Experiences in Developing a Distributed Agent-based Modeling Toolkit with Python

Nicholson T. Collier
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL USA
ncollier@anl.gov

Jonathan Ozik
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL USA
jozik@anl.gov

Eric R. Tatara
Decision and Infrastructure Sciences
Argonne National Laboratory
Lemont, IL USA
tatara@anl.gov

***Abstract*—Distributed agent-based modeling (ABM) on high-performance computing resources provides the promise of capturing unprecedented details of large-scale complex systems. However, the specialized knowledge required for developing such ABMs creates barriers to wider adoption and utilization. Here we present our experiences in developing an initial implementation of Repast4Py, a Python-based distributed ABM toolkit. We build on our experiences in developing ABM toolkits, including Repast for High Performance Computing (Repast HPC), to identify the key elements of a useful distributed ABM toolkit. We leverage the Numba, NumPy, and PyTorch packages and the Python C-API to create a scalable modeling system that can exploit the largest HPC resources and emerging computing architectures.**

***Index Terms*—agent-based modeling, parallel processing, high performance computing, computer simulation**

I. Introduction

Agent-based modeling (ABM) is a method of computing system-level consequences of the behaviors of sets of individuals [1]. Recent improvements in high-performance ABMs have enabled the simulation of a variety of complex systems, including the spread of infectious diseases and community-based healthcare interventions in social systems [2]–[4], and cancer immunotherapy in biological systems [5], [6]. There are two primary ways in which ABMs intersect with high-performance computing (HPC), through the "inner" and "outer" loops [7]. The inner loop refers to the model itself, where the runtime performance is improved by distributing an individual ABM across multiple computing units via technologies like MPI (for internode parallelization) and OpenMP (for intranode parallelization). The outer loop, also referred to as model exploration [8], is the application of analyses such as optimization, uncertainty quantification, and data assimilation on the inner loop model. While the focus of this paper is the inner loop, the consequences of a performance-optimized model is to enable a wider variety of sequential outer loop algorithms involving intensive repeated execution of the inner loop.

The technologies that enable HPC ABMs are complex, creating barriers to wider adoption and utilization. There have emerged application area specific frameworks that attempt to reduce the complexity of developing large-scale ABMs [9], [10] by providing simplified layers. However, these necessarily lack the generalizability for application to diverse scientific areas.

This work is supported by the U.S. Department of Energy under contract number DE-AC02-06CH11357 and by the National Institutes of Health under Award U2CDA050098.

Here we present an initial implementation of a distributed general-purpose ABM toolkit developed with Python as the modeling language. Python has become the *de facto lingua franca* for artificial intelligence (AI) and is widely used across many scientific disciplines for machine learning (ML) and other applications. It also boasts a vibrant ecosystem of libraries that exploit the latest advances in algorithmic development and emerging hardware architectures. Building on our experience in designing and developing the widely used Repast ABM toolkit (https://repast.github.io), we describe the steps we took to identify the requirements, test the component performance, and assemble an initial implementation of Repast4Py.

II. Related Work

In the last decade, there have been several relevant parallel distributed implementations of an ABM toolkit. In this section, we discuss five of these as well as relevant work in distributed discrete event scheduling.

D-MASON [11] is a parallel version of the MASON [12] ABM framework. Implemented in Java, D-MASON distributes a simulation by partitioning the space and the agents to be simulated into regions, These regions are each managed by a worker process, and all agent-to-agent and agent-to-space interaction is local within that worker. Synchronization across workers is performed using a publish-subscribe pattern that propagates agent and environment state information across regions that neighbor each other. D-MASON initially used the Java Message service for communication between processes and thus was more appropriate for use on small clusters rather than larger high performance computing resources. Subsequent iterations [13] retain the publish subscribe pattern, but use MPI as the messaging implementation.

Pandora [14] is an ABM framework written in C++ where models can be written in both C++ and Python. Like D-MASON, the spatial environment of the simulation (a 2D raster) is partitioned over the available processes and agent-to-agent and agent-to-space interaction is local within that process. Each process' environment contains a buffer where neighboring environments overlap. With a novel solution, Pandora avoids race-conditions on the buffer by splitting the space within a process into four equal parts and executing agent behavior and spatial updates within those parts sequentially, synchronizing after each part updates. These parts are sectioned such that they do not share a border with the same numbered part on another process and thus any border update issues are avoided. Cross-process synchronization is accomplished via MPI. Pandora also provides an OpenMP based scheduler that executes certain specified agent behavior in parallel, where such behavior should not cause any race conditions.

Care HPS [15] is an ABM framework for parallel and distributed agent-based modeling written in C++ with a focus on rapid model development and experimentation on HPC resources. The simulation space (a 2D or 3D Euclidian space) and the agents in them can be partitioned via MPI using one of three methods: cluster based partitioning, strip based partitioning and a hybrid solution. The cluster based partitioning divides agents among processes according to their interaction patterns. The strip based partitioning divides the simulation space into strips (1 strip per process rank), and assigns the agents to strips based on their simulated location. The hybrid solution extends the strip based partitioning by executing agent behavior on under-utilized processes in parallel using OpenMP. Care HPS does not use a scheduler to order and execute simulation behavior, rather each agent (a C++ class) implements an `update_agent` method that gets called each simulation step.

The Flexible Large-scale Agent Modeling Environment (FLAME) [16], implemented in C using MPI, has its origins in the large-scale simulation of biological cell growth. Unlike D-MASON, Pandora, and Care HPS where each agent type is implemented as a C++ or Java class, each agent is modeled as an X-Machine. A X-Machine is a finite state machine consisting of a set of finite states, transition functions, and input and output messages. Communication between agents and the environment is by means of messages which are posted to a message board where they can be read by other agents and the environment. The simulation is partitioned into multiple nodes across processes, where each node has a message board and a set of agents. Each agent communicates with the message board, which in turn communicates with other message boards, synchronizing the state of the simulation across processes. Leveraging the fact that agent communication is typically local, agents are clustered into processes (‘neighborhoods’) according to their interactions patterns and sphere of influence.

Flame GPU [17] is a high performance GPU extension to the FLAME framework. Models are specified using the X-Machine Markup Language (XMML), including agent (i.e., X-Machine) definitions, message types, and the execution order of agent functions. Additional behavior scripts can be written in C. These are then parsed and compiled into CUDA code that targets GPU devices. Unlike the previous ABM toolkits, agents are not distributed across processes, but rather across a GPU such that their behavior can be executed in a massively parallel manner.

Other work, although not strictly focusing on the large-scale execution of ABM applications, explores the related area of parallel discrete event simulation (PDES) [18], [19]. Such work typically focuses on parallel event synchronization where logical processes communicate and synchronize with each other through time-stamped events. Three approaches are commonly used: conservative, optimistic and mixed-mode [20], [21]. The conservative approach blocks the execution of the process, ensuring the correct ordering of simulation events [22], [23]. The optimistic approach avoids blocking, but must compensate when events have been processed in the incorrect order. The Time Warp operating system is the pre-eminent example of this approach [24], [25]. Mixed-mode combines the conservative and optimistic approaches [26], [27]. In a more ABM-like context, the trio of Sharks World papers [28]–[30] explore the application of these approaches to the Sharks World simulation (sharks preying on fish inhabiting a toroidal world).

