Argonne NATIONAL LABORATORY

BUILDING A DISTRIBUTED AGENT-BASED MODEL WITH REPAST4PY

NICK COLLIER
Senior Software Engineer
Argonne National Laboratory

JONATHAN OZIK
Principal Computational Scientist
Argonne National Laboratory

20 July 2022
San Diego, CA

OUTLINE

- Introduction and Motivation
- Repast4Py Overview
- Repast4Py Distributed ABM Components
- Building a Model: Hands-on Coding Exercise
- Performance Techniques

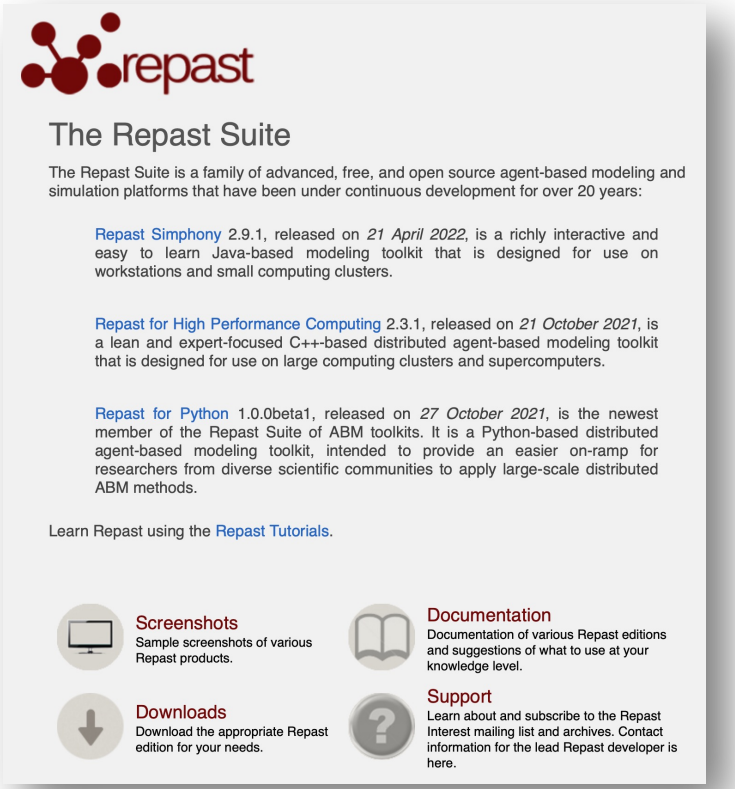
JUPYTER LAB ON BINDER

- https://tinyurl.com/33mynb4b (https://mybinder.org/v2/gh/Repast/repast4py-tutorial.git/annsim_2022)

