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

Vivek Balasubramanian[*¶], Matteo Turilli [*¶], Weiming Hu[†], Matthieu Lefebvre[‡], Wenjie Lei[‡],
Ryan Modrak[‡], 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 exemplar use cases: seismic inversion and adaptive analog ensembles. We perform nine experiments, characterizing EnTK overheads, strong and weak scalability, and the performance of the 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 $O(10^4)$ tasks; and (iv) task-level fault tolerance. We discuss novel computational capabilities that EnTK enables and the scientific advantages arising thereof. We propose EnTK as an important 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 an application with a large but single task, and less on applications 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 (HPC) machine.

When the collective outcome of a set of tasks is of importance, this set is defined to be an ensemble. Individual tasks within the set might be coupled or uncoupled. When coupled, 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) applications, where tasks are typically identical, uncoupled, idempotent and can be executed in any order. Individual tasks within the ensemble may also vary in their type, executable, and resource requirements.

The number and type of applications that can be formulated as ensembles is vast and span many scientific domains. 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 machines presents three main challenges: (1) encoding scientific problems into algorithms that are amenable to distributed and coordinated solution; (2) sizing, acquiring, and managing resources for the execution; and (3) managing the execution of the ensemble. Encoding scientific problems into ensembles requires describing tasks with heterogeneous properties, specifying whether and how tasks are grouped into partitions of the ensemble, and defining an ordering among tasks and partitions. Sizing resources for the ensemble depends on calculating the resources needed by each task and those needed by the set of tasks that can be executed concurrently.

Often, there is a friction between the resource requirements of an ensemble and the traditional resource management of HPC machines. Each task of the ensemble has to be queued onto a HPC machine, incurring a long queue waiting time that adds up to the total time to completion of the ensemble. Usually, at least one compute node of the HPC machine has to be requested for each task, and often for at least one hour, even when tasks may require fewer resources for a shorter duration. Finally, distributing the execution of an ensemble requires tailored coordination and communication infrastructure and protocols, not made readily available to the user via the HPC software provision. These factors make using HPC resources for ensemble applications challenging, when not unfeasible.

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 building blocks approach for the design, development and integration of middleware [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].

This paper offer four main contributions: (1) a description of the design, architecture and implementation of EnTK (§II); (2) a characterization of EnTK overheads on different HPC computing infrastructures (CI) for a variety of task types (§IV-A); (3) an analysis of EnTK weak and strong scaling on a leadership-class HPC CI (§IV-B); and (4) the support of two ensemble-based scientific applications with different characteristics and requirements (§IV-C). This shows that EnTK can support diverse types of ensemble applications, at scale and on several HPC CIs, introducing acceptable overheads. As such, we propose EnTK as an important 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 GitHub repository [6].

A. Requirements

As seen in §I, 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 about 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) be performance independent of the type of CI.

The use cases motivating EnTK require execution of up to $O(10^4)$ ensemble members (tasks). This poses many challenges, that need to be addressed by EnTK and a runtime system (RTS). At this scale, EnTK has to reliably enable sustained task submission rate, tracking of executing tasks and clean termination of tasks. The RTS has instead to integrate with

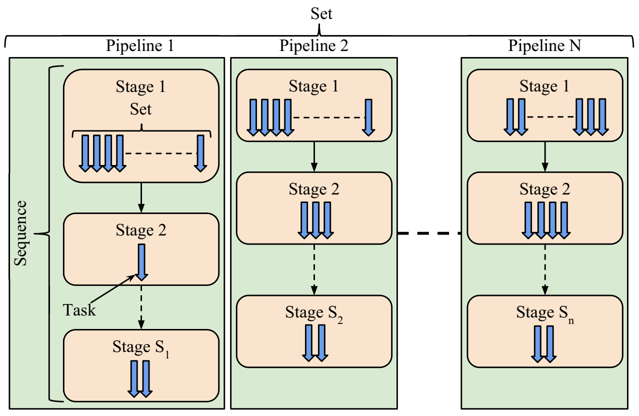


Fig. 1: Diagrammatic representation of an application consisting of a set of pipelines with varying number of stages and tasks.

suitable MPI layers, setting up the execution environment for heterogeneous tasks, managing their data requirements and scheduling tasks across multiple resource partitions. Together, EnTK and RTS have to ensure full resource utilization across the ensemble execution time.

