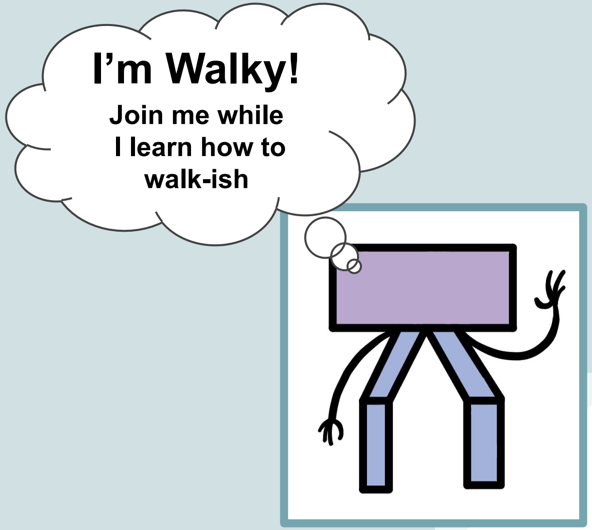
Using a Genetic Algorithm to Teach an Agent How to Walk

May Soshi, Ricky Williams,
Omar Abdel Haq, & Elie Eshoa



What is a Genetic Algorithm? Why Use One?

Genetic algorithms (GA) are a class of optimization algorithms inspired by the natural selection and genetic processes observed in biological evolution

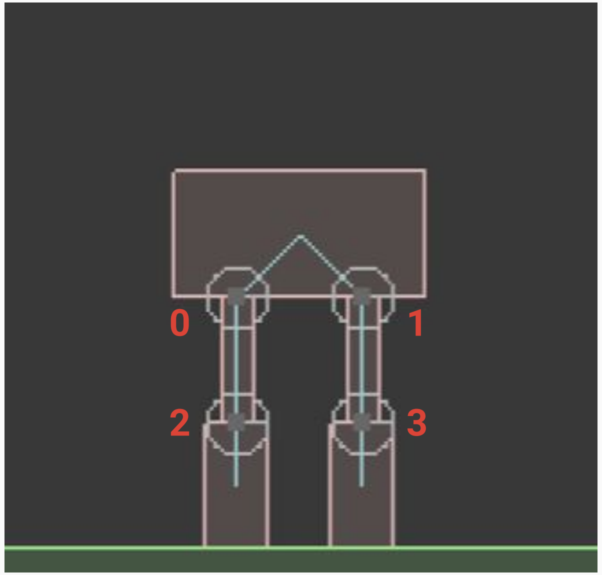
Using evolution as a computational tool can elucidate unconventional, high-performing solutions in some cases

What is a Walker?

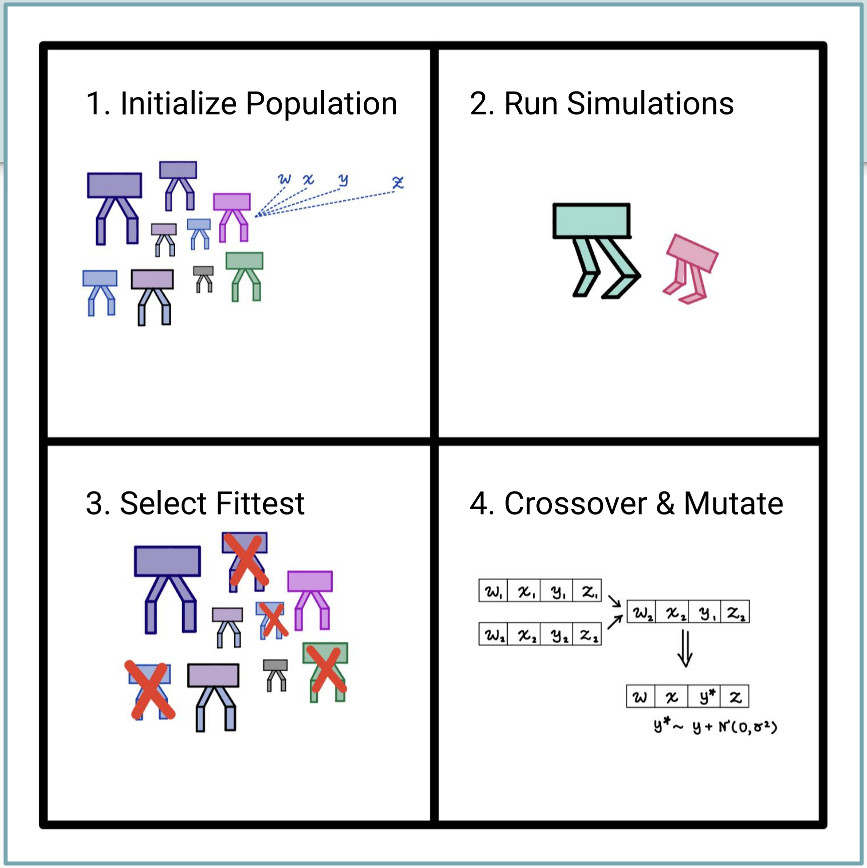
Box2D is a 2D rigid body simulation library

