Extreme-Scale Dynamic Exploration of a Distributed Agent-Based Model With the EMEWS Framework

Jonathan Ozik, Nicholson T. Collier, Justin M. Wozniak, Charles M. Macal, and Gary An

***Abstract*— Agent-based models (ABMs) integrate the multiple scales of behavior and data to produce higher order dynamic phenomena and are increasingly used in the study of important social complex systems in biomedicine, socioeconomics, and ecology/resource management. However, the development, validation, and use of ABMs are hampered by the need to execute very large numbers of simulations in order to identify their behavioral properties, a challenge accentuated by the computational cost of running realistic, large-scale, potentially distributed ABM simulations. In this paper, we describe the Extreme-scale Model Exploration with Swift (EMEWS) framework that is capable of efficiently composing and executing large ensembles of simulations and other "black box" scientific applications while integrating model exploration (ME) algorithms developed with the use of widely available third-party libraries written in popular languages, such as R and Python. EMEWS combines novel stateful tasks with traditional run-to-completion many-task computing and solves many problems relevant to high-performance workflows, including scaling to very large numbers (millions) of tasks, maintaining state and locality information, and enabling effective multiple-language problem solving. We present the high-level programming model of the EMEWS framework and demonstrate how it is used to integrate an active learning ME algorithm to dynamically and efficiently characterize the parameter space of a large and complex, distributed message passing interface agent-based infectious disease model.**

***Index Terms*— Agent-based modeling, high-performance computing (HPC), machine learning, metamodeling, parallel processing.**

I. Introduction

RECENT improvements in high-performance agent-based models (ABMs) have enabled the simulation of a variety of complex systems, including the spread of infectious diseases and community-based healthcare interventions [1], [2], critical materials supply chains [3], and land-use and resource management [4], [5]. As ABMs have become more complex, capturing more salient features of the systems under study, parameters that dictate the structural (e.g., social networks), behavioral, and other dynamical elements of the models have increased in number. Other complex systems modeling approaches (e.g., mathematical modeling and system dynamics) can rely on assumptions about model and parameter space structures to make use of relatively efficient methods for model calibration and optimization. However, the highly nonlinear relationship between ABM input parameters and model outputs, as well as feedback loops and emergent behaviors, require less-efficient ensemble modeling approaches. These approaches execute large numbers of simulations, often in complex iterative workflows driven by sophisticated model exploration (ME) algorithms, such as active learning (AL), which adaptively refine model parameters through the analysis of recently generated simulation results and launch new simulations.

In order to facilitate these dynamic ME-based approaches, we have created the Extreme-scale ME with Swift (EMEWS) framework [6], [7]. EMEWS, which is built on Swift/T [8], offers the capability to run very large, highly concurrent ensembles of simulations of varying types while supporting a wide class of ME algorithms, including those increasingly available to the community via Python and R libraries. Furthermore, it offers a software sustainability solution, in that, ME studies based around EMEWS can easily be compared and distributed. A central EMEWS design goal is to ease software integration while providing scalability to the large-scale (petascale plus) supercomputers, running millions of ABMs, thousands at a time. Initial scaling studies of EMEWS have shown robust scalability [9]. The tools are also easy to install and run on an ordinary laptop, requiring only a message passing interface (MPI) implementation that can be easily obtained from common OS package repositories. By combining novel stateful tasks with traditional run-to-completion many-task computing (MTC), our framework solves many problems relevant to high-performance workflows, including scaling to very large numbers (millions) of tasks, maintaining state and locality information, and the multiple language problem.

EMEWS enables the user to plug in both ME algorithms and models (e.g., ABMs). Thus, researchers in various fields who may not be parallel programming experts can simply incorporate existing ME algorithms and run computational experiments on their scientific application without explicit parallel programming. A key feature of this approach is that the model is unmodified and the ME algorithm is

Manuscript received January 5, 2018; revised June 20, 2018; accepted July 16, 2018. Date of publication August 30, 2018; date of current version September 11, 2018. This work was supported in part by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357, and in part by National Institutes of Health under Award R01GM115839 and Award R01GM121600. *(Corresponding author: Jonathan Ozik.)*

J. Ozik, N. T. Collier, and C. M. Macal are with the Decision and Infrastructure Sciences Division, Argonne National Laboratory, Argonne, IL 60439 USA, and also with the Consortium for Advanced Science and Engineering, The University of Chicago, Chicago, IL 60637 USA (e-mail: jozik@anl.gov).

J. M. Wozniak is with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 USA, and also with the Consortium for Advanced Science and Engineering, The University of Chicago, Chicago, IL 60637 USA.

G. An is with the Department of Surgery, The University of Chicago, Chicago, IL 60637 USA.

Digital Object Identifier 10.1109/TCSS.2018.2859189

2329-924X © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

only minimally aware of its existence within the EMEWS framework. EMEWS uses a novel form of Inversion of Control (IoC), where Swift/T instantiates the ME algorithm, which then provides model parameters back to Swift/T [over interprocess communication (IPC), without returning]. These parameters are distributed to worker processes for model execution. Swift/T provides a variety of methods for integrating models, including via built-in interpreters, command line invocation, and as compiled libraries. Upon completion, the model outputs are registered back to the ME algorithm, which provides more parameters until a convergence criterion is satisfied or a computing budget is exhausted.

EMEWS also relies on the new "many *resident* task computing" capabilities that extend the notion of MTC. This allows running tasks to effectively suspend, waiting for queries. We demonstrate that mixing resident tasks with traditional run-to-completion tasks is a powerful programming model that supports the development of calibrated and validated scientific applications, including realistic ABMs that can be used as *electronic laboratories* to answer important research and policy questions.

This paper offers the following contributions.

1) It describes a software integration model for high-performance workflowlike applications, where advanced algorithms, such as AL, implemented in languages, such as R, can be integrated.
2) It describes a compelling, real-world application infectious disease dynamics and presents results from running a large-scale AL workflow to characterize the parameter space of a distributed ABM.
3) It proposes and investigates novel, flexible concurrence schemes for these workflows.
4) It evaluates the performance and scalability of the application up to 10k cores on a Cray supercomputer.

The remainder of this paper is organized as follows. In Section II, we describe ABMs, ABM ensemble ME methods, and our susceptible-exposed-infected-recovered (SEIR) ABM. In Section III, we describe the EMEWS programming model and its implementation. In Section IV, we describe how the various components in our SEIR model and AL EMEWS workflow are connected. In Section V, we present the results from running a large-scale AL workflow to characterize the SEIR model parameter space. In Section VI, we report the performance numbers for the workflow. In Section VII, we restate our contributions and offer conclusions.

II. ABM, Ensemble Model Exploration Methods, and the SEIR Model

Agent-based modeling and simulation (ABMS) is a method of computing the potential system-level consequences of the behaviors of sets of individuals [10]. ABMS allows modelers to specify the individual behavioral rules for each agent, describe the circumstances in which the individuals reside, and, then, execute the rules, via simulation, in order to determine possible system-level results. Agents themselves are individually identifiable components that usually represent decision makers at some level. Agents often are capable of some level of learning or adaptation ranging from simple parameter adjustment to the use of neural networks, evolutionary algorithms, and market models.

As larger and more complicated models of complex systems are developed, high-performance computing (HPC) resources are increasingly required to run the computational experiments needed for developing validated (i.e., trusted) models that can support decision making. On the one hand, ABMS studies require the execution of many model runs to account for stochastic variation in model outputs and for the various ensemble ME methods that are required to calibrate and analyze them. These methods can be used to carry out the followings:

1) adaptive parametric studies;
2) large-scale sensitivity analyses and scaling studies;
3) optimization and metaheuristics;
4) inverse modeling;
5) uncertainty quantification;
6) data assimilation.

On the other hand, ABMs can also be distributed across processes to accommodate very large numbers of agents (e.g., $>10^9$ [11]) or very complex agents.[1] These facts combine to make ABMs well suited for HPC resources, and through the EMEWS framework, they can easily and efficiently be run as part of large-scale scientific workflows.

