Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds.

DISTRIBUTED AGENT-BASED SIMULATION WITH REPAST4PY

Nicholson Collier
Jonathan Ozik

Decision and Infrastructure Sciences
Argonne National Laboratory
9700 S. Cass Avenue
Lemont, IL 60439, USA

ABSTRACT

The increasing availability of high-performance computing (HPC) has accelerated the potential for applying computational simulation to capture ever more granular features of large, complex systems. This tutorial presents Repast4Py, the newest member of the Repast Suite of agent-based modeling toolkits. Repast4Py is a Python agent-based modeling framework that provides the ability to build large, MPI-distributed agent-based models (ABM) that span multiple processing cores. Simplifying the process of constructing large-scale ABMs, Repast4Py is designed to provide an easier on-ramp for researchers from diverse scientific communities to apply distributed ABM methods. We will present key Repast4Py components and how they are combined to create distributed simulations of different types, building on three example models that implement seven common distributed ABM use cases. We seek to illustrate the relationship between model structure and performance considerations, providing guidance on how to leverage Repast4Py features to develop well designed and performant distributed ABMs.

1 INTRODUCTION

Agent-based models (ABMs) are being applied to increasingly detailed complex systems of societal importance. Leveraging high-performance computing (HPC), distributed ABMs can be used to model large numbers of agents and their complex behaviors and interactions. This tutorial will provide an introduction to Repast4Py, an agent-based modeling framework written in Python that provides the ability to build large, MPI-distributed ABMs that span multiple processing cores. While distributed ABMs enable the capturing of unprecedented details in large-scale complex systems, the specialized knowledge required for developing such ABMs creates barriers to wider adoption and utilization. Repast4Py seeks to improve on existing distributed ABM software by simplifying the process of constructing large-scale ABMs. It is designed to provide an easier on-ramp for researchers from diverse scientific communities to apply distributed ABM methods. A review of previous distributed ABM software can be found in Collier et al. (2020). Repast4Py is released under the BSD-3 license, and leverages the MPI for Python (Dalcin and Fang 2021), Numba (Lam et al. 2015), NumPy (Harris et al. 2020), and PyTorch (Paszke et al. 2019) packages, and the Python C API to create a scalable modeling system that can exploit the largest HPC resources and emerging computing architectures.

Repast4Py is the newest member of the Repast suite. Originally developed in 2001, the first Repast ABM toolkit (Collier et al. 2003) was written in Java and targeted typical desktop machines. Subsequent iterations brought a .NET C# port, an early experimental (and popular) Python-based variant, and a large user base (North et al. 2006). From 2012 the Repast ABM Suite has consisted of two toolkits, Repast Simphony (North et al. 2013) and Repast for High Performance Computing (Repast HPC) (Collier and North 2013). Repast Simphony is a Java-based platform that provides multiple methods for specifying

agent-based models, including pure Java, Statecharts (Ozik et al. 2015) and ReLogo (Ozik et al. 2013), a dialect of Logo. 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++. Repast4Py builds on the lessons learned while implementing Repast HPC, the Chicago Social Interaction Model (ChiSIM) Repast HPC framework (Macal et al. 2018), and models built with Repast HPC (Collier et al. 2015; Khanna et al. 2019; Kaligotla et al. 2020; Ozik et al. 2018; Ozik et al. 2021; Lindau et al. 2021; Tatara et al. 2022).

The remainder of this paper is organized as follows. In Section 2, we describe the components of the Repast4Py ABM toolkit. Section 3 presents the canonical structure of a Repast4Py model, showing how the toolkit components come together. Section 4 describes how Repast4Py implements synchronization of distributed model structures. Section 5 provides guidance on how model performance can be assessed and improved. Section 6 presents three example models and the seven distributed ABM use cases they encapsulate. Section 7 provides illustrative scaling studies. We summarize our contributions and discuss future work directions in Section 8.

2 COMPONENTS

The Repast Simphony 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 (North et al. 2013). 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. Repast4Py, like Repast HPC, adapts these principles and components to the distributed computing context where an ABM's agents and environment are partitioned among multiple compute processes running in parallel and the framework maintains a coherent unified simulation from these separate processes. Unlike Repast HPC, Repast4Py is designed to provide an easier on-ramp for researchers to apply distributed ABM methods. The Repast4Py components are of three types: 1) agents and their environments; 2) logging and data collection; and 3) simulation utility functions (e.g., random number management, input parameter parsing, and so on). We explore these components and their implementations next. Many of the class names of these implementations are prefixed with "*Shared*", and by *shared* we want to emphasize that the global simulation is shared among a pool of processes, each of which is responsible for some portion of it. Instances of these *Shared* classes running on separate processes cooperate to create the unified but distributed simulation. Further details on the components described in this section can be found in the Repast4Py API documentation (Repast Project 2022a).

2.1 Agents

In a distributed Repast4Py simulation, an agent resides on a compute process and is said to be *local* to that process. Each process maintains a collection of agents and is responsible for executing the code that defines the agents' behavior, for example, by iterating over all the agents on that process and calling a method on each. However, in order to maintain a unified model, a process may contain non-local or *ghost* agents copied from another process with which local agents may interact. Agents are implemented as Python classes that must subclass `repast4py.core.Agent`, which is implemented using the Python C-API in the `R4Py_Agent` Python Object. This object has two fields, an agent id (`R4Py_AgentId`) field that uniquely identifies an agent and an integer field identifying the agent's current local process. It also includes functions for accessing the agent id and its elements (e.g., the `uid` property that returns the agent's unique id as a tuple). `R4Py_AgentId` is a native code Python Object consisting of two integers and a long. The first integer specifies the type of agent (where type is a simulation specific type) and the second integer is the process 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 can uniquely identify an agent

```
def save(self) -> Tuple:
    return (self.uid, self.meet_count, self.pt.coordinates)
```

Figure 1: An example agent's save method.

across the entire simulation. We use the C-API to allow other Repast4Py components implemented using the Python C-API to efficiently access an agent id, when it is used as a key in a C++ map, for example.