A Walker consists of

- (1) Head body
- (2) Upper Leg bodies
- (2) Lower Leg bodies
- (4) Joints

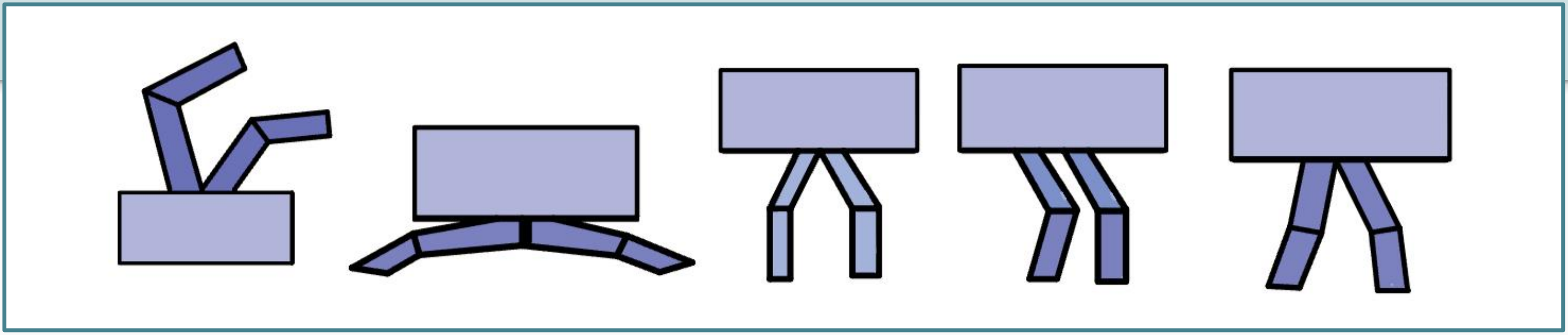


Steps of Our Genetic Algorithm



1. Initialize a population of walkers with different, randomly selected torques
2. Run
3. Select the fittest
4. Crossover and mutate
5. Repeat steps 2-4