A. Ensemble Model Exploration Methods

Depending on the aims of a computational experiment, different dynamic ensemble ME methods are appropriate.[2] In the realm of stochastic optimization, there are simulated annealing [15], adaptive mesh [16], genetic algorithms [17], approximate Bayesian computation [18], [19], and other techniques. Ensemble Kalman filtering [20] and particle filters [21], [22] are useful for combining ensembles of model outputs and empirical observations. AL [23] can be used to efficiently characterize large parameter spaces. These types of techniques are increasingly being used with ABMs [24]. Many of the methods are being actively developed and are implemented as free and open source libraries in popular data analysis programming languages (e.g., R) and general purpose languages (e.g., Python).

While sophisticated ME techniques have been a generally fruitful approach for combining ensemble mathematical (e.g., compartmental) models and empirical observations, for example, in infectious disease modeling [25]–[27], we also see that such events like the 2013 West African Ebola outbreak have exposed some limits to the predictive power of these approaches [28]. The possible reasons for this are many, but some of the simplifying assumptions inherent in the compartmental models that are used for the infectious disease studies might be at issue. Compartmental models use differential equations relating aggregate variables (e.g., the fractions of the population that are susceptible, infected, or recovered/removed)

[1] So-called "thick" agents may include sophisticated and computationally expensive cognitive abilities.

[2] We note that there exist static parameter search techniques (e.g., full factorial design [12], Latin hypercube sampling [13], and Morris method [14]) that *a priori* determine the sampling from a parameter space. While these can be useful for some purposes, they are not adaptive and do not require complex workflow logic and, hence, are not the focus of this paper.

to derive the dynamics of disease progression in a population. But such models are not able to capture "complex social networks and the direct contacts between individuals who adapt their behaviors [29]." By developing more realistic models in the form of ABMs, the complexity, for example, of the interagent and biological–social interactions inherent in many infectious diseases, can be encapsulated in the specification of processes, such as agent activities and decision making, agent interactions over social networks, demographic and geographic heterogeneity, and agent adaptation and learning.

With EMEWS, the ensemble ME techniques that have been applied to simpler modeling paradigms can be carried over to the ME of large, complex, parallel, and distributed ABMs. Furthermore, many of these techniques are being actively developed and are implemented as free and open source libraries in popular programming languages. As indicated earlier, rather than requiring the reimplementation of these algorithms in the Swift/T language, the goal of the EMEWS framework is to be able to have these libraries directly control large-scale HPC workflows, thereby making them more accessible to a wider range of researchers and, at the same time, enable them to run at HPC scales.

B. SEIR Model

Our SEIR model is a distributed parallel ABM of the transmission of a flu-like disease using SEIR model dynamics [30]. The model represents each person in a selected geographical region (e.g., the City of Chicago) as an agent. Each person in the model is in one of four disease states: susceptible, exposed, infected, or recovered. Persons transition through states, moving from susceptible to exposed to infected and ending with recovered. While susceptible, a person can become exposed in the presence of infectious persons. Exposed persons are infectious but not yet infected, i.e., they can infect other persons but are not yet symptomatic. Infected persons are symptomatic and also infectious for at least part of the period of infection. Recovered persons are no longer infected or infectious and, being effectively immune to the disease, will not become susceptible again. The model begins with some specified number (parameter C_I) of persons in the exposed state, who subsequently transition through the infected and recovered state while, in turn, exposing other persons to the disease, triggering the transition of those persons through the disease states.

The transition and duration of each state are determined by model state and user-specifiable input parameters. A susceptible person will transition to exposed in the presence of infectious persons with a base probability ($P_{S\rightarrow E}$) modified by the number of colocated infectious persons. The duration of a person's stay in the exposed state is drawn from a triangular distribution specified by a mode ($\mathrm{Mo}_{t_{\mathrm{inc}}}$), minimum ($\mathrm{Mi}_{t_{\mathrm{inc}}}$), and maximum ($\mathrm{Mx}_{t_{\mathrm{inc}}}$), where the minimum ($\mathrm{Mi}_{t_{\mathrm{inc}}}$) defaults to one day and the maximum ($\mathrm{Mx}_{t_{\mathrm{inc}}}$) to four [31]. After the exposed duration has elapsed, an exposed person enters the infected state. Exposed persons are infectious from one day prior to entering the infected state to seven days after entering it [31]. The length of the infected state is also drawn from a triangular distribution (Mo_{t_I}, Mi_{t_I}, and Mx_{t_I}) with a default minimum of seven days and a default maximum of 14 days [32]. While infected, a person will remain at home, thus avoiding contact with anyone outside the household. With a user-specifiable probability (P_{home_A}), a person will remain at home as soon as they become infected; otherwise, they will remain at home beginning one day after becoming infected. A person will remain at home for either five, six, or seven days depending on a user-specifiable probability (P_{home_B} and P_{home_C}), after which they will resume their normal activities. Once the infected period ends, a person transitions to the recovered state. The parameters of the triangular distributions, the "stay-at-home" probabilities, and the initial number of exposed persons are model parameters (see Table I) and thus can be altered to affect the number of persons in each state as the model progresses.

TABLE I

SEIR MODEL INPUT PARAMETERS

Parameter	Description
C_I	Initial number of infected individuals
$P_{S\rightarrow E}$	Hourly probability of going from susceptible to exposed per each collocated infectious agent
$Mo_{t_{inc}}$, $Mi_{t_{inc}}$, $Mx_{t_{inc}}$	Mode, minimum, and maximum of the triangular distribution for exposed to infected incubation period
Mo_{t_I}, Mi_{t_I}, Mx_{t_I}	Mode, minimum, and maximum of the triangular distribution for time in infected state
P_{home_A}, P_{home_B}, P_{home_C}	Probabilities for infected individual to remain home on first day, sixth day and seventh day of infection

The SEIR model is implemented in C++ using the Repast for HPC (Repast HPC) [33] and the Chicago Social Interaction Model (chiSIM) [34] toolkits. Repast HPC is an ABM framework for implementing ABMs in MPI and C++ on high-performance distributed-memory computing platforms. chiSIM is a framework for implementing the models that simulate the hourly mixing of a synthetic population, in this case, the City of Chicago consisting of approximately 2.9 million individual agents and 1.2 million distinct places. Synthetic populations with baseline sociodemographic data, derived from the combined U.S. Census files, are available from a growing number of sources. chiSIM uses baseline synthetic population data, such as those developed through the National Institutes of Health (NIH) MIDAS network [35], [36]. The sociodemographic attributes of the synthetic population match that of the actual population for Chicago in the aggregate for the Census years of 2000 and 2010. Each agent has a baseline set of sociodemographic characteristics (e.g., race/ethnicity, age, gender, educational attainment, and income). All places are characterized by place type, including households, schools, hospitals, and workplaces, and have a geographic location. In the synthetic population, agents are assigned to households, workplaces, and schools (for those of school age). Places are categorized as having different types of activities that may occur there.

In a chiSIM-based model, such as SEIR, each agent, that is, each person in the simulated population, resides in a place (for example, a household, dormitory, or retirement home/long-term care facility) and moves among other places, such as schools, workplaces, hospitals, jails, and sports facilities.

Agents move between places according to their shared activity profiles. Each agent has a profile that determines at what times throughout the day they occupy a particular location [33]. Our Chicago agent activity profiles are empirically based on 24-h time diaries collected as part of the U.S. Bureau of Labor Statistics annual American Time Use Survey (ATUS) for individuals aged 15 years and older and from the Panel Study of Income Dynamics (PSID) for children younger than 15 years. Both are nationally representative samples and collect diary data on randomly assigned days. In the SEIR model, two profiles (one weekday and one weekend) from ATUS/PSID respondents living in metropolitan areas are assigned to each agent in the model. This is done by stochastically matching each agent with an ATUS or PSID respondent who is either identical or similar with respect to the sociodemographic characteristics. Agents move between places according to their activity profiles. Once in a place, an agent mixes with other agents in some model or domain-specific way. In the case of the SEIR model, infectious agents infect colocated susceptible agents, who are having, become, infected can then in turn infect other agents as they move.

chiSIM itself is a generalization of a model of community-associated methicillin-resistant Staphylococcus aureus (CA-MRSA) [2]. The CA-MRSA model was a nondistributed model, in which all the model components (all the agent, places, and so on) run on a single process. chiSIM retains and generalizes the social interaction dynamics of the CA-MRSA model and allows models implemented using chiSIM to be distributed across multiple processes. Places are created on a process and remain there. Persons move among the processes according to their activity profiles. When a person, agent, selects a next place to move to, the person may stay on its current process or it may have to move to another process if its next place is not on the person's current process. A load balancing algorithm has been applied to the synthetic Chicago population to create an efficient distribution of agents and places, minimizing this cross-process movement of persons and balancing the number of persons on each process [34]. In addition, chiSIM provides the ability to cache any constant agent state, given sufficient memory, lessening the amount of data transferred between processes.