EnTK has to be fault-tolerant at scale, i.e., when both the probability and cost of failure increase. Currently, EnTK is required to support resubmission of failed tasks, without application checkpointing, and restarting of failed RTS and components. In this way, applications can be executed on 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

1) Application Model: We model applications by combining 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 pipelines, stages, and tasks (PST). The application consists of a set of pipelines, where each pipeline is a list of stages, and each stage is 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 PST descriptions can be extended to account for dependencies among groups of pipelines in terms of lists of sets of pipelines. Further, the specification of branches in the execution flow of applications does not require to alter the PST semantics: Branching events can be specified as tasks where a decision is made about the runtime flow. For example, a

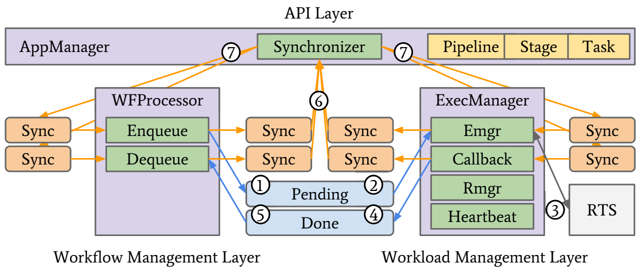


Fig. 2: EnTK architecture and execution model. Components' (purple) subcomponents (green) use queues (blue and orange) to communicate and coordinate the execution of an application via a chosen RTS (gray).

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 codify PST descriptions. The Workflow Management layer retrieves information from the user about available CIs, initializes EnTK, and holds the global state of the application during execution. The Workload Management layer acquires resources via the RTS.

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 Workload 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.

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 from 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.

4) Failure Model: We consider four main sources of failure: EnTK components, RTS, CI, and task executables. All state updates in EnTK are transactional, hence any EnTK component that fails can be restarted at runtime without losing information about ongoing execution. In case of full failure, EnTK can reacquire upon restarting information about the state of the execution up to the latest successful transaction before the failure. Information is synced on disk and hooks are in place to use an external database.

Both the RTS and the CI are considered black boxes. Partial failures of their subcomponents at runtime are assumed to be handled locally, not globally by EnTK. Upon full failure of the RTS, EnTK assumes all the pilot resources and the tasks undergoing execution are lost. EnTK purges any process left over by the failed RTS, starts a new instance of the RTS, acquires new pilot resources, and restarts executing the ensemble until completion. Users can configure the number of times a RTS is restarted during the execution of a single ensemble.

CI-level failures are reported to EnTK indirectly, either as failed pilots or failed tasks. Both pilots and tasks can be restarted, up to a certain number of times configured by the user. Failures are logged and reported to the user at runtime for live or postmortem analysis. EnTK design enables collection of information from both the RTS and the resources via APIs. When RTS and CI expose information about, for example, OS-level faults, application checkpoint, or hardware faults, EnTK can implement advanced fault-tolerant capabilities. Currently, these capabilities are not required by our use cases and application checkpoint is not performed by their executables.

C. Implementation

EnTK is implemented in Python, uses RabbitMQ message queuing system and the RADICAL-Pilot (RP) 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

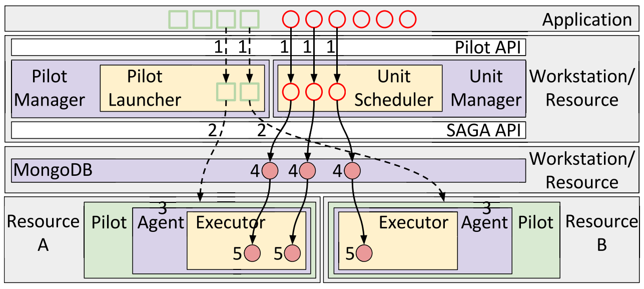


Fig. 3: RADICAL-Pilot (RP) architecture and execution model. Gray: machines; green: pilot; purple: modules; yellow: components; red: tasks.

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.

D. *Runtime System*

Currently, EnTK uses RADICAL-Pilot (RP) [7], [8] as the RTS. RP is a runtime system designed to execute ensemble applications via pilots. Pilots provide a multi-stage execution mechanism: Resources are acquired via a placeholder job and subsequently used to execute the application's tasks. When a pilot is submitted to a CI as a job, it waits in the CI's queue until the requested resources become available. At that point, the CI's scheduler bootstraps the job on the CI's compute nodes. RP does not attempt to 'game' the CI's scheduler: Once queued, the pilot is managed according to the CI's policies.

RP is a distributed system with four modules: PilotManager, UnitManager, Agent and DB (Fig. 3, purple boxes). PilotManager, UnitManager and Agent have multiple components (Fig. 3, yellow boxes), isolated into separate processes. Components are stateless and some of them can be instantiated concurrently to enable RP to manage multiple pilots and tasks at the same time. Concurrent components are coordinated via a dedicated communication mesh, scaling throughput and enabling tolerance to failing components.

Workloads and pilots are described via the Pilot API and passed to the RP runtime system (Fig. 3, 1). The PilotManager submits pilots as jobs (or virtual machines or containers) to one or more CIs via the SAGA API (Fig. 3, 2). The SAGA API implements an adapter for each supported type of CI, exposing uniform methods for job and data management. Once a pilot becomes active on a CI, it bootstraps the Agent module (Fig. 3, 3). The UnitManager schedules each task to an Agent (Fig. 3, 4) via a queue on a MongoDB instance. Each Agent pulls its tasks from the DB module (Fig. 3, 5), scheduling them on the Executor. The Executor sets up the task's execution environment and then spawns the task for execution.