A Full Simulation: A Collection of States

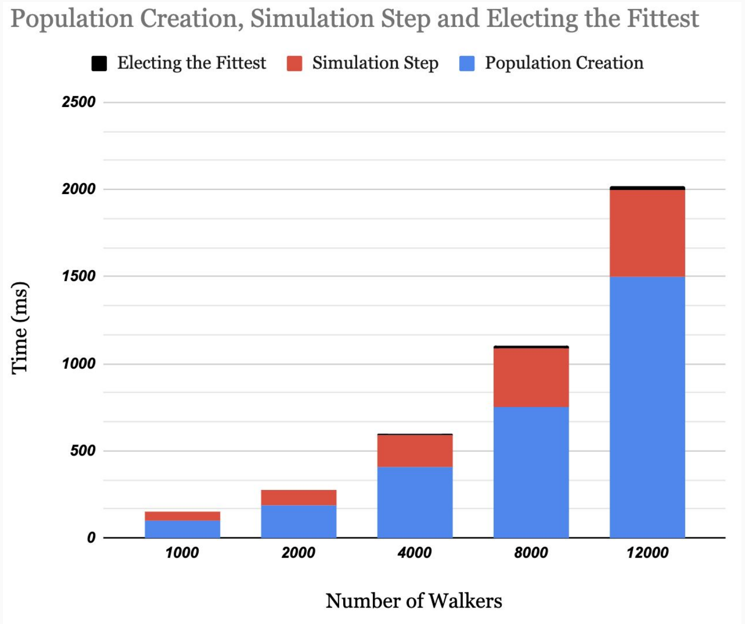


A full simulation comprises of running the genetic algorithm *g* times for *g* generations. After a full simulation, we visualize the progress of the best walker using Box2D.

How Can We Improve?

1. Parallelize Current Serial Implementation: **OpenMP**

2. Increase Problem Size: **MPI**



Parallelization I: OpenMP and Scaling Analysis

Using OpenMP, parallelization was done as discussed in MS4

As for our scaling analysis, we have used a standard job during our project development with a population size of 2000. First, we used that to calculate our serial fraction ***f***. To compute it, we follow the following steps:

- Assumptions: the code has 3 main (potentially) parallelizable components that depend on the size ***n***:
    - Creation, Simulation, and Electing the fittest walkers
- Taking the ratio of the walltime of running the remaining parts over the total walltime can experimentally help us approximate ***f*** , which ended up being around ***~0.04***

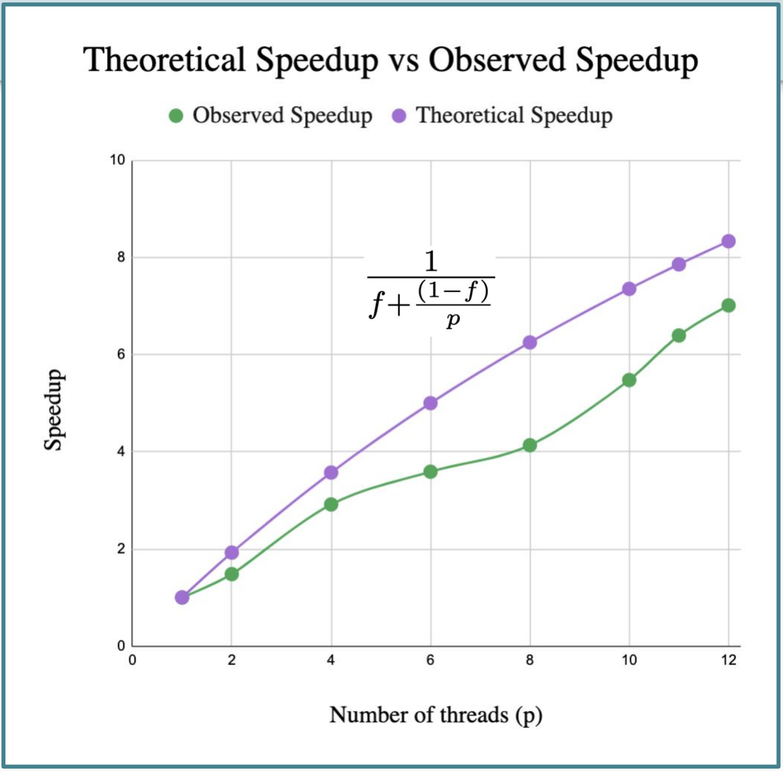
Parallelization I: OpenMP and Strong Scaling

We measured the speedup of the algorithm as a function of the number of threads, while keeping the problem size fixed at 2000. We ran it for 500 iterations.