Sections III–V describe how the EMEWS framework is used to perform an adaptive parametric study of the SEIR model by integrating it with an ME algorithm, in this case AL [23].

III. EMEWS Programming Model

The EMEWS framework is designed to implement a high-level programming model that allows us to coordinate calls to scientific applications, such as large ABMs, as well as various control and analysis scripts over a scalable, MPI-based computing infrastructure. Specifically, EMEWS was implemented to meet the following requirements:

1) the ability to construct a workflow of many (potentially millions of) calls to a scientific application (such as an ABM simulation) with different parameters;
2) the ability to allow simulation results to feedforward into future application parameters;
3) the ability to integrate a complex ME algorithm, such as AL, into the parameter construction;
4) the ability to call into the native-code models and scientific applications (e.g., written in C++) and the third-party implementation of an ME algorithm (written in R in the current application);
5) the ability to maintain the state of the ME algorithm from call to call and programmatically access this state in the system.

We provide an overview of how EMEWS and Swift/T addresses these requirements in the following.

1) The ability to manage an extreme quantity of tasks is a main design feature of the Swift/T implementation [8], [37], which essentially translates the Swift script into an MPI program for execution on the large-scale supercomputers. The Swift–Turbine Compiler [38] optimizes the script using multiple techniques, both conventional and oriented toward novel concurrence. In synthetic tests, Swift/T has been used to run trillions of tasks at over one billion tasks per second. It can also send very small tasks to GPUs at high rates [39], enabling powerful mixed programming models.
2) Swift is a dataflow language. In this model, the user defines the data items (numbers, strings, binary data, and various collections of these) and connects them with functional execution. Swift also offers conventional constructs, such as `if`, `for`, `foreach`, and so on, with their definition only slightly modified for automatic parallelism. Following dataflow (not control flow) functions execute when their inputs are available, possibly concurrently. Thus, typical Swift loops are automatically parallel loops. Dataflow analysis allows common expressions such as `g(f(1),f(2));` to expose the available concurrence (two simultaneous executions of `f()`).
3) Swift/T has rich support for integrating complex logic into workflows, including using scripting languages, such as Python and R. It enables this on HPC machines (where `fork()` may be undesirable or unavailable) by optionally bundling the script interpreters for Python, R, Julia, Tcl, JVM languages, and so on into the Swift/T runtime [40]. These interpreters are called through their native-code interfaces (thus reusing the Swift/T ability to call into native-code libraries), but high-level interfaces are provided for Swift. For example, the following Swift code:

```
string result
          = python("a=2+2","str(a)");
```

would store `"4"` in `result`. The `python()` function takes two string arguments, *code* and an *expression*. The code is executed, and the string expression is returned as a Swift string. Users may set `PYTHONPATH` and load their own modules or third-party modules, such as Numpy and so on. They may also call through these scripting layers into native code. The Repast HPC code is called an MPI library, as described in Section III-A.

4) Various states may be maintained in the Swift/T implementation while remaining outside the main dataflow model. This is typically done in the tasks, avoiding confusion with the dataflow. For example, a configuration file could be loaded from disk by a Python-based task and cached in a global Python variable. This data would be available on the next invocation of a Python task on that process.

Developers can target different parts of the system by using the locality features in Swift/T. These were initially added to allow users to send tasks to data in a compute-node resident file system [41]. However, they can also be used to send tasks to state in a script interpreter. Tasks can be targeted at a rank or a node and be strict or nonstrict. We use strict rank targeting in this paper, while nonstrict, node-based targeting is used in (for example) the cache storage systems.

The rest of this section goes into further details on the EMEWS and Swift/T features that address the programming model requirements.

A. Hierarchical Concurrence

MPI enables the concurrent execution of multiple cooperating multiprocessing codes, each of which can have a separate communication context shared with only the MPI processes executing that code. MPI represents these contexts with communicators that typically form a tree hierarchy, starting from an initial world communicator that encompasses all processes. Given a communicator, new child communicators can be created and passed to libraries for their exclusive use, allowing an application to be constructed through composition of existing parallel libraries and codes.

Our execution model has multiple levels of concurrence and a great deal of flexibility in how the workflow uses the available processing power of a supercomputer. Since the SEIR model itself uses MPI, it must be treated as an MPI library. Swift/T uses the MPI 3.0 `MPI_Comm_create_group()` feature to allocate a new communicator for each task [42]. These are handed to the application for each new task and deallocated (`MPI_Comm_free()`) at the end of the task. The user can specify the number of processes (e.g., p) for each task programmatically with

```
@par=p f(...);
```

When launching a simulation task, Swift/T constructs the communicator and passes it to the SEIR model that is a shared library loaded by Swift/T. The use of MPI in the SEIR model is completely independent of the use of MPI by Swift/T. There is no mixture of control flow from Swift to the SEIR model; once the SEIR task starts, it proceeds with normal MPI/C++ semantics until returning control back to Swift/T.

B. Location-Aware Many-Task Scheduling

MTC workloads, on the one hand, generally allow the *scheduler* a great deal of leeway in determining where tasks will execute. Bag-of-tasks workloads, for example, are the most lenient, allowing tasks to execute anywhere in any order. Programming models, such as MPI, on the other hand, give the *programmer* total control over execution locality.

Swift/T strikes a balance between these two extremes with its *location annotation*. By default, tasks can execute on any worker process, but the programmer has the option of specifying the annotation with `@location=L f()`, where `f()` is the task and `L` is a location value. A location value is constructed from an MPI rank r with optional accuracy and strictness qualifiers. (Swift/T features allow a hostname to be translated to one or more MPI ranks.) The accuracy may be RANK, specifying the process with rank r, or NODE, specifying any process that shares the same network host with r. The strictness may be SOFT, allowing the task to run anywhere in the system if there is nothing else to do at a given point in time, or HARD, specifying that the scheduler should wait until the location constraint can be satisfied (even at the expense of maintaining idle processors).

The location features in Swift/T were originally added for data-intensive workloads [41]. These provide a novel model for best effort, data-aware scheduling, when data are stored on the compute nodes. Compute node-resident storage systems that advertise data locations can be exploited by these programming features. In EMEWS, we extend the utility of this feature by using it to target program state instead of bulk data. By keeping program state resident, we avoid any cost associated with approaches that depend on data serialization. More importantly, we can more easily leverage the third-party libraries as resident programs without extensively modifying them to fit a data serialization-based scheme.

C. Resident Tasks for Ensemble Control

Previous uses of workflow languages to control ME typically take one of two approaches. In the first approach, the ME algorithm is encoded in the workflow language. While some workflow languages provide rich support for arithmetic operations (Swift/T is notable in this regard), many do not. Even so, this approach requires that such algorithms be coded from scratch in the workflow language and makes it impossible to directly reuse code in other languages. In the second approach, the algorithm is provided as a built-in feature of the workflow system. This approach has been taken by Nimrod/O [43] and Dakota [44], among others. It does not allow the end users much control over the ME algorithm used, unless they can modify the source code of the workflow system itself.

EMEWS defines and uses *resident tasks* as a building block to implement the user-defined ME workflows. The key technological feature is the ability to launch a task in a *background* process or thread. *Background* indicates that the *foreground* process or thread returns the control to Swift/T after execution (as a normal task would), but the background task is still running. It retains state and potentially performs ongoing computation. For the current example, the background task maintains the state of an AL ME algorithm. The overarching workflow must simply *query* this task for instruction on what tasks to execute next. To do so, a task is issued to the same location as the resident task that communicates with it over IPC.

