Author Manuscript

HHS Public Access

Author manuscript

SIAM J Sci Comput. Author manuscript; available in PMC 2017 February 10.

Published in final edited form as:

SIAM J Sci Comput. 2016 ; 38(3): C179–C202. doi:10.1137/15M1014784.

MOLNs: A CLOUD PLATFORM FOR INTERACTIVE, REPRODUCIBLE, AND SCALABLE SPATIAL STOCHASTIC COMPUTATIONAL EXPERIMENTS IN SYSTEMS BIOLOGY USING PyURDME

Brian Drawert[†], **Michael Trogdon**[‡], **Salman Toor**[§], **Linda Petzold**[¶], and **Andreas Hellander**[‖]

[†]Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA 93106

[‡]Department of Mechanical Engineering, University of California, Santa Barbara, Santa Barbara, CA 93106

[§]Department of Computer Science, University of Helsinki, Helsinki FI-00014, Finland, and Department of Information Technology, Division of Scientific Computing, Uppsala University, Uppsala, 75105 Sweden

[¶]Departments of Computer Science and Mechanical Engineering, University of California, Santa Barbara, Santa Barbara, CA 93106

Abstract

Computational experiments using spatial stochastic simulations have led to important new biological insights, but they require specialized tools and a complex software stack, as well as large and scalable compute and data analysis resources due to the large computational cost associated with Monte Carlo computational workflows. The complexity of setting up and managing a large-scale distributed computation environment to support productive and reproducible modeling can be prohibitive for practitioners in systems biology. This results in a barrier to the adoption of spatial stochastic simulation tools, effectively limiting the type of biological questions addressed by quantitative modeling. In this paper, we present PyURDME, a new, user-friendly spatial modeling and simulation package, and MOLNs, a cloud computing appliance for distributed simulation of stochastic reaction-diffusion models. MOLNs is based on IPython and provides an interactive programming platform for development of sharable and reproducible distributed parallel computational experiments.

Keywords

simulation software; spatial stochastic simulation; systems biology; computational experiments; cloud computing

---

[‖]Corresponding author. Department of Information Technology, Division of Scientific Computing, Uppsala University, Uppsala, 75105 Sweden (andreas.hellander@it.uu.se).

The authors declare that they have no conflict of interest.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

1. Introduction

In computational systems biology, one of the main goals is to understand how intracellular regulatory networks function reliably in a noisy molecular environment. To that end, discrete stochastic mathematical modeling has emerged as a prominent tool. Stochastic simulation of well-mixed systems is now routinely used [3, 37, 9, 31], and recently, spatial stochastic models have resulted in important scientific insights [10, 21, 36], clearly demonstrating the potential as an analytic tool in the study of cellular control systems. Compared to less detailed models such as ordinary differential equations (ODEs), well-mixed discrete stochastic models, or partial differential equations (PDEs), spatial stochastic models are both more costly to simulate and more difficult to formulate and set up. The large simulation cost of stochastic reaction-diffusion simulations has led to the development of more efficient algorithms; an overview of theory and methods for discrete stochastic simulations can be found in [15]. Several software packages are publicly available, both for mesoscopic, discrete stochastic simulation [16, 18, 5] and microscopic particle tracking based on Brownian dynamics [2, 38, 35, 32]. A recent overview of particle based simulators can be found in [33].

While efficient simulation methods are critical for well-resolved spatial models, practical modeling projects require the support provided by a software framework. In the early stages of the model development process, there is typically no need for large compute resources. In later stages, computational experiments generate large numbers of independent stochastic realizations. This is common to all applications that rely on Monte Carlo techniques. For spatial stochastic models, substantial computational and data handling facilities are required. A simulation framework that focuses on modeler productivity needs to accommodate both interactivity and visual feedback, as well as the possibility of large-scale simulation and data handling. To be cost and resource efficient, it should also support dynamic scaling of compute and storage resources to accommodate the needs in different stages of the modeling process.

Since most successful modeling projects involve a multidisciplinary team of researchers, it is important that models can be shared and understood by team members with different areas of expertise. Formats for model exchange based on static markup language descriptions such as the systems biology markup language (SBML) [19] or Open Modeling EXchange format (OMEX) [4] are useful to standardize descriptions of simple ODE and well-mixed stochastic models, but they fall short when it comes to complex spatial models. Recently, numerous developers of spatial simulation packages have taken another approach and provided application programming interfaces (APIs) for model specification in a scripting language [24, 18, 38], with Python being a popular choice. Our newly developed package PyURDME falls into this category. We will show how PyURDME, being designed with the IPython suite in mind, can be used to program spatial stochastic models as highly interactive and sharable notebooks. In addition, we note that by providing a virtual cloud appliance, not only the models but also the computational experimental workflow including the computing environment becomes easily reproducible.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

In previous work, we have developed the URDME (Unstructured mesh Reaction-Diffusion Master Equation) framework for discrete stochastic simulation of biochemical reaction-diffusion systems [5]. URDME was designed primarily as a traditional, native toolkit that combines MATLAB and COMSOL Multiphysics to form an interactive modeling and simulation environment. The design of URDME has proven useful to support both methods' development and modeling, but the framework has limitations when it comes to assisting large-scale Monte Carlo computational experiments. URDME can be executed on clusters or grid resources [27]. However, doing this typically requires computer science knowledge beyond that of the average practitioner and access to high-performance computing (HPC) environments. This distracts users from the science problems addressed, and it acts as a barrier to scale up the computational experiments as needed for a consistent statistical analysis. Further, the computational experiment becomes hard to reproduce since the provenance relies on specific resources not accessible to third parties.

Based on the above observations, we argue that the classical view of the scientific application (in our case PyURDME), as being separate from the compute, storage, and data analysis tools employed, is restrictive. Enhanced modeling productivity and reproducibility would result if the computational infrastructure and the software stack were combined into a unified appliance. Hence, the aim of this work has been to develop a platform that

1. allows interactive development of spatial stochastic models supported by basic visualization capabilities,
2. facilitates collaboration and reproducibility,
3. allows for convenient and efficient execution of common computational experiments, such as estimation of mean values, variances, and parameter sweeps,
4. is close-to-data and allows for flexible specification of custom postprocessing,
5. allows for flexibility in the choice of computational infrastructure provider and dynamic scaling of computing resources, and
6. requires no more than basic computer science knowledge to deploy and manage.

To meet all these requirements, we have developed MOLNs, a cloud computing appliance that configures, builds, and manages a virtual appliance for spatial stochastic modeling and simulation on public, private, and hybrid clouds. By relying on cloud computing and its resource delivery model, the responsibility for handling the complex setup of the software stack is shifted from the users to the developers since we can prepare virtual machines that are preconfigured and ready to use. With support for the most common public clouds such as Amazon Elastic Compute Cloud (EC2) and HP Helion, we ensure high availability and scalability of computational resources. By supporting OpenStack, an open source cloud environment commonly used for private (in-house) cloud installations, MOLNs brings the flexibility and tools of cloud computing to the user's own servers. Taking it one step further, MOLNs provides support for hybrid deployments in which private and public cloud resources can be combined, allowing the use of in-house resources and bursting to public clouds during particularly compute-intensive phases of a modeling project. Interactivity is

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

achieved by building on Interactive Python (IPython), in particular the web-based IPython Notebook project [28, 30]. See illustration in Figure 1.

We demonstrate the potential of MOLNs to greatly assist computational experimentation in a case study of yeast polarization and evaluate its performance in parallel, distributed performance benchmarks. While the current computational engine is our newly developed Python package PyURDME, we believe that users as well as developers of other spatial simulation tools could benefit greatly from the delivery model proposed in our virtual platform. All components of the software presented here, as well as all models (and many more), are publicly available under open source licenses that permit unlimited redistribution for noncommercial purposes under the GPLv3 license at https://github.com/MOLNs/MOLNs.

2. Stochastic simulation of spatially inhomogeneous discrete biochemical systems

Recent advances in biology have shown that proteins and genes often interact probabilistically. The resulting effects that arise from these stochastic dynamics differ significantly from traditional deterministic formulations and have biologically significant ramifications. This has led to the development of discrete stochastic computational models of the biochemical pathways found in living organisms. These include spatial stochastic models, where the physical extent of the domain plays an important role. For mesoscopic models, similar to popular solution frameworks for partial differential equations (PDEs), the computational domain is discretized with a computational mesh, but unlike PDEs, the reaction-diffusion dynamics are modeled by a Markov process where diffusion and reactions are discrete stochastic events. The dynamics of a spatially inhomogeneous stochastic system modeled by such a Markov process formalism are governed by the reaction-diffusion master equation (RDME) [12].

The RDME extends the classical well-mixed Markov process model [14] to the spatial case by introducing a discretization of the domain into K nonoverlapping voxels. Molecules are point particles, and the state of the system is the discrete number of molecules of each of the species in each of the voxels on Cartesian grids or unstructured triangular and tetrahedral meshes. The RDME is the forward Kolmogorov equation governing the time evolution of the probability density of the system. For brevity of notation, we let $p(\mathbf{x}, t) = p(\mathbf{x}, t|\mathbf{x}_0, t_0)$ for the probability that the system can be found in state $\mathbf{x}$ at time t, conditioned on the initial condition $\mathbf{x}_0$ at time t_0. For a general reaction-diffusion system, the RDME can be written as

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

$$\frac{\mathrm{d}}{\mathrm{dt}}p(\mathbf{x},t)=\sum_{i=1}^{K}\sum_{r=1}^{M}a_{ir}(\mathbf{x}_{i\cdot}-\boldsymbol{\mu}_{ir})p(\mathbf{x}_{1\cdot},\ldots,\mathbf{x}_{i\cdot}-\boldsymbol{\mu}_{ir},\ldots,\mathbf{x}_{K\cdot},t)-\sum_{i=1}^{K}\sum_{r=1}^{M}a_{ir}(\mathbf{x}_{i\cdot})p(\mathbf{x},t)$$
$$+\sum_{j=1}^{N}\sum_{i=1}^{K}\sum_{k=1}^{K}d_{jik}(\mathbf{x}_{\cdot j}-\nu_{ijk})p(\mathbf{x}_{\cdot 1},\ldots,\mathbf{x}_{\cdot j}-\nu_{ijk},\ldots,\mathbf{x}_{\cdot N},t)$$
$$-\sum_{j=1}^{N}\sum_{i=1}^{K}\sum_{k=1}^{K}d_{ijk}(\mathbf{x}_{\cdot j})p(\mathbf{x},t), \quad (1)$$

where $\mathbf{x}_{i\cdot}$ denotes the ith row and $\mathbf{x}_{\cdot j}$ denotes the jth column of the $K \times S$ state matrix $\mathbf{x}$, where S is the number of chemical species. The functions $a_{ir}(\mathbf{x}_i)$ define the propensity functions of the M chemical reactions, and $\boldsymbol{\mu}_{ir}$ are stoichiometry vectors associated with the reactions. The propensity functions are defined such that $a_{ir}(\mathbf{x})$ t gives the probability that reaction r occurs in a small time interval of length t. The stoichiometry vector $\boldsymbol{\mu}_{ir}$ defines the rules for how the state changes when reaction r is executed. $d_{ijk}(\mathbf{x}_i)$ are propensities for the diffusion jump events, and ν_{ijk} are stoichiometry vectors for diffusion events. ν_{ijk} has only two nonzero entries, corresponding to the removal of one molecule of species X_k in voxel i and the addition of a molecule in voxel j. The propensity functions for the diffusion jumps, d_{ijk}, are selected to provide a consistent and local discretization of the diffusion equation, or equivalently the Fokker–Planck equation for Brownian motion.

