Ensemble Toolkit: Scalable and Flexible Execution of Ensembles of Tasks

Vivekanandan Balasubramanian, Antons Treikalis, Ole Weidner*, Shantenu Jha†
Department of Electrical and Computer Engineering, Rutgers University, Piscataway, New Jersey 08854
*Work done when at Rutgers
†Corresponding Author

***Abstract*—There are many science applications that require scalable task-level parallelism and support for flexible execution and coupling of ensembles of simulations. Most high-performance system software and middleware, however, are designed to support the execution and optimization of single tasks. Motivated by the missing capabilities of these computing systems and the increasing importance of task-level parallelism, we introduce the Ensemble toolkit which has the following application development features: (i) abstractions that enable the expression of ensembles as primary entities, and (ii) support for ensemble-based execution patterns that capture the majority of application scenarios. Ensemble toolkit uses a scalable pilot-based runtime system that decouples workload execution and resource management details from the expression of the application, and enables the efficient and dynamic execution of ensembles on heterogeneous computing resources. We investigate three execution patterns and characterize the scalability and overhead of Ensemble toolkit for these patterns. We investigate scaling properties for up to O(1000) concurrent ensembles and O(1000) cores and find linear weak and strong scaling behaviour.**

I. Introduction

Many scientific applications in the field of molecular sciences, computational biology [1], [2], astrophysics [3], weather forecasting [4], bioinformatics [5], [6] are increasingly reliant on ensemble-based methods to make scientific progress. This is true for applications that are both net producers of data, as well as aggregate consumers of data. Computationally, an ensemble is comprised of multiple concurrent units of execution, which are referred to as *tasks*. Tasks may refer to simulations, analysis or any independent computational process that needs to be executed. The set of tasks comprising different ensemble-based application vary in the degree of coupling between tasks, dependency between stages of tasks, as well as the level of heterogeneity across tasks.

In spite of the apparent simplicity of running ensemble-based applications, the scalable and flexible execution of a large and collective set of tasks is non-trivial. Ensemble-based applications are significantly more complex and sophisticated than simple parameter sweeps (bag-of-tasks) in that they support varying degrees of coupling between the ensemble members, such as global synchronization, temporally varying pairwise interactions. In addition, all ensemble members must successfully complete as the collective properties of the entire ensemble is often computed.

There are many functional, performance and usability challenges that currently prevent ensemble-based applications from exploiting the full power of modern high-performance computing (HPC) systems. The situation is exacerbated by the lack of tools that are designed and implemented with ensembles as a fundamental unit. Hitherto, ensemble-based applications have either been retrofitted into one of many *general purpose* workflow systems, or wrapped in specifically constructed scripts that address some, but rarely all of the requirements. The former can in principle support ensemble-based applications as a *special* case but most workflow systems have their design point and performance tuned to managing complex inter-dependencies of tasks, which is not the distinguishing characteristic or dominant requirement of ensemble-based applications. The scripting-based approach presents a significant burden on the user, at least, when it comes to running large ensembles in a fault-tolerant way on heterogeneous resources, as well as in developing sophisticated ensemble-based applications.

In response to these challenges, the growing importance and pervasiveness of ensemble-based applications, we propose, design and implement Ensemble toolkit. Ensemble toolkit promotes ensembles as a first-class entity and has the following design features to meet the requirements of ensemble-based applications: (i) programming abstractions that enable the expression of an ensemble of tasks as the primary entities, independent of the executable, (ii) support for ensemble-based execution patterns, (iii) decoupling of the details of execution and management of these patterns from the expression of the patterns, and (iv) a runtime system that enables the efficient execution of tasks and provides the flexible resource utilization capabilities over a range of HPC platforms.

The implementation of these design features provides advances at both system and application levels. While being novel in concept, its implementation builds upon well-established high-performance computing abstractions and significantly reused existing software components, i.e., its new software footprint is extremely small. Ensemble toolkit incorporates both best practices in software development and systems engineering, designed to capture the *sweet-spot* in the space of control, flexibility and usability, and resulting in simple yet powerful ways of developing and executing ensemble-based applications.

Although the motivation of this toolkit stems from its current need & utility in the molecular science and geoscience community, the concepts, components and use of Ensemble toolkit is agnostic to the application and can be used in any domain. Ensemble toolkit fills a missing gap in the repertoire of production-grade HPC tools.

In Section 2, we discuss a few other tools that share the research space with Ensemble toolkit. In Section 3, we present the Ensemble toolkit, discuss the objectives, design, and implementation. In Section 4, we perform validation and scalability experiments using both toy and real MD workload. We conclude the paper in Section 5 with a comparison of the key features of Ensemble toolkit with other similar tools.

II. Related Work

Many tools have been developed to assist computational scientists in running their workloads on HPC infrastructures, however as alluded to, none have been designed with ensembles as the fundamental entity. For example, there exist multiple tools to run parameter sweeps (e.g., Nimrod [7]) and many to support embarrassingly parallel tasks.

Traditionally, scripting has been a common mode for many "workflows", due to the illusion of providing complete control and flexibility. Scripting requires a priori and complete knowledge of workload and thus has obvious limitations in terms of capability for dynamic decisions, fault handling, and reuse. Some of the simple scripting approaches have also been interfaced with web-services based tools to manage job submission, such as Longbow [8] and GROWL [9]. Both Longbow and GROWL, however, are limited in the different application types that can be supported. Longbow only supports a bag of MD simulations using Amber [10], CHARMM [11], Gromacs [12], LAMMPS [13] or NAMD [14] codes. GROWL is limited to C and C++ based applications.

Workflow systems are at the other end of the spectrum in terms of control and flexibility. Systems such as Pegasus [15] and DAGMan [16] convert given workloads into directed acyclic graphs (DAGs). DAGMan simply schedules the jobs as per the DAG where each edge of the DAG specifies the order of precedence. DAGMan is primarily meant for static workloads. Pegasus, primarily expressed in virtual data language, maps workflows onto select execution sites based on some criteria to create resource specific DAG. DAGMan is then responsible for job execution in the prescribed order. There are workflow systems such as Swift [17], which do not rely on the creation of a DAG but require an explicit mention of the order of execution of the various tasks in a multi-stage application.

There are other dataflow oriented tools such as Copernicus [18], Ruffus [19], Snakemake [20] and COSMOS [21]. Both Ruffus and COSMOS provide a python library to construct the user application whereas Snakemake provides a separate workflow definition language. Snakemake and COSMOS convert their workload into DAGs using tool dependencies and file naming conventions and enable execution on common HPC. Ruffus mainly concentrates on expression & automation of conventional pipeline based workloads.

As demonstrated, many tools exist but very few, if any, are designed for ensemble-based applications. In fact, there are no best practices, nor is there a consensus on suitable tools for ensemble-based applications. For example, ensemble-based enhanced sampling or replica-exchange algorithms in molecular science applications [1], [22]–[24] use widely different approaches, depending upon the number of tasks (typically O(100)-O(1000)), coupling and size of the system being studied. Similarly, in geosciences, ensemble-based simulations (number of tasks is typically between O(10)-O(1000)) are used to study dispersion of atmospheric pollutants [25], [26], effects of contaminant release [27], areas under risk from nuclear releases [28], ozone forecast [29], hurricane prediction [30], precipitation forecast [31], modelling sensitivities in atmospheric circulations [32], water budget estimation [33], estimating uncertainty in future sea level rise [34]. Within the aforementioned geoscience applications, the coordination, orchestration and execution of ensembles are handled very differently.