D. EMEWS Queues for R

To query the state of the AL algorithm, we designate one worker on location L for exclusive use by AL. Interaction with

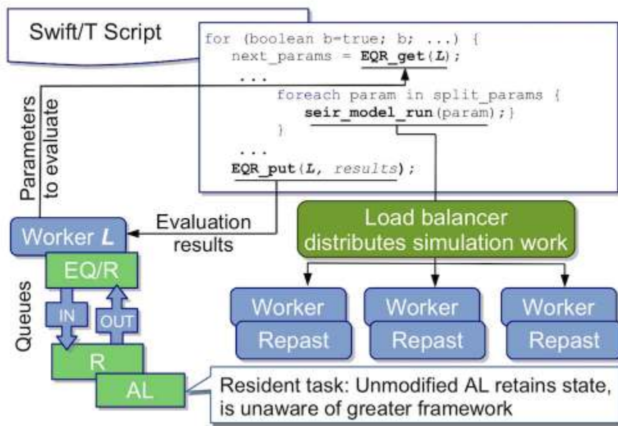

Fig. 1. EQ/R EMEWS workflow with an AL ME resident task.

this worker via the EMEWS Queues for R (EQ/R) extension is shown in Fig. 1. The EQ/R extension allows Swift/T workflows to communicate with a persistent embedded R interpreter on a worker at some location L via two blocking queues, IN and OUT. The extension provides C++ functions that allow string data to be pushed onto and retrieved from these queues. These functions are wrapped in an interface and are accessible to Swift/T and shared with the R environment. Upon initialization, EQ/R adds these functions to the R environment and spawns a thread, in which the R script is run. Through these functions, the R script places string data in the OUT queue, where the Swift/T parent thread can retrieve it with the `EQR_get()` function. Similar functionality exists for the IN queue, and in this way, string data are passed back and forth from the R script to the Swift/T workflow. The queues themselves will block if the queue is empty, allowing the Swift/T workflow to pause and wait for data from the R script, and vice versa. When the R script waits and control returns back to Swift/T, the R interpreter is not deallocated. When subsequent R tasks execute on location L, they have access to the IN and OUT queues via the same functions. Through blocking queues and resident tasks, EQ/R implements an IoC pattern, where the logic embedded within the external ME algorithm, rather than in the Swift/T script, determines the progression of the workflow. As a note, a similar IoC pattern is employed with the EMEWS queues for the Python (EQ/Py) extension and Python-based ME algorithms.

E. Worker Types

Swift/T offers worker types, a powerful, high-level way to map the execution to various parts of the system. The user may specify any number of task types by simply providing a token. Then, functions that are defined with this token will execute only on workers (ranks) configured to accept these task types.

```
1  pragma worktypedef resident_work;
2  @dispatch=resident_work
3    register(string params) {
4      ... //  body
```

Similar to the Swift/T locality features (Section III-B), these offer a tradeoff between automated load balancing and full user control over execution location. They could be used to ensure that a small number of workers are allocated for performance-critical control tasks (e.g., tasks that produce input parameters for many other tasks) or to throttle the number of I/O-intensive tasks running at any point in time.

The EQ/R tasks have their own worker type *resident_work*. This enables R-based analysis code to be used for tasks, such as calculating complex objective functions from simulation outputs, without affecting the R interpreter used by the AL calculations.

F. Contiguous Ranks

In the previous work with Swift/T parallel tasks, worker ranks were assembled into per-task subcommunicators essentially, randomly. This was the most flexible technique and was immune to the fragmentation problems. For EMEWS, we extended the Swift/T parallel tasks feature to additionally support "parmod" (parallel-modulus) tasks. Communicators constructed to run tasks denoted with parmod $= n$ have the following two additional constraints.

1) They must start on a rank r, such that $r \equiv 0$ (mod n).
2) The ranks in the new subcommunicator are contiguous in the parent communicator.

For example, on a computer with 32 cores per node, the user could set parmod $= 32$, then a 32-process (`@par=32`) task would always consume exactly 1 node; when parmod $= 64$, a 64-process task would consume exactly 2 nodes that are topologically neighbors (assuming the MPI implementation is configured to lay out ranks in such a manner).

In this paper, we use parmod tasks for two reasons. The first is simply to gain the benefit of achieving the intranode performance for parallel SEIR model tasks by ensuring that all the processes in that node are running the same model instance. We run each task, that is, each model instance on 256 processes with a per process node count of 8 and thus fully utilize 32 nodes. Second, it allows us to cache data more easily since the task layout is always the same. If the communicator layouts were more random, it would take a great deal, more development time, to correctly manage data cached in the SEIR model, in this case, the initial synthetic population, from one parallel task to the next.

IV. Integration

Our focus is on the identification of the *viable* regions within the parameter space of the SEIR model. These regions represent input parameters resulting in model outputs that fall within the range of plausible flu incidence trajectories. The SEIR model includes stochasticity in two of its key elements. First, the initially exposed population is randomly distributed across the synthetic population. Second, the collocation-based infection dynamics stochastically determine whether an infection has occurred. Thus, as modelers, we are faced with the task of determining how to evaluate the "goodness" of a parameter set. We cannot simply look for time series fits to historical flu trends since the empirical time series are individual trajectories of flu infection dynamics that have been observed. What the empirical data do not show, for example, are all of the flu trajectories that did not occur (or possibly were not identified). Also, since the SEIR model distributes the initially exposed population randomly, it is highly unlikely

that any actual distribution of initially exposed people would match this, and since the infections are not spread in aggregate but through contacts between collocated individuals, the initial spatial distribution has the potential to greatly affect the timing and size of the flu incidence peak. As such, as we describe in the following, we resolved to run 20 stochastic variations for each parameter combination and characterize the parameter set as viable or not based on two *aggregate* statistics.

In this current parameterization of the SEIR model, the inputs that are allowed to vary are the initial number of infected individuals (C_I) and the hourly probability of going from susceptible to exposed per each collocated infectious agent ($P_{S \to E}$). C_I is ranged from 1 to 100 in increments of 1. $P_{S \to E}$ is ranged from 2e−5 to 4e−5 in increments of 0.02e−5. For each combination of these two parameters, the SEIR model outputs a table of newly infected agents for each week of a 35-week period. The objective function we use to characterize the model output calculates the mean and maximum values for each 35-week period. We define a threshold condition using the mean and maximum values within which the model outputs are deemed to adequately resemble empirically observed infection count trends for Chicago, obtained from [45]. The threshold condition used was less than 10 000 newly infected in any single week for the maximum and a mean across all 35 weeks of greater than 100 new infections per week. The computational challenge then becomes one of trying to characterize the SEIR model parameter space into viable and nonviable regions efficiently, that is, without having to run too many simulations to evaluate the viability of parameter combinations. While a number of different ensemble methods could potentially be used for this, the AL approach, described in the following, maps naturally to the problem.

A. AL Algorithm

AL [23] is a promising approach for characterizing the large parameter spaces of computational models (see [46]) with less expensive, reduced order models, or *metamodels*. AL combines the concepts from adaptive design of experiments (see [47]) and machine learning to iteratively and strategically sample from an unlabeled data set. AL works well in situations, where "…unlabeled data may be abundant or easily obtained, but labels are difficult, time-consuming, or expensive to obtain" [23]. The AL approach can be naturally mapped to the characterization of the parameter spaces of computer simulations when one considers the unlabeled data as points in a parameter space and the labeling activity as evaluating those points by running (possibly expensive) simulations.

In this paper, we chose to implement an R-based AL algorithm in order to highlight the types of useful and sophisticated parameter search approaches that can be developed when leveraging the existing functionality in widely used open source data analytics languages. Rather than requiring the time-intensive and error-prone reimplementation of these algorithms in Swift/T for the sole purpose of running large ensembles of simulations, we are able to have these algorithms directly control large-scale HPC workflows.

AL is a general approach which can afford a fair amount of customization in its specific implementation. The overall goal is to iteratively pick points (individual or sets) to sample, where the sampled points are chosen through some query strategy. In our case, we choose an *uncertainty sampling* strategy, where we employ a machine learning classifier on the already collected data and, then, choose subsequent samples close to the classification boundary, i.e., where the uncertainty between classes is maximal. In this way, we *exploit* the information that the classifier provides based on the existing data. To take advantage of the concurrence that we have available on HPC systems, the samples at each round of the AL procedure are batch collected (and evaluated) in parallel. In order to decrease the overlap in reducing classification uncertainty that nearby maximally uncertain sample points are likely to have, we cluster all the candidate points and choose an individual point within each cluster. This ensures a level of diversity in the sampled points and, therefore, a greater expected reduction of uncertainty [48]. We balance the exploitation of the classifier model with an *exploratory* component, where random points in the parameter space are sampled in order to investigate the additional regions that may not have been sampled yet. This can prevent premature convergence to an incorrect or incomplete metamodel.