Building on this related work and on our previous work, in particular the Repast HPC [31] toolkit described below, we identify the following three guiding elements that are likely needed for creating a successful (i.e., widely used) distributed ABM toolkit: 1) a critical need for ease of use to reduce the expertise-impedance mismatches that exist between individual scientific communities and the relevant HPC realms (viz. Pandora’s use of Python), 2) scalable performance that exploits the continually increasing concurrency and hybrid aspects of HPC resources (viz. Flame GPU), and 3) novel strategies for distributing agents and scheduling their actions (viz. Care HPS) to support different types of distributed ABMs. Our development of Repast4Py is guided by these themes. Python was chosen as the modeling language for its ease of use and proliferation across many scientific domains. Scalable performance is achieved through the implementation of optimized ABM components, written using Python’s C-API, and the NumPy [32], Numba [33], and PyTorch [34] Python packages. Finally, the flexibility of Python development makes it amenable to rapid prototyping of novel distribution and scheduling strategies.

III. The Repast Suite, Elements of ABM, and Requirements for Distributed ABM

Developed originally in 2001, the first Repast ABM toolkit [35] was written in Java and targeted typical desktop machines. Subsequent iterations brought a .NET C-Sharp port, an early experimental (and popular) Python-based variant and a large user base [36]. The current Repast ABM Suite consists of two toolkits, Repast Simphony [37] and Repast for High Performance Computing (Repast HPC) [31]. Repast Simphony is a Java-based platform that provides multiple methods for specifying agent-based models, including pure Java, Statecharts [38] and ReLogo [39], a dialect of Logo. The architectural design is based on central principles important to agent modeling and software design, including modular, layered and pluggable components that provide for flexibility, extensibility and ease of use [37]. These principles combine findings from many years of ABM toolkit development and from experience using the ABM toolkits to develop models for a variety of application domains. Repast HPC is a parallel distributed C++ implementation of Repast Simphony using MPI and intended for use on high performance distributed-memory computing platforms. It attempts to preserve the salient features of Repast Simphony for Java and provide Logo-like components similar to ReLogo, while adapting them for parallel computation in C++.

A. Contexts and Projections

One of the major goals of Repast Simphony was flexibility and reusability of its various components. Previous versions suffered from a lack of such flexibility. In particular, the code required to make an agent participate in an environment such as a network made it difficult for that agent to participate in another environment such as a 2D grid space. Repast Simphony overcame this problem through the use of *contexts* and *projections*. The context is a simple container based on set semantics. Any type of object can be put into a context, with the simple caveat that only one instance of any given object can be contained by the context. From a modeling perspective, the context represents a population of agents. The agents in a context are the population of a model. However, the context does not inherently provide any relationship or structure for that population. Projections take the population as defined in a context and impose a structure on it. The structure defines and imposes relationships on the population using the semantics of the projection. Actual projections are such things as a network structure that allows agents to form links (network type relations) with each other, a grid where each agent is located in a matrix-type space, or a continuous space where an agent's location is expressible as a non-discrete coordinate [37]. Projections have a many-to-one relationship with contexts. Each context can have an arbitrary number of projections associated with it. Projections are designed to be agnostic with respect to the agents to which projections provide structure.

B. Simulation Scheduling

Events in Repast simulations are driven by a discrete-event scheduler. These events themselves are scheduled to occur at a particular *tick*. Ticks do not necessarily represent clock-time but rather the priority of its associated event. Ticks determine the order in which events occur with respect to each other. For example, if event A is scheduled at tick 3 and event B at tick 6, event A will occur before event B. Assuming nothing is scheduled at the intervening ticks, A will be immediately followed by B. There is no inherent notion of B occurring after a duration of 3 ticks. Of course, ticks can and are often given some temporal significance through the model implementation. A traffic simulation, for example, may move the traffic forward the equivalent of 30 seconds for each tick. A floating point tick, together with the ability to order the priority of events scheduled for the same tick, provides for flexible scheduling. In addition, events can be scheduled dynamically such that the execution of an event may schedule further events at that same or at some future tick.

C. Repast HPC

Repast HPC was developed in order to enable the implementation of large-scale (e.g., city size) models that are not possible with Repast Simphony or similar non-distributed frameworks. Agent-based models with large populations of agents can be prohibitively slow to execute. The threshold where slow becomes too slow is dependent on the types of questions that the model is intended to answer and the overall experimental design needed to provide those answers. Adequate results may be achieved within a limited period of time. However, to serve as useful electronic laboratories, ABMs often require the execution of many iterated model runs to account for stochastic variation in model outputs and for the various dynamic sampling algorithms to calibrate and analyze them [3]. By distributing an agent population over many processes and reducing the time needed for evaluating individual model runs, Repast HPC allows an ABM to be used as the foundation of an electronic laboratory in which sophisticated, sequential statistical techniques can be used for these types of analyses. See, for example, [3], [40], [2], and [41].

However, while Repast HPC made possible much larger simulations by leveraging the power of HPC resources, it does lack the ease of use of its predecessors. A small part of this difficulty is conceptual: designing and implementing a parallel distributed model is more difficult (perhaps partly due to unfamiliarity) than a single process shared memory one. The majority of the difficulty, though, is due to the implementation language, C++. While extremely powerful, C++ can be difficult to learn, and more importantly, the articulation of a modeler's intention in code can be laborious, particularly for inexperienced programmers. One of our initial hopes for Repast HPC and later our EMEWS [8] framework was to make HPC resources easier to use both for model development and analysis. C++ can be an impediment to that end, especially for persons with little software development experience. Python's popularity for AI and ML applications and its proliferation across scientific domains, when combined with its wide availability on HPC resources, the availability of MPI via the MPI4Py [42] package, and our own familiarity with it [43], made it a natural choice as an implementation language.

D. Requirements for a Distributed ABM Toolkit

In Sections III-A and III-B, we have described the Repast Simphony features that have proven useful in agent-based modeling. Distributed, or what we have called *shared* [31], implementations of these form the initial requirements for a distributed ABM toolkit. By *shared* we want to emphasize the partitioned and distributed nature of the simulation. The global simulation is shared among a pool of processes, each of which is responsible for some portion of it. We provide the complete requirements for a distributed ABM toolkit below.

1) Shared Context: A distributed ABM partitions its agents across multiple processes. A shared context manages the population of agents on a single process, both the agents local to that process and non-local copies of agents on other processes. Critically, it provides the functionality to send and receive agents from other processes. As a container, it can be iterated through, and searched for agents that match some query condition. As agents are added or removed from a context, they are added and removed from the projections within that context.

2) Shared Projections: As mentioned above (III-A), a context does not inherently provide any relationship or structure for the population of agents it contains. Projections take the population as defined in a context and impose a particular structure on it. The structure defines relationships among the population using the semantics of the projection. A shared projection partitions this global structure across individual processes, such that each process is responsible for some section of that structure. For example, in a network projection, the process would be responsible for some subset of nodes and links from the global set of nodes and links. In a spatial projection, the process is responsible for some section of the global area. As parts of a global whole, a shared projection needs to unite the parts into the whole projection via a ghosting or haloing strategy. For example, in a network projection, where there are links between nodes local to a process and those on other processes, the remote nodes can be copied between processes and the appropriate links created between the local and non-local copies (ghosts) of the remote nodes. For spatial projections, such as a discrete 2D grid, neighboring subsections of the space can be slightly extended into their neighboring space together with copies of the agents in that extended (buffered) section.