The RDME is too high-dimensional to permit a direct solution. Instead, realizations of the stochastic process are sampled, using kinetic Monte Carlo algorithms similar to the stochastic simulation algorithm (SSA) [14] but optimized for reaction-diffusion systems. State-of-the-art algorithms such as the next subvolume method (NSM) [7] rely on priority queues and scale as $\mathcal{O}(\log_2(K))$, where K is the number of voxels in the mesh. The computational cost of spatial stochastic simulation depends on the number of reaction and diffusion events that occur in a simulation, since exact kinetic Monte Carlo (KMC) methods sample every individual event. The number of diffusion events in the simulation scales as $O(h^{-2})$, where h is a measure of the mesh resolution. This leads to stochastic stiffness, where diffusion events greatly outnumber reaction events for fine mesh resolutions. This has led to the development of hybrid and multiscale methods to improve the situation. For an overview see [15].

Despite the large computational cost, mesoscopic simulation with the RDME, when applicable, is typically orders of magnitude faster than alternatives such as reactive Brownian dynamics. Individual realizations can be feasibly sampled for fairly complex models in complicated geometries on commodity computational resources such as laptops and workstations. However, since the models are stochastic, single realizations are not sufficient. Rather, large ensembles of independent samples of the process need to be generated to form a basis for statistical analysis. Furthermore, key parameters of the biological process may be known only to an order of magnitude or two, thus necessitating an exploration of parameter space and/or parameter estimation. The need for an infrastructure to manage the computation and data has motivated the development of PyURDME and the MOLNs platform.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

3. Results

3.1. Construction of spatial stochastic models with PyURDME

PyURDME (www.pyurdme.org) is a native Python module for the development and simulation of spatial stochastic models of biochemical networks. It is loosely based on the URDME [5] framework, in that it replicates the functionality of URDME's core and uses modified versions of the stochastic solvers. While URDME was designed as an interactive MATLAB package, using COMSOL Multiphysics for geometric modeling and meshing, PyURDME is a Python module providing an object-oriented API for model construction and execution of simulations. PyURDME relies only on open source software dependencies and uses FEniCS/Dolfin [22] as a replacement for the facilities that COMSOL provided for URDME.

Creating a model in PyURDME involves implementing a class that extends a base model, *URDMEModel*, where information about chemical species, reaction rates, reactions, and the geometry and mesh are specified in the constructor. There is a minimal amount of Python code that is easily readable and powerful enough to extend to more complex models quite intuitively. Then, spatial stochastic solvers, each based on a base-class *URDMESolver*, can be instantiated from a reference of the model. After executing simulations, results are encapsulated in an *URDMEResult* object. The excerpt of an IPython notebook [28] in Figure 2 illustrates specification and execution of a model of spontaneous polarization in yeast [1]. We will use the development and analysis of this model as a case study later in this paper. In the supplementary material, which is linked from the main article webpage, we provide in-depth explanations of the design and workings of the key classes *URDMEModel*, *URDMESolver*, and *URDMEResult*.

The *URDMESolver* class provides an interface to spatial stochastic solvers. The current core solver in PyURDME is a modified version of the NSM [7] core solver in the URDME framework [5]. It is implemented in C, and we follow the same execution mechanism as in [5]. Upon execution of the solver (e.g., the *model.run()* command in Figure 2), PyURDME uses the model specification encoded in *YeastPolarization* to assemble the data structures needed by the core solver. It also generates a C file specifying the reaction propensity functions and compiles a binary for execution of the specific model. The binary solver is then executed as a separate process. The core solver executes the NSM method and generates a spatio-temporal time series data set which is written to the compressed binary file in the HDF5 format [39]. For exact reproduction of a single run, PyURDME's *run()* function takes as an argument *seed=N*, where *N* is the desired value of the random number initialization.

Though all of the functionality of PyURDME is available when using it as a native library on a local client (such as a user's laptop), we provide additional functionality to enhance the usability when integrated in MOLNs. For example, simulation results can be visualized with a three-dimensional (3D) rendering of the mesh or domain inline in an IPython Notebook using the JavaScript library three.js [40] (as illustrated in Figure 2). Additionally, special attention has been paid to make the instances of *URDMEModel*, *URDMESolver*, and *URDMEResult* serializable. This enables PyURDME to integrate with the *IPython.Parallel*

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

library, the distributed computing facilities of IPython, and is an important property that prepares PyURDME for distributed computing. PyURDME models need not be developed in a tightly coupled manner on the MOLNs platform, but a benefit of doing so is that it enables seamless integration with the development and visualization facilities and allows the computational scientists to easily harness the computational power of the large-scale distributed computational cloud computing environment.

3.2. The MOLNs cloud platform

The MOLNs cloud computing platform has three major components, as shown in Figure 3. The first component is the IPython notebook web interface, which provides a widely used and familiar user interface for the MOLNs platform. The second component is the *molnsclient*, a command line interface (CLI) which is responsible for the setup, provisioning, creation, and termination of MOLNs clusters on private or public cloud computing infrastructure services. The final component is the *molnsutil* package, which provides a high-level API for distributed simulation and postprocessing of Monte Carlo workflows with PyURDME. Together, these components make up a powerful and easy to use tool for harnessing the computational power and high availability of cloud computing resources in an interactive, sharable, and reproducible manner.

3.2.1. IPython notebook server—The first component of the MOLNs platform is an IPython notebook server. The IPython notebook is a web-based interactive computational environment where code execution, text, mathematics, plots, and rich media can be combined into a single document. The main goal of the IPython project has been to provide interactive computing for scientists [28], and it has gained widespread use in the scientific community. IPython notebooks are "computable documents," and this makes them ideal to present easily reproducible and shareable scientific results [30]. IPython Notebook was recently suggested in a *Nature* editorial to be a promising tool for addressing the lack of reproducibility of computational biology results [34]. An example of the usage of PyURDME in such a notebook is shown in Figure 2.

While the notebooks contain the information needed to share and reproduce the model and the structure of the computational experiment, other important parts of the provenance of a computational experiment are the compute infrastructure and the software stack. For computational experiments, the software stack is often quite complex, and a notebook does not provide a way to set up an environment in which it can be executed. For spatial stochastic simulations, this is complicated further by the need for complex HPC infrastructure. This is addressed by *molnsclient*.

3.2.2. Molnsclient—The second component of the MOLNs software is the *molnsclient*, which is responsible for the infrastructure management of cloud computing resources. It is a CLI for provisioning the MOLNs clusters, i.e., starting and terminating the virtual machine instances on the cloud computing service providers. This is represented by the gray lines in Figure 3. The configuration of *molnsclient* is organized into *Providers*, *Controllers*, and *Workers*. The CLI allows the user to configure and set up each of these objects.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

A *Provider* represents a cloud Infrastructure-as-a-Service (IaaS) provider, such as public cloud providers Amazon EC2[1] or HP Cloud,[2] or a private installation of cloud IaaS software such as OpenStack[3] or Eucalyptus [26]. To set up a *Provider*, the user simply provides access credentials. Next, *molnsclient* will automate the building of the virtual machine (VM) images. This is done by starting a clean Ubuntu 14.04 seed VM. Then, using package management and source control programs, the set of packages necessary for MOLNs are loaded onto the image. The image is then saved and used for all subsequent provisioning of VMs on this *Provider*.

A *Controller* represents the head node of a MOLNs cluster. It is associated with a specific *Provider*. It hosts the IPython notebook server interface, the parallel computing work queue (IPython parallel controller), and hosts the *SharedStorage* service. If a *Controller* VM has enough CPUs, one or more IPython parallel engines will be started on the node as well. A *Worker* represents one or more *Worker* nodes and is associated with a *Provider* and a *Controller*. It is not required that a *Worker* have the same *Provider* as its associated *Controller*. Indeed, starting *Workers* on a *Provider* different from the *Controller* enables MOLNs's heterogeneous cloud computing capability; see Figure 3. *Workers* host IPython parallel engines for parallel processing, typically one per CPU. *Controllers* and *Workers* can be started independently, and additional workers can be added and removed from a running cluster dynamically, though a *Worker* can only be started if its associated *Controller* is already running.

Together, the infrastructure set up by *molnsclient* and the IPython framework provides an environment that allows interactive and efficient parallel computational experiments. However, the virtual cloud environment adds requirements for handling data not addressed by IPython. Also, directly using the IPython parallel APIs to script scalable Monte Carlo experiments requires some computer science expertise. Hence, there is a need to simplify for practitioners the set up and execution of typical experiments with the spatial stochastic solvers. These issue are addressed by the *molnsutil* package.

3.2.3. Automatic parallelization of systems biology workflows—Providing access to massive computational resources is not sufficient to enable the wider community to utilize them. Efficient use of parallel computing requires specialized training that is not common in the biological fields. Since we have designed MOLNs as a virtual platform, and thus control the whole chain from software stack to virtual compute infrastructure, building upon a parallel architecture (IPython.parallel), we are able to implement a high-level API that provides simple access to the parallel computational resources. From a modeler's perspective, this ensures that computational experiments can be scaled up to conduct proper statistical analysis or large-scale parameter studies without having to deal with managing a distributed computing infrastructure. Instead, the modelers can spend their time on interactively developing and refining postprocessing functions as simple Python scripts.

