What is Ray

Ray provides a simple, universal API for building distributed applications

What is Ray

Ray provides a simple, universal API for building distributed applications

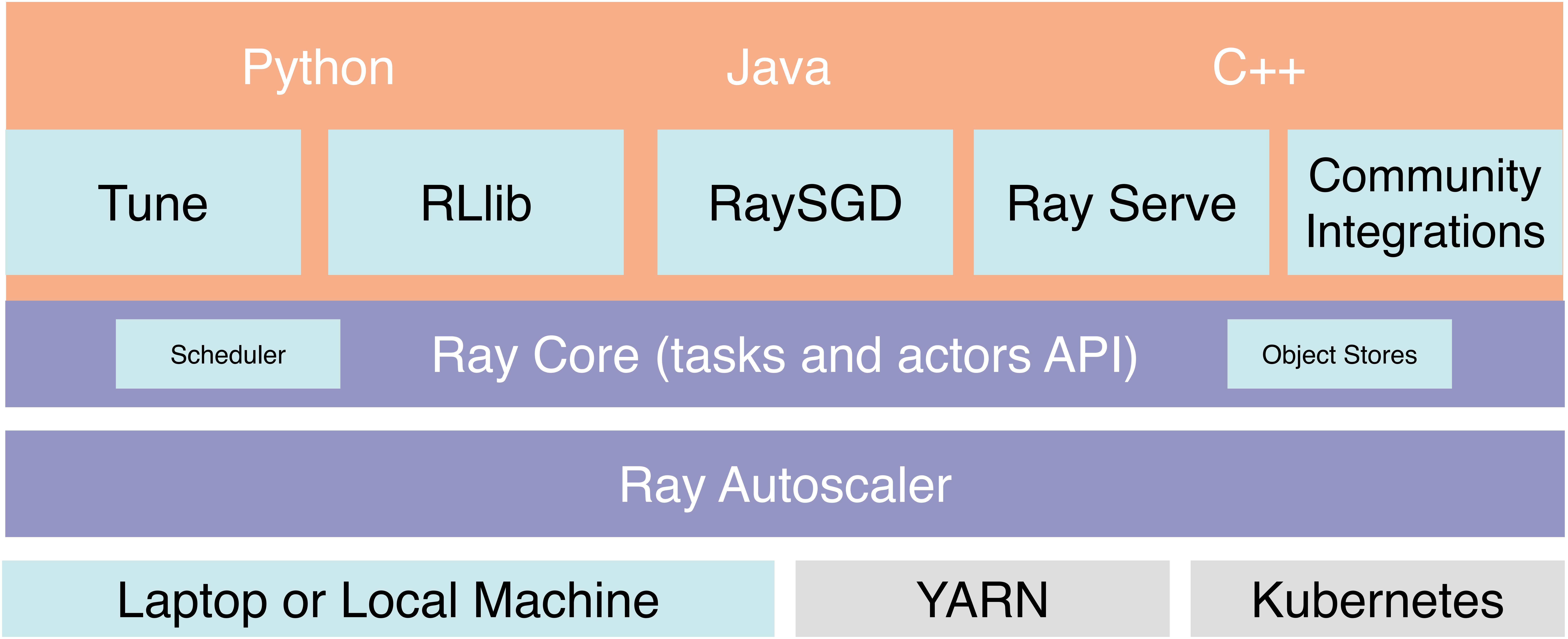
What is Ray

- Primitives for running fault-tolerant distributed applications
- Parallelize single machine code with little to zero code changes
- Large ecosystem of applications/libraries/tools on top of Ray Core

What is Ray

`multiprocess` **meets** **RPC...**

Ray Layer Cake



The Ray Programming Model

Tasks	Actors
Remote Functions	Remote Classes/Objects
Similar to Spark closures	Maintain internal state

The Ray Programming Model

Tasks

```python
import ray
ray.init()

@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

Actors

```python
import ray
ray.init() # Only call this once.

@ray.remote
class Counter(object):
    def __init__(self):
        self.n = 0

    def increment(self):
        self.n += 1

    def read(self):
        return self.n

counters = [Counter.remote() for i in range(4)]
[c.increment.remote() for c in counters]
futures = [c.read.remote() for c in counters]
print(ray.get(futures)) # [1, 1, 1, 1]
```


Source: https://docs.ray.io/en/master/ray-overview/index.html#gentle-intro

The Ray Programming Model

Tasks	Actors
Fine-grained load balancing	Coarse-grained load balancing
Support for object locality	Poor locality support
High overhead for small updates	Low overhead for small updates
Efficient failure handling	Overhead from checkpointing

Source: https://docs.ray.io/en/master/ray-overview/index.html#gentle-intro

Spark

- Coarse grained
- SQL like abstractions (tables)
- Synchronous
- Functional

Ray

- Fine grained
- Actor based abstractions (agents)
- Asynchronous
- Object Oriented

Spark

- Data processing and transformation
- Data heavy workloads
- Iterative MapReduce
- Ex: Pagerank

Ray

- Distributed asynchronous algorithms
- Compute heavy workloads
- Ex: Reinforcement learning

Spark