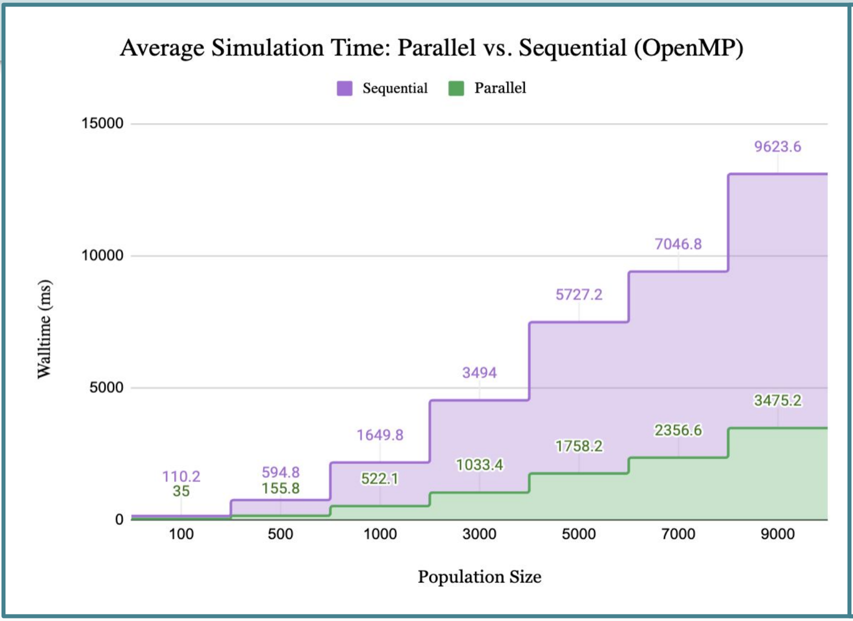
We compare our speedup to the theoretical speedup (***f = 0.04***)

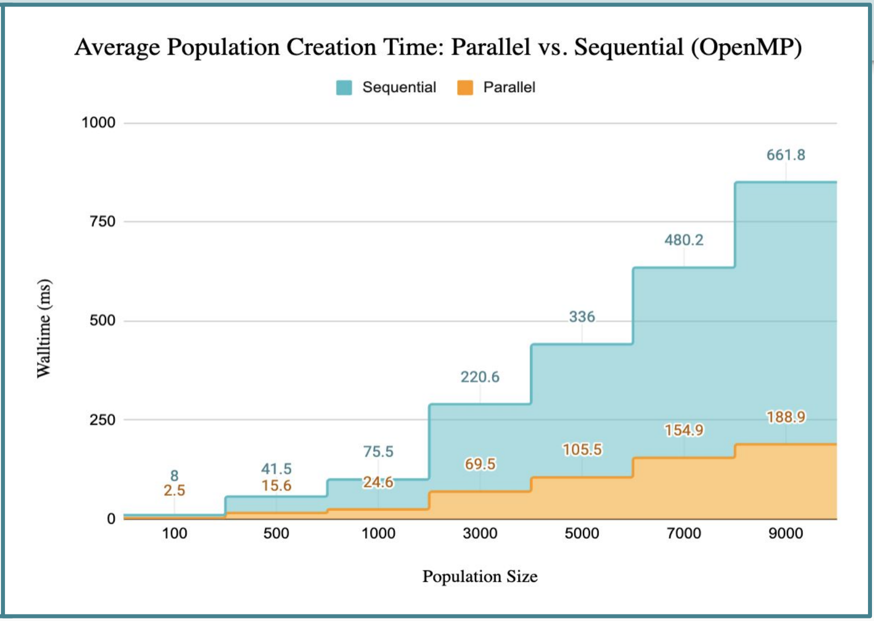
Number of threads	Speedup	Theoretical
12	7.01	8.34
11	6.39	7.86
10	5.47	7.36
8	4.13	6.25
6	3.58	5
4	2.91	3.58
2	1.48	1.93
1	1	1