[1] http://aws.amazon.com/ec2/
[2] https://horizon.hpcloud.com/
[3] http://www.openstack.org/

The role of *molnsutil* is to bridge the gap between the underlying virtual infrastructure provisioned by *molnsclient* and the modeler, providing easy-to-use abstractions for scripting Monte Carlo experiments in the IPython notebook front-end. IPython.parallel provides an API for distributed parallel computing that is easy to use for computational scientists. Although IPython.parallel as deployed in typical cloud computing environments does not offer an environment for truly scalable and low-latency communication-intensive parallel computing, fairly general parallel computing workflows can be implemented and executed in the MOLNs environment, and good performance can be expected for typical Many-Task computing problems. In *molnsutil*, we have used this API to provide high-level access to two such applications and also the two most common computational workflows with PyURDME: the generation of large ensembles of realizations and global parameter sweeps. We also address the question of data management in the cloud, as this issue is out of the scope of the IPython environment. The *molnsutil* library provides an easy-to-use API to store and manage data in MOLNs.

3.2.4. Cluster and cloud storage API—The storage API mirrors the storage layers in the infrastructure; see Figure 3 and Table 1. We define three API-compatible storage classes: *LocalStorage*, *SharedStorage*, and *PersistentStorage*, where the first enables writing and reading of files to the local ephemeral disks of the individual compute nodes, the second uses the cluster-wide network storage, and the third uses the Object Store of the underlying cloud provider. They all fulfill separate needs; *LocalStorage* is used for caching files near compute engines and has the smallest I/O overhead but adds complexity for the developer in dealing with failures that lead to data loss. This storage mode is mainly used internally in *molnsutil* for optimization purposes. *SharedStorage* provides a nonpersistent global storage area that all compute engines can access, making the computations more robust to failing workers. Using *SharedStorage* does not incur any additional cost beyond the cost for the deployed cluster instances.[4] *PersistentStorage* also provides global access to objects, but in addition it makes them persistently available outside the scope of the deployed cluster and visible to other applications (if they share credentials to access the storage buckets). *PersistentStorage* is hence ideal for simulation data that needs to be shared or stored for long periods of time. In public clouds, using *PersistentStorage* incurs extra cost both for storing the objects and for accessing them. As long as the cluster is deployed in a sensible manner, current cost models in the supported clouds permit free network transfer from the object store to the compute engines. In addition to storage abstractions, *molnsutil* contains parallel implementations of two important Monte Carlo computational workflows.

3.2.5. Ensemble statistics—A frequently occurring scenario in computational experiments with spatial stochastic solvers is the generation of very large ensembles of independent realizations from the same model, followed by a statistical analysis based on the ensemble data. Often, a postprocessing function is used to translate the detailed state information $\mathbf{X}$ into information directly related to the biological question being addressed. Hence, this function $g(\mathbf{X})$ is provided by the user. The most common statistics are the mean and the variance. The mean of $g(\mathbf{X})$, $E[g(\mathbf{X})]$, can be computed as

[4]This occurs if all VMs are within the same availability zone.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

$E[g(\mathbf{X})]=\frac{1}{K}\sum_{k=1}^{K}g(\mathbf{X}_k)$, where K is the number of realizations in the ensemble, typically a large number. The variance is given by $V[g(\mathbf{X})]=\frac{1}{K-1}\sum_{k=1}^{K}(g(\mathbf{X}_k)-E[g(\mathbf{X})])^2$, and a 95% confidence interval for the mean is then given by $E[g(\mathbf{X})]\pm 1.96\sqrt{(V[g(\mathbf{X})])/K}$.

In *molnsutil*, this fundamental computation is implemented as part of a *DistributedEnsemble* class. When generating a distributed ensemble, a *URDMESolver* instance is created from a *URDMEModel* class on each of the workers. The stochastic solver is then run (in parallel) independently to create the K realizations. Each realization is represented by an *URDMEResult* object. Hence, the K *URDMEResult* objects contain the $\mathbf{X}_k$ variables in the equations above. To compute the ensemble statistics we apply the postprocessing function to all the results and aggregate them by summation. It is important to note that to exactly reproduce stochastic ensembles requires that the same random seed be used in each of the simulated trajectories. MOLNs provides this facility by simply adding the argument *seed=N*, where N is the desired value of the random number initialization. *molnsuitil* distributes this initialized value to each simulation in the ensemble (or set of ensembles for a parameter sweep) and ensures that each trajectory has a unique random number seed by incrementing the base seed by one.

In Figure 4 we further distinguish two main variants of the execution of this fundamental workflow. The first is where we do not store the intermediary data and instead directly pass it to the next part of the computation, only storing the final postprocessed result (B). The second is where we in a first pass generate the ensemble data and store the intermediary result (the *URDMEResult* objects) (C) and then, in a second pass, apply the postprocessing routines and compute statistics (D). Both of these cases are common in practical modeling projects. In early stages of a project, where the postprocessing functions are being developed, one tends to favor storing the ensemble data and then interactively and dynamically analyzing it while developing the code for the postprocessing analysis. Thus, the lifetime of the data may be hours to days and typically follows the lifetime of the compute cluster (making the use of *SharedStorage* ideal). Later in the project when production runs are conducted, the generation of the ensemble data can require significant CPU time, and one may want to store the simulation data during the lifetime of the project (months to years) for reanalysis, reproducibility, or sharing with another modeler. In this case, the lifetime of the data can be much longer than the lifetime of the cluster resources (making the use of the *PersistentStorage* ideal). In other situations, the stochastic simulations may run fast while the size of the output data set is large. In those cases, it may be preferable to simply recompute the ensemble data in every pass of an analysis step since the cost of recomputation is smaller than the cost of storage.

Figure 5 shows an excerpt from a MOLNs notebook illustrating how the above workflows are executed using *molnsutil*. The user writes the postprocessing function shown in cell *In* [7], and then in cell *In* [8] creates an instance of *DistributedEnsemble* and generates 200 realizations of the model, corresponding to the workflow in Figure 4(C). Then, cell *In* [10] executes the postprocessing workflow in Figure 4(D). Note that in order to change the analysis in a subsequent step, it is only necessary to modify the function *g* in cell *In* [7] and

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

re-executing cell *In* [10]. This gives the modeler the ability to interactively and efficiently develop the analysis functions. While it is not possible or desired to abstract away the user input for the analysis function, as this is where the biology question gets addressed, we have made efforts to abstract away the details of the numerics by encapsulating the data in the *URDMEResult* object and exposing it through simple API calls.

Table 1 summarizes how the storage classes in *molnsutil* maps to the different variants of the workflows. When creating a *DistributedEnsemble*, the default is to use *SharedStorage*, but the user can switch to *PersistentStorage* via a single argument to *add realizations. LocalStorage* is used internally to optimize repeated postprocessing runs by explicitly caching data close to compute nodes.

3.2.6. Parameter sweeps—In most biological models, there is considerable uncertainty in many of the involved parameters. Experimentally determined reaction rate constants, diffusion constants, and initial data are often known to low precision or not known at all. In some cases, phenomenological or macroscopic outputs of the system are available from experiments, frequently in terms of fluorescence image data or coarse-grained time series data for the total number (or total fluorescence intensity) of some of the species. Hence, parameter sweeps are prevalent in modeling projects. Early in a modeling project, they are typically used for parameter estimation, i.e., finding values of the experimentally undetermined parameters that give rise to the experimentally observed phenomenological data. Such brute force parameter estimation may seem like a crude approach, but more sophisticated techniques based on, e.g., parameter sensitivity have yet to be theoretically developed and made computationally feasible for mesoscopic spatial stochastic simulations. Later in a modeling project, when some hypothesis or observation has been made, it is typically necessary to conduct parameter sweeps to study the robustness of this observation to variations in the input data. We also note that studying the robustness of gene regulatory networks in a noisy environment has been a common theme in the systems biology literature [37, 36, 21].

From a computational point of view, a parameter sweep can be thought of as generating a collection of ensembles, one for each parameter point being explored. Since the number of parameter points in a multidimensional parameter space grows very quickly with the number of parameters included in the sweep, the amount of compute time and storage needed for a parameter sweep can be very large, even if relatively small ensembles are generated for each parameter point. The same tradeoffs with respect to storing the ensemble trajectory data as discussed above for a single ensemble applies also to parameter sweeps, but due to the massive amounts of data that is generated even for a moderately large parameter sweep, it will likely be more common to use the execution model where the time course simulation data for each (parameter, ensemble)-pair is not stored. In those cases, the evaluated output metrics for each parameter point will be stored for further analysis and visualization.

The last cell in Figure 5 shows how to execute a parameter sweep in MOLNs. The user simply provides a mapping between any named argument to the constructor in the *URDMEModel* class definition and a value range. The *molnsutil* package then executes the parallel workflow and returns the result.

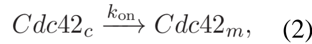
Author Manuscript Author Manuscript Author Manuscript Author Manuscript

3.3. Case study: Interactive model development, simulation, and analysis of yeast polarization

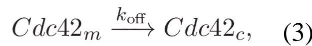
To illustrate the capabilities of MOLNs we created, implemented, and analyzed a model of spontaneous yeast polarization. The cell signaling system in *Saccharomyces cerevisiae* is an ideal candidate to test MOLNs because it is a well-studied, yet not fully understood system in which polarization is critical to the cell cycle. In this case study we describe how we developed our model, parameter sweeps, and postprocessing analysis using MOLNs and reproduced conclusions from the literature.

Yeast cells exist in both haploid and diploid forms, both of which can reproduce through mitosis (i.e., budding). The haploid cells exist in two different types which can mate with each other to form a diploid cell. In both cases, polarization of proteins into a cap within the cell is crucial to establish the point of budding or the point of the mating projection. Cdc42 is a critical protein to the establishment and regulation of polarity [29]. Though many models exist, varying in range of mathematical complexity and physical relevance, we focus on a relatively simple model presented in [1, 20] that makes use of a minimal positive feedback circuit.

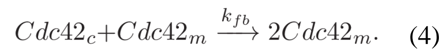
3.3.1. Model specification—The yeast cell is modeled as a sphere with a membrane on the surface of the sphere. The model has three reactions between two species: cytosolic Cdc42 is allowed to spontaneously attach to the membrane with rate k_{on} (2), membrane-bound Cdc42 can likewise spontaneously detach with rate k_{off} (3), and finally membrane-bound Cdc42 can recruit cytosolic Cdc42 to the membrane at rate k_{fb} to close the positive feedback loop (4):



$$Cdc42_c \xrightarrow{k_{\text{on}}} Cdc42_m, \quad (2)$$



$$Cdc42_m \xrightarrow{k_{\text{off}}} Cdc42_c, \quad (3)$$



$$Cdc42_c + Cdc42_m \xrightarrow{k_{fb}} 2Cdc42_m. \quad (4)$$

The cytosolic and membrane bound species can diffuse at rates D_{cyt} and D_{mem}, respectively (the diffusion of the membrane-bound Cdc42 being restricted to the membrane). The geometry, model definition, postprocessing, and visualization are handled completely within the MOLNs environment.