III. Ensemble Toolkit

Having discussed the impressive range of applications and methods for ensemble-based applications, we provide a semi-informal discussion of the requirements of applications comprised of multiple tasks on HPC systems. This motivates the design and implementation of Ensemble toolkit.

A. Requirements

There are a number of requirements that must be met in order to support the wide range of ensemble-based applications. At the application level, there is the requirement for uniform and simple approaches for developing ensemble-based applications without compromising scale and the ability to run over heterogeneous resources. Ensemble-based applications vary in the type of coupling between the tasks, the degree of frequency of interaction, the executable software that is at the center ("kernel") of each ensemble member, and also the volume of information exchange between these tasks. Applications could conform to a MapReduce style application [35] where all tasks are subject to a hard (global) synchronization step. Further, tasks can vary greatly in their individual computational requirements. In some cases, a task could be a single core job, multiple nodes [36] or even hundreds of cores each [37].

One of the primary challenges is the need to provide efficient and flexible resource management over a diverse range of resource types for the ensembles. In many cases, the resource requirements of the total set of tasks are much greater than the total amount of resources that can be acquired. This can be due to the lack of resources, impractically large queue wait times or upper limits enforced by scheduling policies. In either case, it is important to decouple the total resources required from the resources utilized (or available) at any given instant of time. The resource management complexity points

to the need for a special-purpose runtime that insulates the challenges from the applications.

B. Design

Three primary design objectives of Ensemble toolkit are:

1) Support a range of varying ensemble-based science applications without major refactoring of application
2) Manage execution level details while providing capability and performance guarantees
3) Support dynamic resource management so as to decouple resource requirements from the availability of resources.

We observe that the control flow in common ensemble-based applications can be categorized into a few recurring types. We take advantage of this similarity by creating abstract "patterns" which define the control flow agnostic of the type of workload, the number of tasks and resource requirements of tasks. For example, a bag of tasks pattern would create a set of tasks that are independent of each other without specifying the actual work done or resource requirements of the tasks. With these points in consideration, we state the design decisions that were made for Ensemble toolkit:

Pattern-based approach for application development: The toolkit treats the execution pattern as a core component, thereby capturing the control flow of the application. It provides a simple mechanism to develop applications by parametrizing a pre-defined pattern and defining its execution stage(s).

Identify and support unit patterns: We identify simple, unit execution patterns for ensemble-based applications. Many higher order patterns can be developed by combining or extending these unit patterns.

Separation of concerns: User is required to provide only the application control flow logic via the exposed components. The complex details of the task execution, task synchronization, etc. are hidden, from the user, in the runtime system. This partition provides the freedom to make execution level decisions targeting various metrics, namely, throughput, time to completion, data movement, scale, or resource specific decisions without the requirement of user involvement.

The resulting architecture of Ensemble toolkit (Fig. 1) has four components: the execution patterns, the dynamic resource handle, the kernel plugins and the execution plugins. These components ease the development of applications while confining the execution complexity to the runtime system. We discuss each component:

1) **Execution Patterns:** An execution pattern is a high-level object that represents an application's control flow. They can be viewed as a parameterized template for capturing the execution of the ensemble(s). Execution patterns provide placeholder methods for the individual stages of an execution trajectory.
2) **Kernel plugins**: A kernel plugin is an object that abstracts a computational task in Ensemble toolkit. It represents an instantiation of a specific science tool along with the required software environment. Kernel plugins hide kernel-specific peculiarities across different resources as well as differences between the interfaces of the various kernels to the extent possible, thus addressing kernel-level heterogeneity.
3) **Resource Handle**: The resource handle is a representation of a computing infrastructure (CI), providing methods to:
   - allocate resources
   - run an execution pattern on these resources
   - deallocate resources
4) **Execution Plugins**: The Execution Plugin binds the kernel plugins and the execution patterns, and translates the tasks into executable units that, along with resource details from the resource handle, are forwarded to the underlying runtime system. Thus the execution is decoupled from the expression of the application. It currently supports static binding and translation, "intelligence" can be added to the execution plugin so as to support applications that have a dynamic workload or changing resource requirements. To do so requires the use of information about the resource(s) coupled with the requirements from the application, to devise execution strategies [38].

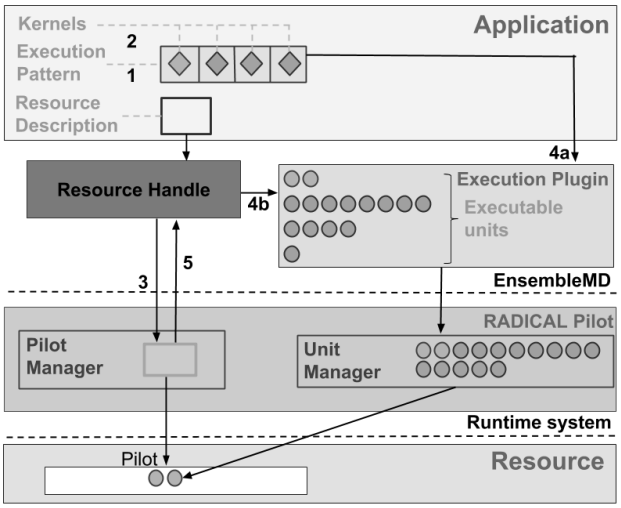

Fig. 1. Ensemble toolkit components: Execution Patterns, Execution Plugins, Resource Handle, Kernel plugins. Five steps: (1) Pick execution pattern for application, (2) Define Kernel plugins in the various stages of the pattern, (3) Create resource handle and submit a request for resources, (4) Execution plugin (a) binds execution pattern and the kernel plugins & (b) executes them on the remote machine, (5) User gets back control.

The resource handle, execution pattern, and kernel plugins are user facing components. They are used to obtain and validate details regarding the resources, the pattern used and the workload in each stage of the pattern. The execution plugin is an internal component of the toolkit which is responsible for extracting the necessary data from the other components and passing it on to the runtime system in an understandable format.

C. Implementation

In order to keep the software footprint of Ensemble toolkit small, it was implemented to utilize existing tools, libraries,

and frameworks wherever possible. In order to be able to use HPC systems, Ensemble toolkit follows a standard job submission language and runtime system that allows the same. Although the criteria of the runtime system is to be simply compatible with this job submission language alone, we select RADICAL-Pilot due to its specific advantages discussed below.

1) Job submission language: The Ensemble toolkit is designed to be compatible with runtime systems that comply with the SAGA Job Description [39] that is derived from the standard Job Submission Description Language [40] proposed by the Global Grid Forum. An example of the values required by the Ensemble toolkit resource handle to construct the job submission script is provided in listing 3.

2) Runtime system: As mentioned previously, Ensemble toolkit relies on a runtime system to manage task execution. One of the design objectives was to provide dynamic resource management, one type of which is to be able to execute more tasks than resources available. Pilot-Job systems provide a simple solution to the static resource management found in most high performance and distributed computing infrastructures [41]. Pilot-Job systems provide placeholders or container jobs that are submitted to the target resource. These container jobs enable application-level scheduling of any number of workload tasks on these resources. A resource placeholder, thus, decouples the acquisition of the resources from their use to execute application tasks.

Of the multiple pilot systems currently developed [42], we select the RADICAL-Pilot [43] implementation. This is due to certain features unique to RADICAL-Pilot: 1) Support for MPI-based applications, 2) support for data-intensive applications 3) support for execution on diverse resources. RADICAL-Pilot builds upon important conceptual theoretical advances (P* Model [41]). Resource-level heterogeneity is addressed in RADICAL-Pilot via an interoperability layer, SAGA-python [44] to get access to diverse DCI middleware.

