StormCloud: Sky Computing for DataStorm, A Coupled Simulation Ensemble Framework

A Preprint

Hans Walter Behrens, Hardik Shah, Yash Shah
Tempe, AZ 85281-3673
{hwb@asu.edu, hrshah5@asu.edu, ygshah@asu.edu}

1 Problem Statement

In 2016, flooding in Texas caused residents to evacuate their homes with only hours of warning, despite hurricane simulations accurately predicting the path of the storm weeks in advance. Unfortunately, rainfall predictions from the hurricane simulations could not be fed directly into hydrographic models which predicted water flows, leading to a failure in preparation. Many such scientific simulators exist, written by domain experts, but compatibility and scalability problems prevent their interoperation.

DataStorm [1] is a workflow engine designed to assist in the creation of these types of coupled, heterogeneous simulation ensembles. In doing so, it solves a wide variety of challenges across many computing subdomains, including data interoperability, scalability, temporal and spatial alignment, and sparse simulation space sampling, among others.

In this work, we propose StormCloud, an extension to the DataStorm framework that allows for the integration of PaaS-based models, as well as hybrid deployments of the cluster across several different cloud service providers. Many simulators are amenable to deployment according to a PaaS paradigm, greatly simplifying the process of readying a model for integration with DataStorm. Additionally, the adaptive scaling offered by PaaS solutions such as Google App Engine (GAE) are well-suited to the periodic but computationally-intensive loads required by DataStorm.

Specifically, StormCloud aims to achieve the following research goals:

1. Permit deployment of model instances to PaaS providers, including Google App Engine.
2. Seamlessly integrate existing DataStorm functionality across different cloud providers.
3. Adaptively scale PaaS-deployed models to incoming simulation workloads.

In the following sections, we will discuss the design and implementation of StormCloud (Section 2), following by a discussion of our system testing, integration, and evaluation process (Section 3). We conclude with a description and detailed walkthrough of the components of StormCloud (Section 4).

2 Design & Implementation

2.1 Architecture

StormCloud is based on the concept of using multiple cloud providers simultaneously, also known as "sky computing" [2], to leverage the complementary strengths of each. Specifically, our system uses ChameleonCloud, an OpenStack-based IaaS provider, as well as Google Cloud Platform's Google App Engine, as part of a hybrid system.

A high-level architectural overview at the instance level can be seen in Figure 1. DataStorm stores much of its data in MongoDB, which works well with temporally-ordered and unstructured data that is common to the model environment.

However, although this figure illustrates the low-level communication channels between the models, it also obscures important semantic and operational relationships that underlie the functionality of the DataStorm/StormCloud system.