INTRODUCTION AND MOTIVATION

REPAST AGENT-BASED MODELING SUITE

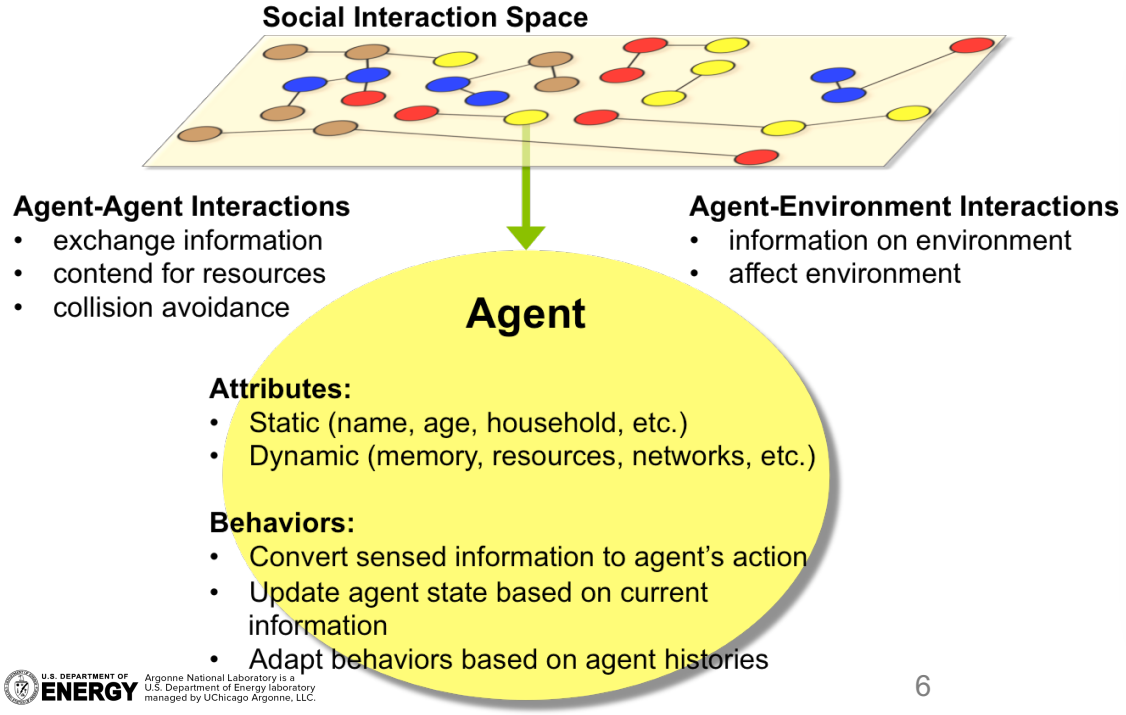
https://repast.github.io

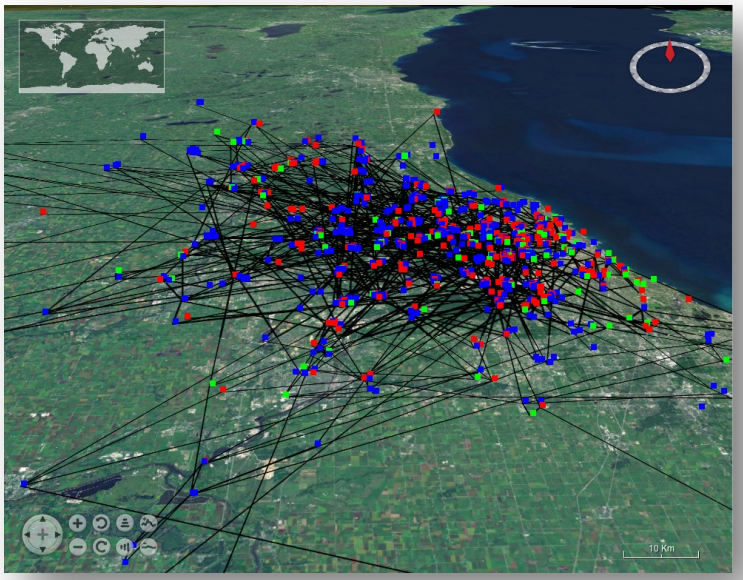


- Family of advanced, free, and open source agent-based modeling and simulation platforms that have collectively been under continuous development for over 20 years
- Over 200,000 downloads globally
- Developed and maintained at Argonne
- Three flavors of agent-based modeling toolkits
  - Repast4Py
  - Repast Simphony
  - Repast for High Performance Computing (Repast HPC)

AGENT-BASED MODELS (ABMS)

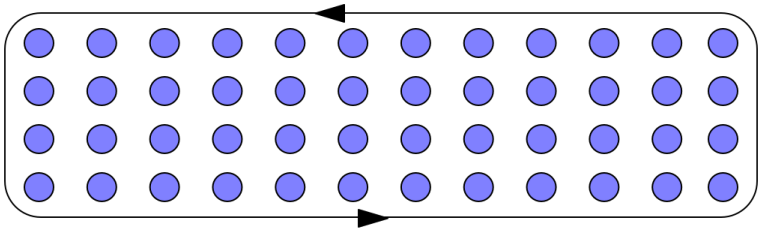
- **Disaggregated description of complex systems:**
  - Method of computing the potential system-level consequences of the behaviors of sets of individuals
  - Effects of interventions can be run with different assumptions



