Parallel Simulations

In the current agent based model, the agents update their infection status and perform movements according to their daily routine in a sequential manner. Both of these operations utilize the simulation's read buffer to calculate required values that are based on the agent's neighborhood. The result of the movement of each agent is reflected in the write buffer. Thus, this procedure requires each agent to perform read operations on the read buffer and write operations on the write buffer. At the end of each time step, the buffers are swapped.

We modify this procedure to allow for agent's to perform their operations parallely. There are no dependencies between agents and the order of execution does not matter. Hence a data-parallel approach can be implemented in which each thread executes a single agent's operations and calculates its updates simultaneously.

Since the buffers are maps (mapping locations to agents), updating them requires collisions to be handled. In a serial implementation, this is straightforward - if a location is already occupied by an agent due to an update in some previous iteration, then the current agent remains at its original location. In a parallel implementation, handling collisions and collecting the results can be done via two methods:

M1: Concurrent data structures with double buffering:
In this approach the buffers are concurrent data structures which allow multiple threads to share the buffer and write to it. Hence, for each agent, each thread can write the agent's updates to the write buffer concurrently.

In Rust, we use DashMap for this purpose. DashMap allows easy collision handling by allowing the following two steps to be done atomically: check if the entry at a certain key is empty and if empty, insert the given value.

The efficiency of this method depends on the efficiency of reads and writes in the concurrent data structure. In DashMap, we found that the time saved by writing updates in parallel was overshadowed by the excess time introduced in reading from a DashMap. This is due to the sharding nature of the data structure that enables concurrent writes, but makes simple reads slow.

M2: Concurrent data structures with double buffering, with "read only view" for read operations:

To overcome the problem of excess time in reading from a DashMap, we maintain a "read only view" of the map, which acts as a regular non-concurrent map. All read operations are performed on this map instead. However, now there is the added cost of updating the "read only view" at each step of the simulation.

M3: Map-reduce approach:
In this approach, we collect the updates calculated in parallel by each agent in a temporary data structure. Then the updates are iterated over serially and reflected in the write buffer. Collisions are handled one by one in sequential order, the same as in serial implementation.

This method allows us to use regular data structures without compromising on read operation speeds however, the agent loop is not parallelized completely.

/

We observe that reflecting the updates in serial is faster than using concurrent data structures in Rust.

Table: *iterations/sec* measured for 1 million agents, averaged over 5 simulations.
Values stored in : epirust_timing (sheet: benchmarks)

Method	Serial	Parallel (8 cores)	Parallel (32 cores)	Parallel (64 cores)
M1	**1.17723942**	**2.68863944**	**4.85818212**	**5.18984214**
M2	**1.1334596**		**4.2243108**	**4.23892338**
M3	**1.82072374**	**3.96251958**	**4.68030536**	**4.77848274**

Observations:

Using a concurrent data structure like DashMap is beneficial when running parallel code at higher number of cores, however the serial code using this method performs poorly compared to using a map-reduce approach.
Note to self: how to mention the additional meta data about agent that needs to be updated in data structures (such as “listeners”) which are not concurrent.