It is important to note that the Ensemble toolkit can be coupled with other general-purpose or special-purpose runtime systems. Ensemble toolkit is actively developed and provides the necessary hooks for this coupling.

D. Ensemble toolkit Execution Patterns

Ensemble toolkit natively supports three execution patterns: pipeline, ensemble exchange (EE) and simulation analysis loop (SAL). These initial patterns were motivated by their immense popularity and use in molecular science and geoscience domains. Although introduced in the context of these two domains, these patterns can be used for any application that follows the same orchestration of tasks by modifying the specific execution kernel plugins. We take a look at these patterns in terms of the coupling and dependencies of these tasks.

1) Pipeline: It consists of a bag of independent pipelines that consist of multiple stages of execution which can contain heterogeneous workload. It follows a linear and unidirectional flow of data and control. Although each stage of a pipeline depends on its predecessor, the pipelines themselves execute independent of each other. Figure 2 is a pictorial representation of the pipeline pattern consisting of N pipelines each with M stages. Many applications in bioinformatics follow the pipeline pattern [6] [5].

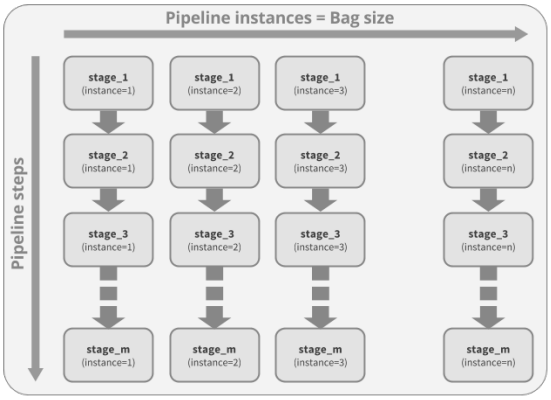

Fig. 2. Diagrammatic representation of the pipeline execution pattern with N pipes each with M stages

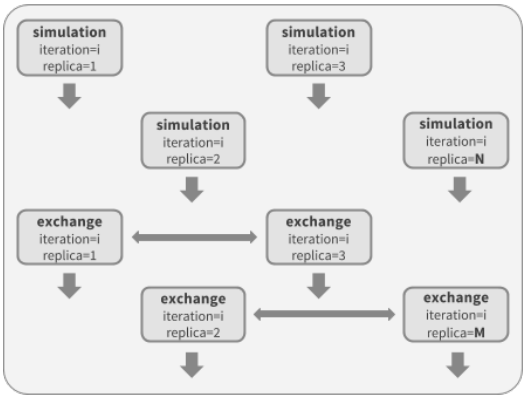

Fig. 3. Diagrammatic representation of the EE execution pattern with N simulation and exchange instances

2) Ensemble Exchange: EE pattern is fully specified by two stages: simulation and exchange. The simulation stage is followed by the exchange stage, where there can be an exchange of information between the various instances. As depicted in Figure 3 execution of both simulation and exchange stages can be performed with various degrees of concurrency. EE pattern is a generalization of popular replica exchange molecular dynamics (REMD) conformational sampling algorithm [24].

3) Simulation Analysis Loop: The SAL pattern consists of iterations of a set of simulation instances and a set of analysis instances. There are also the pre_loop and the post_loop stages which are outside this iterative sequence. Figure 4 represents the SAL pattern with N simulation instances and M analysis instances in each loop. This pattern finds use in many of the MD [1], [22] and geoscience applications [25]–[30], [32] where multiple iterations of simulation and analysis tools need to be performed till a convergence criterion is reached.

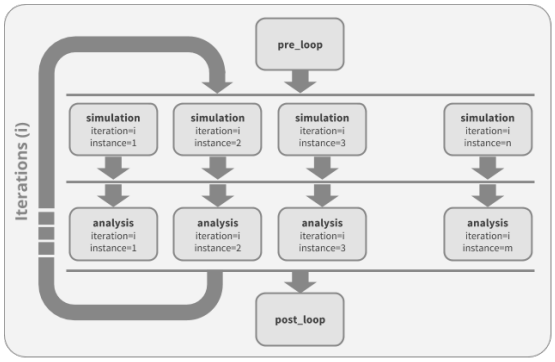

Fig. 4. Diagrammatic representation of the SAL execution pattern with N simulation instances and M analysis instances

E. Ensemble toolkit Interface

We now present the interface of components that are exposed to the user: resource handle, execution patterns and kernel plugins. The purpose is to present how their expression remains agnostic to the resource and underlying execution details, yet provides the much-needed flexibility to compose a wide range of ensemble-based applications. We discuss the interface of the pipeline execution pattern and for completeness outline that of the SAL pattern. We provide an example of an end-to-end application using the EE pattern to show how an application can be mapped to a pattern.

1) Execution Pattern - Pipeline: Ensemble toolkit provides an intuitive method to support the execution of an ensemble of pipelines. Each pipeline executes independent of the other, but within each pipeline, the execution of each stage `i` is preceded by complete execution of stage `i-1`. They can be heterogeneous, i.e. each stage of each pipeline can define its own kernel plugin. Other pipeline parameters, such as the number of stages or the number of pipelines can also be controlled by the user. Listing 1 provides an example of an instance of the pipeline execution pattern with 2 stages and 16 parallel pipelines.

```
from radical.ensemblemd import Pipeline

class MyApp(Pipeline):
    def __init__(self, stages, instances):
        Pipeline.__init__(self, stages, instances)

    def stage_1(self, instance):
        # kernel definition...

    def stage_2(self, instance):
        # kernel definition...

app = MyApp(stages=2,instances=16)
```

Listing 1. Interface of the pipeline execution pattern containing 2 stages and 16 pipe instances

2) Kernel plugins: A kernel plugin is an object that abstracts a computational task. It represents an instantiation of the computational task along with the software environment required for the specific resource. In addition, the kernel object also provides methods to specify input and output data, allocate cores for the each task. We list an example of kernel plugins below.

```
k = Kernel(name="myKernel")
k.arguments = ["--arg1=file1.dat", "--arg2=file2.dat"]
k.upload_input_data = ["file1.dat","file2.dat"]
k.dowdnload_output_data = ["outputfile.dat"]
k.cores=1
k.mpi=False
```

Listing 2. Example of a kernel plugin in Ensemble toolkit that enables the user to specify the arguments the kernel takes, the files to be transferred, MPI enabled and the cores to be used

3) Resource handle: Every Ensemble toolkit script requires a resource handle which requires the following details: resource name, number of cores, walltime of the reservation, username for access, allocation id, database details, submission queue(optional). These are necessary to create a job description in accordance with the SAGA job model. The database details are a consequence of the choice of RADICAL-Pilot as the runtime system which requires it for a persistent point for coordination of tasks.

Listing 3 creates a resource handle object requesting 16 cores for 30 minutes on Stampede supercomputer [45]. Using this object user can perform three operations: allocate the resource, run an execution pattern on allocated resources or deallocate the resource.

```
from radical.ensemblemd import SingleClusterEnvironment

resource = SingleClusterEnvironment(
    resource        = "xsede.stampede",
    cores           = 16,
    walltime        = 30,
    username        = "abc",
    project         = "TG-xyz123",
    queue           = "normal",       #optional
    database_url    = "mongodb://user:pwd@domain",
    database_name   = "myexps"     #optional
)
```

Listing 3. Example of an resource handle targeting XSEDE Stampede requesting 16 cores for 30 mins along with credentials to access the machine and the mongodb

It is important to note that the resource handle is independent of the execution pattern or the workload. The resource handle remains uniform across all possible resources and hides resource heterogeneity from the user. In any application, resource details need to be specified only once, via the resource handle.