In addition to subclassing `repast4py.core.Agent`, the user must implement a `save` method that returns the state of the agent as a tuple. The first element of this tuple must be the agent id. The remaining elements should encapsulate any dynamic agent state. An example of this method as implemented in the *Walker* agent from the Random Walk demonstration model (Section 6.1) can be seen in Figure 1. Here, we return a tuple, consisting of the agent's id (`self.uid`), and the two elements that make up a *Walker* agent's state: `meet_count`, and `pt.coordinates`. The `save` method is complemented by a `restore` function that deserializes the saved state to create an agent on another processes. The `save` method and `restore` function are always required, but certain agent environments, such as a network, also require an agent `update` method (see Section 2.3.3). We will discuss the save, restore, and update API in more detail in Section 4.

2.2 Simulation Event Scheduling

Events in Repast simulations, such as when an agent or agents execute their behavior, are driven by a discrete-event scheduler. The events are scheduled to occur at a particular *tick*. Ticks do not represent clock-time but rather the priority of its associated event. 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 future ticks.

Following Repast HPC (Collier and North 2013), Repast4Py uses a distributed discrete event scheduler that is tightly synchronized across processes. It is implemented in Python, using Python container classes and operations. The events are Python functions or classes that implement a functor type interface via Python's `__call__` method, together with data specifying when the event should occur next. Scheduling is implemented in the `repast4py.schedule` module whose primary user-facing class is the `SharedScheduleRunner`, which provides methods for scheduling different types of events (one-time, repeating, and at simulation end) with different priorities. Additional functions in the module allow the user to initialize and easily access a global `SharedScheduleRunner` and create events from a function and its arguments. Additional details of the `SharedScheduleRunner` implementation can be found in Collier et al. (2020).

2.3 Contexts and Projections

Contexts and *projections* first appeared in Repast Simphony (North et al. 2013) as a way to apply multiple agent environments to a population of agents. A Repast4Py *context* is a simple container, based on set semantics, that encapsulates an agent population, and is implemented in pure Python. The equivalence of elements is determined by the agent id. The *SharedContext* is a context implementation that combines container semantics with methods to synchronize and manage both local and ghost agents.

A context does not inherently provide any relationship or structure for its population of agents. 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 grid-type space, or a continuous space where an agent's location is expressible as a non-discrete coordinate (North et al. 2013). 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. Shared Projections are projections that have been partitioned over multiple processes and cooperate with each other and the *SharedContext* to create a unified agent environment. Table 1 displays the projections provided by Repast4Py. Further details on the implementation of *SharedContext*, *SharedGrid*, *SharedCSpace*, and *SharedValueLayer* can be found in Collier et al. (2020).

Table 1: Repast4Py's shared projections.

Description	Name	Dimensions
Grid type space with discrete coordinates	space.SharedGrid	1D/2D/3D
Continuous space with non-discrete coordinates	space.SharedCSpace	1D/2D/3D
Gridded scalar value field	value_layer.SharedValueLayer	1D/2D
Directed network in which agents are nodes	network.DirectedSharedNetwork	
Undirected network in which agents are nodes	network.UndirectedSharedNetwork	

2.3.1 SharedGrid and SharedCSpace

The `SharedGrid` and the `SharedCSpace` classes implement spaces in which an agent's location is expressible as, respectively, discrete integer or continuous 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 such that each process is responsible for a subsection of the larger area. These subsections are stitched together through the use of a *buffer*, an area of a process subsection duplicated in its process neighbors such that agents in that process can *see* and interact with agents in the neighboring subsection.

2.3.2 SharedValueLayer

A value layer is essentially a grid 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 grid but distributed across processes, each process is responsible for a particular subsection of the larger grid. Also in common with the shared grid and continuous space, a shared value layer has an optional buffer and buffer synchronization mechanism, requirements for any operation (e.g., diffusion or neighborhood queries) that require access to the values in neighboring subsections. The `SharedValueLayer` uses a PyTorch (Paszke et al. 2019) tensor to store its values.

2.3.3 SharedNetwork

The `SharedNetwork` classes implement directed and undirected networks in which agents are the nodes in the network and can form edges with each other. These classes provide typical network functionality such as adding and removing nodes and edges, updating edge attributes, and finding network neighbors. As with the previously discussed projections, each `SharedNetwork` instance running on its own process is responsible for only part of the network containing the nodes (i.e., the agents) that are local to that process. Edges between processes are created using copies (*ghosts*) of the non-local nodes participating in the edges, as illustrated in Figure 2. Here, the global pan-process network contains an edge from node *A1* on process *P1* to node *B2* on process *P2*. In order for *A1* and *B2* to interact with each other along the edge, copies of *B2* and *A1* are made on processes *P1* and *P2* respectively, and the appropriate edges are created. A typical network model involves agents executing some behavior using their network neighbors (polling their neighbors and updating an attribute based on the result of the poll, for example), and the use of ghosts allows local agents to properly access all their network neighbors.