$$\frac{1}{f+\frac{(1-f)}{p}}$$



Weak Scaling Growth is linear *n ~ p*





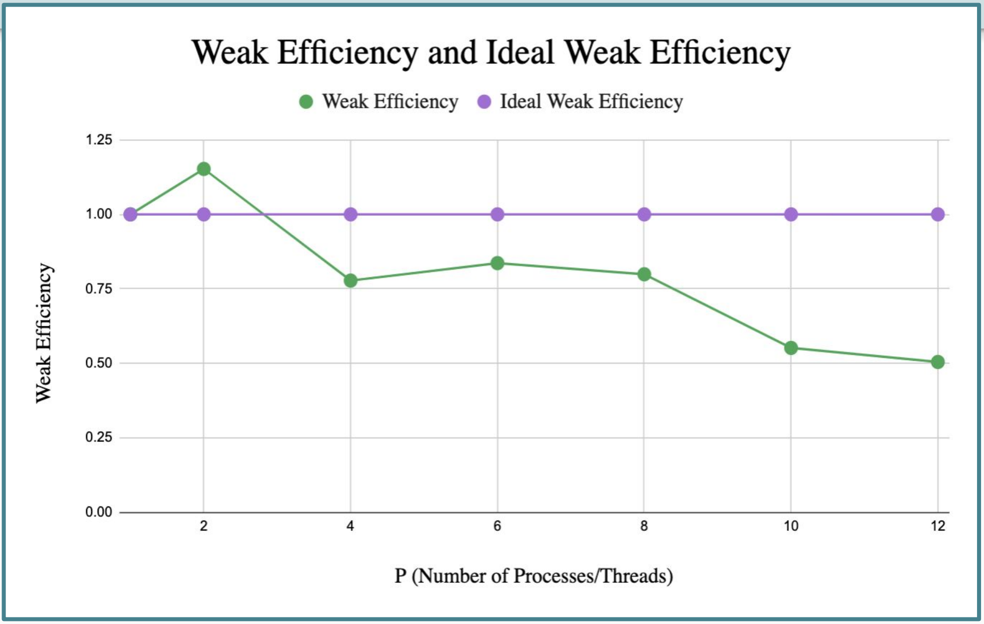
As we see in the graphs above, the time needed on average (i.e. a timestep in our algorithm) increases linearly with the size of the problem (number of walkers, ***n***)

Parallelization I: OpenMP and Weak Scaling

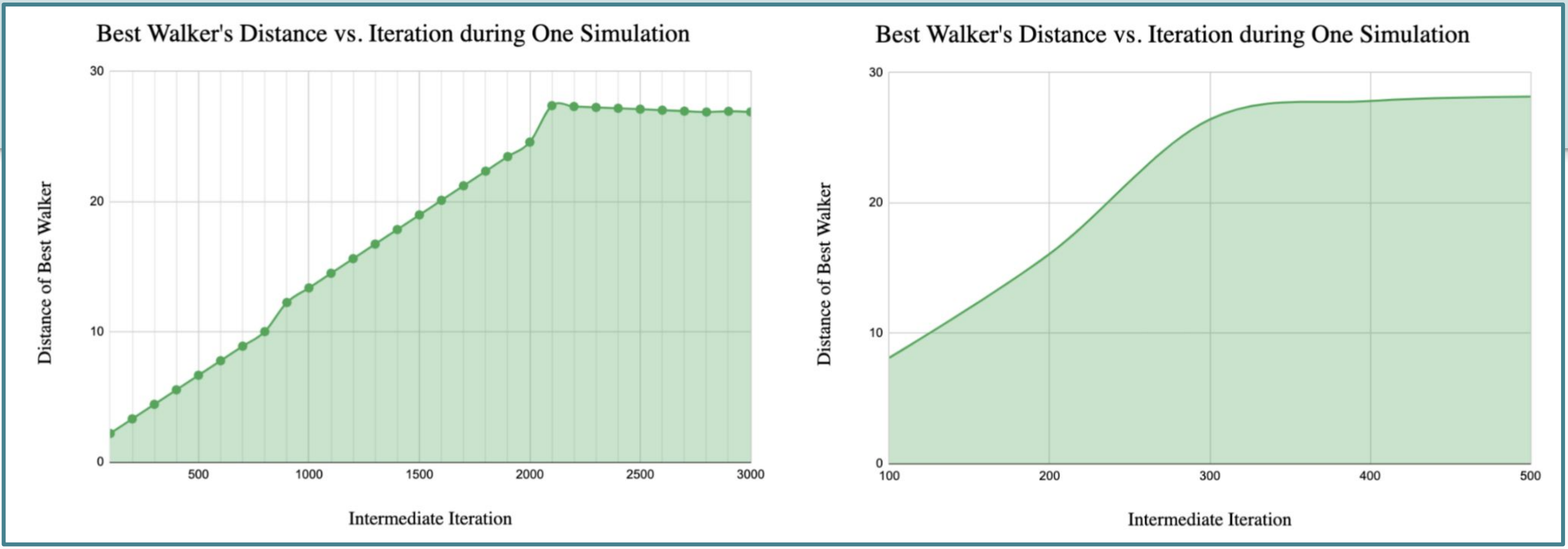
So, we ran 150 iterations (generations) per simulation. We ran simulations with upto ***p = 12*** threads and measured the weak efficiency ***Ep***. Since the relationship is linear, we used the following growth pattern:

n = 200, 400, . . . , 2400 using ***p = 1, 2, . . . , 12*** threads respectively.

The weak efficiency does not stay great because of the probable *overhead* introduced when scaling over multiple threads. This could be explained because the *third-party libraries* may be interfering with that, in addition to higher load imbalance throughout the runs.



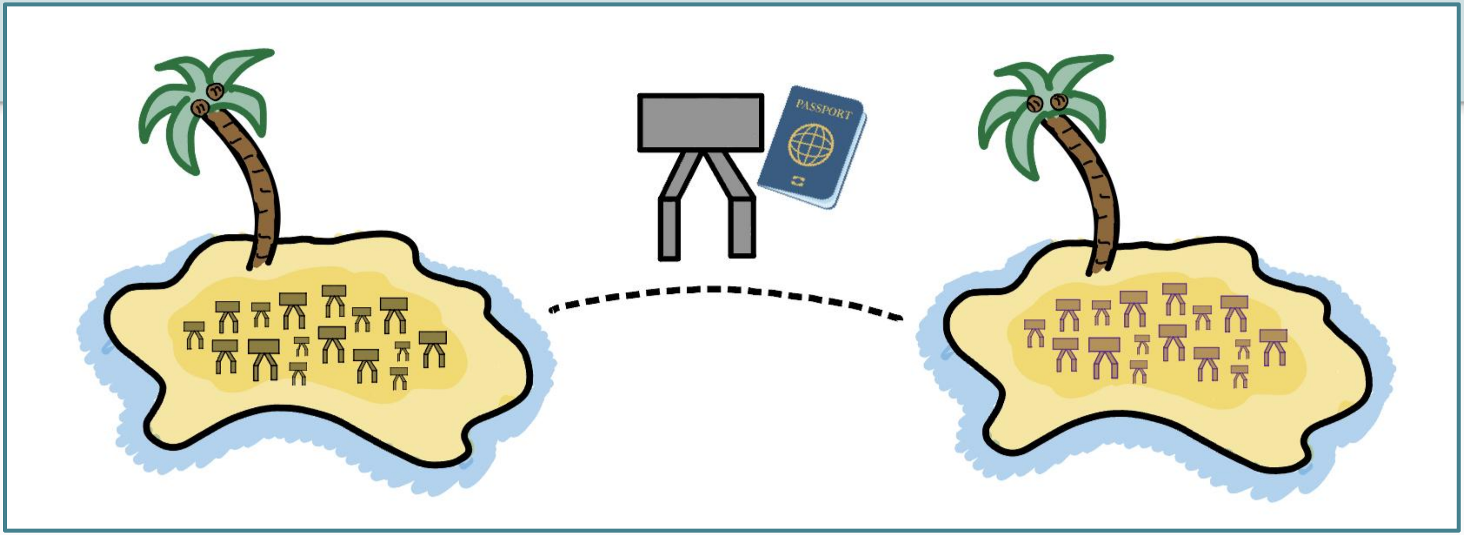
More Populations >> More Generations



Interestingly, we observed a consistent ***plateauing*** behavior in the course of our experiments.