The definition of the yeast polarization model is a Python class that extends the *URDMEModel* class. First, the model parameters and species were defined through *add parameter* and *add species* functions with the expressions and names for model parameters and species (these and all other commands referenced can be seen explicitly in the example code provided in Figure 2). Next, we defined the geometry of the cell using the built-in

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

functionality of the FEniCS/DOLFIN [22, 23] constructive solid geometry (CSG) sphere object. The membrane subdomain is defined by a custom class that marks all voxels on the surface of the sphere and is then added to the model object via the *add subdomain* function. Next the reactions are defined by specifying the reactants, products, rate parameters, and subdomains to which the reactions and species are restricted. In this example problem all reactions are mass action, but PyURDME also has the capability to take custom propensity functions for a reaction, such as Michaelis–Menten. The reactions are added to the PyURDME model object via the *add reaction* function. The last step in the model definition is to provide initial conditions and information about the simulation time span. Here, initial conditions were specified to be a random scattering of 10% of molecules on the membrane and the rest scattered through the cytosol. Although this example is intended to be simple, the design of PyURDME enables easy extension of these modeling definition techniques to much more complex systems. All code and parameter values for this model can be found in the attached example files in the supplementary material.

3.3.2. Model execution, parameter sweep, and postprocessing—Once we have completed the model definition, we execute the simulation within the same IPython notebook with one *run()* command. After model execution, the postprocessing capabilities of MOLNs can be utilized. Having all model parameters, species, geometry, subdomains, and reactions organized within one easily accessible PyURDME model object simplifies the development of postprocessing analysis scripts. All of the postprocessing and data visualization take place right in the same IPython notebook in which model definition and execution occurred. All computation is performed in the cloud, and the users interact via a web browser connected to the IPython notebook interface. In particular, interactive 3D plots of results are rendered in the web browser.

The IPython notebook contains the code that generates plots and the interactive plots themselves within one editable, easily transferable document, which provides the MOLNs user a unique modeling experience that significantly eases the development process. A result can be visualized right along with the code that generated it, and any errors or changes that need to be made will be readily apparent. For this particular example it was of interest to monitor the polarization activity on the membrane. The previous implementations of this positive feedback model [20] made explicit predictions of a density-dependent switch that drives stochastic clustering and polarization (although the physical relevance of this behavior has more recently come into question [11]).

To determine whether the density-dependent switch behavior was in fact observed, we varied the total number of Cdc42 signaling molecules while keeping the volume constant and investigated the polarization behavior. The interactivity of MOLNs allowed useful data to be easily stored and analyzed, which in turn led to the development of metrics quantifying polarization over time.

3.3.3. Result interpretation and case-study summary—One result that the design of MOLNs facilitated was to define a polarization metric that tracks the clustering behavior of the membrane molecules over time. The number of molecules at each voxel is stored for every time point in the PyURDME result object. This allowed the number of membrane

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

molecules to be plotted over time, and once some dynamic equilibrium state is reached, the clustering can be investigated. Here, polarization at any given time was defined by a region making up 10% of the membrane surface area containing more than 50% of the membrane molecules. This metric could be monitored and plotted for each number of signaling molecules to try to discern a qualitative density-dependent switch behavior for polarization.

A parameter sweep was run in parallel for a range of Cdc42 molecule counts. Each parameter point was analyzed using a custom postprocessing function to calculate polarization percent versus time. In this case it was not necessary to store the large amounts of data from the intermediary simulations, but rather return only the output of the post-processing function for each parameter point; thus we used the *No-Storage* method in *molnsutil*. Plots of polarization percent versus time along with the total number of membrane bound Cdc42 molecules versus time for various numbers of total Cdc42 molecules can be seen in Figure 6. Based on the predictions of [20], there should be a critical range for polarization. This range is from a lower critical number of molecules necessary to facilitate polarization to an upper number above which molecules essentially become homogeneous on the membrane (i.e., not polarized). In Figure 6 the time average of the maximum polarization percent is plotted for each Cdc42 molecule count, with error bars corresponding to the standard deviation. As can be seen in Figure 6, there is in fact a density-dependent switch behavior in the model. Below the theoretical critical value calculated from [20] (around 500 molecules for this model) the molecules are in a homogeneous off state, meaning all of the molecules stay in the cytosol. There is an abrupt switch to a high percent polarization above the critical value. As the number of molecules is increased further, they asymptotically approach a homogeneous distribution on the membrane, as predicted by [20].

This case study illustrates the power and ease with which MOLNs users can define and analyze biologically relevant models. Having a coding environment for model and postprocessing development and the interactive visualization of results side by side in one self-contained document with all computation taking place in the cloud makes for a smooth development experience. Also the ability to perform large-scale parameter sweeps efficiently in the cloud and to effectively organize the results is crucial for any modeling task.

3.4. Parallel computing performance

Since MOLNs builds on the IPython suite, it inherits a design focused on interactive parallel computing and dynamic code serialization (enabling the interactivity in the development of the postprocessing routines), and hence programmer productivity and flexibility are areas where MOLNs can be expected to excel. As we have seen, this is enforced by the design of PyURDME. However, parallel performance and scalability are also important factors to consider since they map directly to cost in public cloud environments. Here, we study the performance for our most fundamental computational workflow: generation of a distributed ensemble and subsequent postprocessing by computing the ensemble mean for a given function. We examine the performance in three different clouds: MIST, a privately managed OpenStack Havanna deployment, and the Amazon EC2 and HP Helion public clouds. Finally, we benchmark the system for a hybrid deployment using the HP and EC2 providers.

Author Manuscript

Details regarding the IaaS providers and instance types can be found in the supplementary material.

Figure 7 shows strong and weak scaling experiments when executing the workflows (B)–(D) detailed in Figure 4. Strong scaling shows the execution time for a fixed problem size, here computing an ensemble with 10^2 realizations, with an increasing number of computational engines. We start with a relatively small number of realizations to highlight the impacts of system latency on how much the simulation time can be reduced for a given problem by adding more workers to the MOLNs cluster. Weak scaling, on the other hand, shows the total execution time when both the number of engines and the problem size are increased proportionally so that the total work per engine stays constant. This benchmark shows how well the system lets you increase the problem size by scaling out the compute resources, and the ideal outcome is a constant execution time independent of problem size. In reality, the execution time will never be perfectly constant due to the possibility of exhausting common resources such as disk I/O throughput or network bandwidth (in the case of storing simulation data) or due to scheduling overheads as the number of tasks and workers become numerous. Since these particular workflows map well to the MapReduce programming model, as do many simple Monte Carlo experiments, we will also compare the performance of the MOLNs implementation to a reference implementation using Hadoop streaming on a virtual Hadoop cluster deployed over the same physical resources in our private cloud, MIST. Details of the Hadoop implementation can be found in the supplementary information.

It is not our objective to compare the performance of the different cloud providers in absolute numbers since the underlying hardware differs, although we chose instance types that are as closely corresponding to each other as possible (details can be found in the supplementary material). Rather, we are interested in the scaling properties which we find to be similar for all cloud providers, as can be seen in Figure 7. For strong scaling, irrespective of storage mode, we initially see a rapid reduction in simulation time and a saturation for larger numbers of engines. This is expected due to the total overhead of the system that sets a fundamental limit on the possible speedup. The total simulation time at saturation is less than 20 seconds. For weak scaling, the *SharedStorage* method is faster than using *PersistentStorage* for a smaller number of nodes; however, as the number of workers increases, the *PersistentStorage* is scaled better. We find the crossover point for the performance of these two modes to be approximately five nodes. We also note that for the public clouds (in particular for EC2), the *PersistentStorage* backend results in nearly perfect weak scaling, as the scaling curves parallel the *No-Storage* curves. This result is expected since the Amazon S3 object storage service used by the *PersistentStorage* backend is designed to handle large throughput. In the private cloud MIST, the OpenStack Swift object store uses a single proxy-server, which limits the load-balancing capabilities, and as a result we see a linear scaling of computational time with respect to the total number of requests. In contrast, the *SharedStorage* shows a limited capability to scale out (add nodes to) computations, as the computational time increases sharply as the problem size becomes large. This is a result of saturation of the I/O read and write throughput used by the *SharedStorage* backend on the controller node. In terms of absolute numbers, the EC2 provider outperforms both the HP and the MIST cloud providers. One possible explanation

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

for this would be the fact that the EC2 instances are equipped with SSD-disks which allow for faster I/O throughput.

For comparison, we performed these benchmarks using the widely used Apache Hadoop[5] distributed computing software system. Hadoop MapReduce implements parallel processing of data sets that are typically stored in the Hadoop distributed file system (HDFS). We performed the benchmarks on our private cloud MIST and found that Hadoop with HDFS is slower than MOLNs for all cases. For weak scaling, Hadoop without storage is very close to MOLNs with *No-Storage*, which is expected since the task size is large and system latency has little impact on the computational time.

In addition to benchmarks on single cloud providers, we performed benchmarks on hybrid deployments where the controller node is on one cloud and all of the workers are on a separate cloud provider. Hybrid deployments become useful when users have exhausted their quota in one cloud and want to add more workers in a different cloud, or if they have access to a private cloud but want to burst out to a public cloud for meeting peak loads. For hybrid MOLNs deployments, the performance of computations using *SharedStorage* scales badly due to the network latency for workers writing to the shared disk on the controller in a different cloud provider, to the point that its use cannot be recommended in a hybrid deployment (lower two panels). As can be seen, with *PersistentStorage* or *No-Storage*, a user can benefit from adding workers in a different cloud. It should be noted, however, that the cost of using the *PersistentStorage* in this case will be much higher than in the pure cloud environments since data is moved between cloud providers.

In conclusion, these benchmarks show that MOLNs is not only capable of providing a flexible programming environment for computational experiments, but also a scalable and efficient execution environment, also in comparison with less easy-to-use and less flexible systems such as the industry-standard Apache Hadoop. We estimate the total cost to run this suite of benchmarks was $158 for the HP cloud provider and $50 for the EC2 cloud provider (December 2014 prices). This estimate is based on the monthly billing statement; details can be found in the supplementary material.