```python
text_file = sc.textFile("hdfs://...")

counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://...")
```

Ray

```python
@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

Spark

```python
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(range(0, NUM_SAMPLES)) \
             .filter(inside).count()

print("Pi is roughly %f" % (4.0 * count / NUM_SAMPLES))
```

Ray

```python
@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

Spark

```
# Counts people by age
countsByAge = df.groupBy("age").count()

# Saves countsByAge to S3 in the JSON format.
countsByAge.write.format("json").save("s3a://...")
```

Ray

```
@ray.remote
def f(x):
    return x * x

futures = [f.remote(i) for i in range(4)]
print(ray.get(futures)) # [0, 1, 4, 9]
```

Spark

```python
# Set parameters for the algorithm.
# Here, we limit the number of iterations to 10.
lr = LogisticRegression(maxIter=10)

# Fit the model to the data.
model = lr.fit(df)

# Given a dataset, predict each point's label.
model.transform(df).show()
```

Ray

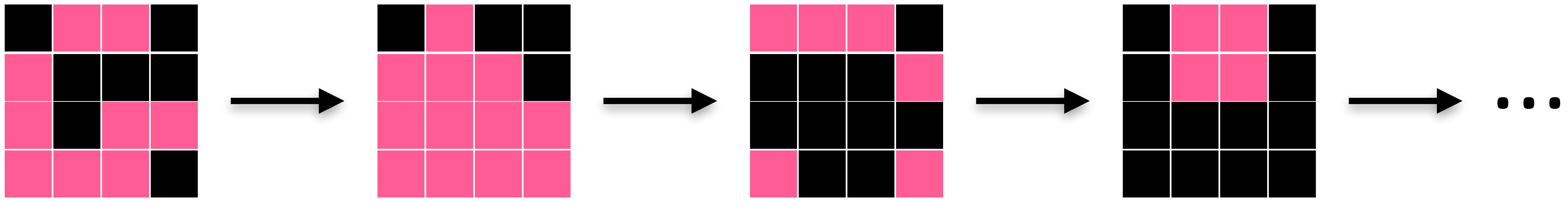
```python
@ray.remote
class DataWorker(object):
    def __init__(self):
        self.model = ConvNet()
        self.data_iterator = iter(get_data_loader()[0])

    def compute_gradients(self, weights):
        self.model.set_weights(weights)
        data, target = next(self.data_iterator)

