltime limit on Titan for a queue with 512 cores is 2 hours, making impossible to run more than 4096 tasks.

Fig. 12 shows that Task Execution Time remains constant up to 1024 tasks/cores but then increases gradually by 100 seconds up to 4096. Analysis of the RTS profiles shows that this behavior is due to delays in the Executor module of the RTS Agent. However, the EnTK Management Overhead remains constant till 2048 tasks as the number of tasks are too small to cause a variation. The overhead, then, increases linearly between 2048 and 4096 tasks: With the increase of the number of concurrent tasks, EnTK requires more resources and starts to strain the resources of the host on which it is executed. This is consistent with what observed in the Strong scaling experiment, where EnTK had to manage only 512 concurrent tasks and with Fig. 5.

C. Use Cases at Scale

We use EnTK to implement and execute at scale the most computationally intensive and fault-prone step of the tomography workflow, and the full adaptive analog workflow (III).

1) Seismic inversion: We use EnTK to encode the forward simulations of the seismic tomography workflow described in [III-A] and depicted in Fig. 3. These simulations account for more than 90% of the total computation time of the current
implementation of the workflow, requiring 384 nodes for each earthquake simulation. When earthquakes are concurrently simulated, they require a sizable portion of Titan and incur a high rate of failures. Without EnTK, these failures result in manual resubmission of computations, adding a significant overhead due to queue wait time on user intervention.

We characterize the scalability of forward simulations with EnTK by running experiments with a varying number of tasks, where each task uses 384 nodes/6144 cores to forward simulate one earthquake. Understanding this scaling behavior contributes to optimize the execution of the whole workflow, both by limiting failure and enabling fault-tolerance without manual intervention. Ultimately, this will result in an increase of the overall efficiency of resource utilization and in a reduction of the time to completion.

The current implementation of forward simulations causes heavy I/O on a shared file system (III-A). This overloads the file system, inducing crashes or requiring termination of the simulations. EnTK and RP utilize pilots to sequentialize a subset of the simulations, reducing the concurrency of their execution and without having to go through Titan’s queue multiple times. This is done by reducing the number of cores and increasing the walltime requested for the pilot.

Fig. 13 shows that increasing concurrency leads to a linear reduction of Task Execution Time, with a minimum of \( \approx 180 \) seconds. Interestingly, reducing concurrency eliminates failures: we encountered no failures in executions with up to \( 2^4 \) concurrent tasks and 6144 nodes. At \( 2^5 \) concurrent tasks and 12288 nodes, 50% of the tasks failed due to runtime issues.

EnTK automatically resubmitted failed tasks until they were successfully executed. In the run with \( 2^5 \) tasks, EnTK attempted to run a total of 157 tasks. The resulting Task Execution Time was \( \approx 360 \) seconds, similar to that of a run with \( 2^4 \) concurrent tasks (Fig. 13).

EnTK and RP enable reasoning and benchmarking the concurrency of an execution without any change in the executable code. This gives insight on how to tailor a given computational campaign on a specific CI. The insight gained via our experiments can be immediately used in production: On Titan, forward simulations are best executed with \( 2^4 \) concurrent tasks. Further, fault-tolerance has an immediate impact on production runs, eliminating one of the most limiting factor of the previous implementation of the workflow.

2) Meteorological Probabilistic Forecasts: We use EnTK to implement the AUA algorithm to iteratively and dynamically identify locations of the analogs. We also implement the status quo method of generating these analogs, i.e., random selection of locations in each iteration. We perform experiments to compare the two implementations and observe the speedup of the proposed algorithm. We repeat the experiment 30 times for statistical accuracy, initializing both implementations using the same initial random locations.

Fig. 14 shows the prediction maps and errors obtained from the two implementations. With 1800 locations calculated for both prediction maps (Fig. 14(b), Fig. 14(c)), the AUA algorithm generates a map with certain areas that have a better representation of the analysis than the map generated by a random selection of pixels.

The box plot in Fig. 14(d) shows the distribution of the errors for both implementations. The error converges faster in the AUA algorithm than in the random selection. The total amount of potential locations (pixels) is 262972; thus both implementations use a small fraction of the available locations but the AUA algorithm is automatically steering the computation at each iteration. EnTK and RP avoid the usual shortcoming of this approach: The evaluation required by the steering can be implemented as a task and iterations do not wait in the HPC queue, even if their number is unknown before execution. These results suggest that the AUA algorithm over random selection of points is well suited for very large domains.