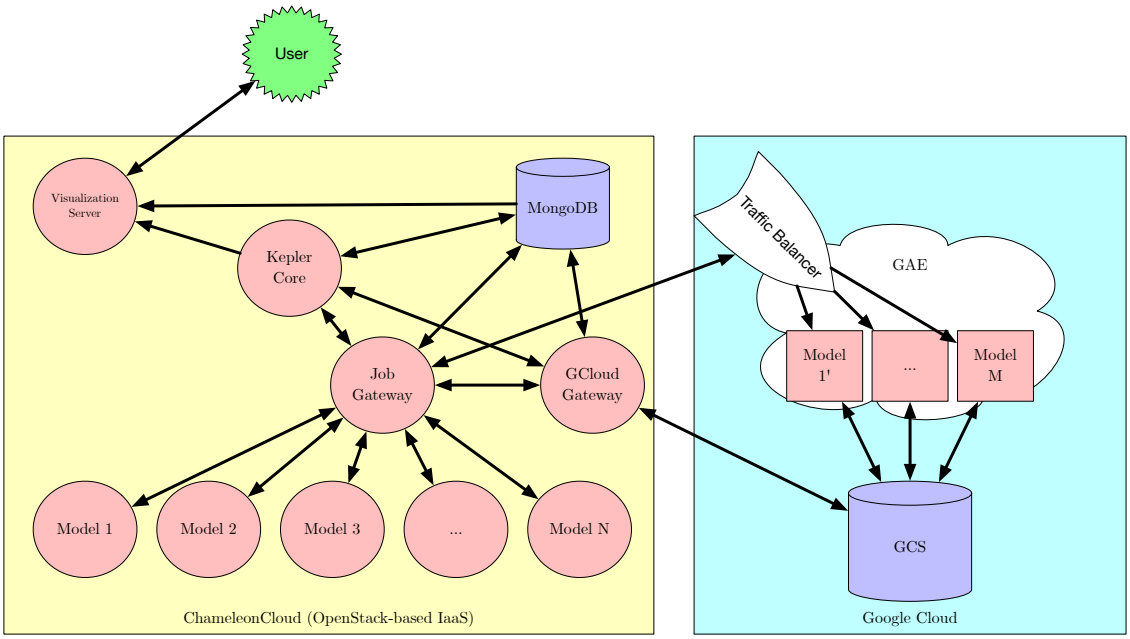

Figure 1: A high-level architectural overview of StormCloud

Specifically, the control flow of the system is managed by the Kepler instance at two levels of granularity. The first level is controlled by the *model dependency graph* (MDG), a map of the relationships between models which affect each other. For example, a hurricane model generates rainfall data, which will intuitively have an effect on the output of a flooding model. An example of a dependency graph can be seen in Figure 2.

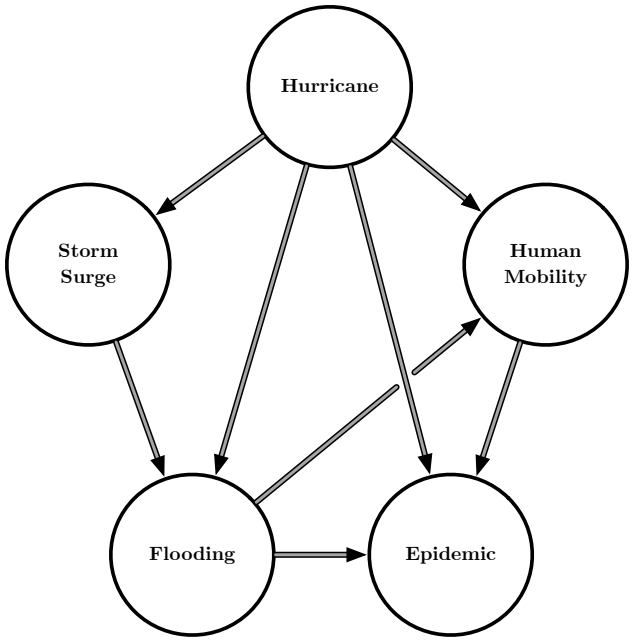

Figure 2: An example of a *model dependency graph* (MDG) between five constituent models (or DS-Actors).

In this example, the hurricane model is the initial model; it does not take any model-produced inputs, as there is very little which can affect the path of a hurricane. This model starts with a *control token*, which is managed by the Kepler workflow system. The dependency graph is then traversed in a breadth-first fashion so that downstream models'

upstream input dependencies can be met. For this reason, it is crucial the dependency graph is directed and acyclic (a DAG), as cycles could lead to dependency deadlocks.

The DAG is traversed *asynchronously*. When traversing a node, the ownership of the control token is not deterministic, but is based instead of the progress towards the goal that the model at each node has made. It may be passed if the model has completed the simulation and reported its results, or it may be retained at a particular state to allow for additional processing to occur. When a single traversal has been completed, another traversal begins immediately to provide additional opportunities for control token resolution to occur. This also provides an opportunity to provide

Each model node is also granular, in the sense that its progress is non-binary. In actuality, each node is represented in the Kepler system as a DS-Actor, a subgraph of processing modules responsible for different phases of computation. The DS-Actor topology can be seen in Figure 3.