3) Shared Scheduler: The shared scheduler determines the order in which events happen in a simulation and synchronizes the execution of events across processes. Each process has its own shared scheduler, executing its own events. These schedulers coordinate such that each process is executing the events for the same time *tick*, maintaining a shared coherent simulation state by preventing a process from running either ahead or behind the other processes.

4) Agents & Agent Identity: In an ABM framework, it is necessary to associate an agent with arbitrary but required data (e.g., an agent to a location) using a unique agent identifier. This is particularly important in a distributed ABM to avoid sending redundant information. Rather than sending the entire agent, a unique identifier for that agent can be sent and the remaining data can be retrieved using a cache lookup, for example [44].

5) Performance: As we have stressed above, an ABM framework must be performant to be useful as the foundation of an experimental laboratory. Performance is primarily achieved by dividing the global population of agents among processes. A typical simulation can loop through all the agents during each iteration of a simulation. The fewer agents to iterate over the faster the simulation. This divide-and-conquer strategy requires both that the cost of moving agents among processes does not exceed the benefit of parallelization and that the agents are properly load balanced across processes to avoid the creation of laggard processes. Issues we have explored in detail in [44] and that continue to guide the implementation that is described in the sections that follow. Component level performance is also achieved by leveraging emerging hybrid CPU/GPU architectures.

IV. Structure and Performance Investigations for Python Distributed ABM Capabilities

Typical distributed simulations (e.g., those developed with Repast HPC) contain either relatively few computationally complex, or heavy, agents per process or a multitude of lighter-weight agents per process. The distinct processes communicate, for example to share agents between themselves, using MPI. Each individual process is responsible for some number of agents *local* to that process. These agents reside in the memory associated with that process and the process executes the code that represents these *local* agents' behavior. In terms of Repast components, each process has a scheduler (III-B), a context (III-A) and the projections (III-A) associated with that context. At each tick, a process' schedule will execute the events scheduled for the current tick, typically some behavior of the agents in the context that makes use of a projection. For example, each agent in a context might examine its network neighbors and take some action based on those links, or each might examine its grid surroundings and move towards or away from some location.

As mentioned above in III-C, Repast HPC enables the creation of performant large-scale simulations and their sequential evaluation through distributed parallelization. With respect to performance, our focus is on creating a useful and usable working framework. A Python implementation need not be as fast as one written in C++, and as long as the Python implementation is scalable, additional computational resources can be used to improve performance. Nevertheless, it does have to be fast enough to allow for intensive repeated execution of simulations within a reasonable amount of time. If a Python implementation cannot achieve this "good enough" level of performance, then it will not be useful. Consequently, we began our investigations of the feasibility of a Python distributed ABM toolkit with some profiling to identify performance hotspots in a pure Python implementation, and to determine where packages such as NumPy [32], Numba [33], and PyTorch [34], and native code implementations using the Python C-API could be used to improve performance. For these benchmarks we began with the *Simple Model*, a toy model intended to exercise agent movement across processes and a computationally expensive agent behavior.

A. The Simple Model

The Simple Model creates a user specified number of agents and 20 "places" on each process. The pseudocode can be seen

```
1:  Given a population of agents $A$, places $P$, distributed over process ranks
    $Pr$,
2:  for each iteration of the model do
3:      for each process $pr \in Pr$ do
4:          for each agent $a \in A$ do
5:              Move $a$ to a random rank $r \in Pr$
6:              Place $a$ in a random place $p \in P$ on $r$
7:          end for
8:          for each place $p \in P$ on $pr$ do
9:              for each agent $a \in p$ do
10:                 Play tag with every other agent $\in p$
11:             end for
12:         end for
13:     end for
14: end for
```

Fig. 1. Pseudo-code for the Simple Model

TABLE I
SIMPLE MODEL RUNTIMES

Component	Subcomponent	Runtime (sec.)
Cross-Process Movement	Serialization	2.37
	alltoall MPI call	2.09
	Deserialization	3.27
Agent behavior	Nested tag loop	30.93
Total		40.84

in Figure 1.

Here lines 4-7 move agents across processes while lines 8-12 mimic computationally expensive agent to agent interactions via a nested loop. The agents themselves are implemented as Python classes with `id` and `tag_count` integer fields, and a `dp` double field. The latter being included to exercise MPI4Py's ability to send tuples of mixed types. When an agent is moved between processes, these 3 values are packed into a tuple and that tuple is sent to the destination process. The agent is removed from the source process and then created using that tuple on the destination process. The 20 places are each implemented as a Python class, essentially a wrapper around a Python list into which an agent is added when it is moved to a place, together with some additional methods for running the game of *tag* in that place (see Listing 1). These initial experiments were run over 4 processes with 3000 agents on each process, and for 300 iterations. Our overarching intention in the initial experiments was to determine the performance feasibility of using MPI4Py to move agents with mixed type attributes between processes, and to investigate the performance characteristics of the pure Python implementation. Representative benchmarked runtimes for a single rank are displayed in Table I. Given that there was little variation in runtimes (10ths of a second) between ranks only those for a single rank are shown.

According to these results, agent movement between processes is only about 20% of the total runtime with the remainder being the agent behavior, that is, the looping through the places on each process and having each agent play *tag* with each other agent in that place. Within the cross-process movement, MPI4Py's `alltoall` call is about a quarter of the total cross-process movement runtime. The remainder being spent in serialization and deserialization. Serialization consists of packing agent attributes into a tuple, and appending these tuples to the list to be sent. Deserialization consists of receiving this list of tuples from the `alltoall` call and creating new agents from them. Some further benchmarking revealed that about half of the deserialization time was spent creating the agents themselves via a constructor call. In previous work [44] we have discussed the cost of object creation and how caching agents on a process rather than recreating them effectively mitigates this cost. Based on these results, namely 1) the relatively small amount of runtime spent in MPI4Py, 2) its ease of use, effectively performing in a few lines of code the same operation as 40 or so lines of C++ code full of error prone displacement calculations and memory allocations, and 3) the known strategies for alleviating deserialization costs, we decided that at least as far as distributing and moving agents among processes, Python seemed to be an excellent candidate for implementing a distributed ABM toolkit.

Listing 1. Python *tag*

```
def run(self):
    for agent1 in self.agents:
        for agent2 in self.agents:
            if agent1 != agent2:
                agent1.tag(agent2)
```

A C++ implementation of the Simple Model was created using Repast HPC components where applicable and was used to provide a best case scenario with which the Python implementation could be compared. The C++ version of the Simple Model ran in slightly over 4 seconds, the majority of which was consumed by agent cross process movement, making the pure Python version approximately 10x slower. The Python *tag* code (Listing 1) is a simple nested loop in which all agents will be both the source and target of a *tag*. The *tag* itself simply increments the tag counter in each agent. In order to improve the performance of this part of the code, we reimplemented the *tag* loop using the Numba Python package [33] to dynamically compile the nested loop into native code. The loop was refactored from a method in the places class into a separate function and decorated with the appropriate Numba decorator: `@numba.jit(nopython=True)`. The Agent class was likewise decorated with a `numba.jitclass` decorator so that it could be referenced in this new function.