The pseudocode for our AL algorithm is shown in Fig. 2. The workflow proceeds until the cross-validation metric, a proxy for out-of-sample model performance, is satisfied. Parallel evaluations of the objective function $F()$—the SEIR model simulation—are performed in lines 11 and 19 over some sample of parameters. At each iteration, the sampled results feed into the classifier R (lines 13 and 21). At the end of the workflow, the final metamodel predictions are generated for the remaining parameter space.

1: **define** P_{all} **as** *all parameter points*
2: **define** L **as** *all evaluated parameter points*
3: **define** P_{unev} **as** $(P_{all} - L)$
4: **define** $F()$ **as** *the objective function*
5: **define** R **as** *Random forest classifier with cross validation*
6: **define** M **as** *trained classifier model*
7: **define** $M.cp(p)$ **as** *classification probability of point p*
8: **define** km **as** *k-means clustering*
9: **define** max_c **as** *maximal number of kmeans clusters*
10: $P_{init} \leftarrow$ **sample**($n_{P_{init}}$ **from** P_{all})
11: **evaluate**($F(), P_{init}$)
12: $L \leftarrow L \cup P_{init}$
13: $M \leftarrow R$.**train**(L)
14: **while** *cross validation metric not satisfied in* M **and** *maximum iterations not exceeded* **do**
15: $P_{thresh} \leftarrow \forall p \in P_{unev} : M.\mathbf{cp}(p) \in (p_{low}, p_{high})$
16: $C = \{c_1, \dots c_{max_c}\} \leftarrow \mathbf{km}(P_{thresh})$
17: $P_{clus} \leftarrow \{p_i \in c_i : M.\mathbf{cp}(p_i)$ *closest to* $0.5\}$
18: $P_{rand} \leftarrow$ **sample**($n_{P_{rand}}$ **from** $P_{unev} - P_{clus}$)
19: **evaluate**($F(), P_{clus} \cup P_{rand}$)
20: $L \leftarrow L \cup (P_{clus} \cup P_{rand})$
21: $M \leftarrow R$.**train**(L)
22: **end while**
23: M.**generate_predictions**(P_{unev})

Fig. 2. Pseudocode for AL algorithm.

```
1  (void v) doAL(location L) {
2
3    for (boolean b = true, //  Loop variables
4         int i = 1;
5         b;                //  Loop condition
6         b = c,            //  Loop updates
7         i = i + 1)
8    {
9      string next_params = EQR_get(L);
10     boolean c;
11     if (next_params == "FINAL")
12     {
13       string results = EQR_get(L);
14       printf("Results: %s", results) =>
15         v = make_void() =>
16         c = false;
17     }
18     else
19     {
20       string res = run_model(next_params,p_count);
21       EQR_put(L, res) =>
22         c = true;
23     }
24   }
25 }
```

Fig. 3. Main Swift/T workflow loop.

B. Inversion of Control Implementation

Our central EMEWS workflow pattern is shown in Fig. 3. For our AL R algorithm, located at location `L`, the `doAL` function is called. The `for` loop continues to iterate while the new sets of parameters are obtained from the AL algorithm via the EQ/R `EQR_get` call. The parameter sets are sent to `run_model`, where they are split up and evaluated concurrently via a Swift/T `foreach` loop (not shown). Objective function results `results`, indicating a viable parameter combination or not, are returned by `run_model` and passed back to the AL algorithm via the EQ/R `EQR_put` call. This loop continues until the `EQR_get` call obtains the special token "`FINAL.`" Note that the `EQR_put` and `EQR_get` calls take the location `L` as a parameter. The implementation of `EQR_get` and `EQR_put` uses this location in a location-aware many-task scheduling annotation, as described in Section III-B.

Also, as described earlier, this qualifies as an IoC pattern since rather than Swift/T, the R-based AL algorithm controls the overall workflow logic. The algorithm produces simulation parameters and consumes results; however, instead of calling the model code directly, the parameters are intercepted and sent to Swift/T for distributed execution, with results seamlessly returned. This powerful pattern allows many third-party algorithms to be easily dropped into our framework and coordinate large-scale ensemble ME workflows.

C. AL EQ/R Communication Interface

As described in Section III-D, the interprocess communication is performed over queues. The queues are implemented in C++ but must also be accessible from Swift and R. The interface to these queues is shown in Fig. 4. Their implementation uses a straightforward Standard Template Library-based locking scheme. This library is exposed to Swift/T by using its SWIG-based library calling technique [49]. It is exposed to R via RInside [50]. Thus, the C++ data structure is available to both the Swift/T workflow and the R-based algorithm via Tcl and R wrapper interfaces, respectively.

```
1  #include <string>
2  void initR(std::string script_file);
3  std::string OUT_get(void);
4  void IN_put(std::string val);
5  void stopIt(void);
6  void deleteR(void);
```

Fig. 4. Queue implementation header: Swift to C++ linkage.

D. SEIR Model as Parallel Leaf Function

Since the SEIR model is an MPI application, it must be compiled as a shared library and wrapped in a Swift/T Tcl interface [42]. Through this interface, Swift/T passes a parameter string that contains all the parameters (i.e., the initial number of exposed persons, the various distribution values, and so on) for the current model run to the SEIR model. In addition, the Tcl interface also passes the MPI communicator for the current run. When the model receives the first set of parameters, it fills a cache with the required input data from the files specified in the parameter string, virtually eliminating I/O overhead in subsequent model runs. The caches are per process and contain the data for that process rank. The input data consist of person, place, and activity definitions. As part of the above-mentioned load balancing scheme, places are assigned to particular process ranks and persons move among processes as they move to the next place in their activity schedules. Each cache then contains the data for its process rank. Consequently, the caching mechanism requires consistent contiguous process ranks, such that the cache originally created on process n, remains on process n during subsequent runs. We make sure this is the case by setting the environment variable ADLB_PAR_MOD to the number of processes required to run the model (i.e., 256), enabling contiguous process ranks in communicators of that size.

For a more in-depth and technical description of the elements within an EMEWS workflow, including a complete AL workflow utilizing a distributed MPI-based model, the reader is referred to the EMEWS tutorial, accessible through the EWEWS site [7].

V. AL RESULTS

All experiments presented in Sections VI and VII were performed on the Cray XE6 *Beagle* at the University of Chicago, hosted at the Argonne National Laboratory. *Beagle* has 728 nodes, each with two AMD Operton 6300 processors, each having 16 cores, for a total of 32 cores per node; the system thus has 23 296 cores in all. Each node has 64 GB of RAM.

For the AL workflow run presented in this section, each SEIR model run was distributed over 256 processes (32 nodes), and we ran up to six models concurrently (192 nodes), demonstrating the hierarchical concurrence that EMEWS workflows can generate. Each model took approximately 7 s to run per simulated week, and we ran them for 35 weeks ($\approx$ 245 s per model run). The initial cache loading of person, place, and activity definitions occurred exactly once across each of the six sets of 32 nodes and took a total of 2 min.