4. Discussion

The issue of reproducibility in scientific computation is both important and difficult to address. MOLNs constructs a templated software environment including virtualization of the computational infrastructure, and the IPython notebooks contain all the code necessary to construct the models and execute the analysis workflows; thus we believe that our system holds promise to allow for easy reproduction and verification of simulation results. In particular, there is no need for a modeler to manage multiple formats of the models, or to develop code or input files specific to a particular HPC environment, as all of the information is contained within the notebooks. This reduces the burden on the practitioner to learn specific computing systems and removes the error prone and technically involved process of scaling up a computational experiment in a way that allows for collaborative

[5] http://hadoop.apache.org/

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

analysis of the simulation results. All in all, we believe that MOLNs showcases a scientific software design that has the potential to greatly increase productivity for computational scientists.

The current version of MOLNs makes the spatial stochastic simulation package PyURDME available as a service. PyURDME was designed from the ground up as a cloud-ready package, but in such a way that it does not rely on MOLNs for its use. Naturally, a modeling process may want to rely on other simulation packages as well. MOLNs's automatic and templated configuration of the environment can easily be extended to make other tools available in the IPython notebooks, provided that they can be accessed from the IPython environment (which is not restricted to Python code). We believe that PyURDME showcases a good design to follow for other simulation packages to benefit from this cloud delivery model. It is our hope that the MOLNs platform will grow to include a larger ecosystem of spatial and nonspatial simulation software to facilitate for practitioners to compare tools and to choose the best one for the task at hand.

We have chosen to focus our efforts in facilitating model development on constructing a programmatic interface; hence use of the service requires basic capabilities in Python programming knowledge. The principal target user group is computational biologists that have basic knowledge of programming in a scripting language. By specifying models as compact Python programs, MOLNs and PyURDME join a community of computational software packages, such as PySB [24], whose objective is to utilize high-level, descriptive programmatic concepts to create intuitive, extensible, and reusable models that integrate advanced software engineering tools and methods to distribute and manage the computational experimental process.

From a computer science perspective, the traditional tradeoffs between interactivity and large-scale computational experiments that motivated the development of MOLNs are not unique to this particular application. Looking at scientific computing in general, applications often follow a traditional black-box execution model in which the results of the computation can be procured after the complete execution process. Such workflows have proven to be successful both for simple and complex applications. Queuing based job schedulers such as Torque/PBS which are typical on university clusters have been the driving force behind this approach. However, lack of interactivity is one of the empirical drawbacks of the black-box execution approach. The cloud paradigm changes the way resources are offered, and therefore it is vital to change the traditional black-box execution model of scientific applications to support more interactivity, something that will enhance productivity, prevent wastage of computational resources, and allow inducing knowledge on-the-fly to further optimize the ongoing analysis process. The issues of traditional computational workflows have been addressed within specialized application domains. Galaxy [13] provides an interactive platform that combines the existing genome wide annotations database with online analysis tools that enables running complex scientific queries and visualization of results. A commercial service, PiCloud[6] [8], provided a service for distributing computation on cloud computing resources. The Control Project [17] at Berkeley focuses on a general

[6]PiCloud is now at http://www.multyvac.com.

purpose interactive computing technique to enhance the human computer interaction for massive dataset analysis and thus provides an effective control over information. This project offers online aggregation, emulation and visualization, and rapid data-mining tools. The authors in [30] present a similar approach based on StarCluster [25] and IPython notebooks for a multitask computing model for reproducible biological insights. MOLNs brings the new style of IT model that the above projects represent to the domain of quantitative modeling in systems biology.

Finally, StochSS (www.stochss.org) is a cloud-computing application developed by the present authors that aims at integrating solvers for many different model levels ranging from ODEs to spatial stochastic simulations. In contrast to MOLNs, the present StochSS application emphasizes ease of use and targets biology practitioners with limited or no programming experience. This is reflected by a graphical web user interface (WebUI) to support modeling and a very high abstraction level for interacting with compute resources. In future work, the MOLNs platform will be consumed as a service within the StochSS application as an alternative to the UI-assisted modeling approach, when the user becomes more and more comfortable with quantitative modeling.

In conclusion, we present MOLNs: a cloud computing virtual appliance for computational biology experiments. It has the capability to create computational clusters from a heterogeneous set of public and private cloud computing infrastructure providers and is bundled with the *molnsutil* package to organize distributed parallel computational workflows. It uses an IPython notebook user interface designed to enable interactive, collaborative, and reproducible scientific computing. We also present PyURDME, a software package for modeling and simulation of spatial stochastic systems. It features an intuitive and powerful model description API based on Python objects, efficient handling of complex geometries with FEniCS/Dolfin [22], fast stochastic solvers, and an extensible framework for development of advanced algorithms [5, 6]. Additionally, we demonstrate the capabilities of MOLNs with a computational biology study of yeast polarization. Finally, we demonstrate shareability and reproducibility by including all the IPython notebooks used in the writing of this paper as supplemental material, and we also distribute them as examples in the MOLNs software.

Acknowledgments

This work was supported by National Science Foundation (NSF) award DMS-1001012, U.S. Army Research Office ICB award W911NF-09-0001, NIBIB of the NIH awards R01-EB014877 and R01-GM113241, (U.S.) Department of Energy (DOE) award DE-SC0008975, and the Swedish strategic research program eSSENCE. The content of this paper is solely the responsibility of the authors and does not necessarily represent the official views of these agencies.

We would like to acknowledge Benjamin B. Bales for useful discussions on the design and Stefan Hellander for helpful comments on the manuscript. BD and AH developed the software; BD and AH conceived and designed the study; BD, MT, ST, and AH performed the experiments and data analysis; all authors wrote the manuscript.

REFERENCES

1. Altschuler SJ, Angenent SB, Wang Y, Wu LF. On the spontaneous emergence of cell polarity. Nature. 2008; 454:886–889. [PubMed: 18704086]

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

2. Andrews SS, Addy NJ, Brent R, Arkin AP. Detailed simulations of cell biology with smoldyn 2.1. PLoS Comput. Biol. 2010; 6:e1000705. [PubMed: 20300644]
3. Barkai N, Leibler S. Biological rhythms: Circadian clocks limited by noise. Nature. 2000; 403:267–268. [PubMed: 10659837]
4. Bergmann FT, Adams R, Moodie S, Cooper J, Glont M, Golebiewski M, Hucka M, Laibe C, Miller AK, Nickerson DP, Olivier BG, Rodriguez N, Sauro HM, Scharm M, Soiland-Reyes S, Waltemath D, Yvon F, Le Novére N. COMBINE archive and OMEX format: One file to share all information to reproduce a modeling project. BMC Bioinformatics. 2014; 15:369. [PubMed: 25494900]
5. Drawert B, Engblom S, Hellander A. URDME: A modular framework for stochastic simulation of reaction-transport processes in complex geometries. BMC Syst. Biol. 2012; 6:76. [PubMed: 22727185]
6. Drawert B, Lawson MJ, Petzold L, Khammash M. The diffusive finite state projection algorithm for efficient simulation of the stochastic reaction-diffusion master equation. J. Chem. Phys. 2010; 132:074101. [PubMed: 20170209]
7. Elf J, Ehrenberg M. Spontaneous separation of bi-stable biochemical systems into spatial domains of opposite phases. Syst. Biol. 2004; 1:230–236.
8. Elkabany, K., Staley, A., Park, K. SciPy 2010 Python for Scientific Computing Conference. Austin, TX: 2010. Picloud-cloud computing for science. Simplified.
9. Elowitz MB, Levine AJ, Siggia ED, Swain PS. Stochastic gene expression in a single cell. Science. 2002; 297:1183–1186. [PubMed: 12183631]
10. Fange D, Elf J. Noise induced Min phenotypes in E. coli. PLoS Comput. Biol. 2006; 2:e80. [PubMed: 16846247]
11. Freisinger T, Klünder B, Johnson J, Müller N, Pichler G, Beck G, Costanzo M, Boone C, Cerione RA, Frey E, Wedlich-Söldner R. Establishment of a robust single axis of cell polarity by coupling multiple positive feedback loops. Nat. Commun. 2013; 4:1807. [PubMed: 23651995]
12. Gardiner, CW. Handbook of Stochastic Methods for Physics, Chemistry and the Natural Sciences. 3rd. Springer-Verlag, Berlin: Springer Ser. Synergetics 13; 2004.
13. Giardine B, Riemer C, Hardison RC, Burhans R, Elnitski L, Shah P, Zhang Y, Blankenberg D, Albert I, Taylor J, Miller W, Kent WJ, Nekrutenko A. Galaxy: A platform for interactive large-scale genome analysis. Genome Res. 2005; 15:1451–1455. [PubMed: 16169926]
14. Gillespie DT. A general method for numerically simulating the stochastic time evolution of coupled chemical reacting systems. J. Comput. Phys. 1976; 22:403–434.
15. Gillespie DT, Hellander A, Petzold LR. Perspective: Stochastic algorithms for chemical kinetics. J. Chem. Phys. 2013; 138:170901. [PubMed: 23656106]
16. Hattne J, Fange D, Elf J. Stochastic reaction-diffusion simulation with MesoRD. Bioinformatics. 2005; 21:2923–2924. [PubMed: 15817692]
17. Hellerstein JM, Avnur R, Chou A, Hidber C, Olston C, Raman V, Roth T, Haas PJ. Interactive data analysis: The control project. Computer. 1999; 32:51–59.
18. Hepburn I, Chen W, Wils S, Schutter ED. STEPS: Efficient simulation of stochastic reaction-diffusion models in realistic morphologies. BMC Syst. Biol. 2012; 6:36. [PubMed: 22574658]
19. Hucka M, Finney A, Sauro HM, Bolouri H, Doyle JC, Kitano H, the rest of the SBML Forum. Arkin AP, Bornstein BJ, Bray D, Cornish-Bowden A, Cuellar AA, Dronov S, Gilles ED, Ginkel M, Gor V, Goryanin II, Hedley WJ, Hodgman TC, Hofmeyr J-H, Hunter PJ, Juty NS, Kasberger JL, Kremling A, Kummer U, Le Novére N, Loew LM, Lucio D, Mendes P, Minch E, Mjolsness ED, Nakayama Y, Nelson MR, Nielsen PF, Sakurada T, Schaff JC, Shapiro BE, Shimizu TS, Spence HD, Stelling J, Takahashi K, Tomita M, Wagner J, Wang J. The systems biology markup language (SBML): A medium for representation and exchange of biochemical network models. Bioinformatics. 2003; 19:524–531. [PubMed: 12611808]
20. Jilkine A, Angenent SB, Wu LF, Altschuler SJ. A density-dependent switch drives stochastic clustering and polarization of signaling molecules. PLoS Comput. Biol. 2011; 7:e1002271. [PubMed: 22102805]
21. Lawson MJ, Drawert B, Khammash M, Petzold L, Yi T-M. Spatial stochastic dynamics enable robust cell polarization. PLoS Comput. Biol. 2013; 9:e1003139. [PubMed: 23935469]