By design, the user only interacts with the above three components in any application development process. Their interface presents the simplicity in setting up these components to construct a complete application. To reiterate the point, we briefly discuss the interface for the SAL pattern; the salient features of the interfaces discussed in the context of the pipeline pattern apply here too: separation of resource and execution description from the application (pattern, kernel) description, flexibility of composition.

Simulation Analysis Loop: The SAL pattern consists of 4 stages: 2 iterative, simulation and analysis stages, and 2 non-iterative, pre-loop and post-loop stages. For the SAL pattern, the user can specify the number of loops of the simulation and analysis stages, number of simulations and number of analysis instances per iteration.

```
from radical.ensemblemd import SimulationAnalysisLoop
```

```
class MyApp(SimulationAnalysisLoop):
    def __init__(self, maxiterations, simulation_instances
     =1, analysis_instances=1):
        SimulationAnalysisLoop.__init__(self, maxiterations
     , simulation_instances, analysis_instances)

    def pre_loop(self):
        # kernel definition...

    def simulation_stage(self):
        # kernel definition...

    def analysis_stage(self):
        # kernel definition...

    def post_loop(self):
        # kernel definition...

app = MyApp(maxiterations=10, simulation_instances=16,
     analysis_instances=1)
```

Listing 4. Interface of the SAL execution pattern containing 10 iterations of 16 simulation instances followed by 1 analysis instance

4) End-to-end example: In the end-to-end EE application (Listing 5) we insert `md.namd` and `md.re_exchange` kernel plugins in the EE pattern. We create the resource handle targeting `localhost` which launches a pilot occupying 2 CPU cores for 15 minutes.

```
class MyApp(ReplicaExchange):

    def __init__(self, workdir_local=None, cycles=1,
     replicas=1):
        ReplicaExchange.__init__(self, workdir_local,
     cycles, replicas)

    def prepare_replica_for_md(self, replica):

        k = Kernel(name="md.namd")
        k.arguments            = ['alanin_base_{0}_{1}.namd
    '.format(replica.id,replica.cycle)]
        k.upload_input_data    = ['alanin_base_{0}_{1}.namd
    '.format(replica.id,replica.cycle)]
        k.download_output_data = ['']alanin_base_{0}_{1}.
    history'.format(replica.id,replica.cycle)']

        replica.cycle += 1
        return k

    def prepare_replica_for_exchange(self, replica):

        k = Kernel(name="md.re_exchange")
        k.arguments = ["--calculator=namd_matrix_calculator
    .py",
                       "--replica_id=" + str(replica.id),
                       "--replica_cycle=" + str(replica.
    cycle-1),
                       "--replicas=" + str(self.replicas),
                       "--replica_basename=alanine_base"]

        k.upload_input_data    = "namd_matrix_calculator.py
    "
        k.download_output_data = "col_{0}_{1}.dat".format(
    replica.cycle-1,replica.id )

if __name__ == "__main__":

  cluster = SingleClusterEnvironment(
            resource="local.localhost",
            cores=2,
            walltime=15,

            database_url='mongodb://user:password@ds039145
    .mongolab.com:39145/enmd'
        )

  # Allocate the resources.
  cluster.allocate()

  workload = MyApp(cycles=1, replicas=1024)

  # run EE simulation
  cluster.run(workload)

  cluster.deallocate()
```

Listing 5. EE execution pattern for a replica exchange application

Application development using the Ensemble toolkit consists of five basic steps (Figure 1):

Step 1: The user interacts with three components - execution pattern, kernel plugins, and resource handle. The user first picks the execution pattern that best represents the application. Every execution pattern requires pattern specific parameters such as the number of cycles, the number of stages, etc.

Step 2: The user then picks kernel plugins to fill in the various stages of the execution pattern. These kernel plugins define what each stage should execute and what data needs to be moved in and out of the stage.

Step 3: The next step is to create a resource handle for a remote machine with details of the resource request. Once created, the user can now request an allocation for the resources.

Step 4: Once the allocation request is made, we can pass the entire execution pattern + kernel plugins to the execution plugin where they are translated to executable units and submitted to the remote machine.

Step 5: Once the execution is completed, control returns back to the user. The user can now run another pattern if required or simply deallocate the resources.

There are currently three execution patterns supported by Ensemble toolkit. The toolkit can be extended to support new execution patterns by creating new execution plugins and defining an interface for the same. Creating the execution plugin requires familiarity with RADICAL-Pilot, but once created it can be reused for multiple applications following this pattern. Keeping the space and scope in mind, we skip the description of how new patterns can be added, but its information can be found in the Ensemble toolkit documentation [46].

In this section, we identified the requirements for the Ensemble toolkit. We designed the components of Ensemble toolkit to meet these requirements: execution patterns to provide the control flow, kernel plugins to abstract the computational task, resource handle to hide resource heterogeneity and access to resources, execution plugins to translate tasks to executable units. With these components, a range of ensemble-based applications can be developed. With the functionality of the toolkit established, we discussed the usability of the toolkit by describing the interface to the various components. We discussed, step-by-step, the application development process to show that user effort is only towards defining the application and does not involve execution level details. In the next section, we move to discussing the performance of the toolkit.

IV. Characterization and Validation

In this section, we attempt to characterize the toolkit and validate some of the design decisions that we made. We begin with a description of the workload and the HPC machines

used in each of the experiments. We, then, discuss some of the parameters and definitions used before moving to the experiment results and their analysis.

We divide our experiments into two tracks: validation and performance characterization. We characterize the execution patterns by executing the same workload using three patterns. We discuss the individual timing components in detail. Next, we attempt to validate the kernel plugins by simply choosing one of the patterns from the previous experiment, switching the kernel plugins and comparing the timing details in both cases validating the hypothesis that switching the kernels does not change the overhead and hence the total time. So far, in the paper, we have addressed the aspects of functionality and usability. In the second experiments track, we characterize the scalability of Ensemble toolkit- we run strong and weak scaling tests for the EE and SAL execution patterns. Strong scaling tests observe the variation of the time to completion (TTC) with the number of cores for a problem of fixed total size. Weak scaling tests observe the variation of the TTC with the number of cores when the problem size per core is kept fixed. We also characterize the capability in Ensemble toolkit to support MPI tasks.

A. Experiment Setup

Prior to discussion of the experiments, we present a description of the workload and the HPC machine used in each of the experiments.

HPC description: In the first two experiments, we use the XSEDE Comet [47] cluster from the San Diego Supercomputing Center. Comet is an Intel Xeon cluster with 1984 nodes and 24 cores per node and 120GB memory per node. In the scaling experiments, we use XSEDE Stampede [45] from Texas Advanced Computing Center and XSEDE SuperMIC [48] from Lousiana State University. Stampede is an Intel Xeon cluster with 6400 nodes and 16 cores per node and 32GB memory per node. SuperMIC is an Intel Xeon Phi cluster with 360 nodes with 20 cores per node and 60GB memory per node.

Workload description: We describe the workload in each of the three experiments below:

1) We use a toy application which contains 2 stages. The first stage contains tasks which create a file with random characters. The second stage consists of tasks which perform a character count on these files.
2) We construct the DM-d-MD application [1] using the SAL pattern. The physical system used was a decalanine residue with each simulation run for 1ps using the Gromacs [49] MD engine.
3) We run scaling experiments for the EE pattern and the SAL pattern. We use a solvated alanine dipeptide molecule containing 2881 atoms and run each simulation using the Amber [50] MD engine.