V. RELATED WORK

Ensemble applications are becoming increasingly common in molecular sciences [11, 15, 24, 3], climate sciences [16, 17, 18, 19], bio-informatics [20] and uncertainty quantification [21], to name a few domains.
Many tools have been developed to run scientific multi-task applications on HPC systems. Among these, workflow systems like Pegasus [22], Kepler [23], or Triana [24] are general-purpose frameworks that provide encompassing functionalities. These systems have relatively monolithic design and assume end-to-end executions. This makes it difficult to tailor them to specific user requirements, and redundant for applications with comparatively simple workflows.

Among other workflow systems, Swift [25] enables execution on HPC systems but requires users to program their applications in a domain specific language. Airflow [26], Taskflow [27] and Luigi [28] provide a simple interface, automation and visualization but are not designed to utilize shared HPC systems or run scientific applications.

EnTK contributes to the research that focuses on the specificity of ensemble applications. These applications can be expressed as workflows but their distinguishing patterns permit a simplification of the graph structure while requiring better handling of task parallelism and runtime adaptivity [29]. Systems like Tigres [30] focus on usability while EnTK is designed to support tailored APIs and to reduce the complexities of resource and execution management typically associated to ensemble applications and HPC systems.

VI. Conclusion

The results of our experiments show that the design and implementation of EnTK meet the requirements of diverse use cases. The performance of EnTK is shown to be invariant of workload and platform. EnTK was shown to have ideal weak and strong scaling up to currently required scales. Importantly, any deviation from ideal scaling was explicable, and the causes are candidates for future enhancements. The use of EnTK with Specfem at large scales on Titan at ORNL led to unprecedented reductions in time-to-completion, insulation against failures (e.g., hardware and software), and improved reliability.

Abstractions exposed by EnTK permit algorithmic innovations. For the meteorological probabilistic forecast use case, the independence from direct resource management permits new adaptive formulations of the Analog Ensemble method, which in turn leads to improved accuracy in predictions, with reduced time to completion and usage of compute resources.

We provide initial demonstrations of how EnTK has facilitated the full potential of ensemble methods (“power of many”). EnTK will allow similar methodological advances for other ensemble applications, which have so far been hindered by the lack of suitable tools. EnTK is also a validation of the building block approach to middleware: it is demonstrably extensible to application specific frameworks in the upward direction [4], as well as being agnostic to the specific RTS below.

Having provided fundamental advances for ensemble applications at the largest scales currently available (~66% of Titans’ nodes), EnTK will be engineered to provide a pathway to pre-exascale levels without disruption in production capabilities for users of Titan. Specifically, EnTK will provide capabilities for: (i) dynamic mapping of tasks onto heterogeneous resources, and (ii) and adaptive execution strategies to enable optimal resource utilization.

Acknowledgements

This work is supported by NSF ICER 1639694, DOE ASCR [de-sc0016280] and an INCITE award to J.T. Resources from the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory are supported by the Office of Science of the U.S. Department of Energy, Contract No. DE-AC05-00OR22725.