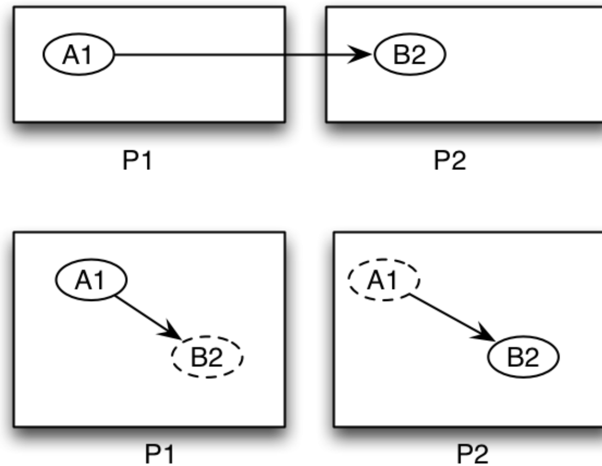

Figure 2: Ghost agents in a SharedNetwork.

The use of ghost agents requires the agent class to implement an `update` method, in addition to the required `save` method, that is called by the synchronization mechanism to update the internal state of a ghost agent from its source agent. `update` works as a complement to the `save` method discussed in Section 2.1, taking the agent's state that was serialized in `save` as an argument. The code in Figure 3, adapted from the Repast4Py Rumor Network demonstration model, illustrates a simple save and update in which the agent's state, the `received_rumor` variable is saved and updated. Ghost agents can be requested from specific ranks using the `SharedContext`'s `request_agents` method which will query those ranks for agents with the specified ids and create ghost copies of these on the requesting rank. This functionality is used by the network creation utility code (Section 2.5.3) to create ghost nodes and the edges between the ghost and local nodes, given a network partition description.

```
class RumorAgent(core.Agent):
    def save(self):
        return (self.uid, self.received_rumor)

    def update(self, data: bool):
        self.receive_rumor = data
```

Figure 3: The save method serializes the agent's state (self.received_rumor) and the update method receives it as an argument, updating the agent's state.

The network implementation consists of a base `SharedNetwork` class that contains the core network functionality and `DirectedSharedNetwork` and `UndirectedSharedNetwork` classes, which extend `SharedNetwork`, specializing it for directed and undirected networks, respectively. `SharedNetwork` is itself a wrapper around a networkx (Hagberg et al. 2008) `Graph` instance and delegates all of the network operations to that object, excluding those that may affect the distributed structure of the network, such as adding and removing nodes. `SharedNetwork` exposes the networkx `Graph` instance as its `graph` field that can be used for any network operations without structure altering side effects, iterating through an agent's network neighbors, for example (`network.graph.neighbors(agent)`).

2.4 Logging

The Repast4Py logging module is used to record output produced by a simulation to a file. The logging is of two types: 1) a *reduce*-type where the user specifies the individual column values to be logged, and Repast4Py performs a cross-process reduce operation (e.g., summing across processes) on those values; and 2) a more general type where the user explicitly logs an entire row of data, and the rows are concatenated across processes.

The *reduce* type is oriented around three concepts: data sources, data loggers, and data sets. A data source is a named source of typed numeric data. A data logger logs data from a single source, and can perform the reduction operation on the logged data. A data set represents a table of data, consisting of a collection of data loggers. The data set logs the data produced from each of its data loggers to a single file where each data source is a column in that file. Repast4Py implements the data source as a `DataSource` protocol and provides a `DCDataSource` that implements the protocol, adapting it for Python's `dataclass` objects such that the fields in the `dataclass` behave as data sources. The `ReducingDataLogger` implements the data logger, retrieving data from its data source (e.g., the current value of a `dataclasse's` field), and adding them to a NumPy array. The `ReducingDataSet` class implements a data set, containing a collection of `ReducingDataLoggers`. To log data, `ReducingDataSet.log` is called, which calls `log` on each of its `ReducingDataLoggers`. In practice, successive `log` calls result in the collection of NumPy arrays in the `ReducingDataLoggers` filling with the logged data on each process. Once the arrays reach a specified size, a MPI reduce operation is performed on rank 0 where the reduced data can be written out.

The code in Figure 4 illustrates how to initialize and log data using a Python `dataclass` as the data source. The intention here is to log the total number of persons and the total number of infected persons in an epidemiological model. These statistics are recorded in a `StatsLog` dataclass on each process. The `logging.create_loggers` function creates the data loggers using the `StatsLog` fields as data sources. This function defaults to creating loggers for all the fields in a `dataclass`. A subset of them can be logged by specifying the fields to log in the `names` argument of the function. In Figure 4, we call `logging.create_loggers` to create loggers for each of the fields in the StatsLog, passing the `StatsLog` instance, the reduction operation, and the rank of the current process. The resulting list of loggers is then passed to a `ReducingDataSet` constructor, an instance of which writes the output to `output_file`. Calling `ReducingDataSet`'s `log` method will then retrieve the current values of the `stats_log` fields and reduce (in this case sum) as necessary.

```
@dataclass
class StatsLog:
    total_persons: int = 0
    total_infected: int = 0

stats_log = StatsLog()
loggers = logging.create_loggers(stats_log, op=MPI.SUM, rank=rank)
data_set = logging.ReducingDataSet(loggers, MPI.COMM_WORLD, output_file)
...
data_set.log(current_tick)
```

Figure 4: Dataclass logging example.

The second type of logging, where the user explicitly provides the column values for each row, is implemented in the `TabularLogger` class. No reduction occurs. Rather, all the rows are collected on each process, concatenated on process 0, and written out. Figure 5 illustrates how to initialize and use the `TabularLogger`. The `TabularLogger` is initialized with the simulation's MPI communicator, the path to the output file, and a list of headers. Data is logged using the `log_row` method where each

argument is a column in that row. This is particularly useful for logging agent attributes. For example, in Figure 5, each agent's *energy* attribute is being logged per tick.

```
logger = logging.TabularLogger(MPI.COMM_WORLD,
                 output_file, ['tick', 'agent_id', 'energy'])
for agent in context.agents():
    logger.log_row(current_tick, agent.uid, agent.energy)
```

Figure 5: Explicit row logging example.