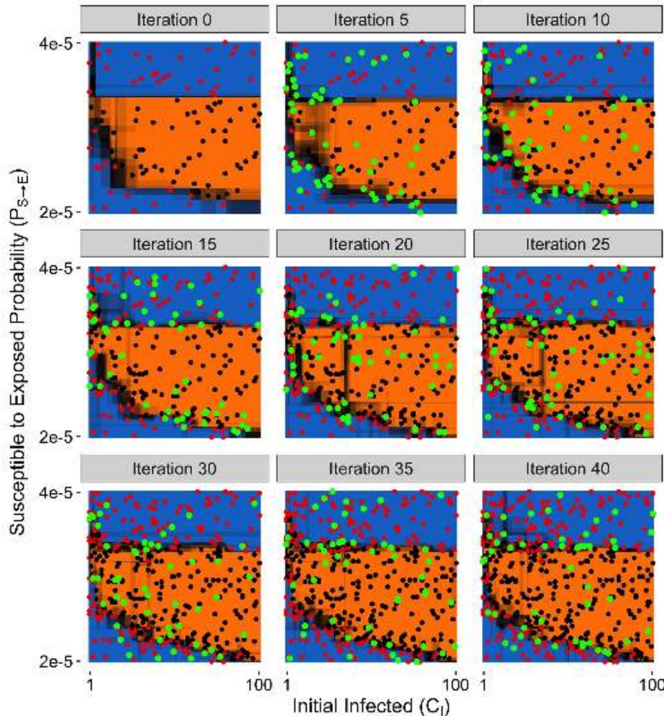

Fig. 5. Progression of the AL workflow, where the black/red dots indicate the evaluated (viable/nonviable) points, green points are newly added points since the previous panel, and orange/blue regions correspond to the out-of-sample predictions for (viable/nonviable) regions.

Fig. 5 shows the progression of the AL algorithm evaluating parameter points, training the random forest model, and generating predictions for the out-of-sample points in the 2-D C_I versus $P_{S\to E}$ parameter space over 40 iterations, where the parameter space was gridded into 10 100 discrete points (101×100). Each parameter point evaluation consists of 20 model runs of that parameter combination with the random seed varied for each of the runs and with the viability of the parameter set determined, as described in Section IV. Iteration 0 shows the initial design, where 100 randomly chosen points were evaluated. The black and red dots signify the parameter sets evaluated to be *viable* and *nonviable*, respectively. The orange and blue regions indicate the random forest metamodel out-of-sample prediction for *viable* and *nonviable* parameter space regions, respectively. The shading between the orange and blue regions represents the uncertainty in these predictions, where the darkest regions represent maximal uncertainty, i.e., equal probability of being *viable* or *nonviable*. As the iterations progress, points that were newly added since the last iteration panel are indicated by the green dots. For this particular AL workflow, at each iteration, we added five points close to the classification boundary (exploit) and five randomly sampled points (explore), for a total of 10 new points per iteration. Thus, at the end of iteration 40, about 5% of the parameter space was sampled. What can be observed is that as the AL progresses, the initial prediction boundary is gradually refined as additional points along it are evaluated, while the rest of the parameter space, where there is less uncertainty in the model prediction, e.g., the central part of the viable region, is not as densely explored. Importantly, regions of high uncertainty are seen to be reduced in width, sharpening the distinction between the two categories of interest. This pattern of parameter space evaluation is useful from the point of view of efficiently utilizing a computational budget, as the boundary points are the main drivers of an accurate metamodel. While the exploitation/exploration balance that we used appears to sufficiently cover and characterize our parameter space, other parameter spaces with, e.g., different dimensionality or granularity, may benefit from a different ratio.

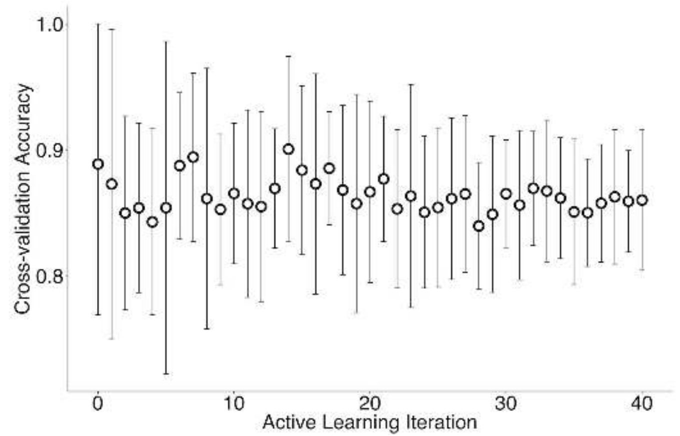

Fig. 6. CV accuracy means and standard deviation based on tenfold CV of the random forest metamodel at each AL iteration.

An iterative ME algorithm needs a termination condition. This can be based simply on a predetermined computation budget or some expected performance metric. In this example, we chose to monitor the cross-validation (CV) accuracy, both its sample mean and standard deviation. At each AL algorithm iteration, the random forest model is trained and tenfold cross validation is applied in order to get an estimate of the expected out-of-sample model performance. Fig. 6 shows the progression of the CV accuracy and standard deviation. What is observed is that while the CV accuracy is near constant, the standard deviation gradually decreases. This indicates that as the metamodel is being improved at each iteration with the addition of more data, we are able to better trust its out-of-sample performance level. This also reflects the increased certainty of the metamodel as seen by the reduction of shaded regions in Fig. 5. Finally, this also suggests the additional AL experiments, such as varying the number of initial samples or the number of samples chosen at each iteration, to observe the effects on the trajectory of CV accuracy or other CV metrics.

VI. Performance Results

A. Task Parallelism

In our application model (see Section III-A), there are multiple potential concurrence modes. Here, we describe the task-specific parallelism. As described in Section II-B, the SEIR model can be load balanced to run on any number of processes, parameterized by `p_count`. For these task parallelism experiments, we configured it to run on `p_count`=4, 8, 16, 32, 64, 128, and 256 processes. We measured the average time it took for the SEIR model to simulate one week within the workflow and reported it in Fig. 7.

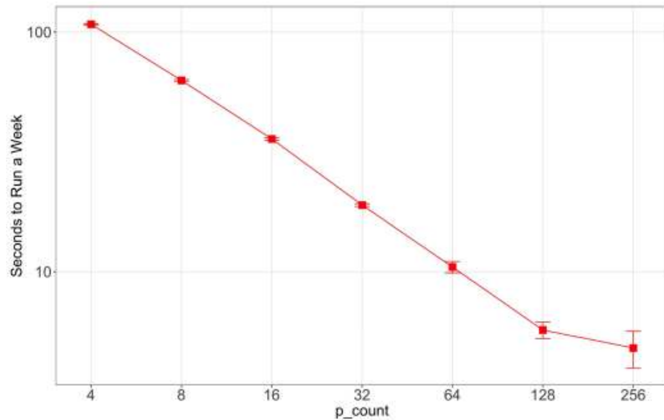

Fig. 7. Average time for the SEIR model to run a week as a function of `p_count`. Error bars are the sample standard deviation from 210 simulated weeks.

The results show that the SEIR model scales well to 128 and potentially to 256 processes. This scaling is important, as many of the ensemble methods of interest are iterative in nature, such that any performance increases that can be achieved for the simulation runs themselves are generally multiplied by the number of iterations required for the complete workflow if the necessary concurrence is available. Thus, the simulation developer has the option to retain a model's complexity rather than simplify it, such that it "…be amenable to comprehensive and systematic analysis" [24].

B. Total Time to Completion

For our SEIR/AL workflow performance evaluation, we constructed test AL workflows using a one ZIP code version (~44k agents) of the SEIR model. The tests in this paper exercised the full set of AL workflow components to observe their individual and collective performance characteristics. The cross-validation metric condition was modified to run past satisfaction to produce a consistent number of tasks (and thus always ran to the provided maximum number of iterations).

Our performance objective was to determine how the workflow overheads might affect the total time to complete the AL workflow. For each number of total processes, we ran the workflow at `p_count` = 4. This is the most challenging case for Swift, as higher `p_count` values reduce the number of tasks running at a time (as each task has more processes). For each increasing number of total processes, we increased the workload size (weak scaling). The total number of tasks in each workflow was hand-specified by selecting a maximum iterations number multiplied by the number of total processes; thus, the AL convergence criterion was disabled. The total number of tasks for each run was set to the number of total processes, and the maximal concurrence per round ($P_{\text{rand}} + P_{\text{clus}}$) was equal to the number of total processes divided by 4, and thus, there were four iterations. We recorded the total runtime reported by Swift and plotted it in Fig. 8.

As shown, the total workflow time is only minimally affected by scale. In our largest run, on 10 240 cores of Beagle, there is no utilization loss due to workflow overheads, demonstrating the robust scalability of EMEWS.