B. Parameters and Definitions

We establish the definitions and notations used in the discussion of the results. In each of the experiments, we measure the TTC which can be decomposed as follows:

$$TTC = T_{EnTK\ overhead} + T_{execution} + T_{data\ transfer} \quad (1)$$

- $T_{EnTK\ overhead}$: The total time taken by Ensemble toolkit in construction and scheduling of the various tasks. This includes the time that components, external to Ensemble toolkit but invoked by it take, such as RADICAL-Pilot.
- $T_{execution}$: Execution time of the each stage of the pattern.
- $T_{data\ transfer}$: Total time taken in data transfer.

As we aim to understand Ensemble toolkit, its overhead can be decomposed further to analyze its variation, if any, across configurations and the different patterns:

$$T_{EnTK\ overhead} = T_{core\ overhead} + T_{pattern\ overhead} + T_{RP\ overhead} \quad (2)$$

- $T_{core\ overhead}$: Time spent by Ensemble toolkit to submit and cancel a resource reservation.
- $T_{pattern\ overhead}$: Time spent to create all the tasks of the pattern
- $T_{RP\ overhead}$: Overhead that arises from RADICAL-Pilot. The RADICAL-Pilot overhead is in turn comprised of many sub-component which have been investigated [43]. As the focus is on Ensemble toolkit, we do not decompose the RP overhead any further.

C. Characterization of Execution Patterns

We present and validate the three existing execution patterns in Ensemble toolkit: pipeline, EE, and SAL. We create a two-stage application using each of the patterns. We use the `mkfile` kernel in the first stage to create a file in each task and the `ccount` kernel in the second stage to count the number of characters in each of those files.

In this experiment, we vary the number of tasks and cores from 24-192, keeping their ratio 1:1. We use the XSEDE allocated Comet cluster [47]. A decomposition of the total time using the three different patterns is given in Fig 5. The first three subplots depict the execution time of the application using the three patterns. The following subplots represent the decomposition of the Ensemble toolkit overhead and the data transfer time when using the pipeline pattern. As expected, we observed similar overhead and data transfer timings when using the other patterns. Keeping space considerations in mind, we avoid repeating their overheads, data transfer timings.

From the first three subplots, it can be observed that the application execution times remain relatively same at all the configurations across patterns. This is due to the fact that each task has the same workload and all the tasks execute concurrently in all the patterns. In subplots 4 and 5, we present the behaviour of the overheads when using the pipeline pattern. The EnTK Core overhead which is independent of the pattern, remains constant in all the configurations. The EnTK Pattern overhead is the specific time consumed by the pattern and, as expected, depends on the number of tasks. The RP overhead

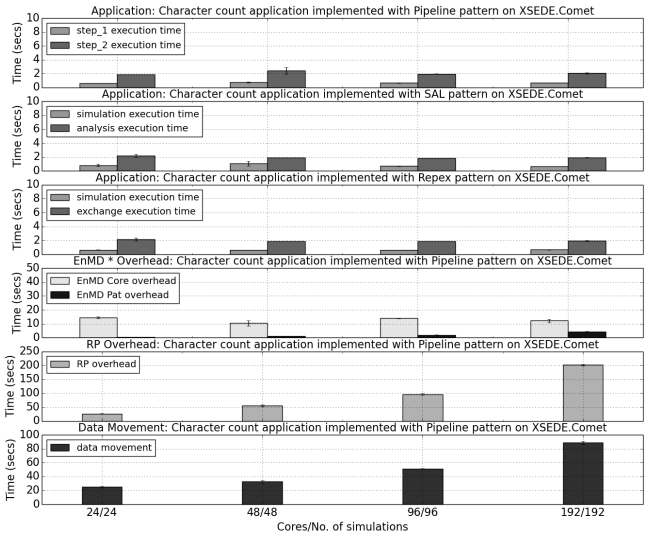

Fig. 5. Character count application implemented with pipeline, SAL and EE pattern on the XSEDE Comet. We vary, in same proportion, the number of tasks and cores from 24-192, thus all the tasks are concurrently executed. The first three subplots show the application execution times using the three patterns. The next 3 subplots show the overhead and data transfer times.

increases quite rapidly with the number of tasks and is quite high as compared to the other overheads. One of the main reasons for this magnitude of the overhead is the high amount of communication, and thus the latency, that occurs between the RP layer and the coordinating database [43]. In the last subplot, we see that the data movement time increases with increase in the number of tasks. This is due to the fact that the amount of data increases with increase in the number of tasks.

It is important to observe that the patterns themselves do not influence the workload. As is evident from the results, given the exact same workload, we observe similar execution times across the different patterns.

D. Validation of Kernel Plugins

The objective of this experiment is to present and validate the support for kernel abstractions in Ensemble toolkit. We pick the SAL pattern from the previous experiment and replace the kernel plugins with actual MD tools: GROMACS [49] in the simulation_stage and LSDMap [1] in the analysis_stage. We execute the application again on the Comet cluster in XSEDE [47] and keep the scales in the same range (24-192).

Figure 6 presents a decomposition of the total time. We skip the data movement time since it is a factor of the amount of data and the network between the client and remote. We observe that, for the same range of scales on the same target machine, the overheads obtained in this experiment are very similar to the ones presented in Figure 5 for the SAL pattern. It can be inferred that changing the kernel plugins, hence the workload does not effect the overhead presented by Ensemble toolkit. Note that due to change in workload, we now have a preloop stage and different values for the application execution time.

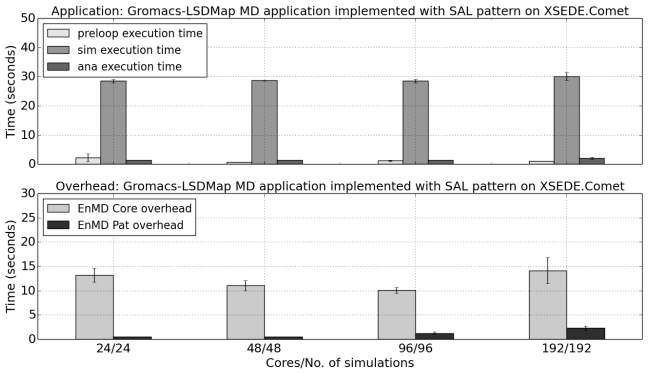

Fig. 6. Gromacs-LSDMap application implemented with SAL execution pattern on the Comet cluster in XSEDE. We vary the number of tasks from 24-192 but also vary the number of cores similarly.

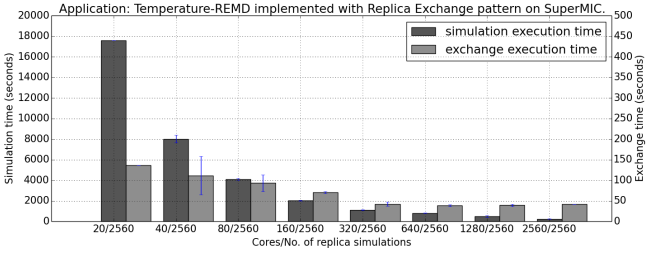

Fig. 7. Strong scaling test for EE execution pattern on XSEDE Supermic using Amber-Temperature Exchange kernel plugins and the alanine dipeptide molecule with 2560 replicas over a (20-2560) range of number of cores hence varying problem size per core.

E. Characterization of Scalability

We have now validated both the support for different execution patterns and kernel plugins in Ensemble toolkit. We now test the scalability of the toolkit with real science workloads. We perform strong and weak scaling tests for EE pattern and the SAL pattern.