2.5 Utilities

Repast4Py also provides modules for model input parameter parsing, random number management, and network creation and partitioning.

2.5.1 Parameter Parsing

The `parameters` module contains two functions, `create_args_parser` and `init_params`, which work together to parse command line input into a dictionary of parameters that can be used to initialize a model. `create_args_parser` creates a command line argument parser with two positional arguments, a parameters file in yaml format composed of key value pairs and a json format parameter string. `init_params` takes these as arguments and returns a dictionary of parameters. Any parameters named in the json parameter string will override those named in the parameters file. This structure easily enables parameter sweeps or model explorations where some parameters change from run to run and are specified in the json string while others are kept constant in the parameters file. Figure 6 illustrates how the two functions are used together.

```
parser = parameters.create_args_parser()
args = parser.parse_args()
params = parameters.init_params(args.parameters_file, args.parameters)
```

Figure 6: Command line argument parsing.

2.5.2 Random Number Generation

Repast4Py uses an instance of `numpy.random.Generator` (NumPy Developers 2022) for random number generation. `numpy.random.Generator` provides access to a wide range of distributions and uses PCG64 as its default BitGenerator. The Repast4Py random module includes a module level `default_rng` variable and an `init` function for instantiating and seeding this variable. If the model input parameters, discussed in Section 2.5.1, define a `random.seed` parameter then `default_rng` will be automatically initialized with that seed during parameter initialization.

2.5.3 Network Creation

The `network` module provides functions for creating a distributed network definition from a `networkx.Graph` object, writing that definition to a file, and creating a `SharedNetwork` (see Section 2.3.3) from that file. This is particularly useful as the networkx package contains graph generators for a wide variety of networks, and these functions can be used to generate `SharedNetworks` from the produced graphs. The code in Figure 7 illustrates the use of these functions. Here, networkx is used to create a Watts-Strogatz small-world graph with *n_agents* number of nodes. `write_network` writes that network to the *fname* file, assigning it the name *ws_network*, and partitioning it over *n_ranks* ranks using the *metis* partition method. Partitioning the network entails assigning each node to a process. The `partition_method` specifies

how this is done, either randomly (the default) or by using the Metis (Karypis 2011) graph partitioning package. Metis will attempt to minimize the number of cross process edges when partitioning the graph, which can result in increased performance given the resulting smaller number of ghost agents to update across processes. Once the partitioned network definition is written, the file can be read to produce a `SharedNetwork`. `read_network` reads the file and creates the appropriate local and ghost nodes on each process using user specified `create_agent` and `restore_agent` functions. `create_agent` creates a local agent from the node specification in the network definition file, and `restore_agent` creates a ghost agent from serialized agent data. An example of the two functions from the Rumor Spreading demonstration model can be seen in Figure 8.

```
import networx as nx
g = nx.connected_watts_strogatz_graph(n_agents, 2, 0.25)
write_network(g, 'ws_network', fname, n_ranks, partition_method='metis')
...
context = ctx.SharedContext(comm)
read_network(fname, context, create_agent, restore_agent)
net = self.context.get_projection('ws_network')
```

Figure 7: Network creation and partitioning.

```
def create_rumor_agent(nid, agent_type, rank, **kwargs):
    return RumorAgent(nid, agent_type, rank)

def restore_agent(agent_data):
    uid = agent_data[0]
    return RumorAgent(uid[0], uid[1], uid[2], agent_data[1])
```

Figure 8: Create and restore functions.

3 MODEL STRUCTURE

In the previous section we discussed the various components used to implement a complete distributed ABM. This section will describe the canonical structure of a Repast4Py simulation that combines these components to create a running model. This structure has four parts: 1) agents implemented as classes including the required `save` and optional `update` methods; 2) a *model*-type class to initialize and manage the simulation; 3) a function to handle restoring agents as they move between processes; and 4) additional code to run the simulation from the command line. Of course, in a more complex simulation the responsibilities and behavior of the agent and model classes can be factored out into additional classes, functions, and modules as necessary, but the overall organization remains the same. Figures 9 and 10 adapted from the Simple Random Walk demonstration model, are examples of this structure, focusing in particular on the initialization and management responsibilities of the *model*-type class. Here the agent is implemented in the `Walker` class. The `Model` class encapsulates the simulation, and is responsible for initialization and the overall iterating behavior of the simulation. Typically, the initialization occurs in the model classes constructor. In `Model.__init__` in Figure 9 we can see how the `SharedScheduleRunner` (Section 2.2), the agent environment, a `SharedGrid` (Section 2.3.1), the agents themselves, and logging (Section 2.4) are all initialized. Each of these initializations uses an entry or entries from the model input `params` dictionary produced by the parameter parsing utility functions (Section 2.5.1). The behavior that propels the simulation forward in time is implemented in the `Model.step` method (Figure 10) and scheduled for execution using the `SharedScheduleRunner`: `self.runner.schedule_repeating_event(1, 1,` **self.step**). In `step`, we iterate through each agent in the `context` and execute their *walk* behavior. Then `SharedContext.synchronize` is called to synchronize the global simulation across processes

(see Section 4 for more details), and lastly the output data for the current time step is logged, and the `meet_log` (Section 2.4) counts are reset to 0.

```
class Walker(core.Agent):
    ...

class Model:
    def __init__(self, comm: MPI.Intracomm, params: Dict):
        self.runner = schedule.init_schedule_runner(comm)
        self.runner.schedule_repeating_event(1, 1, self.step)
        ...
        self.context = SharedContext(comm)
        ...
        self.grid = SharedGrid(name='grid', bounds=box,
            borders=BorderType.Sticky, occupancy=OccupancyType.Multiple,
            buffer_size=2, comm=comm)
        self.context.add_projection(self.grid)
        for i in range(params['walker.count']):
            ...
            walker = Walker(i, rank, pt)
            self.context.add(walker)

        # initialize the logging
        self.agent_logger = TabularLogger(comm, params['agent_log_file'],
            ['tick', 'agent_id', 'agent_uid_rank', 'meet_count'])
        ...
        self.data_set = ReducingDataSet(loggers, MPI.COMM_WORLD,
            params['meet_log_file'])
```

Figure 9: Model initialization.

```
class Model:
    def step(self):
        for walker in self.context.agents():
            walker.walk(self.grid)
        self.context.synchronize(restore_walker)
        ...
        tick = self.runner.schedule.tick
        self.data_set.log(tick)
        self.meet_log.max_meets = self.meet_log.min_meets =
            self.meet_log.total_meets = 0
```

Figure 10: Model step method.