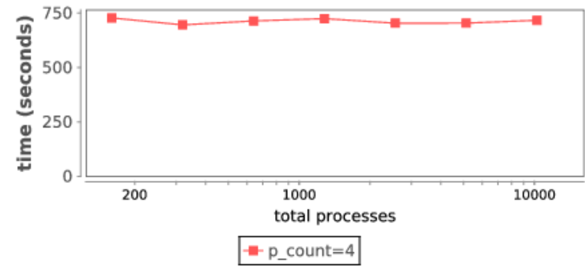

Fig. 8. Total makespan times for the one ZIP code SEIR model.

VII. Conclusion

In this paper, we have presented EMEWS, a framework for running large ensembles of simulations, in which the sophisticated ME algorithms can iteratively and adaptively refine simulation parameters through the analysis of recently generated results and launch new scientific applications based on the refined parameters. The mechanism itself has been implemented by using the Swift/T dataflow language and exhibits a novel form of IoC using location-aware many-task scheduling, resident tasks, and nontrivial IPC over HPC resources.

Using EMEWS, we developed an AL workflow through the selective reuse of third-party R packages, highlighting the multiple parallel programming language and runtime innovations, including novel features for parallel tasks (see Sections III-A and III-F), task locality (Section III-B), and stateful tasks (Section III-C), that make such a workflow possible. We demonstrated how the AL workflow was able to efficiently characterize the parameter space of a stochastic, large-scale, distributed SEIR model into viable and nonviable regions while sampling only a small fraction of possible parameters.

Performance results illustrate the basic scalability of EMEWS on a typical supercomputer. We demonstrated that a flexible range of concurrence strategies are within the performance envelope of our tools, enabling anything from a massive battery of single-process simulations to a mixture of varying multiprocess runs. Furthermore, while the focus here was the use of EMEWS for ABMs, EMEWS is being effectively applied to a variety of modeling methods (e.g., microsimulation [51], machine learning hyperparameter optimization [52], and biophysical modeling [53]) that require calibration, parameterization, or optimization achieved through the iterative execution of large numbers of computations.

We believe that as application teams consider good uses of near-exascale resources, they will observe that the defensible scientific investigations will have to be backed by large and novel many-task ensemble studies.

EMEWS has been released as an open source framework for the community [7], and we intend to continue to refine and improve it while continuing to develop additional use case examples that exploit widely available ME libraries. Ultimately, the goal of EMEWS is to democratize the use of HPC resources by allowing the nonexpert researchers to tap into advanced third-party ensemble ME methods, such as optimization or AL algorithms, to take advantage of

the extreme-scale systems that will become available in the upcoming years.

Acknowledgment

The authors would like to thank the Beagle System and the Research Computing Center, The University of Chicago, Chicago, IL, USA, for the resources provided by them.

References

[1] T. C. Germann, K. Kadau, I. M. Longini, Jr., and C. A. Macken, "Mitigation strategies for pandemic influenza in the United States," *Proc. Nat. Acad. Sci. USA*, vol. 103, no. 15, pp. 5935–5940, Apr. 2006.

[2] C. M. Macal *et al.*, "Modeling the transmission of community-associated methicillin-resistant *Staphylococcus aureus*: A dynamic agent-based simulation," *J. Transl. Med.*, vol. 12, no. 1, p. 124, May 2014.

[3] M. Riddle, C. M. Macal, G. Conzelmann, T. E. Combs, D. Bauer, and F. Fields, "Global critical materials markets: An agent-based modeling approach," *Resour. Policy*, vol. 45, pp. 307–321, Sep. 2015.

[4] J. Ozik *et al.*, "Simulating water, individuals, and management using a coupled and distributed approach," in *Proc. Winter Simulation Conf. (WSC)*. Piscataway, NJ, USA: IEEE Press, 2014, pp. 1120–1131.

[5] F. Bert, M. North, S. Rovere, E. Tatara, C. Macal, and G. Podestá, "Simulating agricultural land rental markets by combining agent-based models with traditional economics concepts: The case of the Argentine Pampas," *Environ. Model. Softw.*, vol. 71, pp. 97–110, Sep. 2015.

[6] J. Ozik, N. T. Collier, J. M. Wozniak, and C. Spagnuolo, "From desktop to large-scale model exploration with Swift/T," in *Proc. Winter Simulation Conf. (WSC)*. Piscataway, NJ, USA: IEEE Press, 2016, pp. 206–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042094.3042132

[7] *EMEWS: Extreme-Scale Model Exploration With Swift*. Accessed: Aug. 17, 2018. [Online]. Available: http://emews.org

[8] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *Proc. CCGrid*, 2013, pp. 95–102.

[9] J. Ozik, N. T. Collier, and J. M. Wozniak, "Many resident task computing in support of dynamic ensemble computations," in *Proc. 8th Workshop Many-Task Comput. Clouds, Grids, Supercomput. (MTAGS)*, 2015. [Online]. Available: http://datasys.cs.iit.edu/events/MTAGS15/p03.pdf

[10] M. J. North and C. M. Macal, *Managing Business Complexity: Discovering Strategic Solutions With Agent-Based Modeling and Simulation*, 1st ed. London, U.K.: Oxford Univ. Press, Mar. 2007.

[11] J. T. Murphy, "Computational social science and high performance computing: A case study of a simple model at large scales," in *Proc. Annu. Conf. Comput. Social Sci. Soc. Amer.*, Santa Fe, NM, USA, Oct. 2011, pp. 1–12.

[12] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*, 2nd ed. Hoboken, NJ, USA: Wiley, 2005.

[13] M. D. Mckay, R. J. Beckman, and W. J. Conover, "Comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979.

[14] M. D. Morris, "Factorial sampling plans for preliminary computational experiments," *Technometrics*, vol. 33, no. 2, pp. 161–174, 1991.

[15] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," *J. Statist. Phys.*, vol. 34, nos. 5–6, pp. 975–986, 1984.

[16] R. Verfürth, "A posteriori error estimation and adaptive mesh-refinement techniques," *J. Comput. Appl. Math.*, vol. 50, nos. 1–3, pp. 67–83, 1994.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: Bradford Book, 1992.

[18] M. A. Beaumont, "Approximate Bayesian computation in evolution and ecology," *Annu. Rev. Ecol., Evol., Systematics*, vol. 41, no. 1, pp. 379–406, 2010.

[19] F. Hartig, J. M. Calabrese, B. Reineking, T. Wiegand, and A. Huth, "Statistical inference for stochastic simulation models—Theory and application," *Ecol. Lett.*, vol. 14, no. 8, pp. 816–827, Aug. 2011.

[20] G. Evensen, *Data Assimilation: The Ensemble Kalman Filter*, 2nd ed. Berlin, Germany: Springer-Verlag, 2009.

[21] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEE Proc. F-Radar Signal Process.*, vol. 140, no. 2, pp. 107–113, 1993.

[22] M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking," *IEEE Trans. Signal Process.*, vol. 50, no. 2, pp. 174–188, Feb. 2002.

[23] B. Settles, *Active Learning* (Synthesis Lectures on Artificial Intelligence and Machine Learning), vol. 6. San Rafael, CA, USA: Morgan & Claypool, Jun. 2012, pp. 1–114.

[24] J. C. Thiele, W. Kurth, and V. Grimm, "Facilitating parameter estimation and sensitivity analysis of agent-based models: A cookbook using NetLogo and R," *J. Artif. Societies Social Simul.*, vol. 17, no. 3, p. 11, 2014.

[25] J. Shaman and A. Karspeck, "Forecasting seasonal outbreaks of influenza," *Proc. Nat. Acad. Sci. USA*, vol. 109, no. 50, pp. 20425–20430, Dec. 2012.

[26] J. Shaman, A. Karspeck, W. Yang, J. Tamerius, and M. Lipsitch, "Real-time influenza forecasts during the 2012–2013 season," *Nature Commun.*, vol. 4, Dec. 2013, Art. no. 2837.

[27] W. Yang, A. Karspeck, and J. Shaman, "Comparison of filtering methods for the modeling and retrospective forecasting of influenza epidemics," *PLoS Comput. Biol.*, vol. 10, no. 4, p. e1003583, Apr. 2014.