When required, the input data of a task are either pushed to the Agent or pulled from the Agent, depending on data locality and sharing requirements. Similarly, the output data of the task are staged out by the Agent and UnitManager to a specified destination, e.g., a filesystem accessible by the Agent or the user workstation. Both input and output staging are optional, depending on the requirements of the tasks. The actual file transfers are enacted via SAGA, and currently support `(gsi)-scp`, `(gsi)-sftp`, `Globus Online`, and local and shared filesystem operations via `cp`. Consequently, the size of the data along with network bandwidth and latency or filesystem performance determine the data staging durations and are independent of the performance of the RTS.

III. Use Cases

To help understand the initial scope and design of EnTK, we describe two motivating use cases, focusing on their computational and functional requirements.

A. *Seismic Inversion*

Inversion of full-waveform, wide-bandwidth seismic data [9] is one of the most powerful tomographic technique to study the Earth's interior. Scaling this technique is challenging, mostly because of the amount of computational resources and human labor it needs. These challenges require a more automated approach to the management and execution of the workflow, such as the one implemented by the EnTK.

Figure 4 shows a high level view of the workflow we use to perform seismic tomography. We record seismic data (i.e., seismograms) as time series of a physical quantity, like displacement, velocity, acceleration or pressure. Our goal is to iteratively minimize differences between observed 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 gets closer to reality.

We run the workflow of Figure 4 in production, assimilating data from about 1,000 earthquakes. Forward (1) and adjoint (3) simulations are the most computationally expensive parts of the workflow, each running on 384 GPUs for a total of 10 millions core-hours per iteration. Data processing (2) is relatively computationally inexpensive, utilizing about 48,000 core-hours in each iteration. Post-processing (4) takes about 10,000 core-hours while optimization (5) takes about 1 million core-hours.

Currently, each part of the workflow relies on a Python-based proto-workflow management system. However, scaling to higher resolutions and assimilating data from 6,000 earthquakes requires more automation to ensure reliability, minimize errors at the user level and lower the overall time to solution. Further, we need to interleave simulation tasks with data-processing tasks, each requiring respectively leadership-scale systems and moderately sized clusters. During the workflow execution, we need to save between 0.15 to 1.5GB per seismogram.

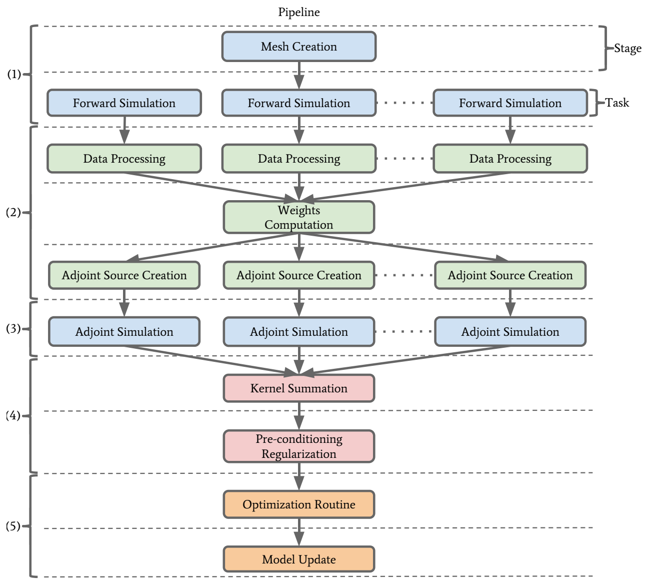


Fig. 4: Simplified seismic tomography workflow encoded into the PST model.

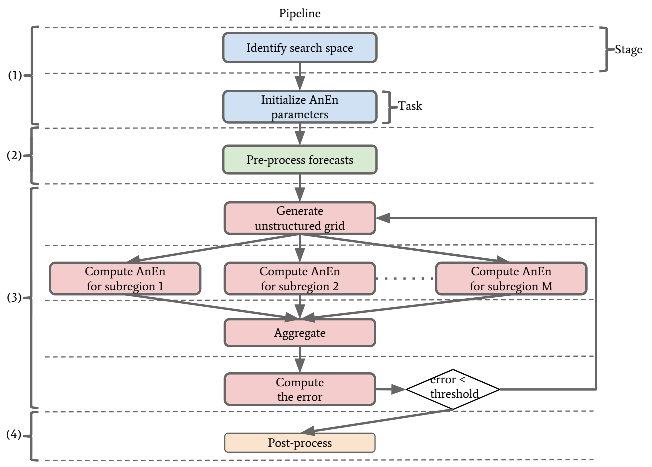
Ensemble-based applications are particularly well-suited to encode the seismic tomography workflow. In EnTK, we can describe the simulation and analysis phases as stages of a pipeline, avoiding to use dedicated MPI application to execute multiple simulation concurrently. EnTK and RP also allow the execution 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 we experienced with full-fledged workflow systems. We encode data dependencies and staging directives via the EnTK API, enabling data management at runtime on per-task, stage, and pipeline basis.