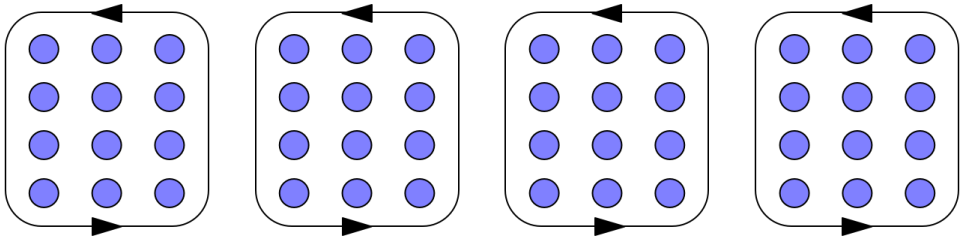


MOTIVATION FOR DISTRIBUTED ABMS

- Current desktop / laptop CPUs typically contain multiple processing cores.
- HPC clusters and supercomputers can contain thousands or more cores.
- Distributed ABM
  - partitions the agents over multiple processes, resulting in a *faster runtime.*
  - allows for greater numbers of agents and more complex behavior.



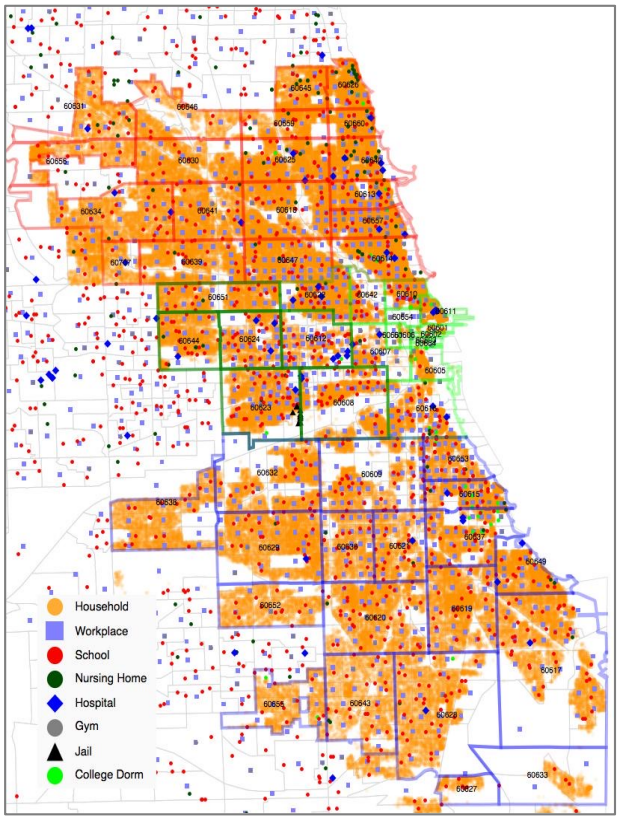
Agents Executing Sequentially on a Single Process



Agents Executing Sequentially on Multiple Concurrent Processes

ChiSIM: THE CHICAGO SOCIAL INTERACTION MODEL

SIMULATING CHICAGO AND EVERYONE IN IT



- 2.7(-8.5) million individual people agents
- Moving to/from 1.2(-5.2) million geo-located places on an hourly basis according to individual activity schedules
- Agents interact with other co-located agents at those places

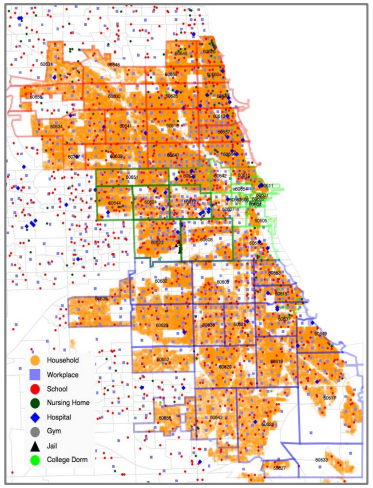
Macal, C. M., N. T. Collier, J. Ozik, E. R. Tatara, and J. T. Murphy. 2018. "CHISIM: AN AGENT-BASED SIMULATION MODEL OF SOCIAL INTERACTIONS IN A LARGE URBAN AREA." In *2018 Winter Simulation Conference (WSC)*, 810–20. https://doi.org/10.1109/WSC.2018.8632409.