Using the Numba version of the code dropped the runtime of the place loop from approximately 31 seconds to approximately 8.5 seconds. This speedup was enough to convince us to invest in continued experimentation, moving from the exceedingly simple example model to one that required more typical ABM framework functionality.

B. The Zombies Model

The Zombies Model consists of human and zombie agents placed at random on a 2D grid and in a 2D continuous space. Each agent has a discrete integer coordinate (e.g., $(30, 22)$)

in the grid and a corresponding floating pointing coordinate (e.g., $(30.3323, 22.02343)$) in the continuous space. When an agent moves, the agent's locations are updated in both the grid and continuous space. For example, if a human moves from $(1.4, 2.2)$ to $(2.1, 2.4)$ in the continuous space, the human also moves from $(1, 2)$ to $(2, 2)$ in the grid. The grid and continuous space are implemented as shared projections (see III-D2) partitioned across all the processes. Consequently, each process is responsible for some portion of the global area. When a human or zombie moves out of its process' local area, it is moved to the process responsible for the area into which it has moved. Additionally, the grid and space also have a buffer zone that extends their local area by some user specified amount. This buffer is populated with non-local copies of the humans and zombies that occupy this buffer area on the adjacent processes. By using a shared projection, a human on the border of a process' local grid can *see* the nearby zombies even though those zombies reside on another process and, having seen the neighboring zombies, can then act accordingly. The agent population and cross-process synchronization is managed by a shared context (see III-D1) and a shared scheduler (see III-D3) is used to execute simulation events, driving the simulation forward.

The pseudocode for the model logic can be seen in Figure 2. The implementation of this required the development of: 1) a pan-process, synchronized and buffered 2D grid shared projection; 2) a pan-process, synchronized and buffered 2D continuous space shared projection; 3) spatial neighborhood query functionality and 4) the ability to move agents across processes. The initial implementation incorporated two optimizations. First, we used a quad tree to quickly find agents within the buffered area in the continuous space, thus speeding up the cross-process buffer synchronization and, second, we used NumPy to speed up the neighborhood queries in the shared grid. Two NumPy arrays define x and y offsets from a central point that, when added to that point, result in the neighboring coordinates of x and y. Additional NumPy operations eliminate any points that are outside the global bounds. Using NumPy arrays for this operation eliminates any expensive explicit loops. We also applied Numba's jit compilation to that code.

The baseline zombie model experiment consists of the model as described above, run with 8000 humans and 400 zombies spread across 4 processes and run for 50 iterations. In this scenario the Python implementation runs in approximately 23 seconds with some small (under a second) variation depending how the agents are initially located across the space and grid as determined by the random seed. For comparison, the C++ version of the model, an example model included with the Repast HPC distribution, runs in approximately 4 seconds with the caveat that while executing the same agent behavior, the C++ model is written in Repast HPC's ReLogo dialect. It does, however, illustrate how fast a roughly equivalent native code implementation can be expected to run, and is thus useful as a target comparison.

Some initial profiling on the distributed Zombies model

```
1: Given a population of zombies $Z$, and humans $H$, a continuous space $S$ and a grid $G$
2: for every iteration of the model do
3:     Synchronize $Z$ and $H$ across processes: any zombies or humans that moved out of the local bounds of $G$ and those of a neighboring process are moved to that process and removed from the originating process
4:     Update the buffer zone of $S$ from neighboring processes
5:     Update the buffer zone of $G$ from neighboring processes
6:     for each zombie $z \in Z$ do
7:         Find the grid location $g$ within $z$'s local Moore neighborhood $\in G$ with the most number of humans
8:         Move $z$ 0.25 units in the direction of $g$ to a new location $l$
9:         if $l$ contains any humans then
10:            Select a human $h$ at $l$ and infect $h$
11:        end if
12:    end for
13:    for each human $h \in H$ do
14:        if $h$ is infected then
15:            Increment time since infection $t_i$
16:            if $t_i$ is $> 10$ then
17:                Replace $h$ with a zombie
18:            end if
19:        end if
20:        if $h$ is not a zombie then
21:            Find the grid location $g$ within $h$'s local Moore neighborhood in $G$ with the fewest zombies
22:            if $g$ is not $h$'s current location then
23:                Move $h$ 0.5 units in the direction of $g$
24:            end if
25:        end if
26:    end for
27: end for
```

Fig. 2. Pseudo-code for Zombies Model

revealed that, as with the Simple Model, the time spent moving agents across processes was negligible when compared to the time spent executing the actual agent behavior, only about 2% $(0.378/19.84$ sec$)$ of the total runtime. Confident that the distributed aspects of the model were not performance hotspots, and to further investigate the agent behavior we profiled the model running on only a single process. Our profiling strategy was to use Python's cProfile package to determine which top-level functions or methods were the performance hot spots and then use Python's kernprof package [45] to drill down into the details of those calls. Scaling the runs to a single process (2000 humans, 100 zombies and 50 iteration) from 4 processes and using Python's cProfile module, we saw that 79% of the total runtime was being spent in executing the humans' behavior, namely, querying the space for the current location, querying the grid for the minimum number of zombies at each neighboring location, and moving to that location. Further profiling with the Python kernprof package revealed that of human behavior 40% of the time was spent in that movement around the grid and space. Furthermore, according to the cProfile benchmarking, movement by both humans and zombies was 41% of the total runtime. In comparison, the next highest fraction of the total runtime was retrieving the agents (zombies and humans) from a grid location at 15%. That movement would be a performance hotspot is not surprising, given that it is a crucial aspect of the agents' behavior. The movement code for an individual agent is called, in a representative model

```
 1: Given an agent $A$, a space $S$, and a point $P$ in $S$ to move $A$ to
 2: Transform $P$ according to the $S$'s boundary conditions (e.g., periodic) to $P_t$
 3: Remove $A$ from its current location
 4: if $P_t$ is outside the local bounds process bounds then
 5:     Add $A$ to the list of agents to move to neighboring processes
 6: else
 7:     Add $A$ to collection of agents at $P_t$
 8: end if
 9: if $S$ is a continuous space then
10:     Update $S$'s quad tree with $A$ and $P_t$
11: end if
```

Fig. 3. Pseudo-code for Movement in a Grid or Continuous Space

run, about 95,000 times, that is, 2100 agents moving roughly once every iteration for 50 iterations. Moreover, this amount of movement is not atypical in any ABM where movement in the environment is part of agent behavior. Movement then is an obvious candidate for optimization.