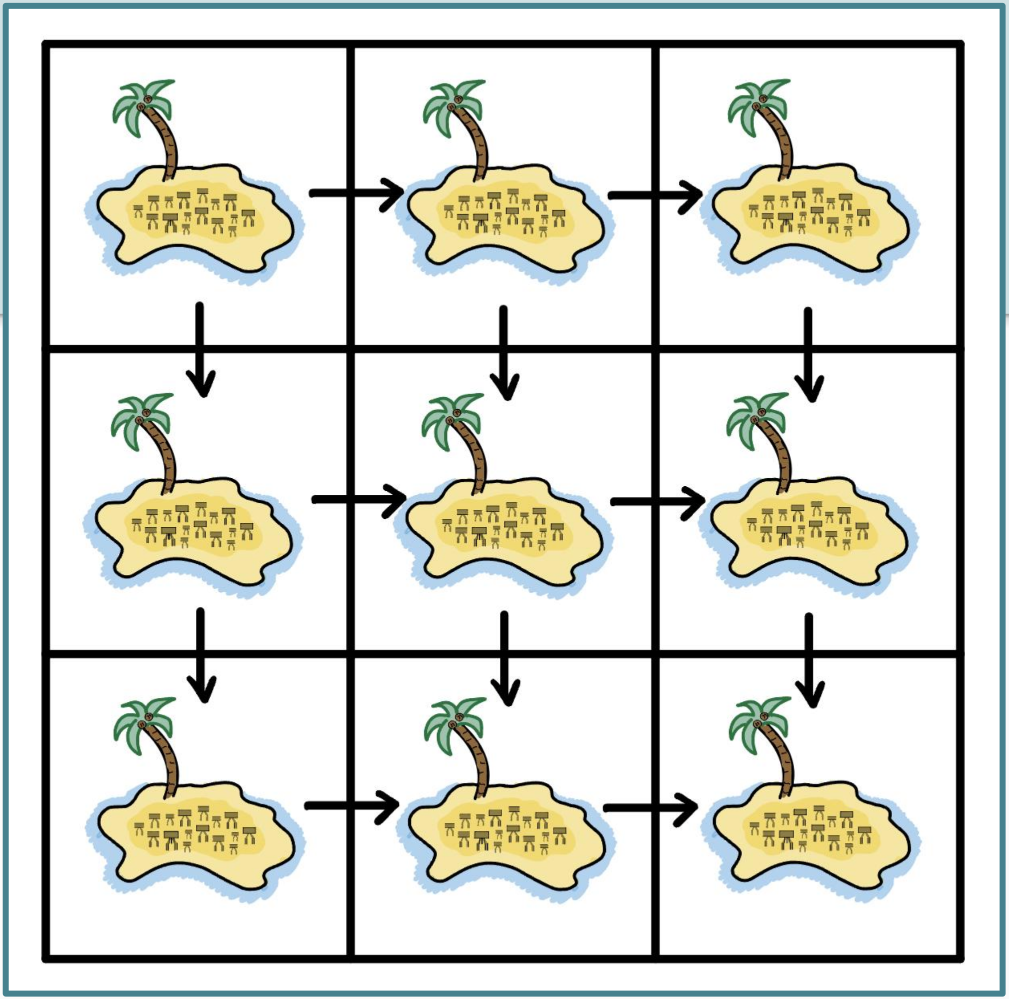
This trend can be attributed to the fact that we generate populations at random and persist with them until they can no longer generate improved generations of walkers, limited by genetic mutations. Our investigations have indicated that this plateauing phenomenon is present across all population sizes that we examined. Thus, initializing more random populations is more integral than running a specific population for an increased number of iterations (generations)