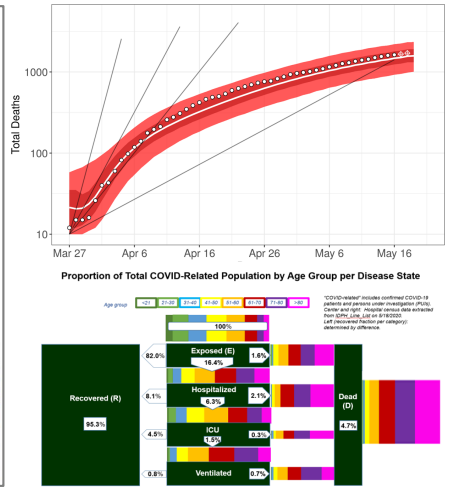
RAPIDLY EVOLVING POLICY AND EPIDEMIOLOGY QUESTIONS

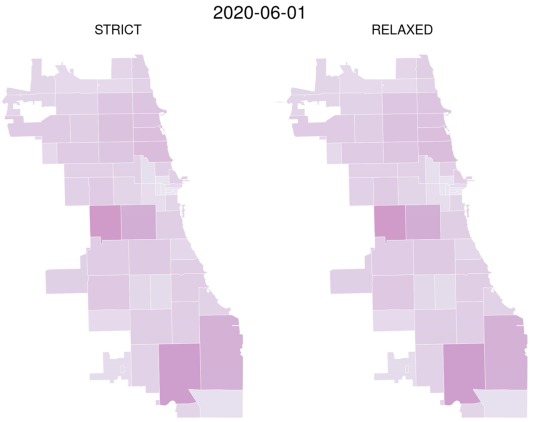
repast

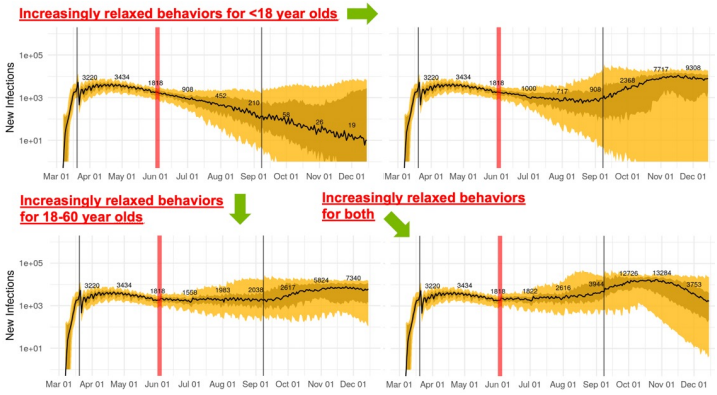
EMEWS

CityCOVID model for Chicago, 2.7 M people (agents) move hourly between 1.2 M locations









What mitigation strategies should be considered?	How should we ease mitigations?	What are place/occupation based risks?	How can we reopen schools/universities?	Policy
How will COVID-19 affect populations?	How does mobility affect transmission?	How do behaviors affect transmission?	How do different age groups behave differently?	Epidemiology

March April May June July

Argonne NATIONAL LABORATORY

REPAST4PY OVERVIEW

REPAST4PY

- Repast4Py is an agent-based modeling framework written in Python that provides the ability to build large, **distributed agent-based models** (ABMs) that span multiple processing cores.
- Repast4Py is released under the **BSD-3 open source license**, and leverages the MPI for Python, 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.
- **Homepage**: https://repast.github.io/repast4py.site/index.html
- **User Guide**: https://repast.github.io/repast4py.site/guide/user_guide.html
- **API Docs**: https://repast.github.io/repast4py.site/apidoc/index.html
- **Github Repo**: https://github.com/Repast/repast4py

Table of Contents

Repast for Python (Repast4Py) User Guide

Version 1.0 October 2021

1. Getting Started

Repast for Python (Repast4Py) is the newest member of the Repast Suite of free and open source agent-based modeling and simulation software. It builds on Repast HPC, and provides the ability to build large, distributed agent-based models (ABMs) that span multiple processing cores. Distributed ABMs enable the development of complex systems models that capture the scale and relevant details of many problems of societal importance.[1][2] Where Repast HPC is implemented in C++ and is more HPC expert focused, Repast4Py is a Python package and is designed to provide an easier on-ramp for researchers from diverse scientific communities to apply large-scale distributed ABM methods. Repast4Py is released under the BSD-3 open source license, and leverages 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. See our paper on Repast4Py for additional information about the design and implementation.[3]

1.1. Requirements

Repast4Py requires Python 3.7+

Repast4Py can run on Linux, macOS and Windows provided there is a working MPI implementation installed and mpi4py is supported. Repast4Py is developed and tested on Linux. We recommend that Windows users use the Windows Subsystem for Linux (WSL). Installation instructions for WSL can be found here.