4 SYNCHRONIZATION

Synchronization harmonizes the model state across each process in response to changes in the model's shared projections. This involves: 1) moving agents between processes when agents move beyond the local dimensions of a bounded projection (i.e., a `SharedGrid` or `SharedCSpace`, Section 2.3.1) and into the area controlled by another process; 2) filling bounded projection buffers with ghost agents in order to unify the agent environment; and 3) updating the network nodes and structure (adding and removing edges, and nodes) in a `SharedNetwork` to reflect structural changes on other processes. All of these use

the save and restore API. In previous Sections (2.1 and 2.3.3) we discussed how each agent is required to implement a `save` method (Figure 1) that serializes the agent's state to a Tuple. We have also seen how the `update` method updates an existing ghost agent's state with that serialized state (Figure 3). Unlike `save` and `update`, restore is not the name of an agent method, but rather a required function that takes the Tuple produced by `save` and returns an agent deserialized from that data. A restore function is used to create an agent on a destination rank from an agent on a source process. The code in Figure 11 from Repast4Py's Random Walk demonstration model illustrates a *restore* function that uses a cache. Often agents, when located on the border of a bounded projection, will move back-and-forth between the processes adjacent to the border. In this case, a typical performance optimization is to cache agents created in the *restore* function to avoid recreating them as they alternate between neighboring processes. In Figure 11, if the serialized agent's unique agent id (`uid`) is found in the `walker_cache` then the cached agent is updated with the `walker_data` state and returned. Otherwise, a new agent is created using that state, cached, and returned.

```
walker_cache = {}
def restore_walker(walker_data: Tuple):
    uid = walker_data[0]
    pt_array = walker_data[2]
    pt = dpt(pt_array[0], pt_array[1], 0)

    if uid in walker_cache:
        walker = walker_cache[uid]
    else:
        walker = Walker(uid[0], uid[2], pt)
        walker_cache[uid] = walker

    walker.meet_count = walker_data[1]
    walker.pt = pt
    return walker
```

Figure 11: A restore method creates a Walker from serialized data.

Synchronization is performed by calling `SharedContext.synchronize` which takes a restore function as an argument. `synchronize` initiates all three of the operations mentioned above (Section 4), moving agents, filling buffers, and updating a network. To move agents between processes, each `SharedContext` queries its associated bounded projections for any agents that have moved beyond the local projection bounds for that process, calling `save` on each of those agents. Using a MPI `alltoall` call, the serialized agent data, consisting of the agent states and their current locations in the bounded projection, are sent to the appropriate destination ranks. There, the restore function is used to create each agent and it is added to the `SharedContext` and bounded projection location on that rank. To fill bounded projection buffers, `synchronize` delegates most of the buffer filling operation to its associated bounded projections. When created, each bounded projection determines what part of the buffer in its local area should be sent to each process neighbor. When synchronization occurs, each bounded projection serializes those agents in the pre-calculated area together with their coordinate locations, and sends that information to the appropriate neighboring ranks using MPI. Once the serialized data has been received, the neighboring projections create the ghost agents using the restore function and place them at the correct locations.

Similar to the buffer filling operation, `synchronize` also delegates network synchronization to `SharedNetwork`. In a `SharedNetwork` synchronization is necessary 1) when an edge between a ghost node and local node is created, removed, or updated; 2) when a node that is participating in an edge with a ghost node is removed; and 3) to update the state of ghost nodes from local nodes. In all three,

the implication is that an edge is mirrored on another rank and the local and ghost agents are reversed as explained in Section 2.3.3 and seen in Figure 2. Consequently, any change involving the local agent that is ghosted on another rank needs to be reflected on that rank. When synchronizing edges added between local nodes and ghosts, `SharedNetwork` sends the unique agent ids of the edge's nodes and the edge's attributes. If the local agent has not already been ghosted to the ghost's rank, then the agent's state is serialized and sent as well. When received, the ghost agent is created via the restore function, if necessary, and the edge is created. When synchronizing removed edges, `SharedNetwork` sends the unique agent ids of the nodes participating in the edge to the process of the ghost node where the edge is then removed. When synchronizing edge attributes, the updated attributes, and node agent ids are sent to the rank of the edge's ghost node where the mirrored edge's attributes are updated. When synchronizing removed nodes, the node agent ids are sent via an MPI call to the rank of the edge's ghost node and the removed node is removed from that rank, together with any edges it participated in. To synchronize ghost node state, the states of local nodes ghosted to other ranks are serialized via the `save` method and sent via an MPI call to the appropriate ranks where the ghost agent states are updated via the `update` method.