The pseudocode for agent movement, applicable to both a discrete grid and a continuous space is shown in Figure 3. To optimize this, we ultimately decided to implement the grid and continuous space projections in C++ using Python's C-API. (The implementation is discussed in more detail in section V-D1.) Several requirements made applying Numba to the Python implementation impractical. Both the grid and continuous space map agents to n-dimensional points and, equally importantly, n-dimensional points to agents. Given an agent, a modeler can query for its location, and given a location, they can query for all the agents at that location. Consequently, movement in a grid and space is not numerically orientated, but is rather composed of dictionary and list operations, updating the mapping between an agent and a location and vice versa. While it was possible to apply Numba to much of the n-dimensional point related computations, especially as the points themselves are implemented as NumPy arrays, that left much of the remaining mapping related code unoptimized. Given our experience in C++ development with Repast HPC, a native code implementation using Python's C-API was a natural option. Finally, as this code would be hidden from users behind a Python API, it would not detract from the usability of the Python toolkit.

The Zombies model was updated to use these new native code implementations of the discrete grid and continuous space and again profiled using cProfile and kernprof. The total runtime dropped from 19.3 seconds to 6.7 seconds. The human agent's behavior again accounted for most of that time (4.1 seconds or 61%). However, of that behavior, movement now consumed only 9% of the runtime, down from 40% in the Python implementation.

V. Repast4Py Implementation

Informed by the profiling activities described above and the requirements for a distributed ABM toolkit outlined in Section III-D, our initial Repast4Py implementation is a mix of Python and native code, and utilizes the NumPy, Numba, and PyTorch packages. The remainder of this section presents the initial implementation of the core components.

A. Agents & Agent Identity

Every agent in a Repast4Py model subclasses a `R4Py_Agent` Python Object. This object has a single member, a `R4Py_AgentId` that uniquely identifies that agent. The id is primarily used to map an agent to any arbitrary data as required by the toolkit components' implementation. For example, the id is used to map agents to the locations in a grid and continuous space as well as for some caching operations when agents are transferred between processes. The `R4Py_AgentId` itself is a native code Python Object consisting of two integers and a long. The first integer specifies the type of agent (where type is some simulation specific type, e.g., zombie or human) and the second integer is the process rank on which the agent with this id was created. The long is used to distinguish between agents of the same type created on the same process. Taken together these three values should uniquely identify an agent across the entire simulation.

B. Shared Context

The context component is a simple container, based on set semantics, that encapsulates an agent population, and is implemented in Python. The equivalence of elements is determined by the agent id. Each process has at least one context to which local and non-local agents are added and removed. The context component provides iterator implementations that can be used to iterate over all the agents in the context or only those of a specific type. Projections are also contained by the context such that any agent added to a context becomes a member of the associated projections (occupying a grid cell, for example). Cross-process synchronization is also managed via the context, during which agents are moved between contexts and the projections that those contexts contain are updated with the new agents.

C. Shared Scheduler

Following RepastHPC [31], Repast4Py uses a distributed discrete event scheduler that is tightly synchronized across processes. In this, it has much in common with conservative PDES synchronization algorithms such as [22], [23]. It is implemented in Python, using Python container classes and operations. Our experience is that an ABM scheduler, rather than having to order and execute many thousands of events each iteration of the model, typically contains at the most perhaps a few hundred distinct repeated events, and often much fewer than that. Consequently, it was deemed unlikely that a pure Python implementation would become a bottleneck. The events themselves are Python functions or classes that implement a functor type interface via Python's `__call__` method, together with some data specifying when the event should occur next.

The scheduler runs in a loop until a simulation stop event is executed. At each iteration of the loop the scheduler determines the next tick that should be executed, pops the events for that tick off of a priority queue, and executes the function or functor for those events. The scheduler synchronizes across processes when it determines the next tick to execute. Each individual scheduler on each process passes the next tick it will execute to an all_reduce call and receives the global minimum next tick as a result. If this global minimum is equal to the local next tick, then the local scheduler executes. Synchronization is thus two-fold: 1) the all_reduce call is a barrier preventing individual processes from getting too far ahead or too far behind; 2) by only executing the global minimum tick, we ensure that all processes are executing the same tick. This type of synchronization is, of course, restrictive and implies that each process is under a similar load and that the coherency of the simulation as a whole benefits from such synchronization. However, this does fit reasonably well with many types of currently implemented agent simulations.

D. *Shared Projections*

Repast4Py currently provides two projections: a grid and a continuous space (a network projection is planned as future work). Additionally a value layer, which while not imposing a structure directly on the agents, does provide a way for agents to query and interact with an underlying, gridded scalar value field. Each of these is implemented according to share projection semantics (see III-D2). If the agents require no spatial or network relationship with each other, then none of these projections need to be created and none of the concomitant computational costs, such as synchronization, are paid. Only the actual projections used need to be created, and only the costs associated with the created projections need to be paid.

1) Shared Grid & Shared Continuous Space: The shared grid projection implements a cross-process discrete grid where agents are located in a matrix type space. The shared continuous space implements a cross-process continuous space where an agent's location is expressible as floating point coordinates. Both implement typical ABM grid / space functionality: moving agents around the grid or space, retrieving grid or space occupants at particular locations, getting neighbors within a certain extent, periodic and non-periodic borders, and so forth. Both are distributed over all processes, and each process is responsible for a particular subsection of the larger area. The agents in a particular subsection are local to that process. Grids and spaces are typically used to define an interaction topology between agents such that agents in neighboring cells or locations interact with each other. In order to accommodate this usage, a buffer can be specified during grid and space creation. The buffer (see Figure 4) will contain non-process-local neighbors, that is, agents in neighboring subsections. Grid and space creation leverages MPI_Cart_create so that processes managing neighboring subsections can communicate efficiently.

The shared grid and shared continuous space also allow agents to move out of one subsection and into another. This entails migrating the agent from one process to another such that the migrated agent is now local to the destination process. The inter-process agent movement is achieved through a shared context (V-B).

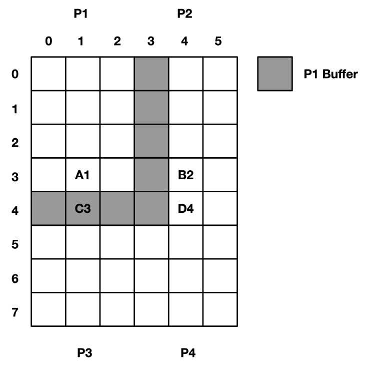

Fig. 4. Shared Grid Buffer. The grid is equally divided among four processes: P1 in the upper left, P2 in the upper right, P3 in the lower left and P4 in lower right. The highlighted section indicates those parts of the grid running on processes P2 and P3 that are shared with P1 as P1's buffer. Equivalent sections form the buffers for the other processes. Agent C3 is local to P3, and also a non-local agent member of P1's buffer. The same applies to agents D4 and B2 for P2 and P4.

As mentioned above (IV-B), profiling revealed that agent movement within a pure Python grid and shared continuous space was a bottleneck, but was significantly improved in the native code implementation. At its root, the native code implementation begins with a templated C++ `BaseSpace` class that provides the core add, remove, move, and query functionality. (With respect to this `BaseSpace`, the grid has a discrete point type as a template argument and the continuous space has a continuous point type.) This `BaseSpace` class is wrapped as a member field by a `DistributedSpace` class that adds the cross-process distributed functionality, delegating the core functionality to a `BaseSpace`. The `DistributedSpace` class uses a cartesian space communicator to determine the process rank of its neighbors in the pan-process global space, and manages the data necessary for 1) moving agents from one process to another when an agent crosses out of its local bounds; and 2) synchronizing the buffer between neighboring subsections of the space. Specialized with the appropriate template arguments, the `DistributedSpace` types are exposed to Python as Python C-API objects. The Python classes `SharedGrid` and `SharedCSpace` are implemented as subclasses of those objects. In concert with the `SharedContext`, these Python classes use the agent movement and buffer data managed by their C++ parent classes to synchronize agents across processes using MPI4Py.