[28] J. Shaman, W. Yang, and S. Kandula, "Inference and forecast of the current west african ebola outbreak in guinea, sierra leone and liberia," *PLoS Currents*, Oct. 2014. [Online]. Available: http://currents.plos.org/outbreaks/article/inference-and-forecast-of-the-current-west-african-ebola-outbreak-in-guinea-sierra-leone-and-liberia, doi: 10.1371/currents.outbreaks.3408774290b1a0f2dd7cae877c8b8ff6.

[29] J. M. Epstein, "Modelling to contain pandemics," *Nature*, vol. 460, no. 7256, p. 687, Aug. 2009.

[30] F. Brauer, P. van den Driessche, and J. Wu, Eds., "Compartmental models in epidemiology," in *Mathematical Epidemiology*. Berlin, Germany: Springer, 2008, ch. 2, pp. 19–79.

[31] Centers for Disease Control. (2016). *How Flu Spreads*. Accessed: Mar. 25, 2016. [Online]. Available: http://www.cdc.gov/flu/about/disease/spread.htm

[32] Centers for Disease Control. (2016). *Flu Symptoms*. Accessed: Mar. 25, 2016. [Online]. Available: http://www.cdc.gov/flu/consumer/symptoms.htm

[33] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *Simulation*, vol. 89, no. 10, pp. 1215–1235, Nov. 2012.

[34] N. Collier, J. Ozik, and C. M. Macal, "Large-scale agent-based modeling with repast HPC: A case study in parallelizing an agent-based model," in *Proc. Eur. Conf. Parallel Process.*, Vienna, Austria, Aug. 2015, pp. 454–465, doi: 10.1007/978-3-319-27308-2_37.

[35] W. D. Wheaton *et al.*, "Synthesized population databases: A US geospatial database for agent-based models," *Methods Rep.*, vol. 2009, no. 10, p. 905, 2009. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2875687, doi: 10.3768/rtipress.2009.mr.0010.0905.

[36] S. Gallagher, L. Richardson, S. L. Ventura, and W. F. Eddy. (Jan. 2017). "SPEW: Synthetic populations and ecosystems of the world." [Online]. Available: https://arxiv.org/abs/1701.02383

[37] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, *Programming Models for Parallel Computing*, P. Balaji, Ed. Cambridge, MA, USA: MIT Press, 2015.

[38] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proc. SC*, 2014, pp. 299–310.

[39] S. J. Krieder *et al.*, "Design and evaluation of the GeMTC framework for GPU-enabled many-task computing," in *Proc. HPDC*, 2014, pp. 153–164.

[40] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster, "Interlanguage parallel scripting for distributed-memory scientific computing," in *Proc. WORKS SC*, 2015, Art. no. 6.

[41] F. R. Duro, J. G. Blas, F. Isaila, J. Carretero, J. M. Wozniak, and R. Ross, "Experimental evaluation of a flexible I/O architecture for accelerating workflow engines in ultrascale environments," *Parallel Comput.*, vol. 61, pp. 52–67, Jan. 2017.

[42] J. M. Wozniak *et al.*, "Dataflow coordination of data-parallel tasks via MPI 3.0," in *Proc. EuroMPI*, 2013, pp. 1–6.

[43] D. Abramson, A. Lewis, T. Peachey, and C. Fletcher, "An automatic design optimization tool and its application to computational fluid dynamics," in *Proc. SuperComputing*, 2001, p. 47.

[44] B. M. Adams *et al.*, "DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: Version 5.0 user's manual," Sandia, Albuquerque, NM, USA, Tech. Rep. SAND2014-4633, Nov. 2015.

[45] J. Shaman. (2016). *Columbia Prediction of Infectious Diseases*. [Online]. Available: http://cpid.iri.columbia.edu

[46] M. Cevik, M. A. Ergun, N. K. Stout, A. Trentham-Dietz, M. Craven, and O. Alagoz, "Using active learning for speeding up calibration in simulation models," *Med. Decision Making*, vol. 36, no. 5, pp. 581–593, Oct. 2015, doi: 10.1177/0272989X15611359.

[47] R. Jin, W. Chen, and A. Sudjianto, "On sequential sampling for global metamodeling in engineering design," in *Proc. 28th Design Automat. Conf.*, Jan. 2002, pp. 539–548.

[48] Z. Xu, R. Akella, and Y. Zhang, "Incorporating diversity and density in active learning for relevance feedback," in *Advances in Information Retrieval. ECIR* (Lecture Notes in Computer Science), vol. 4425, G. Amati, C. Carpineto, and G. Romano, Eds. Berlin, Germany: Springer, 2007, pp. 246–257, doi: 10.1007/978-3-540-71496-5_24.

[49] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster, "Toward interlanguage parallel scripting for distributed-memory scientific computing," in *Proc. CLUSTER*, 2015, pp. 482–485.

[50] D. Eddelbuettel and R. Francois. *RInside CRAN Package*. Accessed: Aug. 17, 2018. [Online]. Available: https://cran.r-project.org/web/packages/RInside

[51] C. Rutter, J. Ozik, M. DeYoreo, and N. Collier. (Apr. 2018). "Microsimulation model calibration using incremental mixture approximate Bayesian computation." [Online]. Available: https://arxiv.org/abs/1804.02090

[52] J. M. Wozniak *et al.*, "CANDLE/supervisor: A workflow framework for machine learning applied to cancer research," *BMC Bioinf.*, to be published. [Online]. Available: https://bmcbioinformatics.biomedcentral.com

[53] J. Ozik *et al.*, "High-throughput cancer hypothesis testing with an integrated PhysiCell-EMEWS workflow," *BMC Bioinf.*, to be published. [Online]. Available: https://bmcbioinformatics.biomedcentral.com

Jonathan Ozik received the Ph.D. degree from the University of Maryland, College Park, MD, USA, in 2005.

He is currently a Computational Scientist with the Decision and Infrastructure Sciences Division, Argonne National Laboratory, Argonne, IL, USA, and a Senior Scientist with the Consortium for Advanced Science and Engineering, The University of Chicago, Chicago, IL, USA. He leads the Repast agent-based modeling toolkit and the EMEWS framework for large-scale model exploration.

Nicholson T. Collier received the Ph.D. degree from The University of Chicago, Chicago, IL, USA, in 1998.

He is currently a Software Engineer with the Decision and Infrastructure Sciences Division, Argonne National Laboratory, Argonne, IL, USA, and a Research Staff with the Consortium for Advanced Science and Engineering, The University of Chicago. He is also the Lead Developer of the Repast, agent-based modeling toolkit, and a Core Developer of the EMEWS Framework.

Justin M. Wozniak received the Ph.D. degree from the University of Notre Dame, Notre Dame, IN, USA, in 2008.

He is currently a Computer Scientist with the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA, and a Scientist with the Consortium for Advanced Science and Engineering, The University of Chicago, Chicago, IL, USA. He is the Lead Developer of the Swift/T, a parallel scripting language, and a Core Developer of the EMEWS Framework.

Charles M. Macal received the Ph.D. degree from Northwestern University, in 1989.

He has been a Registered Professional Engineer at the State of Illinois, since 1980. He is currently a Senior Systems Engineer, an Argonne Distinguished Fellow, the Group Leader of the Decision and Infrastructure Sciences Division, Social, Behavioral and Decision Science Group, Argonne National Laboratory, Argonne, IL, USA, and a Senior Scientist with the Consortium for Advanced Science and Engineering, The University of Chicago, Chicago, IL, USA. He is recognized globally as a Leader in the field of agent-based modeling and simulation and has led interdisciplinary research teams in developing innovative computer simulation models in application areas, including global and regional energy markets, critical materials, electric power, healthcare and infectious diseases, environment and sustainability, and technology adoption.

Gary An received the M.D. degree from the University of Miami, Coral Gables, FL, USA, in 1988.

He is currently an Associate Professor of surgery with the Department of Surgery, The University of Chicago, Chicago, IL, USA. His current research interests include the development of mechanism-based computer simulations in conjunction with biomedical research labs, high-performance/parallel computing architectures for agent-based models, artificial intelligence systems for modular model construction, and community-wide metascience environments, all with the goal of facilitating transformative scientific research. Toward these areas, he has developed agent-based models of sepsis, multiple organ failure, wound healing, surgical site infections, necrotizing enterocolitis, tumor metastasis, breast cancer, C. difficile colitis, and the link between oncogenesis and inflammation.