1) Ensemble Exchange Pattern: We run the EE pattern for a solvated alanine dipeptide molecule containing 2881 atoms. We use the Amber MD Engine for the simulations and perform a temperature exchange during the exchange stage. We perform both the experiments on the SuperMIC cluster in XSEDE [48]. Figure 7 and Figure 8 present the results of the strong and weak scaling experiments respectively.

For strong scaling experiments, we keep the number of replicas constant at 2560 and vary the number of cores between 20-2560. Each replica is run on 1 core for 6ps before exchange. In Figure 7, note that since the simulation and exchange times are different by orders of magnitude, we use two y-axes as labeled. From Figure 7, we can observe that the simulation time decreases to half its value when the number of cores are doubled. The exchange times, on the other hand, remain constant as they depend on the number of replicas, which is constant for this experiment.

In weak scaling experiments, we keep problem size per core constant, i.e., we keep the ratio of the number of replicas to the number of cores constant. We vary the number of replicas from 20-2560 and the number of cores proportionately. Each

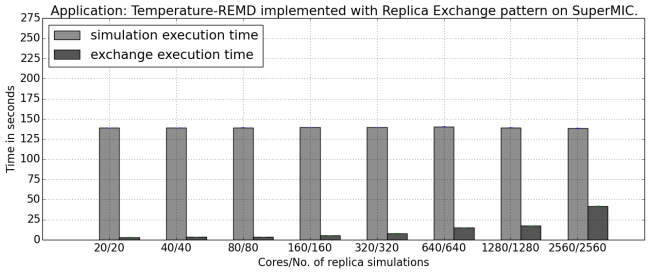

Fig. 8. Weak scaling test for EE execution pattern on XSEDE Supermic using Amber-Temperature Exchange kernel plugins and the alanine dipeptide molecule with fixed problem size per core.

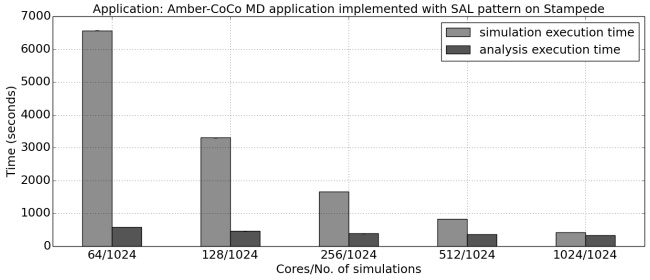

Fig. 9. Strong scaling test for Simulation-Analysis-Loop execution pattern on XSEDE Stampede using Amber [50]-CoCo [22] kernel plugins and the alanine dipeptide molecule with 1024 simulations over a (64-1024) range of number of cores hence varying problem size per core.

replica is run on 1 core for 6ps before the exchange. Our results in Figure 8 show that the simulation time remains relatively constant. The exchange times, however, increases as this depends on the number of replicas.

2) Simulation Analysis Loop Pattern: We implement the iterative collective coordinates algorithm [22] using the SAL pattern. We use a solvated alanine dipeptide molecule containing 2881 atoms as our physical system. Each simulation is executed using the Amber MD Engine for 0.6 ps followed by the CoCo analysis of all simulations. Figure 9 and Figure 10 present the results of our strong and weak scaling experiments.

In the strong scaling experiment, we keep the number of simulations fixed at 1024 and use one core per simulation. We vary the number of cores used from 64-1024. Similar to the results with the EE pattern, we can observe that the simulation time reduces to half its value when the number of cores is doubled. The analysis algorithm is executed in serial and thus depends on the number of simulations. Hence, the analysis execution time remains constant for all configurations.

Similar to the EE pattern, we perform weak scaling experiments. The number of cores is varied between 64-4096. At each of these configurations, the simulation execution time is observed to be constant. Similar to the previous case, the analysis is executed in serial. Note that the absolute performance of the analysis kernel is unrelated to the scalability of Ensemble toolkit.

3) Analysis of scaling results: In the strong scaling experiments, as we increase the number of cores, the number of simulations executing in parallel increase. Since the total

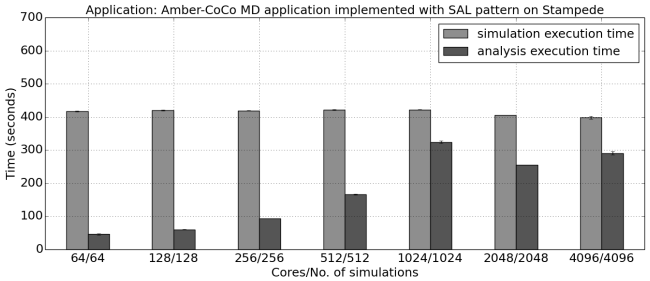

Fig. 10. Weak scaling test for Simulation-Analysis-Loop execution pattern on XSEDE Stampede using Amber-CoCo kernel plugins and the alanine dipeptide molecule with number of replicas equal to the number of cores hence fixed problem size per core.

problem size, i.e. the number of simulations, is constant, the simulation execution time decreases. In the weak scaling experiments, we keep the problem size per core constant by using as many cores as there are simulations. Thus, all simulations execute in parallel at all configurations (of cores and simulations) and we observe constant execution time. The exchange stage in the EE pattern and the analysis stage in the SAL pattern execute in serial and thus dependent on the number of replicas. As expected, their execution time increases with increase in the number of simulations.

The linear behaviour obtained with both patterns is evidence that scalability is invariant of the patterns. The linear scaling is a consequence of the capabilities of the runtime system. This behaviour attests as both an important feature of Ensemble toolkit and validates the choice of using RADICAL-Pilot as the runtime system.

It important to note that although we have used one core for each simulation in these experiments, running multi-core simulations will not change the scaling behaviour. This is due to the fact the overheads, both from Ensemble toolkit and RADICAL-Pilot, depend on the number of tasks as opposed to the size of each task. RADICAL-Pilot is extensible to support most multi-core execution modes [43]. Whereas the size of each task might affect the absolute execution time, the behaviour across different configurations would remain consistent with the above results.

4) MPI capability: We briefly demonstrate the MPI support in Ensemble toolkit by using the same Amber-CoCo MD application, but now with more than one core per simulation. In Figure 11, we fix the number of concurrent simulations at 64, increase the duration of simulation ten-folds to 6ps and vary the number of cores per simulation as 1,16,32,64 (and hence the total no. of cores 64, 1024, 2048, 4096). We see that the execution time of the simulations drops linearly with the number of cores used. Along with the strong and weak scaling experiments, this attests to the fact that given access to the required amount of resources, O(1000) tasks of both MPI and non-MPI type can be supported by Ensemble toolkit.

V. Discussion and Conclusion

After a brief survey of existing approaches to execute ensemble-based applications in Section II, we outlined the

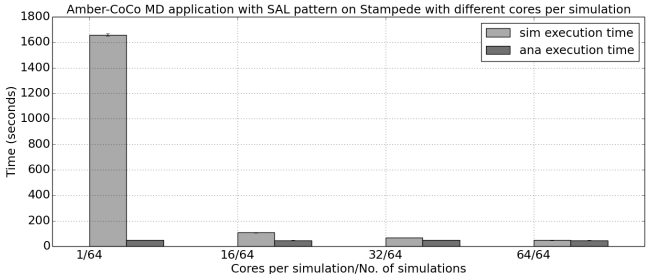

Fig. 11. Amber-CoCo MD application using the SAL execution pattern on XSEDE Stampede with varying no. of cores per simulation for a fixed no. of simulations

primary requirements, design and implementation of Ensemble toolkit in Section III. We introduced three execution patterns that in our assessment represent the majority of ensemble-based applications. Section III, concluded with a discussion of the interface of the user-facing components and an end-to-end example, which served to demonstrate the separation of concerns (ease of development and flexibility versus execution). Experiments in Section IV, validated the design of execution patterns and kernel plugins.