Under Linux, MPI can be installed using your OS's package manager. For example, under Ubuntu 20.04 (and thus WSL), the mpich MPI implementation can be installed with:

```
$ sudo apt install mpich
```

A typical campus cluster, or HPC resource will have MPI and mpi4py installed. Check the resource's documentation on available software for more details.

1.2. Installation

Repast4Py can be downloaded and installed from PyPI using pip. Since Repast4Py includes native MPI C++ code that needs to be compiled, the C compiler `CC` environment variable must be set to the `mpicxx` (or `mpic++`) compiler wrapper provided by your MPI installation.

```
env CC=mpicxx pip install repast4py
```

REPAST4PY

- Repast4Py is an agent-based modeling framework written in Python that provides the ability to build large, **distributed agent-based models** (ABMs) that span multiple processing cores.
- Repast4Py is released under the **BSD-3 open source license**, and leverages the MPI for Python, 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.
- **Homepage**: https://repast.github.io/repast4py.site/index.html
- **User Guide**: https://repast.github.io/repast4py.site/guide/user_guide.html
- **API Docs**: https://repast.github.io/repast4py.site/apidoc/index.html
- **Github Repo**: https://github.com/Repast/repast4py

REPAST4PY DISTRIBUTED ABM COMPONENTS

REPAST4PY AND MPI

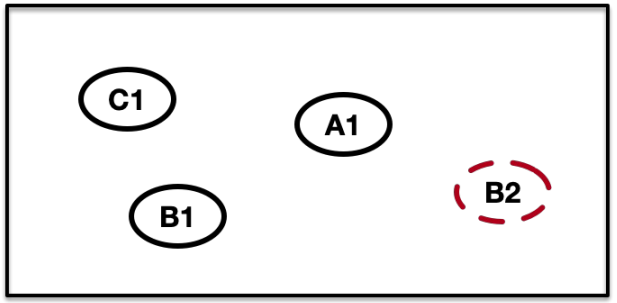
- Repast4Py is implemented in MPI (Message Passing Interface), using the mpi4py Python package.
- MPI applications run on **separate**, **concurrently** executing processes that **do not share memory** and communicate through passing messages.

```
1  from mpi4py import MPI
2
3  size = MPI.COMM_WORLD.Get_size()
4  rank = MPI.COMM_WORLD.Get_rank()
5
6  print(f'Hello, World! I am rank {rank} of {size}')
7
```

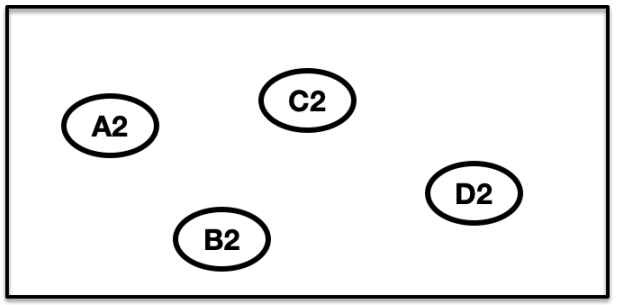
- In a distributed Repast4Py model, agents and their environments run on separate processes that need to be combined into a coherent unified model.

UNIFYING A DISTRIBUTED MODEL 1: GHOST AGENTS

- A *ghost* agent is a copy of an agent from one process copied onto another.
- Agents *local* to a process can interact with agents from other processes by interacting with the ghosts of those agents.
- Coder provides code defining the agent data to copy and how to create an agent from that data.
- Repast4Py performs the copy from one process to the other, creating the agent, and adding it to the destination process agent population as a ghost.