B. High Resolution Meteorological Probabilistic Forecasts

We implemented the Analog Ensemble (AnEn) [10] methodology to generate high-resolution, probabilistic forecasts for environmental variables like temperature or cloud cover. We used relationships between current and past forecasts from the Weather Research and Forecast model (WRF) data to generate an analog ensemble for a given time and location. Our implementation finds the most similar historical forecasts, based on a similarity metric. The observations associated with the most similar past forecasts are used as analogs.

We implemented a dynamic iterative search process, named the Adaptive Unstructured Analog (AUA) algorithm, which generates analogs at specific geographical locations, and interpolates the analogs using an unstructured grid. In this way, we avoid computing analogs at every available location, noting that for some output variables, such as temperature, the highest resolution of the analogs is required only at specific regions, where drastic gradient changes occur.

The AUA algorithm is iterative, and at each iteration it performs a variable number of operations. EnTK addresses the resource management challenges arising from such variations



Pipeline
Stage
Task
(1)
Identify search space
Initialize AnEn parameters
(2)
Pre-process forecasts
(3)
Generate unstructured grid
Compute AnEn for subregion 1
Compute AnEn for subregion 2
Compute AnEn for subregion M
Aggregate
Compute the error
error < threshold
(4)
Post-process

Fig. 5: Adaptive unstructured analog algorithm workflow encoded into the PST model.

by allocating diverse amount of computing resources, depending on the size of the search space required to achieve the desired prediction accuracy.

Figure 5 shows the workflow of the AUA algorithm. The initialization step specifies the search space and the test space, and sets up starting parameters for the AnEn. The preprocessing step generates preparatory data for the subsequent steps. The largest amount of computation occurs in the iterative computation step where analogs are computed and aggregated multiple times until the available resources are exhausted, or the prediction error is below a given threshold. The post- processing task interpolates the analogs to generate the forecast solution.

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).

IV. Experiments

We perform experiments to characterize the overheads of 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, Gromacs [11], Specfem [12], and Canalogs [13]. Sleep and Gromacs enable control of the duration of task execution and to compare EnTK overheads across task executables. Gromacs and the NTL9 protein serve as workload 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 applications.

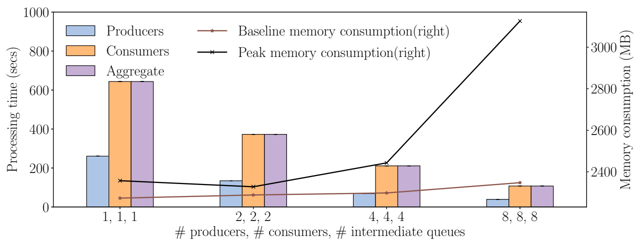


Fig. 6: Execution time and memory consumed by EnTK prototype with multiple producers and consumers and 10^6 tasks.

A. Characterization of EnTK Performance

We use a prototype of EnTK to benchmark its performance, providing a reference hardware configuration to support execution of up to $O(10^6)$ tasks. We then perform four experiments to characterize the overheads of EnTK.

1) Performance of EnTK Prototype: We prototyped the most computationally expensive functionality 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 an empty RTS module. We benchmarked configurations with 10^6 tasks and a different number of producers, consumers, and queues, measuring: producers and consumers time; total execution time; base memory consumption when the components are instantiated; and peak memory consumption during the execution.

Fig. 6 shows that tuning of the prototype can reduce the processing time linearly, at the cost of increased memory usage. Eight producers and consumers require 107 seconds to process 10^6 tasks, with a peak memory consumption of 3,126MB. Uneven distributions of producers and consumers resulted in lower efficiencies than when using even distributions.

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

2) Overheads, Data Staging and Task Execution Time: We characterize EnTK overhead against four parameters that are likely to vary among applications: Task executable; task duration; 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. We measured the overheads that dominate EnTK and RTS runtime alongside the total task execution time and, when required, the total data staging time:

- **EnTK Setup Overhead**: Time taken to setup the messaging infrastructure, instantiate components and subcomponents, and validate 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 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.
- **Data Staging Time**: Time taken to copy data between tasks using the functionality available on the resource (in this case, the Unix POSIX `cp` command).
- **Task Execution Time**: Time taken by the task executables to run on the CI.

We designed four experiments (Table I) to characterize the overheads added by EnTK and the RP RTS to the time taken to execute an application, excluding the time taken by the resources to become available. These experiments execute applications with different task executable (Experiment 1, Fig. 7a); task duration (Experiment 2, Fig. 7b); CI (Experiment 3, Fig. 7c); and application structure, i.e., the number of pipelines, stages and tasks per application (Experiment 4, Fig. 7d).

Fig. 7 shows that EnTK Setup Overhead is ≈0.1s across Experiment 1, 2, 4, and ≈0.05s for Titan in Experiment 3. We attribute this difference to the host from which EnTK was executed. All the experiments on XSEDE machines were performed from the same virtual machine (VM) hosted at TACC, while experiments on Titan had to be performed from an ORNL login node. The ORNL login nodes have faster memory and CPU than the VM.