5 PERFORMANCE

To achieve sufficient performance, for example for running model exploration campaigns (Ozik et al. 2016), Repast4Py is implemented in pure Python and C++ using the Python C-API, and leverages the NumPy (Harris et al. 2020) and PyTorch (Paszke et al. 2019) Python packages where possible. In a previous paper (Collier et al. 2020), we have described our experiences in creating Repast4Py as a performant ABM toolkit in Python, exploring code profiling, how the `SharedGrid` and `SharedCSpace` are implemented using the Python C-API, and how PyTorch is used in the `SharedValueLayer` component. The Repast4Py User Guide (Repast Project 2022b) also includes an example (the `GridNghFinder` class) of how the Numba (Lam et al. 2015) package can be used in conjunction with NumPy to speed up calculating the neighboring grid coordinates of a location, a crucial part in determining an agent's neighborhood. Numba is a just-in-time compiler for Python. It can compile certain kinds of Python functions and classes into optimized native code that bypasses the slower Python interpreter. It is particularly useful for code that is numerically oriented and uses NumPy arrays. We identified this neighboring coordinate calculation as a potential area of improvement using Python's built-in cProfile package and the third-party kernprof package (Robert Kern 2020). Given that the computation can be easily expressed using NumPy arrays, Numba was a natural fit. Our intention is that Repast4Py users will use this as an example when performance tuning their own code: profile using cProfile and kernprof to identify performance hotspots; and re-code hotspots using NumPy and Numba to the extent possible, as well as investigating additional applicable performance techniques, such as those described in Gorelick and Ozsvald (2020). Particularly for distributed ABMs, developing efficient load balancing methods can also contribute to performance gains (Collier et al. 2015; Mucenic et al. 2021).

6 REPAST4PY EXAMPLE MODELS

The Repast4Py User Guide (Repast Project 2022b) contains an in-depth explanation of three demonstration models, illustrating seven distributed ABM use cases (UCs): UC1) movement in a discrete 2D grid; UC2) neighborhood queries in a 2D grid; UC3) logging aggregate and individual-level data; UC4) working with multiple agent types; UC5) movement in a continuous space; UC6) creating and partitioning a network; and UC7) network neighborhood queries. Here we provide a brief overview of the 3 example models and the use cases that they combine.

6.1 Random Walk Model (UC1, UC2, UC3)

The Random Walk model consists of a population of agents moving at random around a two-dimensional grid and logging the aggregate and agent-level co-location counts. Each timestep: 1) all the agents (*walkers*)

choose a random direction and move one unit in that direction; 2) all the agents count the number of other agents they meet at their current location by determining the number of co-located agents at their grid locations; and 3) the sum, minimum, and maximum number of agents met are calculated across all processes, and these values are logged. The model implements UC1, UC2, and UC3.

6.2 Zombies Model (UC1, UC2, UC3, UC4, UC5)

The Zombies model is a predator-prey type model in which zombie agents search for and pursue human agents while the human agents search for and flee the zombie agents. Once caught, the human agents become zombies. Each agent has a floating point coordinate (e.g., (30.3323, 22.02343)) location in a `SharedCSpace` and a corresponding discrete coordinate (e.g., (30,22)) location in a `SharedGrid`. The `SharedCSpace` is used for movement and the `SharedGrid` for neighborhood searches. Each zombie queries its immediate neighboring grid locations for the location with the greatest number of humans, and then moves in that direction in the `SharedCSpace`, attempting to infect any humans in the grid cell corresponding to its new location. Humans behave similarly, finding the adjacent grid location with the fewest zombies and moving towards it. The model implements UC1, UC2, UC3, UC4, and UC5.

6.3 Rumor Spreading (UC6, UC7)

The Rumor Spreading model is a simple network model that illustrates Repast4Py's network features. The simulation models the spread of a rumor through a networked population. During initialization a number of agents (network nodes) are marked as rumor spreaders. At each iteration of the simulation, a random draw is made to determine if the neighbors of any rumor-spreading nodes have received the rumor. This draw is performed once for each neighbor. After all of the neighbors that can receive the rumor have been processed, the collection of rumor spreaders is updated to include those nodes that received the rumor. The network itself is created using a networkx graph generator and the `create_network` function described in Section 2.5.3. The model implements UC6 and UC7.

7 SCALING STUDIES

To explore the scaling performance of Repast4Py and how a model's type and implementation can affect scaling, we ran two experiments using the Random Walk (Section 6.1), and Rumor Spreading (Section 6.3) demonstration models. We performed the experiments on Bebop, an HPC cluster managed by the Laboratory Computing Resource Center at Argonne National Laboratory, using our EMEWS (Ozik et al. 2016) framework. 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. We ran the Random Walk model using 3.6×10^6 *walker* agents, and global grid dimensions of 2016 x 2016, distributed over 36 processes (corresponding to 1 node), 72 ranks (2 nodes), 144 ranks (4 nodes), and 288 ranks (8 nodes). As the process count doubled, we reduced the number of agents on each rank by half. To keep the total size of the simulation's projection constant, we reduced the width and height of each grid's local dimensions by $\sqrt{2}$. The mean runtime over 30 stochastic replicates of 100 time steps was measured for each process count. The results can be seen in Table 2.

The Random Walk model scales well through 144 processes and then sees a drop off at 288 processes. As the process counts double, we see speedups of 1.88x (36 processes to 72), 1.85x (72 to 144), and 1.28x (144 to 288). As is typical in distributed models, the cost of inter-process communication eventually outweighs the performance advantage of executing fewer agents on each process. There is also an additional inter-process communication cost when distributing the model over more than one compute node, that is, over more than 36 processes on Bebop Broadwell nodes. A MPI implementation can optimize communication within a node using shared memory, but inter-node communication will necessarily be slower. This adds additional communication overhead as we increase the node count in each experiment. However, models with larger agent populations or more computationally expensive agent behaviors should continue to scale

well at higher process counts, to the extent that the benefits of dividing the increased computational costs among more processes are not outweighed by the increase in communication costs.

Table 2: Random walk model strong scaling (constant workload with increased process count).