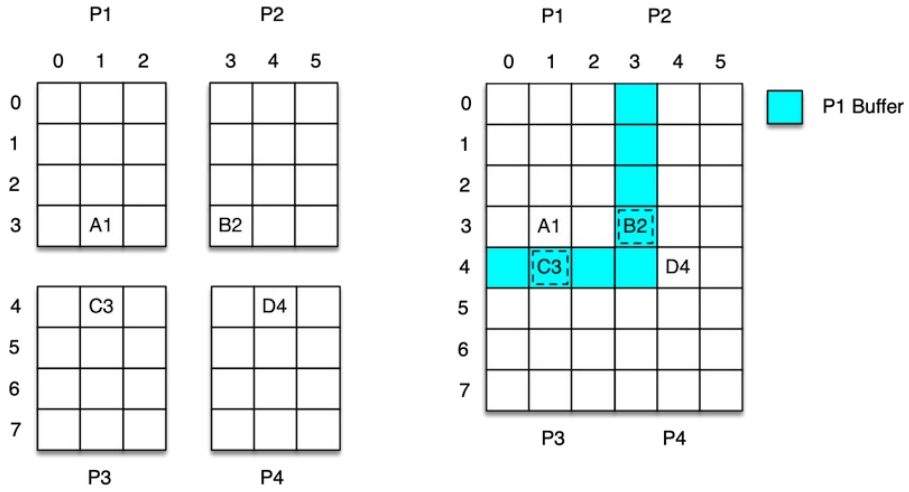
Process P1

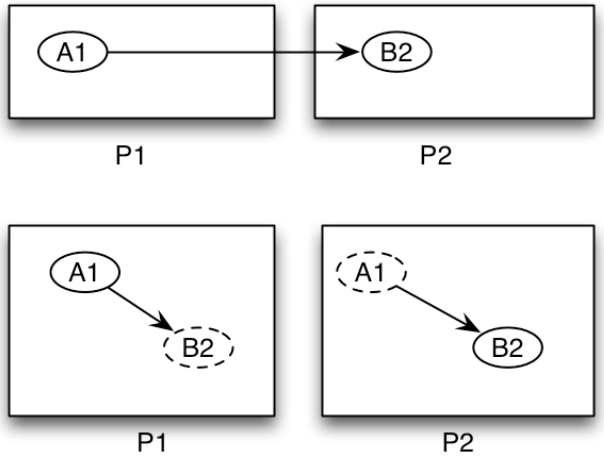


Process P2

UNIFYING A DISTRIBUTED MODEL 2: ENVIRONMENT BUFFERS

- Buffers are adjacent areas of a global environment duplicated between processes.
- Repast4Py handles the buffer synchronization automatically.





REPAST4PY COMPONENTS: CONTEXTS AND PROJECTIONS

- Contexts and projections first appeared in Repast Simphony as a way to apply multiple agent environments to a population of agents.
- A 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.
- A projection (an agent environment) takes the population as defined in a context and imposes a structure on it, where the structure is defined by the semantics of the projection. For example, a network projection provides a network structure with agents as nodes.
- Repast4Py implements ***shared*** contexts and projections as Python classes and modules. ***Shared*** emphasizes the distributed nature of the simulation, where the global simulation is shared among a pool of processes, each of which is responsible for some portion of it, and stitched together using ghost agents and buffered projections.

SHARED CONTEXT

- A shared context
  - encapsulates and provides access to local agents, ghosts, and projections
  - synchronizes ghost agents and projection buffers
- Each process has a shared context.
- The shared context is implemented by the Repast4Py `repast4py.context.SharedContext` class.
- API documentation provides the details on the various methods for iterating through agents, accessing projections, and synchronizing the model across processes.

repast4py.context module

class repast4py.context.**SharedContext**(*comm*)

Bases: **object**

Encapsulates a population of agents on a single process rank.

A SharedContext may have one or more projections associated with it to impose a relational structure on the agents in the context. It also provides functionality for synchronizing agents across processes, moving agents from one process to another and managing any ghosting strategy.

Parameters: **comm** (*mpi4py.MPI.Intracomm*) – the communicator used to communicate among SharedContexts in the distributed model

add(*agent*)

Adds the specified agent to this SharedContext.

The agent will also be added to any projections currently in this SharedContext

Parameters: **agent** (*_core.Agent*) – the agent to add

add_projection(*projection*)

Adds the specified projection to this SharedContext.

Any agents currently in this context will be added to the projection.

Parameters: **projection** (*repast4py.core.SharedProjection*) – the projection add

add_value_layer(*value_layer*)

Adds the specified value_layer to the this context.