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

22. Logg, A.Mardal, K-A., Wells, GN., editors. Automated Solution of Differential Equations by the Finite Element Method. Berlin, New York: Springer; 2012.
23. Logg A, Wells GN. DOLFIN: Automated finite element computing. ACM Trans. Math. Software. 2010; 37:20.
24. Lopez CF, Muhlich JL, Bachman JA, Sorger PK. Programming biological models in Python using PySB. Mol. Syst. Biol. 2013; 9:646. [PubMed: 23423320]
25. MIT starcluster. 2010. http://web.mit.edu/stardev/cluster/
26. Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The eucalyptus open-source cloud-computing system. Proceedings of Cloud Computing and Its Applications. 2008
27. Östberg, P-O., Hellander, A., Drawert, B., Elmroth, E., Holmgren, S., Petzold, L. Abstractions for scaling eScience applications to distributed computing environments; Proceedings of the 3rd International Conference on Bioinformatics Models, Methods and Algorithms; 2012. p. 290-294.
28. Pérez F, Granger BE. IPython: A system for interactive scientific computing. Comput. Sci. Eng. 2007; 9:21–29.
29. Pruyne D, Bretscher A. Polarization of cell growth in yeast: I. Establishment and maintenance of polarity states. J. Cell Sci. 2000; 113:365–375. [PubMed: 10639324]
30. Ragan-Kelley B, Walters WA, McDonald D, Riley J, Granger BE, Gonzalez A, Knight R, Perez F, Gregory Caporaso J. Collaborative cloud-enabled tools allow rapid, reproducible biological insights. ISME J. 2013; 7:461–464. [PubMed: 23096404]
31. Raser JM, O'Shea EK. Noise in gene expression: Origins, consequences and control. Science. 2005; 309:2010–2013. [PubMed: 16179466]
32. Schöneberg J, Noé F. ReaDDy - a software for particle-based reaction-diffusion dynamics in crowded cellular environments. PLoS One. 2013; 8:e74261. [PubMed: 24040218]
33. Schöneberg J, Ullrich A, Noé F. Simulation tools for particle-based reaction-diffusion dynamics in continuous space. BMC Biophys. 2014; 7:11. [PubMed: 25737778]
34. Shen H. Interactive notebooks: Sharing the code. The free IPython notebook makes data analysis easier to record, understand and reproduce. Nature. 2014; 515:151–152. [PubMed: 25373681]
35. Stiles JR, Van Helden D, Bartol TM, Salpeter EE, Salpeter MM. Miniature endplate current rise times less than 100 microseconds from improved dual recordings can be modeled with passive acetylcholine diffusion from a synaptic vesicle. Proc. Nat. Acad. Sci. U.S.A. 1996; 93:5747–5752.
36. Sturrock M, Hellander A, Aldakheel S, Petzold L, Chaplain MAJ. The role of dimerisation and nuclear transport in the Hes1 gene regulatory network. Bull. Math. Biol. 2014; 76:766–798. [PubMed: 23686434]
37. Swain PS, Elowitz MB, Siggia ED. Intrinsic and extrinsic contributions to stochasticity in gene expression. Proc Natl. Acad. Sci. USA. 2002; 99:12795–12800. [PubMed: 12237400]
38. Takahashi K, Tänase-Nicola S, ten Wolde PR. Spatio-temporal correlations can drastically change the response of a MAPK pathway. Proc. Natl. Acad. Sci. USA. 2010; 107:2473–2478. [PubMed: 20133748]
39. The HDF Group. Hierarchical Data Format, version. 1997–2014; 5 http://www.hdfgroup.org/HDF5/.
40. three.js: Javascript 3d library. 2014 http://threejs.org/.

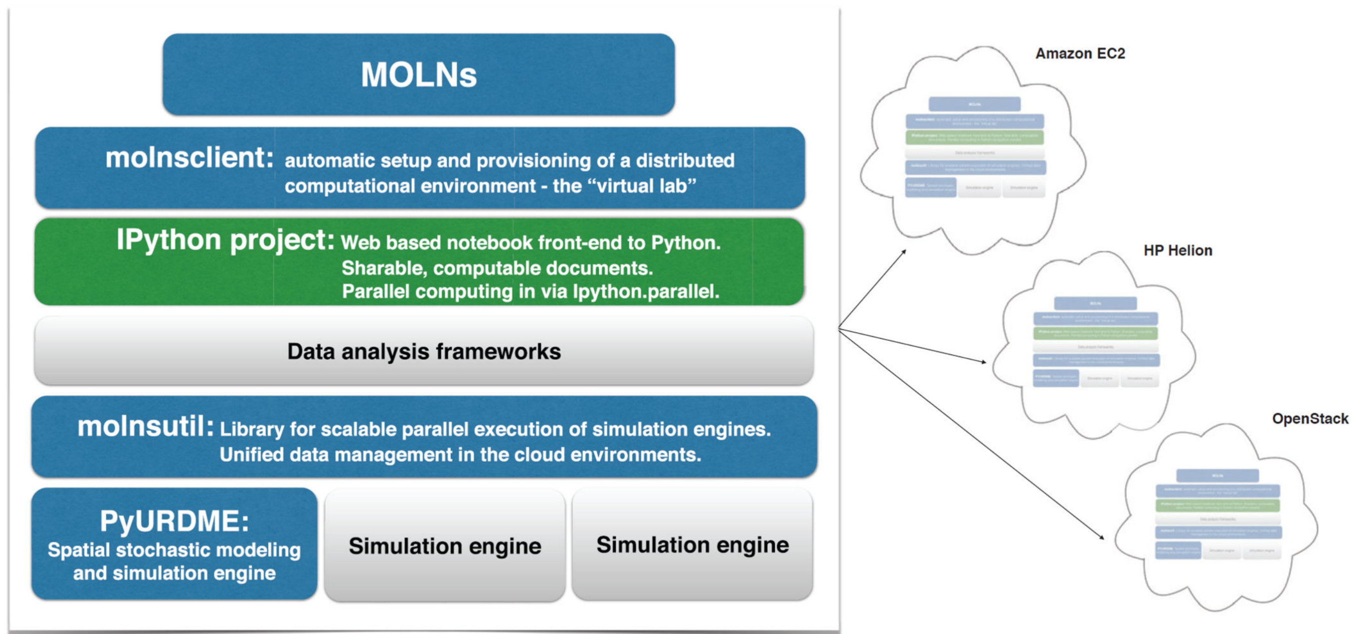


Fig. 1.

MOLNs harnesses the power of cloud computing for biologists to use. Using MOLNs, biologists can take advantage of scalable cloud computing for compute-intensive computational experiments based on stochastic models of reaction-diffusion kinetics. Interactive modeling and scalable Monte Carlo experiments are provided through the use of the IPython Notebook and the newly developed libraries PyURDME and *molnsutil*. Reproducibility of computational experiments requires more than sharing the model, or even the computational workflow that is used for the analysis. By creating a templated computational environment, MOLNs makes the entire "virtual lab" sharable, offering the flexibility to reproduce it in the infrastructure provider of choice, be that public cloud providers or in-house private clouds. This ensures high availability and scalability. Illustrated above are the main components of MOLNs, the newly developed ones are depicted in blue. Grey boxes illustrate the possibility of building on the proposed infrastructure and adding additional data analysis tools to the virtual platform, such as Hadoop, Spark, or other simulation engines. Color is available online only.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

```
In [2]: class Membrane(dolfin.SubDomain):
            def inside(self, x, on_boundary):
                return on_boundary
```

```
In [3]: class YeastPolarization(pyurdme.URDMEModel):
            def __init__(self, Nval=1000, model_name="YeastPolarization"):
                pyurdme.URDMEModel.__init__(self, model_name)
                # Define Parameters
                k_on = pyurdme.Parameter(name="k_on", expression=0.0001/60)
                k_off = pyurdme.Parameter(name="k_off", expression=9.0/60)
                k_fb = pyurdme.Parameter(name="k_fb", expression=10.0/60)
                self.add_parameter([k_on, k_off, k_fb])
                # Define Species
                A = pyurdme.Species(name="A", diffusion_constant=10)
                B = pyurdme.Species(name="B", diffusion_constant=0.0053)
                self.add_species([A, B])
                #Define Geometry
                sphere = mshr.Sphere(dolfin.Point(0.0, 0.0, 0.0), 5.0)
                self.mesh = pyurdme.URDMEMesh(mesh=mshr.generate_mesh(sphere, 14))
                # Define Subdomains
                self.add_subdomain(Membrane(), 2)
                # Define Reactions
                R1 = pyurdme.Reaction(reactants={A:1}, products={B:1}, rate=k_on,  restrict_to=2)
                R2 = pyurdme.Reaction(reactants={B:1}, products={A:1}, rate=k_off, restrict_to=2)
                R3 = pyurdme.Reaction(reactants={A:1, B:1}, products={B:2}, rate=k_fb)
                self.add_reaction([R1, R2, R3])
                #Define initial populations
                A_initial = int(0.9*Nval)
                B_initial = Nval - A_initial
                self.set_initial_condition_scatter({A:A_initial},[1])
                self.set_initial_condition_scatter({B:B_initial},[2])
                # Define simulation timespan
                self.timespan(range(10000))
```

```
In [6]: model = YeastPolarization()
        %time result = model.run()

        CPU times: user 1.8 s, sys: 60.6 ms, total: 1.86 s
        Wall time: 6.19 s
```

```
In [5]: result.display('B', 950, wireframe=False)
```

```
In [6]: model.mesh
```

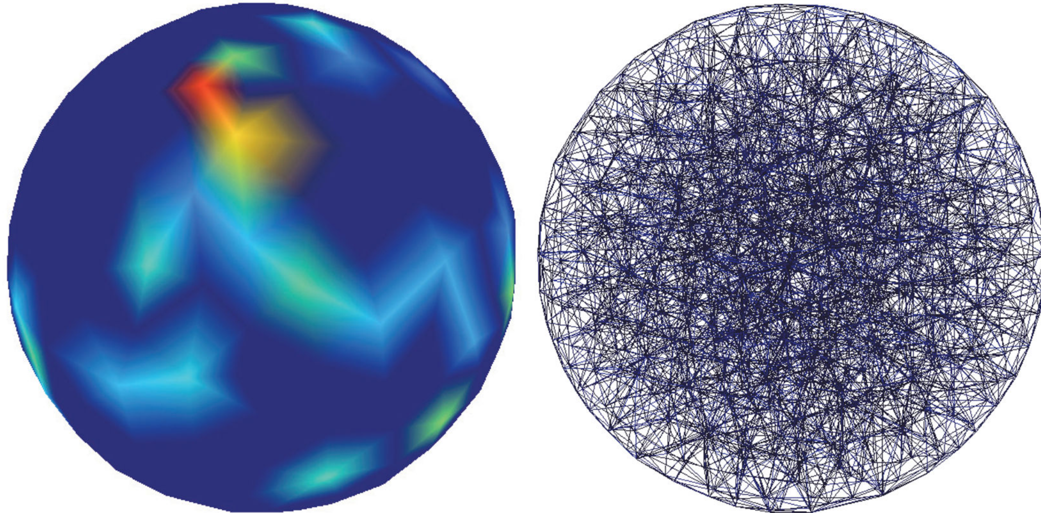


Fig. 2.