Fig. 7 shows a similar behavior between EnTK Setup Overhead and EnTK Management Overhead. EnTK Management Overhead measures ≈10s on all the runs but those performed on Titan where it measures ≈3s. Also in this case, we attribute this difference to the performance of the VM and login nodes from which EnTK was executed.

In Fig. 7, EnTK Tear-Down Overhead and RTS Tear-Down Overhead vary across all four experiments with values between ≈1 and ≈10s for EnTK Tear-Down Overhead and ≈3 and ≈80s for RTS Tear-Down Overhead. We attribute these variations to the time taken by Python to terminate processes and threads. The higher values of RTS Tear-Down Overhead are expected as RP uses significantly more processes and threads than EnTK.

We explain the variations of RTS Overhead in Fig. 7 by noticing that, at runtime, RP initiates communications between the CI and a remote database, and reads and writes to the shared file system of the CI to create the execution environment of each task. Further, RP uses third party tools to distribute the execution of tasks across compute nodes. A detailed analysis of the interplay among network latency, I/O performance, and the performance of third party tools and libraries is beyond the scope of this paper. This is consistent with EnTK design: the RTS (RP in this case) is assumed to be a black box.

Fig. 7 shows that for tasks executing more than 1s, RP overheads have little impact on Task Execution time: As per experiment design, executables of Experiment 1 (Fig. 7a) run for ≈300s and those of Experiment 3 (Fig. 7c) for ≈100s on all four CIs. In Experiment 2 (Fig. 7b), tasks set to run for 1s, run for ≈5s due to RP overhead but tasks set to run for 10s, 100s, and 1,000s run in about that amount of time. In Experiment

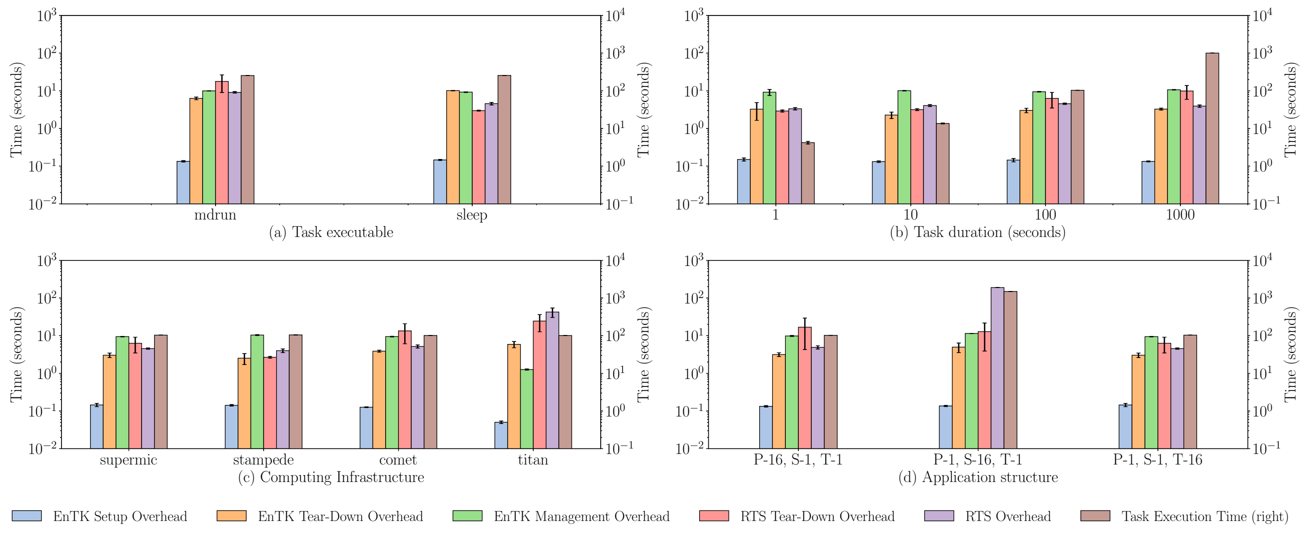


Fig. 7: Overheads and Task Execution Time as function of (a) Task Executable (Experiment 1), (b) Task Duration (Experiment 2) (c) Computing Infrastructure (Experiment 3) (d) Application Structure (Experiment 4).

TABLE I: Parameters of the experiments plotted in Figure 7.

ID	Computing Infrastructure (CI)	Pipeline, Stage, Task	Executable	Task Duration	Data
1	SuperMIC	(1,1,16)	`mdrun`, `sleep`	300s	TDB
2	SuperMIC	(1,1,16)	`sleep`	1s, 10s, 100s, 1,000s	None
3	SuperMIC, Stampede, Comet, Titan	(1,1,16)	`sleep`	100s	None
4	SuperMIC	(16,1,1), (1,16,1), (1,1,16)	`sleep`	100s	None

4 (Fig. 7c), for runs with 16 pipelines and 16 tasks, all the tasks execute concurrently and hence Task Execution Time is ≈100s. However, with 16 stages, tasks execute sequentially, resulting in Task Execution Time of ≈1,600s.