Parameters: **value_layer** (*repast4py.value_layer.SharedValueLayer*) – the value layer to add.

agent(*agent_id*)

Gets the specified agent from the collection of local agents in this context.

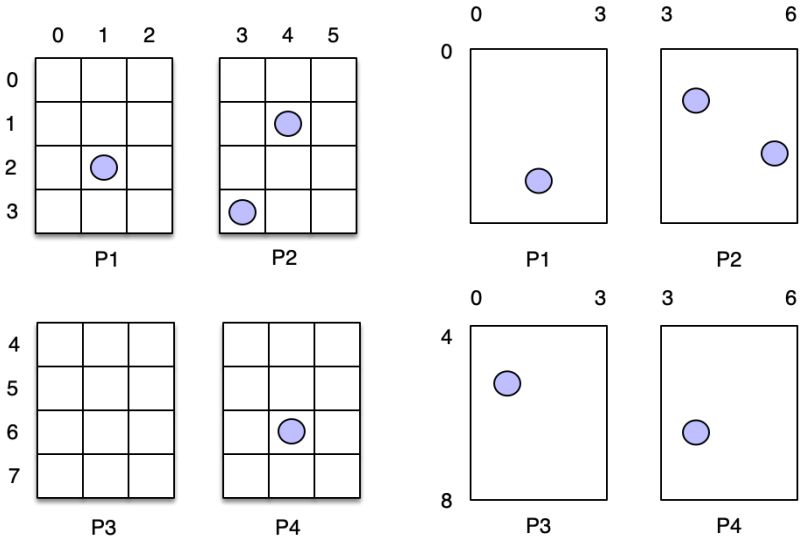
Parameters: **agent_id** – the unique id tuple of the agent to return
Returns: The agent with the specified id or None if no such agent is found.
Return type: _core.Agent

Examples

```
>>> ctx = SharedContext(comm)
>>> # .. Agents Added
>>> agent1 = ctx.agent((1, 0, 1))
```

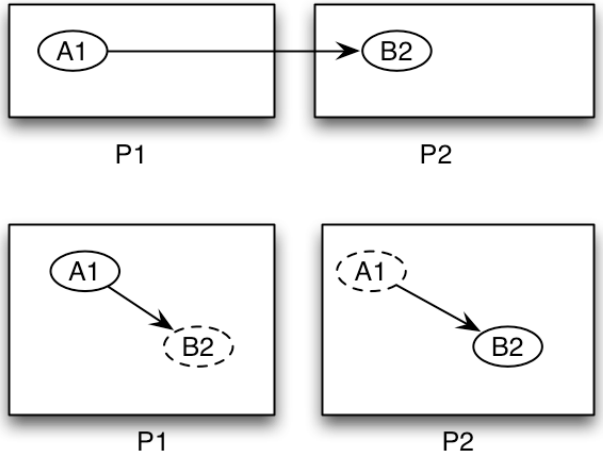
SHARED PROJECTIONS 1

- Shared projections are agent environments partitioned over multiple processes that cooperate with each other and the shared context to create a unified agent environment.
- `space.SharedGrid` is a space in which an agent's location is expressible as a discrete integer coordinate. Agents can move around the grid and query it for agents at a specified location.
- `space.SharedCSpace` is a space in which an agent's location is expressible as a continuous floating point coordinate. Agents can move around the space and query it for agents at a specified location or within a specified bounds.



SHARED PROJECTIONS 2

- The `SharedNetwork` classes implement directed and undirected networks in which agents are the nodes 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.
- The SharedValueLayer class is a grid of numeric values that can be retrieved or set via an indexing coordinate.



	0	1	2
0	1.2	0	0
1	3.2	1	4
2	3.1	11	3
3	12	0	33

P1

3	4	5
8	1.1	4.2
7	2.3	6.2
8.2	8.1	0
0	0	0

P2

4	13	87	1.2
5	5	5	6
6	8	5	2
7	0	0	0

P3

0	1.2	3.4
1.1	0	5
3.2	11	0
12	0	8

P4

REPAST4PY AGENTS

- Synchronizing cross-process ghosts and buffers consists of moving and copying agents between processes. This entails saving the agent state, moving or copying that state to another process, and then restoring the agent state as an agent on the destination process. For this to work:
- All agents in a Repast4Py model must extend the `repast4py.core.Agent` class.
- All agents must have a unique id tuple with 3 elements:
  - An integer that uniquely identifies the agent on the process rank on which it was created
  - An integer that identifies the model specific type of agent
  - The process rank on which the agent was created
- All agents must implement a `save` method that provides enough of the internal state of the agent that a copy can be created.
- The coder must provide a `restore` function that creates an agent from the saved state.