Definition of the yeast polarization model, and examples of simulation and visualization that PyURDME and MOLNs provide within the IPython notebook interface. This simple workflow demonstrates the usage of PyURDME.

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

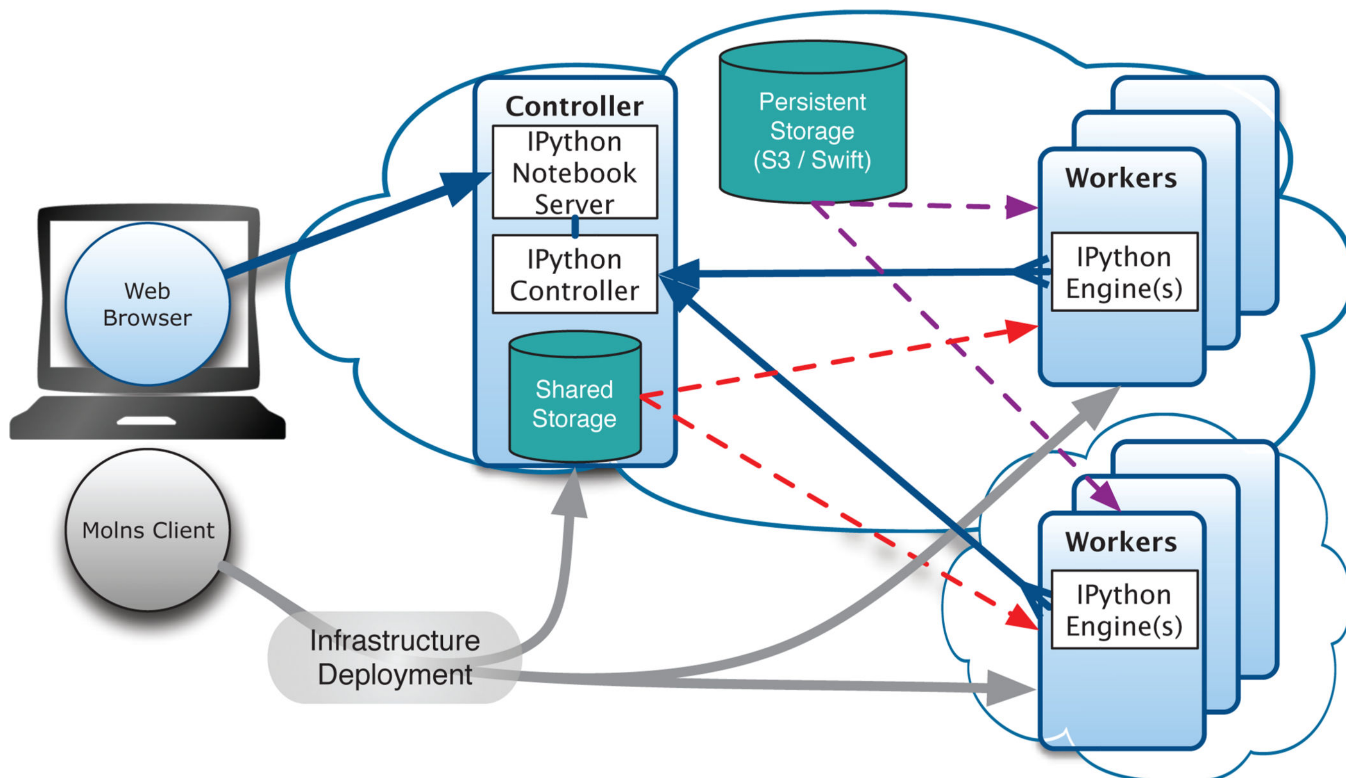


Fig. 3.

MOLNs cluster architecture and communication. Users interact with MOLNs in two ways: using the *molnsclient* and a web browser. The *molnsclient* is used to create, start, and stop a MOLNs cluster by provisioning *Controllers* and *Workers* on multiple clouds (gray arrows). Once a cluster is active, the web browser is used to connect to the IPython notebook web-based interactive computational environment, which provides an interface to PyURDME for modeling and simulation, and to *molnsutil* for distributed computational workflows which utilize the *Workers* of the MOLNs cluster. *Molnsutil* distributes the computations via the IPython controller and IPython engines (blue arrows) and is able to store simulation data in either a transient shared storage (red arrows) or the persistent cloud object storage (i.e., Amazon S*3* or OpenStack Swift, purple arrows). Color is available online only.

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

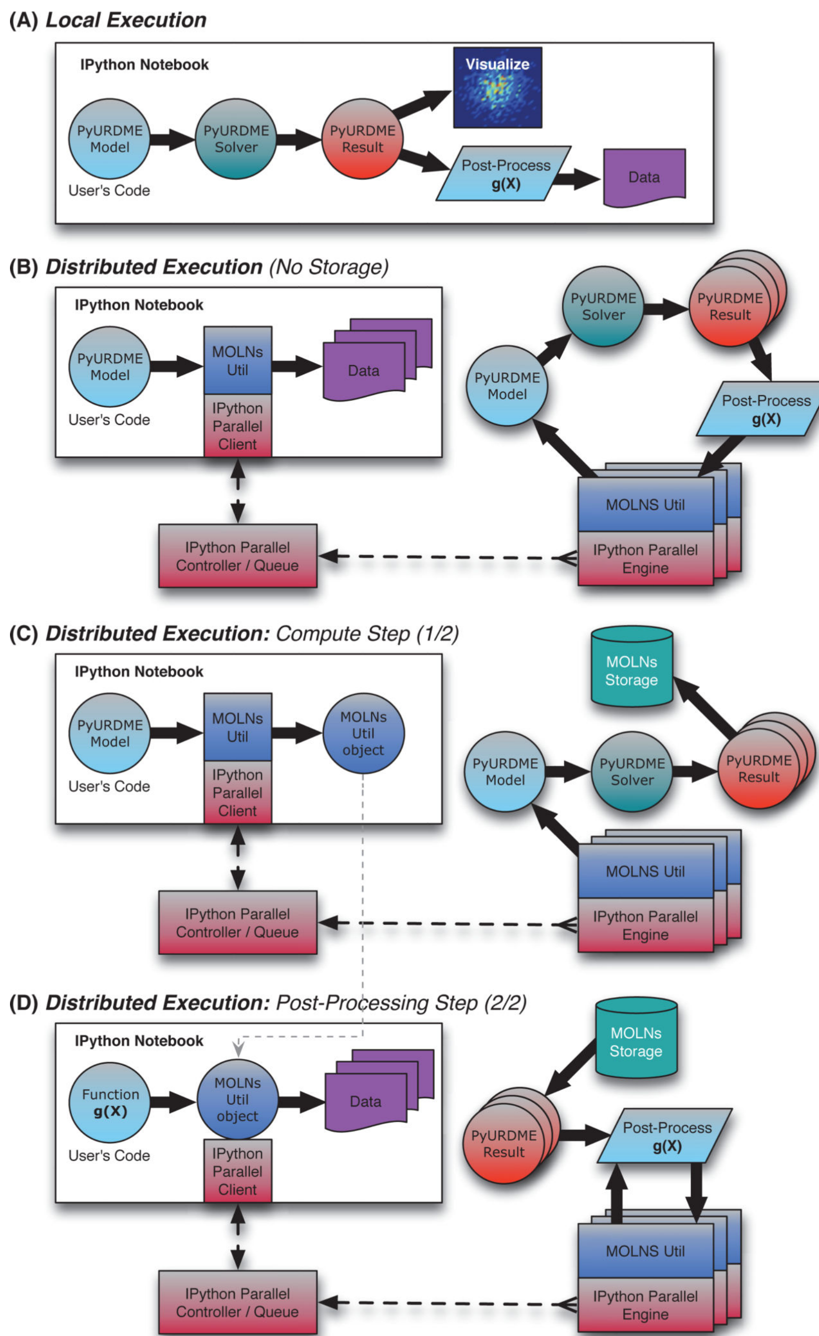


Fig. 4.

MOLNs workflows. *(A)* Basic workflow executed within the IPython notebook. The user develops a biological model, and the model is executed by the solver to produce a result object. The results are either visualized using functionality in PyURDME or passed to a user-defined postprocessing function g(x). This local simulation workflow does not require *molnsutil* and can hence be developed locally without cloud resources. *(B)* Distributed computational workflow. The user develops a biological model and a postprocessing function and passes them to *molnsutil*, which arranges the distributed execution into tasks

and enacts it using IPython parallel. Each task executes the model to produce one or more result objects which are processed by the user-supplied g(x). The resulting data is aggregated and returned to the user's IPython notebook session. *(C)* In many cases it is advantageous to separate the generation of the result objects from the postprocessing. This shows the distributed workflow of generating the results and storing them in the integrated storage services so that subsequent runs of the postprocessing analysis scripts *(D)* can be done multiple times, allowing interactive development and refinement of these scripts.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

```
In [7]: def g(result):
            """ Mapper to extract the values of A at the endpoint. """
            import numpy
            A = numpy.sum(result.get_species("A", -1))
            return A

In [8]: ensemble = DistributedEnsemble(model_class=SimpleDiffusion)
        res  = ensemble.add_realizations(number_of_realizations=200, chunk_size=10)
        print "Time to compute:", res["wall_time"]

        Generating 200 realizations of the model (chunk size=10)

        Time to compute: 22.976152

In [10]: %time mean_val = ensemble.mean(mapper=g)
         print "Mean of g:", mean_val

         Running mapper & aggregator on the result objects (number of results=200,
         chunk size=33)

         Running reducer on mapped and aggregated results (size=7)
         CPU times: user 162 ms, sys: 16.9 ms, total: 179 ms
         Wall time: 1.47 s
         Mean of g: 1000.0

In [14]: plist = [100,500,1000,5000]
         ps = ParameterSweep(model_class=SimpleDiffusion, parameters={'N':plist})
         %time psm = ps.mean(mapper=g, number_of_realizations=200)
         print psm

         Generating 200 realizations of the model at 4 parameter points (chunk size
         =134)

         Running mapper & aggregator on the result objects (number of results=800,
         chunk size=134)

         Running reducer on mapped and aggregated results (size=2)
         CPU times: user 7.7 s, sys: 965 ms, total: 8.67 s
         Wall time: 1min 16s
         [{'N': 100} => 100.0, {'N': 500} => 500.0, {'N': 1000} => 1000.0, {'N': 50
         00} => 5000.0]
```

Fig. 5.