        self.model.zero_grad()
        output = self.model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        return self.model.get_gradients()
```

Differences from Spark

- No built in primitives for partitioning data across cluster.
- Stateful computation with Actors.
- Asynchronous execution.

Agent Based Modeling

Dead cell
Live cell

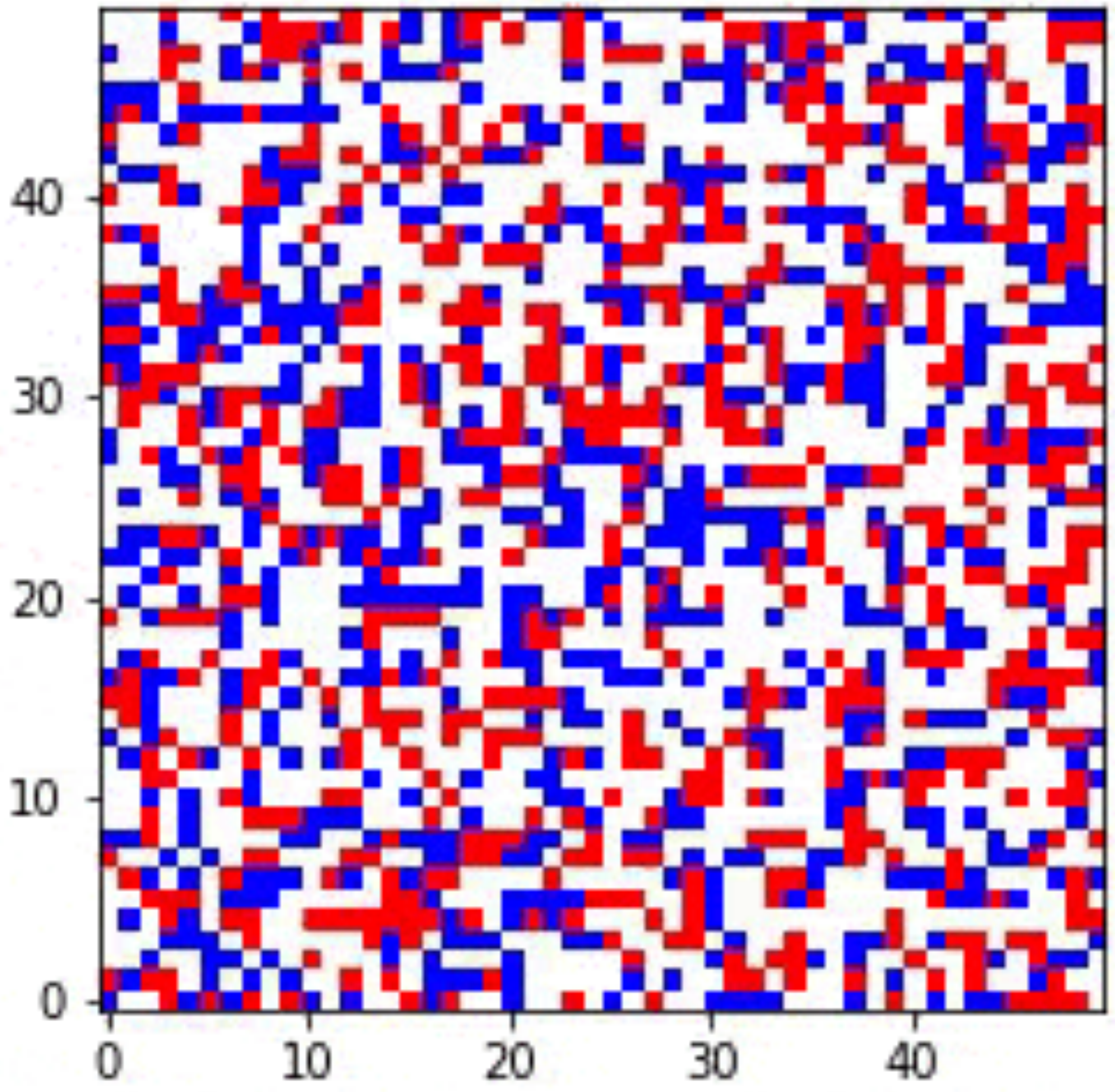


1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

Schelling’s Model of Segregation

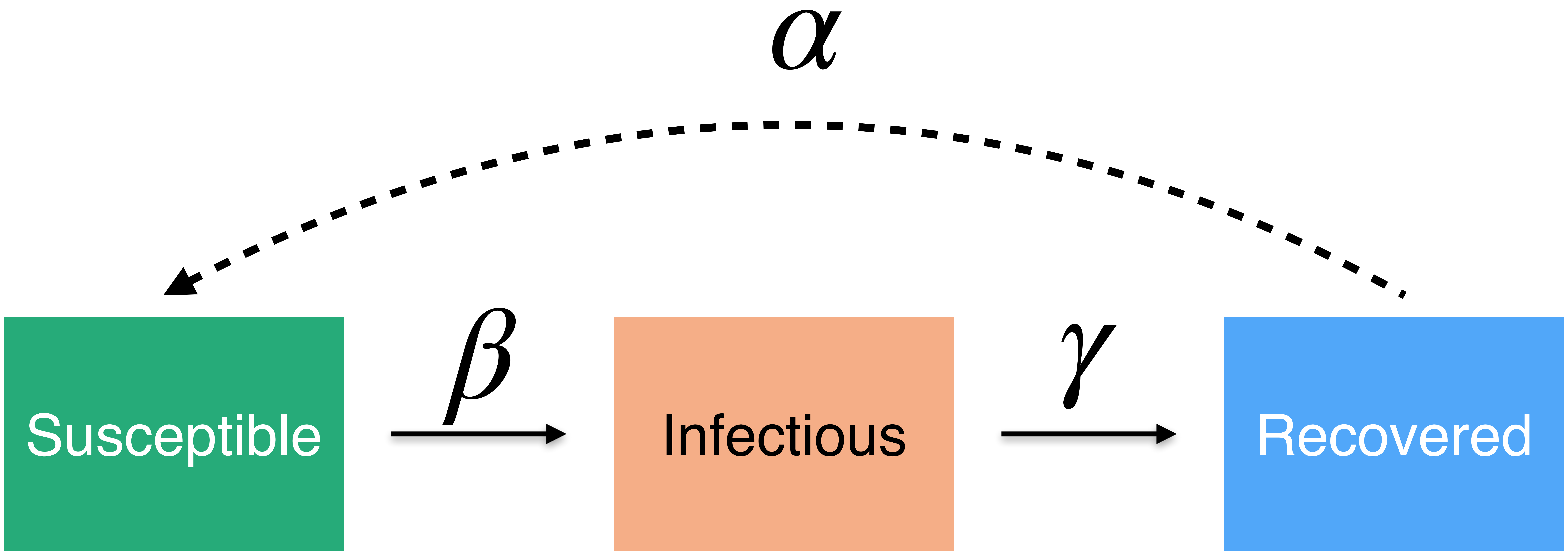
- Agents belongs to one of two groups (0 or 1).
- The agents live in a two-dimensional grid world.
- Agents with at least three neighbors in the same group are happy.
- Agent keeps moving one step to new locations until happy.

Schelling's Model of Segregation



Source: Blaqdolphin, CC BY-SA 4.0, via [Wikimedia Commons](https://commons.wikimedia.org)

SIR Epidemic model



S(t): susceptible individuals who have not yet been infected at time t
I(t): number of infectious individuals at time t
R(t): number of individuals who have recovered (and are immune) at time t

Pearson

SIR Epidemic model

- Agents belongs to one of three groups (S, I, or R).
- Agents only interact with other agents in their neighborhood.
- An infected agent passes disease to a susceptible one with probability β
- An infected agent recovers with probability γ