SHARED SCHEDULE

- Events in Repast4Py models, such as when an agent or agents execute their behavior, are driven by a discrete-event schedule.
- Events are scheduled at a particular *tick* that determines when that event will occur.
- In Repast4Py, events are Python functions or classes that implement the `__call__` method (i.e., `Callable`).
- Functions and callables are scheduled for execution on a `SharedScheduleRunner`.
- The `SharedScheduleRunner` synchronizes across processes such that no individual schedule gets ahead of those on other processes.

```
def step():
    for agent in ctx.agents():
        agent.do_something()

runner.schedule_repeating_event(1, 1, step)
runner.schedule_stop(10)
```

LOGGING

- The `ReducingDataSet` class logs data from a Python `dataclass` instance. The instance's fields become columns in the tabular format output.
- A user-specified cross-process operation (e.g., summation) is performed on each of the fields and the result is recorded to a file.
- The `TabularLogger` class logs user supplied rows of data in a tabular format.
- The rows are concatenated across processes and written to a file.
- `ReducingDataSet` is appropriate for aggregate-level data.
- `TabularLogger` is approprate for individual level data (e.g., logging agent attributes), but can be slow for large amounts of data.

```
@dataclass
class Counts:
    total_agents: int
    infected_agents: int
```

HANDS-ON CODING EXERCISE: THE RANDOM WALK MODEL

THE RANDOM WALK MODEL

- In the Random Walk Model:
  - agents move at random around a two-dimensional grid
  - co-location counts are logged
- Each timestep the following occurs:
  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 at their grid location.
  3. The sum, minimum, and maximum number of co-located agents are calculated across all process ranks, and these values are logged.

THE RANDOM WALK MODEL COMPONENTS

- A `Walker` class that implements the agent state and behavior.
- A `Model` class responsible for initialization and managing the model.
- A `restore_walker` function used to create an individual `Walker` when that `Walker` has walked to another process.
- A `run` function that creates and starts the model.
- A main block that allows the model to be run from the command line.
- We will begin with a skeleton, and incrementally add code to implement these components.
- Binder Jupyter Lab: https://tinyurl.com/33mynb4b

PERFORMANCE CONSIDERATIONS

SCALING

- Inter-process communication cost can eventually outweigh the performance advantage of executing fewer agents on each process.
- Models with more agents and more complex agent behavior will typically scale further.
- Investigating where the sweet spot between dividing computational cost among processes and inter-process communication cost lies is an important point of distributed model development.

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

Random Walk Model Strong Scaling

NUMBA

- Numba is a **just-in-time compiler** for Python.
- It can compile certain types of Python functions and classes into **optimized native code** that bypasses the slower Python interpreter.
- It works particularly well for code that is numerically oriented and uses NumPy arrays.

```
spec = [
    ('mo', int32[:]),
    ('no', int32[:]),
    ('xmin', int32),
    ('ymin', int32),
    ('ymax', int32),
    ('xmax', int32)
]


@jitclass(spec)
class GridNghFinder:

    def __init__(self, xmin, ymin, xmax, ymax):
        self.mo = np.array([-1, 0, 1, -1, 0, 1, -1, 0, 1], dtype=np.int32)
        self.no = np.array([1, 1, 1, 0, 0, 0, -1, -1, -1], dtype=np.int32)
        self.xmin = xmin
        self.ymin = ymin
        self.xmax = xmax
        self.ymax = ymax

    def find(self, x, y):
        xs = self.mo + x
        ys = self.no + y

        xd = (xs >= self.xmin) & (xs <= self.xmax)
        xs = xs[xd]
        ys = ys[xd]

        yd = (ys >= self.ymin) & (ys <= self.ymax)
        xs = xs[yd]
        ys = ys[yd]

        return np.stack((xs, ys, np.zeros(len(ys), dtype=np.int32)), axis=-1)
```

QUESTIONS?
THANK YOU FOR ATTENDING!