Example usage of the DistributedEnsemble and ParameterSweep classes in *molnsutil* inside an IPython notebook. The bars are animated progress bars.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

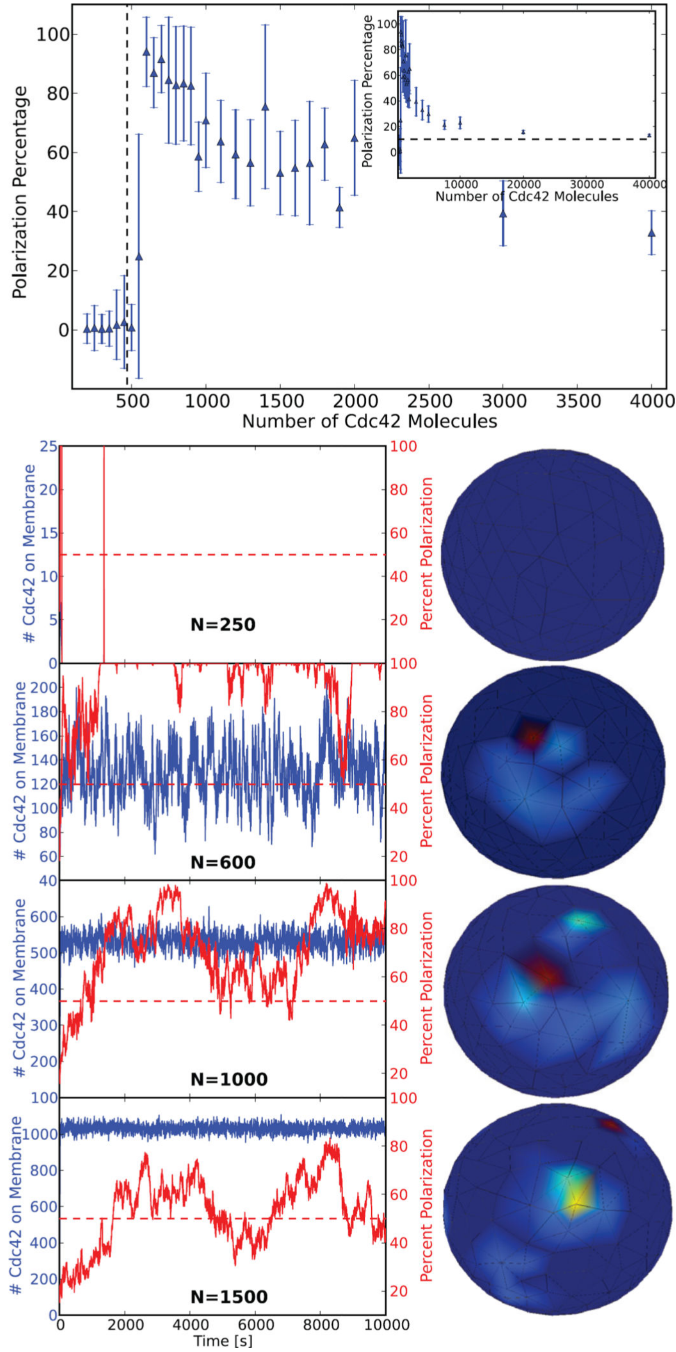


Fig. 6.

The results of a parameter sweep over the number of Cdc42 signaling molecules, N, with volume held constant, performed in parallel. Each model with a given parameter value of N was run to time 10, 000 seconds. Plotted (top) is the time average of the maximum percent of Cdc42 molecules found in any region corresponding to 10 percent surface area on the cell membrane for each N value, with error bars depicting the standard deviation. The dotted line represents the theoretical switch location calculated from [20]. The model captures both the theoretical density dependent switch behavior and the asymptotic decrease to a

homogeneous distribution, which corresponds on average to a maximum of 10 percent of molecules in any 10 percent region on the membrane. Plotted (bottom) is explicit polarization percentage and number of Cdc42 molecules versus time for various values of N along with a characteristic 3D visualization for each. It is important to note that at N = 250 there is no membrane bound Cdc42, as it all remains in the cytoplasm throughout the simulation, which will always be the case below the switch value.

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

Author Manuscript Author Manuscript Author Manuscript Author Manuscript

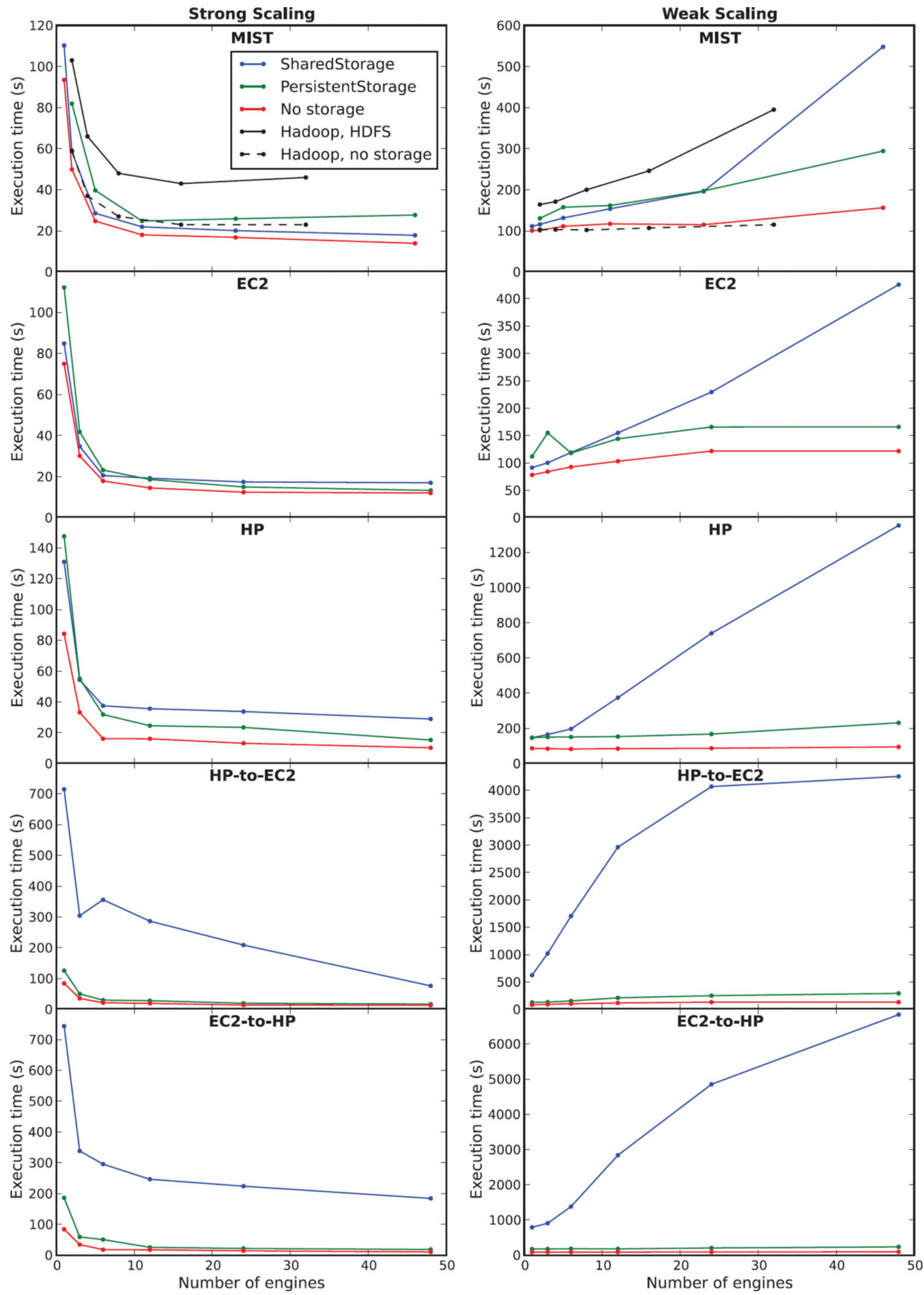


Fig. 7.

Benchmarks of MOLNs for a computation and analysis workflow of a PyURDME distributed ensemble. The left column shows strong scaling tests which demonstrate parallel efficiency: a constant number of jobs (100) executed on a varying number of engines. The right column shows weak scaling tests which demonstrate efficiency of scaling up the problem size: a constant number of jobs per worker (100 × # CPUs) executed on a varying number of engines. The tests were performed on five different compute configurations: the MIST OpenStack private cloud (top row), the Amazon EC2 cloud (2nd row), the HP public

Author Manuscript

Author Manuscript

Author Manuscript

Author Manuscript

cloud (3rd row), a hybrid cloud deployment with the MOLNs controller in the HP cloud and workers in the Amazon EC2 cloud (4th row), and a hybrid cloud deployment with the MOLNs controller in the Amazon EC2 cloud and workers in the HP cloud (5th row). We executed each test with the *SharedStorage*, *PersistentStorage*, and *No-Storage* methods of *molnsutil*. For the MIST cloud we also executed benchmarks of Hadoop MapReduce of the same workflow for comparison.

Author Manuscript
Author Manuscript
Author Manuscript
Author Manuscript

Table 1

Comparison of storage types available to MOLNs distributed workflows.

Type	Advantages	Disadvantages
SharedStorage	No additional cost for read/write Fastest throughput for small clusters No management of remote data	Total storage limited to Controller disk size Nonredundant storage Throughput limited on large clusters
PersistentStorage	Persistent data Designed for extreme scalability	Storage and access incur cost
LocalStorage	Best data locality High I/O throughput	Nonrobust to worker failure Increased complexity for developer
No-Storage	Best parallel scaling No cost for data storage	Data must be recomputed for analysis