Scalability experiments for the EE pattern and SAL pattern with real workload showed linear strong and weak scaling behavior for up to O(1000) tasks. As measured by overheads, Ensemble toolkit does not impose any significant or fundamental scalability limits; performance and scalability are essentially determined by the pilot-based runtime. RADICAL-Pilot has been engineered to support up to 8K tasks on XSEDE Stampede and 2K tasks on Cray machines without loss of performance [43], [51]; O(10,000) tasks are being tested currently on NSF Blue Waters machines [52]. A technical roadmap exists for O(100,000) concurrent tasks. These runtime performance enhancements will be seamlessly applicable to Ensemble toolkit.

The "building blocks" approach provides an effective middle ground between restrictive and inflexible tools on the one hand, and completely unstructured approaches such as scripting. Ensemble toolkit provides building blocks for developing and executing an ensemble-based application. The execution pattern provide common control-flow options, while the user provides workload definition and resource information. This results in a simple yet flexible approach to compose ensemble-based workflows, which is critical as there is a need to support a range of requirements without wholesale refactoring.

Some tools however, may choose to expose fewer degrees-of-freedom and customization options. For example, ExTASY [53] – which can be viewed as a domain specific workflow tool, uses Ensemble toolkit, but limits the type of execution patterns supported. Ensemble toolkit thus provides some empirical data in the "schism" and debate between general-purpose but monolithic workflow system versus building blocks and abstractions based approach to constructing functionally specialized tools to support special-purpose workflow.

Ensemble toolkit currently in prototype stage supports several active ensemble-based science projects [53], [54]. There are however, multiple enhancements to Ensemble toolkit that are planned or underway: One avenue of research is to identify a *complete set* of unit patterns that can be used to compose any higher order "complex" pattern. The ability to express these higher order patterns as functions of unit patterns will enhance the ability to support more applications. For example, applications involving seismic inversion [55], atmospheric forecasting of air quality, ozone forecast, modelling hurricanes, risk from nuclear release are extensions of existing unit patterns.

Currently, workloads are adapted to resources that are chosen based upon user choice and independent of the dynamic state of workloads. Ref [38] formalized and introduced *execution strategies* as the time-ordered set of decisions needed to execute a workload on dynamic resources. This requires integrating both application-level and resource information, which will see the execution plug-in transition from being a simple translation layer to an intelligent middleware component. The transition from static workload-resource mapping to dynamic mapping will enable efficient and optimized execution capabilities. It will lead to the ability to efficiently select resources for a given workload, as well as form the basis to adapt workloads to optimally utilize a pre-specified of resources.

Many applications do not have pre-defined kernels, number and sequence of tasks. Ensemble toolkit will progressively support more adaptive scenarios, for example the ability to kill-replace tasks [56] and vary the number of tasks between stages. The complexities arise not only in the adaptive execution and workloads, but also in capturing user-facing components. In general, Ensemble toolkit forms an initial prototype of a software system upon which to develop advanced adaptive simulation algorithms, some initial ideas of which are sketched in [57].

Software and Data: Ensemble toolkit can be found at [46] and is released under the MIT license. Raw data and scripts to reproduce experiments can be found at [58].

Acknowledgements: We thank members of the RADICAL group for significant discussion in the design, testing and documentation of Ensemble toolkit: Ming Tai Ha for his feedback and work on testing and documentation, Mark Santcroos and Andre Merzky for help with RADICAL-Pilot, Nikhil Shenoy for initial experiments. We also thank Iain Bethune (EPCC) for testing and feedback on software, and other members of the ExTASY project. We also thank Thomas Cheatham III and Peter Kasson for useful discussion about adaptive execution patterns. We also thank Matthieu Lefebvre, Ryan Modrak and Jeroen Tromp for insight into ensemble applications in seismic tomography. This work was funded by NSF CHE-1265788 and NSF ACI 1440677.

REFERENCES

[1] Jordane Preto and Cecilia Clementi. Fast recovery of free energy landscapes via diffusion-map-directed molecular dynamics. *Physical Chemistry Chemical Physics*, 16(36):19181–19191, 2014.

[2] Thomas E Cheatham III and Daniel R Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science & Engineering*, 17(2):30–39, 2015.

[3] Edwin Sirko. Initial conditions to cosmological n-body simulations, or, how to run an ensemble of simulations. *The Astrophysical Journal*, 634(2):728, 2005.

[4] Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.

[5] Anjani Ragothaman, Sairam Chowdary Boddu, Nayong Kim, Wei Feinstein, Michal Brylinski, Shantenu Jha, and Joohyun Kim. Developing ethread pipeline using saga-pilot abstraction for large-scale structural bioinformatics. *BioMed research international*, 2014, 2014.

[6] Jeffrey Martin, Vincent M Bruno, Zhide Fang, Xiandong Meng, Matthew Blow, Tao Zhang, Gavin Sherlock, Michael Snyder, and Zhong Wang. Rnnotator: an automated de novo transcriptome assembly pipeline from stranded rna-seq reads. *BMC genomics*, 11(1):663, 2010.

[7] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/g: killer application for the global grid? In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 520–528, 2000.

[8] Longbow. http://www.hecbiosim.ac.uk/wikis/index.php/Longbow (accessed January 2016).

[9] Mark Hayes, Lorna Morris, Rob Crouchley, Daniel Grose, Ties Van Ark, Rob Allan, and John Kewley. Growl: A lightweight grid services toolkit and applications. In *4th UK e-Science All Hands Meeting, Nottingham, UK*, 2005.

[10] David A Pearlman, David A Case, James W Caldwell, Wilson S Ross, Thomas E Cheatham, Steve DeBolt, David Ferguson, George Seibel, and Peter Kollman. Amber, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules. *Computer Physics Communications*, 91(1):1–41, 1995.

[11] Bernard R Brooks, Charles L Brooks, Alexander D MacKerell, Lennart Nilsson, Robert J Petrella, Benoît Roux, Youngdo Won, Georgios Archontis, Christian Bartels, Stefan Boresch, et al. Charmm: the biomolecular simulation program. *Journal of computational chemistry*, 30(10):1545–1614, 2009.

[12] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. Gromacs: fast, flexible, and free. *Journal of computational chemistry*, 26(16):1701–1718, 2005.

[13] Steve Plimpton, Paul Crozier, and Aidan Thompson. Lammps-large-scale atomic/molecular massively parallel simulator. *Sandia National Laboratories*, 18, 2007.

[14] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[15] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

[16] Condor Team. Dagman: A directed acyclic graph manager. *See website at http://www. cs. wisc. edu/condor/dagman*, 2005.

[17] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[18] Sander Pronk, Iman Pouya, Magnus Lundborg, Grant Rotskoff, Bjrn Wesn, Peter M. Kasson, and Erik Lindahl. Molecular simulation workflows as parallel algorithms: The execution engine of copernicus, a distributed high-performance computing platform. *Journal of Chemical Theory and Computation*, 11(6):2600–2608, 2015. PMID: 26575558.

[19] Leo Goodstadt. Ruffus: a lightweight python library for computational pipelines. *Bioinformatics*, 26(21):2778–2779, 2010.