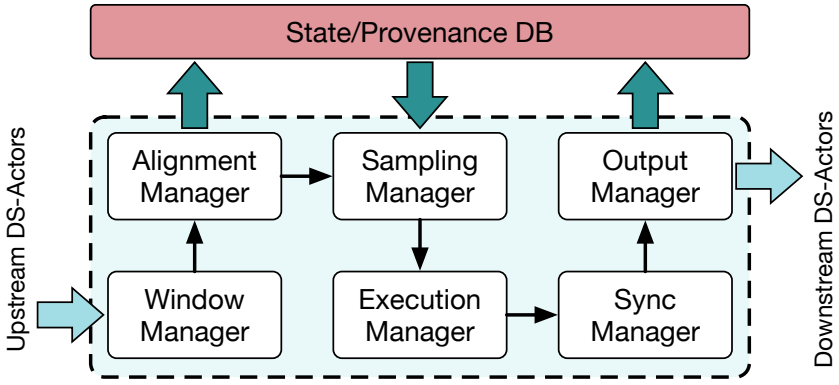

Figure 3: Detailed view of a single DS-Actor module (and its submodules).

The role that the DS-Actor submodules play are summarized as follows:

1. **Window Manager**: Responsible for enforcing input availability (satisfiability) constraints from upstream models, and spatiotemporal input partitioning.
2. **Alignment Manager**: Up- or down-samples inputs (if necessary) to ensure spatial and temporal alignment between upstream resolutions and local requirements.
3. **Sampling Manager**: Considers previous results, user inputs, computational budgets, and parametric inputs to generate a high-dimensional simulation space tensor, representing all possible simulation configurations which could be executed. It them samples from this space, generating a number of simulation jobs constrained by the budget while maximizing usefulness. Also manages job allocation for IaaS-based model instances, to allow for load-adaptive balancing [3].
4. **Execution Manager**: Provides a unified interface for running distributed, heterogeneous simulator implementations in parallel when possible, and in sequence if not.
5. **Post-Synchronization Manager**: Provides an opportunity to aggregate or prune raw simulation results, greatly constraining the potential future simulation space, or increase predictive accuracy through result clustering and other analytical post-processing.
6. **Output Manager**: Responsible for notifying the visualization server that new results are ready to be displayed, enabling push-based real-time monitoring of evolving conditions as results become available. Users operating under the interactive mode, or who are in collaboration lobbies, will not be affected by this notification.

Result tracking, provenance management, and model explicability have been elided from this report, as they are out of scope with respect to the contributions of StormCloud.

2.2 Autoscaling

The number of jobs allocated to a particular simulator is determined by the SamplingManager, which takes into account existing upstream data and user preferences to generate a number of simulation jobs. When IaaS-based instances are used, jobs are allocated to specific instances for each model, to allow for informed, adaptive workloads. That is, the

SamplingManager can use its insight into the relative difficulty of each job so that the total level of effort is equalized across the available instances.

Since DataStorm relies on Kepler to keep the workflow running in a continuous, asynchronous loop, once the jobs are allocated by the SamplingManager, responsibility for their execution is handed off (with the control token) to the ExecutionManager. The ExecutionManager is then responsible for actually starting the simulations on their respective jobs. For non-GAE models, this is done by issuing commands by referencing a cluster topology for state encoding.

However, for PaaS-enabled systems such as the flood model prototype enabled by StormCloud, the job queues generated by the SamplingManager are not used. Instead, jobs are allocated to the back-end system simultaneously using a Flask shim behind a Gunicorn load balancer integrated with the cloud provider. Then, the adaptive balancing of the cloud provider is used to ensure load fairness among the available workers, as well as the dynamic creation and destruction of workers to balance the load for longer-running jobs.

One important factor lies in the transfer of data between the central temporal repository in MongoDB and the GAE workers themselves, which necessarily must have the (potentially large) upstream input data available to them in order to execute. This data hand-off is managed through an additional component, the `gcloud-gateway`. It is responsible for translating the information (if necessary) before handing it off to the GAE workers, as well as exfiltrating the data from the MongoDB, via the intranet, to Google Cloud Storage (GCS), which provides highly scalable storage sufficient for any possible input file size.

Google Cloud Storage is also used as a simple glue layer between the ExecutionManager and the GAE workers, to ensure that jobs do not begin processing until all components have been loaded and are ready. We chose this as the data size limitations were much less restrictive than TaskQueues.