EnTK setup, management, and tear-down overheads vary minimally with the four parameters of task execution we measured. Setup and management overheads depend on the memory and CPU performance of the host on which EnTK is executed, while the tear-down overhead on the Python version utilized. This validates EnTK design and implementation against its requirements: EnTK can be used in various scientific domains, with different task executables, and across heterogeneous CIs.

In absolute terms, EnTK overheads are between ≈10 and 20 seconds but Experiment 3 shows that these overheads can be reduced by running EnTK on a host with better performance. RP RTS shows overheads up to ≈80s, limiting its utilization to applications with at least minutes-long tasks. These limitations are mostly due to the use of Python and its process and thread termination time: EnTK and RP should be coded, at least partially, in a different language to manage the execution of applications of tasks that are O(1) seconds.

B. Scalability

We perform two experiments to characterize weak and strong scalability of EnTK. As with Experiment 1–4, we measure and compare all overheads, Data Staging Time and Task Execution Time. Weak scaling relates these measures to the amount of

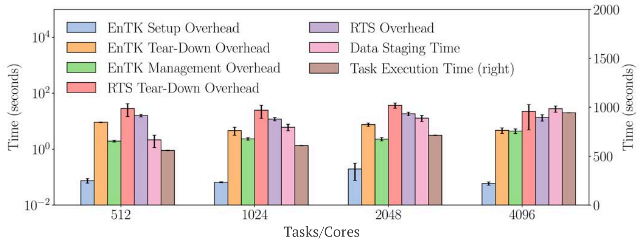


Fig. 8: Weak scalability on Titan: 512, 1,024, 2,048, and 4,096 1-core tasks executed on the same amount of cores.

concurrency used to execute the application's tasks; Strong scaling to the amount of serialization.

1) Weak scalability: To investigate weak scaling, we run four applications on Titan, each with 1 pipeline, 1 stage per pipeline, and 512, 1,024, 2,048, or 4,096 tasks per stage. Each task executable is Gromacs `mdrun`, configured to use 1 core for ≈600 seconds. The number of acquired cores is equal to the number of the application's tasks. Each task requires 4 input files: 3 soft links of 130B each and 1 file of 550KB.

Fig. 8 (right axis) shows that Task Execution Time increases gradually and therefore does not have ideal weak scaling. Analysis of the RTS profiles shows that this behavior is due to delays in the Executor module of the RTS Agent and, specifically, in the current implementation of the Agent scheduler and the ORTE distributed virtual machine of OpenMPI. Ref. [7]

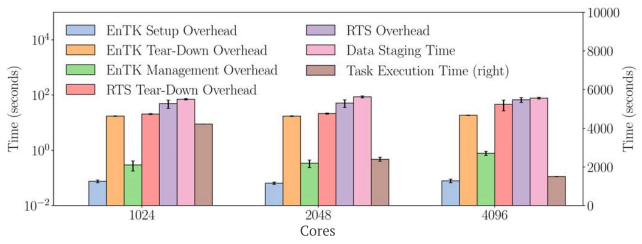


Fig. 9: Strong scalability on Titan: 8,192 1-core tasks are executed on 1,024, 2,048 and 4,096 cores.

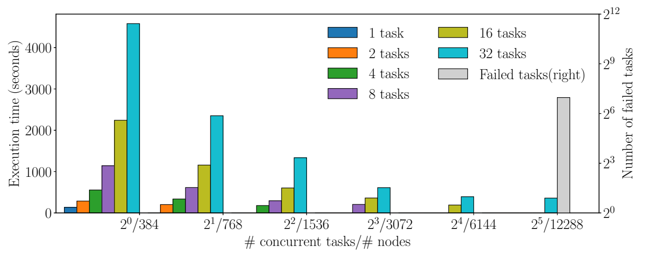


Fig. 10: Task Execution Time of forward simulations using EnTK at various values of concurrency.

characterizes these delays and their causes.

EnTK Management Overhead remains almost constant till 2,048 tasks as the number of tasks are too small to cause a variation. The overhead, then, increases between 2,048 and 4,096 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. The other EnTK and RTS overheads appear to be consistent with those already noted in Experiments 1–4.

EnTK neither controls, nor contributes to Data Staging time. Data staging is performed by the RP RTS that, in this experiment, creates 1 directory for each task, writing 3 soft links and copying 1 file within it for a total of ≈1MB. RP uses Unix commands to perform these operations on the OLCF Lustre filesystem. By default, RP is configured with 1 stager and hence files are staged sequentially. Multiple staging workers can be used to parallelize data staging but trade offs with the filesystem performance must be taken into account.

Data Staging time grows linearly with the number of tasks executed: from ≈11s for 512 tasks to ≈88s for 4,096 tasks. As this time mostly depends on the performance of Lustre, a less linear behavior is expected with larger (amount of) files.