[20] Johannes Köster and Sven Rahmann. Snakemakea scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[21] Erik Gafni, Lovelace J Luquette, Alex K Lancaster, Jared B Hawkins, Jae-Yoon Jung, Yassine Souilmi, Dennis P Wall, and Peter J Tonellato. Cosmos: Python library for massively parallel workflows. *Bioinformatics*, page btu385, 2014.

[22] Charles A Laughton, Modesto Orozco, and Wim Vranken. Coco: a simple tool to enrich the representation of conformational variability in nmr structures. *Proteins: Structure, Function, and Bioinformatics*, 75(1):206–216, 2009.

[23] Cameron Abrams and Giovanni Bussi. Enhanced sampling in molecular dynamics using metadynamics, replica-exchange, and temperature-acceleration. *Entropy*, 16(1):163–199, 2013.

[24] Yuji Sugita and Yuko Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chemical physics letters*, 314(1):141–151, 1999.

[25] Andreas D Lattner and Guido Cervone. Ensemble modeling of transport and dispersion simulations guided by machine learning hypotheses generation. *Computers & Geosciences*, 48:267–279, 2012.

[26] S Galmarini, R Bianconi, W Klug, T Mikkelsen, R Addis, S Andronopoulos, P Astrup, A Baklanov, J Bartniki, JC Bartzis, et al. Ensemble dispersion forecastingpart i: concept, approach and indicators. *Atmospheric Environment*, 38(28):4607–4617, 2004.

[27] G Cervone, P Franzese, Y Ezber, and Z Boybeyi. Risk assessment of atmospheric emissions using machine learning. *Natural Hazards and Earth System Science*, 8(5):991–1000, 2008.

[28] S Galmarini, R Bianconi, R Bellasio, and G Graziani. Forecasting the consequences of accidental releases of radionuclides in the atmosphere from ensemble dispersion modelling. *Journal of Environmental Radioactivity*, 57(3):203–219, 2001.

[29] Vivien Mallet, Gilles Stoltz, and Boris Mauricette. Ozone ensemble forecast with machine learning algorithms. *Journal of Geophysical Research: Atmospheres*, 114(D5), 2009.

[30] Thomas M Hamill, Michael J Brennan, Barbara Brown, Mark DeMaria, Edward N Rappaport, and Zoltan Toth. Future of ensemble-based hurricane forecast products. *Bull. Amer. Meteor. Soc*, 2010.

[31] Paul J Roebber, David M Schultz, Brian A Colle, and David J Stensrud. Toward improved prediction: High-resolution and ensemble modeling systems in operations. *Weather and Forecasting*, 19(5):936–949, 2004.

[32] Hui Wan, Philip J Rasch, Kai Zhang, Yun Qian, Huiping Yan, and Chun Zhao. Short ensembles: an efficient method for discerning climate-relevant sensitivities in atmospheric general circulation models. *Geoscientific Model Development*, 7(5):1961–1977, 2014.

[33] Augusto CV Getirana, Emanuel Dutra, Matthieu Guimberteau, Jonghun Kam, Hong-Yi Li, Bertrand Decharme, Zhengqiu Zhang, Agnes Ducharne, Aaron Boone, Gianpaolo Balsamo, et al. Water balance in the amazon basin from a land surface model ensemble. *Journal of Hydrometeorology*, 15(6):2586–2614, 2014.

[34] Aixue Hu and Clara Deser. Uncertainty in future regional sea level rise due to internal climate variability. *Geophysical Research Letters*, 40(11):2768–2772, 2013.

[35] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[36] S. K. Sadiq, D. W. Wright, O. A. Kenway, P. V. Coveney, "Accurate Ensemble Molecular Dynamics Binding Free Energy Ranking of Multidrug-Resistant HIV-1 Proteases", Journal of Chemical Information and Modeling, 50, 890-905, (2010), DOI: 10.1021/ci100007w.

[37] Hugh S. C. Martin, Shantenu Jha, and Peter V.Coveney. Comparative analysis of nucleotide translocation through protein nanopores using steered molecular dynamics and an adaptive biasing force. *Computational Chemistry*, 35(9):692–702, April 2014. http://onlinelibrary.wiley.com/doi/10.1002/jcc.23572/abstract.

[38] Matteo Turilli, Feng (Francis) Liu, Zhao Zhang, Andre Merzky, Michael Wilde, Jon Weissman, Daniel S. Katz, and Shantenu Jha. Integrating Abstractions to Enhance the Execution of Distributed Applications. In *Proceedings of 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016. http://arxiv.org/abs/1504.04720.

[39] Tom Goodale, Shantenu Jha, Harmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. SAGA: A Simple API for Grid applications, High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[40] Ali Anjomshoaa, Fred Brisard, Michel Drescher, Donal Fellows, An Ly, Stephen McGough, Darren Pulsipher, and Andreas Savva. Job submission description language (jsdl) specification, version 1.0. In *Open Grid Forum, GFD*, volume 56, 2005.

[41] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P*: A model of pilot-abstractions. *IEEE 8th International Conference on e-Science*, pages 1–10, 2012. http://dx.doi.org/10.1109/eScience.2012.6404423.

[42] Matteo Turilli, Mark Santcroos, and Shantenu Jha. A Comprehensive Perspective on Pilot-Jobs, 2016. (under review) http://arxiv.org/abs/1508.04180.

[43] Andre Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. RADICAL-Pilot: Scalable Execution of Heterogeneous and Dynamic Workloads on Supercomputers, 2015. (under review) http://arxiv.org/abs/1512.08194.

[44] Andre Merzky, Ole Weidner, and Shantenu Jha. SAGA: A standard-

ized access layer to heterogeneous distributed computing infrastructure. *Software-X*, 2015. DOI: 10.1016/j.softx.2015.03.001.
[45] Tacc stampede. https://www.xsede.org/stampede (accessed January 2016).
[46] Ensemble toolkit. http://radicalensemblemd.readthedocs.org/en/latest (accessed January 2016).
[47] Sdsc comet. https://portal.xsede.org/sdsc-comet/ (accessed January 2016).
[48] Lsu supermic. https://portal.xsede.org/lsu-supermic (accessed January 2016).
[49] Gromacs. http://www.gromacs.org/ (accessed October 2015).
[50] Amber. http://www.ambermd.org/ (accessed October 2015).
[51] Mark Santcroos, Ralph Castain, Andre Merzky, Iain Bethune, and Shantenu Jha. Executing Dynamic Heterogeneous Workloads on Crays with RADICAL-Pilot. In *CRAY User Group 2016*, 2016.
[52] http://www.nsf.gov/awardsearch/showAward?AWD_ID=1516469&HistoricalAwards=false.
[53] Extasy. http://extasy-project.org/ (accessed January 2016).
[54] Antons Treikalis, Andre Merzky, Darrin York, and Shantenu Jha. RepEx: A Flexible Framework for Scalable Replica Exchange Molecular Dynamics Simulations, 2016. (under review) http://arxiv.org/abs/1601.05439.
[55] Carl Tape, Qinya Liu, and Jeroen Tromp. Finite-frequency tomography using adjoint methodsmethodology and examples using membrane surface waves. *Geophysical Journal International*, 168(3):1105–1129, 2007.
[56] http://www.nsf.gov/awardsearch/showAward?AWD_ID=1521728.
[57] Shantenu Jha and Peter M. Kasson. High-level software frameworks to surmount the challenge of 100x scaling for biomolecul ar simulation science. White Paper submitted to NIH-NSF Request for Information (2015) http://www.nsf.gov/pubs/2016/nsf16008/nsf16008.jsp. http://dx.doi.org/10.5281/zenodo.44377.
[58] https://github.com/radical-experiments/enmd-pattern-testing.