3 Testing & Evaluation

3.1 Considerations

System evaluation for a simulation ensemble tool like DataStorm carries implicit challenges, since it attempts to improve the accuracy of predictions for future events. As such, there can be no ground truth data to evaluate against until after the events have occurred.

Consequently, it makes sense to instead evaluate it against historical events, and compare the eventual predictions against the ground truth of what actually occurred. However, the spatial and temporal resolution of modern data, and its availability to the public at all, complicates this process for older events. The quantity, detail, and format of this historical data is not comparable to modern day data collection techniques, which means that inference mining in the simulation sampling space encounters much more noise than with newer data. This can lead to decreases in accuracy.

Therefore, the most reliable metric by which DataStorm can be evaluated is through expert evaluation. Specifically, we feed in recent available data, and domain experts can then comment on the relative accuracy or believability of the predictions, weighted by the confidence values DataStorm associated to those predictions.

However, for this project, the goals are less about improving the predictive accuracy of the system, and more about increasing the flexibility of deployment for future models. Therefore, our testing regime focuses on the scalability and flexibility of the system, rather than its accuracy.

3.2 Testing

To that end, we have employed the following testing and evaluation scheme:

1. Deploy DataStorm to a cluster instantiated using IaaS, using the OrchestrationManager, a deployment tool written in Python and integrated with Ansible and Paramiko. This is easiest to accomplish using OpenStack, as it can leverage the availability of OS APIs to simplify the deployment process.
2. Utilize an existing cluster configuration (DS-Config) that describes a simple two-model MDG, with 'hurricane' leading to 'flood'.
3. Deploy the Google App Engine-compatible flood model to GAE using `gcloud app deploy`.
4. Start DataStorm.
5. Observe the creation and display of correct results using the visualization component.

In order to test the Google App Engine

3.3 Results

The most easily understood results of DataStorm, in the absence of subject matter experts, can be seen in the data visualization component. An example of composited data layers can be seen in Figure 4. Additional functionality for data exploration is available in this component, but is not discussed in detail as it is outside of the scope of StormCloud.