Process Count	Runtime (sec.)	Speedup	Parallel Efficiency
36	320.15	1x	1
72	170.09	1.88x	0.94
144	91.88	3.48x	0.87
288	71.92	4.39x	0.55

For the Rumor Model experiment, we used the `create_network` function described in Section 2.5.3 to partition a 6×10^6 node small world network (Newman and Watts 1999) across 36, 72, 144, and 288 processes, using *metis* as the partition method, and seeding the network with 3600 initial rumor spreaders. As the process count doubled we halved the number of agents, or network nodes, on each rank. We also ensured that, for each process count, each rank had roughly the same number agents and number of initial rumor spreaders, maintaining the proportion of initial rumor spreaders to the total number of nodes on a rank across process counts. We measured the mean runtime for at least 80% of the network to receive the rumor over 30 stochastic replicates. The results can be seen in Table 3.

Table 3: Rumor spreading model scaling.

Process Count	Runtime (sec.)	Speedup	Parallel Efficiency
36	508.94	1x	1
72	357.85	1.42x	0.71
144	324.73	1.57x	0.39
288	294.62	1.75x	0.22

Unlike the Random Walk model, where all the agents *walk* each time step, the computational cost of executing the agent behaviors in the Rumor Spreading model does not remain constant across time steps. The model iterates through all the rumor spreading agents, adding more rumor spreaders at each time step as the rumor spreads. Consequently, the computational cost is lower for much of the simulation and the benefits of distributing this small amount of work can be easily overwhelmed by the inter-process communication cost of synchronizing the network. This is reflected in reduced parallel efficiency for the scaling results when compared to the Random Walk model across all process counts and highlights how structural differences between even simple models can dramatically affect scaling performance. These results illustrate the importance of understanding how the implementation of a model contributes to the computational cost and thus where the sweet spot between dividing that computational cost among processes and inter-process communication cost lies.

8 SUMMARY AND FUTURE WORK

In this tutorial, we have presented Repast4Py, an agent-based modeling framework written in Python that provides the ability to build large, MPI-distributed ABMs that span multiple processing cores. We began by discussing the core software components of Repast4Py, including agents, their environments, scheduling, logging, and utility code. We then described the canonical Repast4Py model structure which combines the components into an executable simulation. Exploring how the individual process instance components cooperate to create a unified simulation, we provided additional details on the cross-process synchronization mechanisms. We also discussed how performance enhancements of Repast4Py models can occur through multiple pathways. Distributed ABM use cases were presented in the context of the Repast4Py example models. We closed with two scaling studies illustrating how model specification and implementation determines the scaling characteristics of distributed simulations. Future work will focus on

additional performance updates, continued API usability improvements, additional example models, and a shared GIS-enabled projection to support distributed geographic style queries.

ACKNOWLEDGMENTS

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). This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357. Repast4Py is being used in multiple projects, including the NIH funded projects R01AI136056, U2CDA050098, R01MD014703, R21MH128116, R01AI146917, R01AI158666 and is informed by their requirements and continuing development.

REFERENCES

Collier, N., T. Howe, and M. North. 2003. "Onward and upward: The transition to Repast 2.0". In *Proceedings of the First Annual North American Association for Computational Social and Organizational Science Conference*, Volume 122, 136. Pittsburgh, PA: North American Association for Computational Social and Organizational Science.

Collier, N., and M. North. 2013. "Parallel agent-based simulation with Repast for High Performance Computing". *SIMULATION* 89(10):1215–1235.

Collier, N., J. Ozik, and C. M. Macal. 2015. "Large-Scale Agent-Based Modeling with Repast HPC: A Case Study in Parallelizing an Agent-Based Model". In *Euro-Par 2015: Parallel Processing Workshops*, edited by S. Hunold, A. Costan, D. Giménez, A. Iosup, L. Ricci, M. E. G. Requena, V. Scarano, A. L. Varbanescu, S. L. Scott, S. Lankes, J. Weidendorfer, and M. Alexander, Number 9523 in Lecture Notes in Computer Science, 454–465. Cham, Switzerland: Springer International Publishing.

Collier, N. T., J. Ozik, and E. R. Tatara. 2020. "Experiences in Developing a Distributed Agent-based Modeling Toolkit with Python". In *2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, 1–12. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Dalcin, L., and Y.-L. L. Fang. 2021. "mpi4py: Status Update After 12 Years of Development". *Computing in Science & Engineering* 23(4):47–54.

Gorelick, M., and I. Ozsvald. 2020. *High Performance Python*. 2nd ed. Sebastapol, CA: O'Reilly.

Hagberg, A. A., D. A. Schult, and P. J. Swart. 2008. "Exploring Network Structure, Dynamics, and Function using NetworkX". In *Proceedings of the 7th Python in Science Conference*, edited by G. Varoquaux, T. Vaught, and J. Millman, 11 – 15. Pasadena, CA.

Harris, C. R., K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. 2020. "Array Programming with NumPy". *Nature* 585(7825):357–362.

Kaligotla, C., J. Ozik, N. Collier, C. M. Macal, K. Boyd, J. Makelarski, E. S. Huang, and S. T. Lindau. 2020. "Model Exploration of an Information-Based Healthcare Intervention Using Parallelization and Active Learning". *Journal of Artificial Societies and Social Simulation* 23(4):1.

Karypis, G. 2011. "METIS and ParMETIS". In *Encyclopedia of Parallel Computing*, edited by D. Padua, 1117–1124. Boston, MA: Springer US.