2) Shared Value Layer: A value layer is essentially an array or matrix of numeric values, where the value at each location can be retrieved or set via an index coordinate. A shared value layer is such a matrix but distributed across processes, and as in the shared grid and continuous space, each process is responsible for a particular subsection of the larger matrix. Also in common with the shared grid and continuous space, a shared value layer has an optional buffer and buffer synchronization, a requirement for any operation (e.g., diffusion or neighborhood queries) that require access to the values in neighboring subsections.

The value layer implementation begins with a pure Python `ValueLayer` class that uses a PyTorch tensor to store its values. It implements the typical value layer methods for getting and setting values, slice operations, neighborhood queries, and support for non-zero matrix origins via NumPy array based coordinate translation. This latter essentially translates from model space coordinates to that of the PyTorch tensor. Given that value layer operations are for the most part numeric it is possible to use Numba here, although most of the speed comes from using a PyTorch tensor, rather than something sub-optimal, such as a list of lists to represent the matrix. Informed by our work with Numba and NumPy arrays for neighborhood queries in the Zombies model (IV-B), we applied Numba to the neighborhood query code: the value layer neighborhood queries consist of a set of Numba compiled functions that use NumPy arrays to compute slice indices into the tensor.

The `SharedValueLayer` is also a pure Python class that extends the `ValueLayer`, adding support for the buffer and buffer synchronization using a cartesian topology communicator to determine neighboring ranks. The buffer synchronization uses MPI4Py's optimized buffer object support to share the buffered values. Each process' value tensor is sliced appropriately, the resulting tensors are converted to NumPy arrays, and then sent and received via an MPI4Py `Alltoall` call. Where possible, the results of the slicing and conversion are *views* of the underlying value tensor, and thus avoid the overhead cost of, for example, memory allocation.

When used in an ABM, a single value layer is rarely used alone. Typically, agents will read from a *read* layer and write to a *write* layer with perhaps an additional operation (e.g., diffusion) updating the write layer from the read layer. The two value layers are then swapped at the end of a time step. In this way, the value layer appears the same to all agents, avoiding any artifacts produced by the order of agent iteration. The `ReadWriteValueLayer` encapsulates this functionality in a single class. It contains two `SharedValueLayers` such that read operations use the read layer, and write operations use the write layer. The two can then be swapped. It also includes support for applying a function to the read layer to update the write layer.

VI. Performance and Scaling

In this section, we present a CPU vs. GPU performance comparison of the `ReadWriteValueLayer` and scaling experiments using our benchmark Zombies model. The `ReadWriteValueLayer` experiments were performed on the University of Chicago's Research Computing Center's Midway2 cluster. Midway2's GPU partition contains 6 nodes, each of which contains two Intel E5-2680 v4 CPUs and 2 Nvidia K80 GPUs and 64 GB of RAM. The scaling experiments were performed on Bebop, an HPC cluster managed by the Laboratory Computing Resource Center at Argonne National Laboratory. Bebop has 1024 nodes comprised of 672 Intel Broadwell processors with 36 cores per node and 128 GB of RAM and 372 Intel Knights Landing processors with 64 cores per node and 96 GB of RAM.

TABLE II
ReadWriteValueLayer Diffusion Runtimes

Size	Device	Runtime (sec.)	Improvement
1252x1252	CPU	3.42	1x
1252x1252	GPU	0.13	26.3x
2504x2504	CPU	21.8	1x
2504x2504	GPU	0.48	45.4x
5008x5008	CPU	90.78	1x
5008x5008	GPU	1.88	48.2x
10016x10016	CPU	376.39	1x
10016x10016	GPU	7.50	50.84x

For the `ReadWriteValueLayer` performance comparisons, we implemented a diffusion algorithm to be run over a non-periodic 2D grid and applied that to the `ReadWriteValueLayer`. We ran grid sizes of 10016x10016, 5008x5008, 2504x2504, and 1252x1252 on a single process for 50 iterations. The comparative CPU vs. GPU performance as measured by the mean runtime over 20 runs can be seen in Table II.

As expected, the GPU performance is much better than the CPU performance, and improving as the grid size gets larger. The GPU performance itself scales linearly with respect to the grid size. As the size decreases by 4x, the GPU is performance is roughly 4x as fast.

To examine how the `ReadWriteValueLayer` performs in an ABM, we incorporated it into the Zombies model. During each iteration, every human now emits a unit of pheromone into the value layer at their grid locations. This pheromone is then diffused through the value layer. Rather than query their local neighborhood for humans, zombies now look for the greatest concentration of pheromones in their local neighborhood by querying the value layer, and then move toward that direction. This model was run with a non-periodic grid size of 2504 x 2504, 30,000 humans and 1,000 zombies for 100 iterations under a CPU, GPU and Hybrid scenario.

In the model, the value layer PyTorch tensor matrices are accessed in three separate ways. The first is via the diffusion operation during which the entire read and write layers are used. The second is by the humans when they deposit their unit of pheromone into a single location in the matrix. The third is by the zombies when they query the matrix around their local neighborhood for the maximum value. In the CPU scenario all the matrix operations are performed on the CPU, while the GPU scenario performs them on the GPU. The Hybrid approach performs only the diffusion on the GPU and the remaining operations on the CPU. The mean runtimes for each scenario over 10 runs, varying the random seed each run, can be seen in Table III.

The GPU scenario is the slowest, followed by the CPU and then the Hybrid. While the GPU is certainly the fastest at performing the diffusion operations, the thousands of simple get and put access operations required for the humans and zombies behavior during each iteration involve time consuming data

TABLE III
DIFFUSION ZOMBIES RUNTIMES

Scenario	Runtime (sec.)
CPU	387.66
GPU	589.66
Hybrid	331.57

TABLE IV
ZOMBIES MODEL STRONG SCALING

Process Count	Runtime (sec.)	Speedup	Parallel Efficiency
36	826.96	1x	1
72	405.98	2.04x	1.02
144	201.71	4.09x	1.02
288	112.98	7.32x	0.92

transfers from the GPU to the CPU. This overhead results in the GPU scenario being slower than the CPU scenario. The Hybrid scenario eliminates these transfers by performing the human and zombie matrix access on the CPU. The data transfer overhead is not eliminated entirely though. The tensor matrices are moved to the GPU before the diffusion operation, and then moved back to the CPU prior to executing the human and zombie behaviors. Future work will explore patterns for more optimally mixing the hybrid styles of value layer access under different types of agent and value layer interactions.