2) Strong scalability: To investigate strong scaling, we run four applications on Titan, each with 1 pipeline, 1 stage per pipeline, 8,192 tasks per stage and a total of 1,024, 2,048 or 4,096 cores. Each task executable is Gromacs `mdrun`, configured to use 1 core for ≈600 seconds. In this way, we execute at least 2 generations, each with 4,096 tasks, within the 2 hours walltime imposed by Titan's queuing policies. Data staging is as in the weak scalability experiment.

Fig. 9 shows that Task Execution Time reduces linearly with increase in the number of cores. The availability of more resources for the fixed number of tasks explains this linear reduction in the Task Execution Time. EnTK Management Overhead is ≈1s, confirming what already observed in the previous experiments.

All the other overheads and Data Staging Time remain constant across the experiment runs. This suggests that both EnTK and RP overheads mostly depend on the number of managed tasks, not on the size of the pilot on which they are executed. This is confirmed for RP in Ref. [7].

Fig. 6 shows that EnTK can be configured to execute 10^6 tasks in less than 200 seconds and consuming less than 4GB of memory. Extrapolating and accounting for a faster CPU on Titan's login nodes, EnTK should manage enough tasks to fill all of Titan cores with an overhead of less than 20 seconds.

C. Use Cases at Scale

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

1) Seismic inversion: We use EnTK to encode the forward simulations of the seismic tomography workflow described in §III-A and depicted in Fig. 4. These simulations account for more than 90% of the computation time of the workflow, requiring 384 nodes for each earthquake simulation, and 40MB of input data each. 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/6,144 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. 10 shows that increasing concurrency leads to a linear reduction of Task Execution Time, with a minimum of ≈180 seconds. Interestingly, reducing concurrency eliminates failures: we encountered no failures in executions with up to 2^4 concurrent tasks and 6,144 nodes. At 2^5 concurrent tasks and 12,288 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

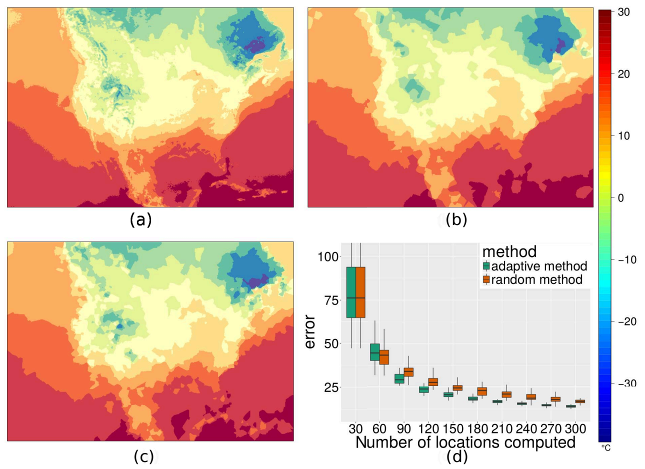


Fig. 11: Predictions from random and adaptive methods. (a) theoretical true value, (b) the interpolated map from 1,800 randomly picked locations, (c) the interpolated map from 1,800 locations identified using AUA, (d) box plots of the errors for both implementations.

Time was ≈360 seconds, similar to that of a run with with 2^4 concurrent tasks (Fig. 10).

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. 11 shows the prediction maps and errors obtained from the two implementations. With 1,800 locations calculated for both prediction maps (Fig. 11(b), Fig. 11(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. 11(d) shows the distribution of the errors for the two implementations. The error converges faster in the AUA algorithm than in the random selection. The total amount of potential locations (pixels) is 262,972; 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

Executing ensemble applications on HPC systems requires knowledge of resource, data and execution management, specific to the HPC system. Several “middleware” [14] frameworks have been developed to abstract execution details and enable execution of ensemble applications. Software development kits such as gSOAP [15] enable web services for HPC applications. Ninf-G [16] and OmniRPC [17] provide client/server-based frameworks for distributed programming. These solutions provide methods to launch application tasks on remote machines but leave the details of task scheduling, resource and data management, and fault tolerance to the user.

Hadoop and its ecosystem have been ported to HPC systems [18], [19], [20], enabling the use of the MapReduce programming model. While some ensemble applications are data-flow oriented and thus amenable to be implemented with MapReduce, EnTK adopts a more flexible and coarse-grained notion of tasks, where a task in EnTK can support multiple programming models, including MPI. Further, EnTK does not assume a specific runtime system and, in conjunction with RP, can use Hadoop on HPC [21].

Feature-rich workflow systems such as Kepler [22], Swift [23], and Pegasus [24] provide end-to-end capabilities such as resource and execution management, fault tolerance, monitoring and provenance. Encoding applications using these systems requires acquiring specific knowledge, including learning new languages and paradigms. Adapting these systems to user requirements is non-trivial due to their feature richness and end-to-end design. Ruffus [25], COSMOS [26], and GXP Make [27] limit the capabilities and prioritize interface simplicity. Galaxy [28], Taverna [29], BioPipe [30], and Copernicus [31] focus on providing tailored interfaces to domain scientists.