Khanna, A. S., J. A. Schneider, N. Collier, J. Ozik, R. Issema, A. di Paola, A. Skwara, A. Ramachandran, J. Webb, R. Brewer, W. Cunningham, C. Hilliard, S. Ramani, K. Fujimoto, and N. Harawa. 2019. "A Modeling Framework to Inform Preexposure Prophylaxis Initiation and Retention Scale-up in the Context of 'Getting to Zero' Initiatives:". *AIDS* 33(12):1911–1922.

Lam, S. K., A. Pitrou, and S. Seibert. 2015. "Numba: A LLVM-Based Python JIT Compiler". In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, 1–6. New York, NY: Association for Computing Machinery.

Lindau, S. T., J. A. Makelarski, C. Kaligotla, E. M. Abramsohn, D. G. Beiser, C. Chou, N. Collier, E. S. Huang, C. M. Macal, J. Ozik, and E. L. Tung. 2021. "Building and Experimenting with an Agent-based Model to Study the Population-level Impact of CommunityRx, a Clinic-based Community Resource Referral Intervention". *PLOS Computational Biology* 17(10):e1009471.

Macal, C. M., N. T. Collier, J. Ozik, E. R. Tatara, and J. T. Murphy. 2018. "ChiSIM: An Agent-Based Simulation Model of Social Interactions in a Large Urban Area". In *Proceedings of the 2018 Winter Simulation Conference*, edited by M. Rabe, A. A. Juan, N. Mustafee, A. Skoogh, S. Jain, and B. Johansson, 810–820. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Mucenic, B., C. Kaligotla, A. Stevens, J. Ozik, N. Collier, and C. Macal. 2021. "Load Balancing Schemes for Large Synthetic Population-Based Complex Simulators". In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 985–988. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Newman, M., and D. Watts. 1999. "Renormalization Group Analysis of the Small-world Network Model". *Physics Letters A* 263(4):341–346.

North, M. J., N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko. 2013. "Complex Adaptive Systems Modeling with Repast Simphony". *Complex Adaptive Systems Modeling* 1(1):3.

North, M. J., N. T. Collier, and J. R. Vos. 2006. "Experiences Creating Three Implementations of the Repast Agent Modeling Toolkit". *ACM Transactions on Modeling and Computer Simulation* 16(1):1–25.

NumPy Developers 2022. "Random Generator". https://numpy.org/doc/stable/reference/random/generator.html. accessed 28th April 2022.

Ozik, J., N. Collier, T. Combs, C. M. Macal, and M. North. 2015. "Repast Simphony Statecharts". *Journal of Artificial Societies and Social Simulation* 18(3):11.

Ozik, J., N. T. Collier, J. T. Murphy, and M. J. North. 2013. "The ReLogo Agent-based Modeling Language". In *Proceedings of the 2013 Winter Simulations Conference*, edited by R. Pasupathy, S.-H. Kim, A. Tolk, R. R. Hill, and M. E. Kuhl, 1560–1568. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Ozik, J., N. T. Collier, J. M. Wozniak, C. M. Macal, and G. An. 2018. "Extreme-Scale Dynamic Exploration of a Distributed Agent-Based Model With the EMEWS Framework". *IEEE Transactions on Computational Social Systems* 5(3):884–895.

Ozik, J., N. T. Collier, J. M. Wozniak, and C. Spagnuolo. 2016. "From Desktop to Large-scale Model Exploration with Swift/T". In *Proceedings of the 2016 Winter Simulation Conference*, edited by T. M. Roeder, P. I. Frazier, E. Z. Robert Szechtman, T. Huschka, and S. E. Chick, 206–220. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Ozik, J., J. M. Wozniak, N. Collier, C. M. Macal, and M. Binois. 2021, September. "A Population Data-driven Workflow for COVID-19 Modeling and Learning". *The International Journal of High Performance Computing Applications* 35(5):483–499.

Paszke, A., 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. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, 8024–8035. Curran Associates, Inc.

Repast Project 2022a. "Repast for Python API Documentation". https://repast.github.io/repast4py.site/apidoc/index.html. accessed 31st January 2022.

Repast Project 2022b. "Repast for Python User Guide". https://repast.github.io/repast4py.site/guide/user_guide.html. accessed 31st January 2022.

Robert Kern 2020. "Kernprof Line Profiler". https://github.com/pyutils/line_profiler. accessed 8th September 2020.

Tatara, E., A. Gutfraind, N. T. Collier, D. Echevarria, S. J. Cotler, M. E. Major, J. Ozik, H. Dahari, and B. Boodram. 2022, March. "Modeling hepatitis C Micro-elimination among People who Inject Drugs with Direct-acting Antivirals in Metropolitan Chicago". *PLOS ONE* 17(3):e0264983.

AUTHOR BIOGRAPHIES

Nicholson Collier is a Software Engineer at Argonne National Laboratory, and Staff Software Engineer in the Consortium for Advanced Science and Engineering at the University of Chicago. As the lead developer for the Repast project for agent-based modeling toolkits and co-developer of the Extreme-scale Model Exploration with Swift (EMEWS) framework, he develops, architects, and implements large-scale agent-based models and frameworks in a variety of application areas, and large-scale model exploration workflows across various domains, including agent-based modeling, microsimulation and machine/deep learning. His e-mail address is ncollier@anl.gov.

Jonathan Ozik is a Principal Computational Scientist at Argonne National Laboratory, Senior Scientist with Department of Public Health Sciences affiliation in the Consortium for Advanced Science and Engineering (CASE) at the University of Chicago, and Senior Institute Fellow in the Northwestern Argonne Institute of Science and Engineering (NAISE) at Northwestern University. He applies large-scale agent-based models and large-scale model exploration across research domains and computational modeling methods. He leads the Repast project for agent-based modeling toolkits and the Extreme-scale Model Exploration with Swift (EMEWS) framework for large-scale model exploration on high-performance computing resources. His e-mail address is jozik@anl.gov.