To test the strong scaling performance of Repast4Py, we ran the original Zombies model (without the diffusion component) with a total human count of 3,000,000, a total zombie count of 6,000, and global grid and space dimensions of 1008 x 1008 using process counts of 36 (corresponding to 1 node), 72 (2 nodes), 144 (4 nodes), and 288 (8 nodes). As the process count doubles, the number of agents on each process is halved, and the width and height of each process-local space and grid are adjusted by about $\sqrt{2}$ each. The runtime was measured as the mean over 30 runs for each process count, varying the random seed for each of the 30 runs. The runs themselves were performed on Bebop using our EMEWS [8] framework. Here we exploited the ability for EMEWS, through the underlying Swift/T [46] workflow engine, to launch a concurrent set of parallel MPI tasks within an allocated pilot job MPI communicator. The results can be seen in Table IV and Figure 5.

The Repast4Py components and the Zombies model scale well through 144 processes, and then begin to show a drop off after. As the process count doubles, we see speedups of 2.04x (36 processes to 72), 2.01x (72 to 144) and 1.79x (144 to 288). At 288 processes, the cost of inter-process communication begins to outweigh the performance advantage of executing fewer agents on each process. This trade off is obviously model dependent. Larger agent populations or more computationally complex agents should continue to scale well at higher numbers of processes. An important part of ABM development is experimenting with various process counts in order to determine the 'sweet spot' that enables the ABM to be effectively used in sequential computational experiments.

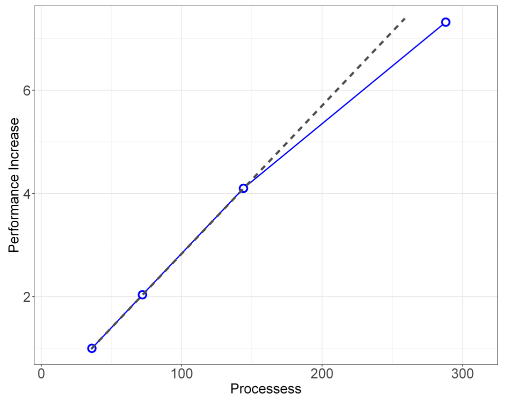

Fig. 5. Zombies Model Strong Scaling.

VII. CONCLUSION AND FUTURE WORK

In this article we have presented our experiences in developing a distributed ABM toolkit with Python. We first provided an overview of existing distributed ABM toolkits and presented the characteristics we believe are needed for creating a widely used and performant tool that can help promote large-scale ABM methods to a wider scientific community. Then, relying on our long history of developing the Repast ABM toolkits, we identified a list of the basic components needed in a distributed ABM toolkit. We proceeded to describe the process by which we determined the feasibility of a Python-based toolkit. We iteratively profiled various Python implementations of key distributed ABM functionality, identified performance hotspots, and found solutions to improve efficiency.

We presented the initial implementation of the core components of Repast4Py. The implementation combines pure Python, Python C-API native code, and uses optimized Python packages. We leveraged Numba and NumPy for optimized and just-in-time compilation of numeric operations, and PyTorch for GPU computation. Our primary concern is with creating a useful and usable working system that can serve as the foundation for developing ABMs of complex systems and enabling scientific discovery through computational experimentation. We wanted to ensure scalable performance that exploits the increasing concurrency and hybrid aspects of HPC resources. To this end, we demonstrated a performant hybrid GPU/CPU approach and good strong scaling. Future work will focus on implementing additional toolkit features, such as a shared network projection, profiling and tuning hybrid GPU/CPU approaches to general ABM functionality (for example, by exploring unified memory support), and providing tutorials and documentation to accompany an initial public release.

ACKNOWLEDGMENT

This research was completed with resources provided by the Research Computing Center at the University of Chicago (Midway2 cluster) and the Laboratory Computing Resource Center at Argonne National Laboratory (Bebop cluster).

References

[1] C. M. Macal, "Everything you need to know about agent-based modelling and simulation," *Journal of Simulation; Abingdon*, vol. 10, no. 2, pp. 144–156, May 2016. [Online]. Available: http://search.proquest.com/docview/1787875572/abstract/BA7FA1B56BE34DC7PQ/1

[2] C. M. Macal, N. T. Collier, J. Ozik, E. R. Tatara, and J. T. Murphy, "Chisim: An agent-based simulation model of social interactions in a large urban area," in *2018 Winter Simulation Conference (WSC)*, 2018, pp. 810–820.

[3] J. Ozik, N. T. Collier, J. M. Wozniak, C. M. Macal, and G. An, "Extreme-scale dynamic exploration of a distributed agent-based model with the emews framework," *IEEE Transactions on Computational Social Systems*, vol. 5, no. 3, pp. 884–895, 2018.

[4] C. Kaligotla, J. Ozik, N. Collier, C. M. Macal, S. Lindau, E. Abramsohn, and E. Huang, "Modeling an Information-Based Community Health Intervention on the South Side of Chicago," in *2018 Winter Simulation Conference (WSC)*, Dec. 2018, pp. 2600–2611.

[5] J. Ozik, N. Collier, J. M. Wozniak, C. Macal, C. Cockrell, S. H. Friedman, A. Ghaffarizadeh, R. Heiland, G. An, and P. Macklin, "High-throughput cancer hypothesis testing with an integrated PhysiCell-EMEWS workflow," *BMC Bioinformatics*, vol. 19, no. 18, p. 483, Dec. 2018. [Online]. Available: https://doi.org/10.1186/s12859-018-2510-x

[6] J. Ozik, N. Collier, R. Heiland, G. An, and P. Macklin, "Learning-accelerated discovery of immune-tumour interactions," *Molecular Systems Design & Engineering*, vol. 4, no. 4, pp. 747–760, 2019. [Online]. Available: https://pubs.rsc.org/en/content/articlelanding/2019/me/c9me00036d

[7] DOE Office of Science Advanced Scientific Computing Research, "Doe national laboratory program announcement: Artificial intelligence and decision support for complex systems," https://science.osti.gov/-/media/grants/pdf/lab-announcements/2020/LAB_20-2321.pdf, 2020, [Online; accessed 11-September-2020].

[8] J. Ozik, N. T. Collier, J. M. Wozniak, and C. Spagnuolo, "From desktop to large-scale model exploration with swift/t," in *Proceedings of the 2016 Winter Simulation Conference*, ser. WSC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 206–220. [Online]. Available: http://dl.acm.org/citation.cfm?id=3042094.3042132

[9] A. Bershteyn, J. Gerardin, D. Bridenbecker, C. W. Lorton, J. Bloedow, R. S. Baker, G. Chabot-Couture, Y. Chen, T. Fischle, K. Frey, J. S. Gauld, H. Hu, A. S. Izzo, D. J. Klein, D. Lukacevic, K. A. McCarthy, J. C. Miller, A. L. Ouedraogo, T. A. Perkins, J. Steinkraus, Q. A. ten Bosch, H.-F. Ting, S. Titova, B. G. Wagner, P. A. Welkhoff, E. A. Wenger, C. N. Wiswell, and f. t. I. f. D. Modeling, "Implementation and applications of EMOD, an individual-based multi-disease modeling platform," *Pathogens and Disease*, vol. 76, no. 5, Jul. 2018, publisher: Oxford Academic. [Online]. Available: http://academic.oup.com/femspd/article/76/5/fty059/5050059