References

[1] T. E. Cheatham III and D. B. Roe, “The impact of heterogeneous computing on workflows for biomolecular simulation and analysis,” *Computing in Science & Engineering*, vol. 17, no. 2, pp. 30–39, 2015.
[2] J. D. Chodera and F. Noé, “Markov state models of biomolecular conformational dynamics,” *Current opinion in structural biology*, vol. 25, pp. 135–144, 2014.
[3] F. Noé, C. Schütte, E. Vanden-Eijnden, L. Reich, and T. R. Weikl, “Constructing the equilibrium ensemble of folding pathways from short off-equilibrium simulations,” *Proceedings of the National Academy of Sciences*, vol. 106, no. 45, pp. 19011–19016, 2009.
[4] “Designing workflow systems using building blocks.” [https://arxiv.org/ab/1609.03484](https://arxiv.org/ab/1609.03484)
[5] “Towards common components for open workflow systems,” [https://arxiv.org/abs/1710.06774](https://arxiv.org/abs/1710.06774)
[6] “EnTK,” [https://github.com/radical-cybertools/radical-entk](https://github.com/radical-cybertools/radical-entk)
[7] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, “Executing dynamic and heterogeneous workflows on super computers,” 2017, (under review) [http://arxiv.org/abs/1512.08194](http://arxiv.org/abs/1512.08194)
[8] J. Virieux and S. Operto, “An overview of full-waveform inversion in exploration geophysics,” *GEOPHYSICS*, vol. 74, no. 6, pp. WCC1–WCC26, Nov. 2009. [Online]. Available: [http://dx.doi.org/10.1190/1.3238367](http://dx.doi.org/10.1190/1.3238367)
[9] E. Bozdag, D. Peter, M. Lefebvre, D. Komatitsch, J. Tromp, J. Hill, N. Podhorszki, and D. Pugmire, “Global adjoint tomography: first-generation model,” *Geophysical Journal International*, vol. 207, no. 3, pp. 1739–1766, 2016. [Online]. Available: [http://dx.doi.org/10.1093/gji/ggw356](http://dx.doi.org/10.1093/gji/ggw356)
[10] G. Cervone, L. Clemente-Harding, S. Alessandrini, and L. Delle Monache, “Short-term photovoltaic power forecasting using artificial neural networks and an analog ensemble,” *Renewable Energy*, vol. 108, pp. 274–286, 2017.
[11] S. Lertrattanapanich and N. K. Bose, “High resolution image formation from low resolution frames using delaunay triangulation,” *IEEE Transactions on Image Processing*, vol. 11, no. 12, pp. 1427–1441, 2002.
[12] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers,” *SoftwareX*, vol. 1, pp. 19–25, 2015.
[13] D. Komatitsch and J. Tromp, “Spectral-element simulations of global seismic wave propagation—i, validation,” *Geophysical Journal International*, vol. 149, no. 2, pp. 390–412, 2002.
[14] S. Alessandrini, L. Delle Monache, S. Sperati, and G. Cervone, “An analog ensemble for short-term probabilistic solar power forecast,” *Applied energy*, vol. 157, pp. 95–110, 2015.
[15] Y. Sugita and Y. Okamoto, “Replica-exchange molecular dynamics method for protein folding,” *Chemical physics letters*, vol. 314, no. 1, pp. 141–151, 1999.
[16] H. Wan, P. J. Rasch, K. Zhang, Y. Qian, H. Yan, and C. Zhao, “Short ensembles: an efficient method for discerning climate-relevant sensitivities in atmospheric general circulation models,” *Geoscientific Model Development*, vol. 7, no. 5, pp. 1961–1977, 2014.
[17] P. Bauer, A. Thorpe, and G. Brunet, “The quiet revolution of numerical weather prediction,” *Nature*, vol. 525, no. 7567, pp. 47–55, 2015.
[18] G. Cervone, F. Franzese, Y. Ezber, and Z. Boybeyi, “Risk assessment of atmospheric emissions using machine learning,” *Natural Hazards and Earth System Sciences*, vol. 8, no. 5, pp. 991–1000, 2008.
[19] T. M. Hamill, M. J. Brennan, B. Brown, M. DeMaria, E. N. Rappaport, and Z. Toth, “Future of ensemble-based hurricane forecast products,” *Bull. Amer. Meteor. Soc.*, 2010.
[20] J. Martin, V. M. Bruno, Z. Fang, X. Meng, M. Blow, T. Zhang, G. Sherlock, M. Snyder, and Z. Wang, “Rnnotator: an automated de novo transcriptome assembly pipeline from stranded RNA-Seq reads,” *BMC Genomics*, vol. 11, no. 1, p. 663, 2010.

[21] S. N. L. Los Alamos National Laboratory, National Energy Research Scientific Computing Center, “APEX Workflows,” 2016.

[22] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good et al., “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[23] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the kepler system,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[24] I. Taylor, M. Shields, I. Wang, and A. Harrison, “The triana workflow environment: Architecture and applications,” *Workflows for e-Science*, pp. 320–339, 2007.

[25] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, “Swift: A language for distributed parallel scripting,” *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011.

[26] “Airflow,” https://airflow.incubator.apache.org/.

[27] “Taskflow,” https://wiki.openstack.org/wiki/TaskFlow.

[28] “Luigi,” https://github.com/spotify/luigi.

[29] V. Balasubramanian, A. Trekalin, O. Weidner, and S. Jha, “Ensemble Toolkit: Scalable and flexible execution of ensembles of tasks,” 2016, 45th Int. Conference on Parallel Processing (ICPP).

[30] V. Hendrix, J. Fox, D. Ghoshal, and L. Ramakrishnan, “Tigres workflow library: Supporting scientific pipelines on hpc systems,” in *Cluster, Cloud and Grid Computing (CCGRID), 16th IEEE/ACM Int. Symposium on*, 2016, pp. 146–155.