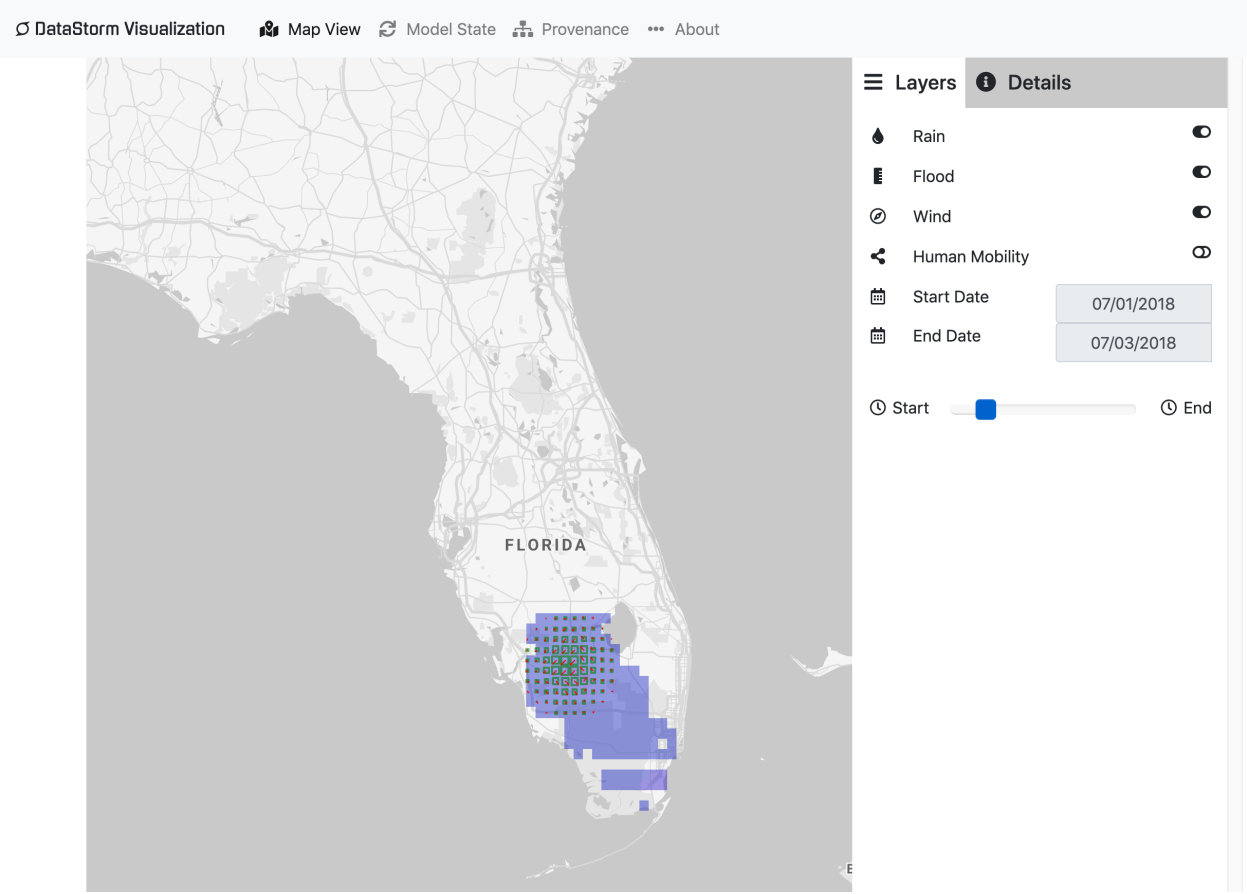

Figure 4: An example screenshot from the data visualization, showing both hurricane output from ChameleonCloud and flood output from GAE.

4 Code

4.1 Deliverables

We have provided code for `gcloud-gateway.py` and for `gcloud-flood-modeller.py` at the following repository: `https://github.com/hwbehrens/cse546-s19-proj2`. It also contains a copy of this report. Note that these files are extracted from their deployment context inside DataStorm, for simplicity of evaluation.

The code base for the parent project can be found here: `https://github.com/EmitLab/DataStorm`. We have not provided detailed documentation for the DataStorm project, as it is outside the scope of this work[1].

4.2 Component: `gcloud-gateway`

This component is responsible for providing a compatibility shim between the data storage system of DataStorm (MongoDB) and the storage used by the GAE-deployed components, which use Google Cloud Storage. Specifically, it identifies, fetches, and collates the data from MongoDB, converts it from BSON to a UTF-8 encoded JSON string, and then pushes that information to the cloud as a new job-associated object.

[1]This exclusion was approved by Dr. Zhao, as deploying an instance of a DataStorm cluster can be extremely complex.

Once all the information has been uploaded, it then notifies the model workers (described below) that a new job is waiting to be completed. It then waits for a response from the worker. Once this response is received, it processes the resulting data, embeds the necessary provenance information, and injects the results back into MongoDB for later processing and display.

4.3 Component: `gcloud-flood-modeller`

This component is a simplified simulation model suitable for demonstration; it collects rainfall data from upstream models, and considers the effect that this will have on terrestrial water depths using a simple mathematical flood model. It also includes a Flask shim to allow communication between the GAE workers that instantiate this code and the external gateway coordinators that will be submitted jobs to those workers.

Once it has received a job, it records the task ID, and fetches all the necessary information to operate on that task from Google Cloud Storage. Once the information has been fetched, it runs the model, collects the results, and stores them back into the Google Cloud Storage. It then notifies the coordinator that initiated that job that the results are ready for collection.

References

[1] H. W. Behrens, K. S. Candan, X. Chen, A. Gadkari, Y. Garg, M.-L. Li, X. Li, S. Liu, N. Martinez, J. Mo *et al.*, "Datastorm-fe: a data-and decision-flow and coordination engine for coupled simulation ensembles," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1906–1909, 2018.

[2] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky computing," *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[3] H. W. Behrens, M.-L. Li, A. Gadkari, Y. Garg, X. Chen, S. Liu, and K. S. Candan, "Load-adaptive continuous coupled-simulation ensembles with datastorm and chameleon," *Chameleon User Meeting*, 2019.