[10] J. J. Grefenstette, S. T. Brown, R. Rosenfeld, J. DePasse, N. T. Stone, P. C. Cooley, W. D. Wheaton, A. Fyshe, D. D. Galloway, A. Sriram, H. Guclu, T. Abraham, and D. S. Burke, "FRED (A Framework for Reconstructing Epidemic Dynamics): an open-source software system for modeling infectious diseases and control strategies using census-based populations," *BMC Public Health*, vol. 13, p. 940, 2013. [Online]. Available: http://dx.doi.org/10.1186/1471-2458-13-940

[11] G. Cordasco, R. D. Chiara, A. Mancuso, D. Mazzeo, V. Scarano, and C. Spagnuolo, "Bringing together efficiency and effectiveness in distributed simulations: The experience with d-mason," *SIMULATION*, vol. 89, no. 10, pp. 1236–1253, 2013. [Online]. Available: https://doi.org/10.1177/0037549713489594

[12] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan, "Mason: A multiagent simulation environment," *SIMULATION*, vol. 81, no. 7, pp. 517–527, 2005. [Online]. Available: https://doi.org/10.1177/0037549705058073

[13] G. Cordasco, A. Mancuso, F. Milone, and C. Spagnuolo, "Communication strategies in distributed agent-based simulations: The experience with d-mason," in *Euro-Par 2013: Parallel Processing Workshops*, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 533–543.

[14] X. Rubio-Campillo, "Pandora: A versatile agent-based modelling platform for social simulation," in *SIMUL 2014, The Sixth International Conference on Advances in System Simulation*, 2014, pp. 29–34.

[15] F. Borges, A. Gutierrez-Milla, E. Luque, and R. Suppi, "Care hps: A high performance simulation tool for parallel and distributed agent-based modeling," *Future Generation Computer Systems*, vol. 68, pp. 59 – 73, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16302758

[16] S. Chin, D. Worth, C. Greenough, S. Coakley, M. Holocombe, and M. Gheorghe, "Flame-ii: a redesign of the flexible large-scale agent-based modelling environment," Science and Technology Facilities Council, Tech. Rep. RAL-TR-2012-019, November 2012.

[17] P. Richmond and M. K. Chimeh, "Flame gpu: Complex system simulation framework," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017, pp. 11–17.

[18] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[19] ——, *Parallel and Distributed Simulation Systems*. Wiley Interscience, 2000.

[20] K. S. Perumulla, "Scaling Time Warp-based Discrete Event Execution to 10^4 Processors on a Blue Gene Supercomputer," in *Proceedings of the 4th International Conference on Computing Frontiers*, 2007, pp. 69–76.

[21] ——, "Parallel and Distributed Simulation: Traditional Techniques and Recent Advances," in *Proceedings of the Winter Simulation Conference (WSC)*, 2006.

[22] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, vol. 24, no. 4, pp. 198–205, 1981.

[23] ——, "Distributed Simulation: A Case-study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE, no. 5, pp. 440–452, 1979.

[24] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time Warp Operating System," *ACM SIGOPS Operating Systems Review*, vol. 21, no. 5, pp. 77–93, 1987.

[25] D. R. Jefferson, "Virtual Time," *ACM Transactions of Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[26] K. S. Perumulla, "μsik - A Micro-kernel for Parallel/Distributed Simulation Systems," in *Workshop on Principles of Advanced and Distributed Simulations*, 2005, pp. 59–68.

[27] V. Jha and R. Bagrodia, "A Unified Framework for Conservative and Optimistic Distributed Simulation," *ACM SIGSIM Simulation Digest*, vol. 24, no. 1, pp. 12–19, 1994.

[28] R. L. Bagrodia and W. Liao, "Parallel Simulation of the Sharks World Problem," in *Proceedings of the 22nd Winter Simulation Conference*, 1990, pp. 191–198.

[29] M. T. Presley, P. L. Reiher, and S. F. Bellenot, "A Time Warp Implementation of the Sharks World," in *Proceedings of the 22nd Winter Simulation Conference*, 1990.

[30] D. M. Nicol and S. E. Riffe, "A 'Conservative' Approach to Parallelizing the Sharks World Simulation," in *Proceedings of the 22nd Winter Simulation Conference*, 1990.

[31] N. Collier and M. North, "Parallel agent-based simulation with Repast for High Performance Computing," *SIMULATION*, Nov. 2012.

[32] T. Oliphant, "NumPy: A guide to NumPy," USA: Trelgol Publishing, 2006–, [Online; accessed ¡today¿]. [Online]. Available: http://www.numpy.org/

[33] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[35] N. Collier, T. R. Howe, and M. J. North, “Onward and Upward: The Transition to RePast 2.0,” *First Annual North American Association for Computational Social and Organizational Science Conference*, 2003.

[36] M. J. North, N. T. Collier, and J. R. Vos, “Experiences creating three implementations of the repast agent modeling toolkit,” *ACM Transactions on Modeling and Computer Simulation*, vol. 16, no. 1, pp. 1–25, Jan. 2006. [Online]. Available: https://doi.org/10.1145/1122012.1122013

[37] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, “Complex adaptive systems modeling with Repast Simphony,” *Complex Adaptive Systems Modeling*, vol. 1, no. 1, p. 3, Mar. 2013.

[38] J. Ozik, N. Collier, T. Combs, C. M. Macal, and M. North, “Repast Simphony Statecharts,” *Journal of Artificial Societies and Social Simulation*, vol. 18, no. 3, p. 11, 2015. [Online]. Available: http://jasss.soc.surrey.ac.uk/18/3/11.html

[39] J. Ozik, N. T. Collier, J. T. Murphy, and M. J. North, “The ReLogo agent-based modeling language,” in *2013 Winter Simulations Conference (WSC)*, Dec. 2013, pp. 1560–1568, iSSN: 0891-7736, 1558-4305.

[40] C. M. Macal, M. J. North, N. Collier, V. M. Dukic, D. T. Wegener, M. Z. David, R. S. Daum, P. Schumm, J. A. Evans, J. R. Wilder, L. G. Miller, S. J. Eells, and D. S. Lauderdale, “Modeling the transmission of community-associated methicillin-resistant Staphylococcus aureus: a dynamic agent-based simulation,” *Journal of Translational Medicine*, vol. 12, no. 1, p. 124, May 2014.

[41] B. H. Park, H. M. Abdul Aziz, A. Morton, and R. Stewart, “High performance data driven agent-based modeling framework for simulation of commute mode choices in metropolitan area,” in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*, 2018.

[42] L. Dalcin, R. Paz, M. Storti, and J. D’Elia, “Mpi for python: Performance improvements and mpi-2 extensions,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 655–662, 2008.

[43] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof, C. G. Cardona, B. V. Essen, and M. Baughman, “Candle/supervisor: a workflow framework for machine learning applied to cancer research,” *BMC Bioinformatics*, vol. 19, no. 18, p. 491, Dec 2018. [Online]. Available: https://doi.org/10.1186/s12859-018-2508-4

[44] N. T. 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 *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, 2015, pp. 454–465. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27308-2_37

[45] Robert Kern, “Kernprof line profiler,” 2020, https://github.com/pyutils/line_profiler, Last accessed on 8 September 2020.

[46] 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 *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 95–102.