EnTK contributes (i) programmability, (ii) portability across CIs and RTSs, and (iii) generality to the research on 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 [32]. Consequently, EnTK exposes an API tailored towards encoding of ensemble applications, focusing on task concurrency and sequentiality.

One of the limitations observed in the existing frameworks is that functional and performance enhancements are localized to one framework and cannot be easily ported to other systems. EnTK avoids framework lock-in by enabling composability with diverse runtime systems and rapid development of user-facing, special-purpose application libraries. In this way, EnTK builds upon the idea of composing applications from execution patterns, also explored by systems like Tigres, and extends it to middleware for HPC for better programmability, portability, and generality.

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 ($\approx$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.

Acknowledgments: 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. R. 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. 19 011–19 016, 2009.

[4] "Designing workflow systems using building blocks," https://arxiv.org/abs/1609.03484.

[5] "Towards common components for open workflow systems," https://arxiv.org/abs/1710.06774.

[6] "EnTK," https://github.com/radical-cybertools/radical.entk.

[7] A. Merzky, M. Turilli, M. Maldonado, and S. Jha, "Design and performance characterization of RADICAL-Pilot on Titan," 2017, (under review) http://arxiv.org/abs/1512.08194.

[8] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Executing dynamic and heterogeneous workloads on super computers," 2017, (under review) http://arxiv.org/abs/1512.08194.

[9] 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

[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] 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.

[12] D. Komatitsch and J. Tromp, "Spectral-element simulations of global seismic wave propagationi. validation," *Geophysical Journal International*, vol. 149, no. 2, pp. 390–412, 2002.

[13] 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.

[14] M. Stonebraker, "Too much middleware," *ACM Sigmod Record*, vol. 31, no. 1, pp. 97–106, 2002.

[15] G. Aloisio, M. Cafaro, D. Lezzi, and R. van Engelen, "Secure web services with globus gsi and gsoap," *Euro-Par 2003 Parallel Processing*, pp. 421–426, 2003.

[16] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-g: A reference implementation of rpc-based programming middleware for grid computing," *Journal of Grid computing*, vol. 1, no. 1, pp. 41–51, 2003.

[17] M. Sato, T. Boku, and D. Takahashi, "OmniRPC: a Grid RPC system for parallel programming in cluster and grid environment," in *IEEE/ACM CCGrid*. IEEE, 2003, pp. 206–213.

[18] S. Krishnan, M. Tatineni, and C. Baru, "myhadoop: Hadoop-on-demand on traditional hpc resources," *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.

[19] W. C. Moody, L. B. Ngo, E. Duffy, and A. Apon, "Jummp: Job uninterrupted maneuverable mapreduce platform," in *IEEE Int. Conf. on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[20] "Magpie," https://github.com/LLNL/magpie(accessed December 2017).

[21] A. Luckow, I. Paraskevakos, G. Chantzialexiou, and S. Jha, "Hadoop on hpc: integrating hadoop and pilot-based dynamic resource management," in *IEEE IPDPS Workshops*. IEEE, 2016, pp. 1607–1616.

[22] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. B. 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.

[23] 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.

[24] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[25] L. Goodstadt, "Ruffus: a lightweight python library for computational pipelines," *Bioinformatics*, vol. 26, no. 21, pp. 2778–2779, 2010.

[26] E. Gafni, L. J. Luquette, A. K. Lancaster, J. B. Hawkins, J.-Y. Jung, Y. Souilmi, D. P. Wall, and P. J. Tonellato, "Cosmos: Python library for massively parallel workflows," *Bioinformatics*, vol. 30, no. 20, pp. 2956–2958, 2014.

[27] K. Taura, T. Matsuzaki, M. Miwa, Y. Kamoshida, D. Yokoyama, N. Dun, T. Shibata, C. S. Jun, and J. Tsujii, "Design and implementation of gxp makea workflow system based on make," *Future Generation Computer Systems*, vol. 29, no. 2, pp. 662–672, 2013.

[28] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[29] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat *et al.*, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[30] S. Hoon, K. K. Ratnapu, J.-m. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke, and E. Stupka, "Biopipe: a flexible framework for protocol-based bioinformatics analysis," *Genome Research*, vol. 13, no. 8, pp. 1904–1915, 2003.

[31] S. Pronk, I. Pouya, M. Lundborg, G. Rotskoff, B. Wesen, P. M. Kasson, and E. Lindahl, "Molecular simulation workflows as parallel algorithms: The execution engine of copernicus, a distributed high-performance computing platform," *Journal of chemical theory and computation*, vol. 11, no. 6, pp. 2600–2608, 2015.

[32] V. Balasubramanian, A. Treikalis, O. Weidner, and S. Jha, "Ensemble toolkit: Scalable and flexible execution of ensembles of tasks," in *45th Int. Conf. on Parallel Processing (ICPP)*. IEEE, 2016, pp. 458–463.