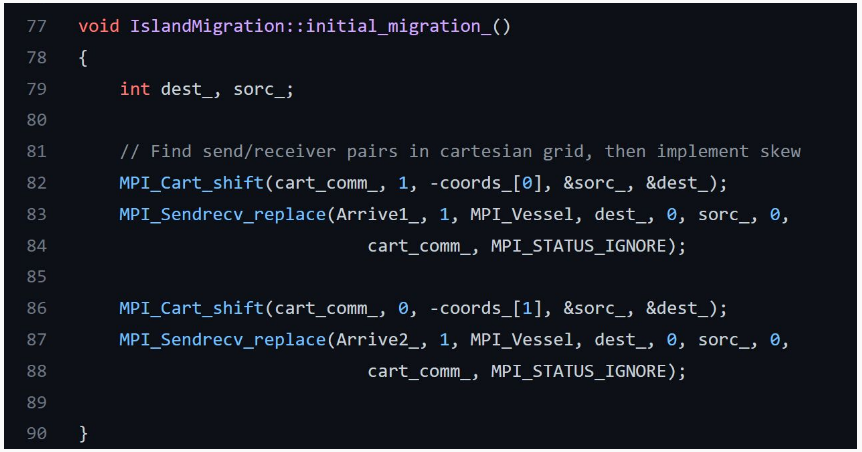
Parallelization II: MPI



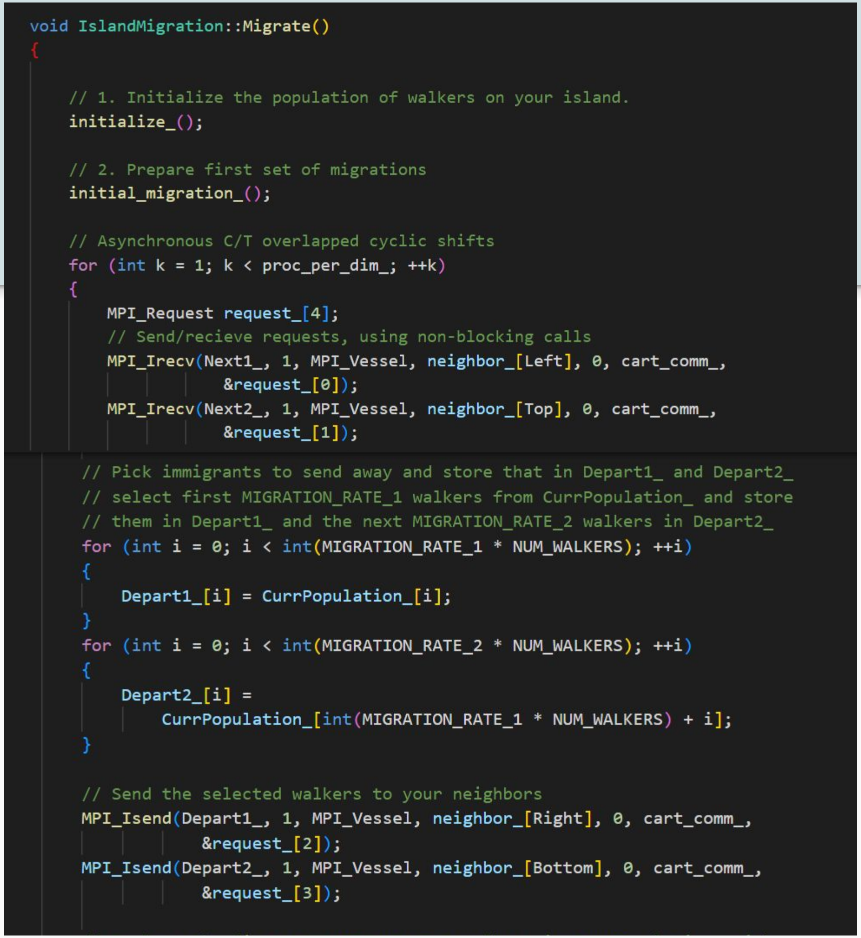
What if instead of having one population evolve over generations, we have tens of different populations, each on their own "island"?

A Cartesian Topology

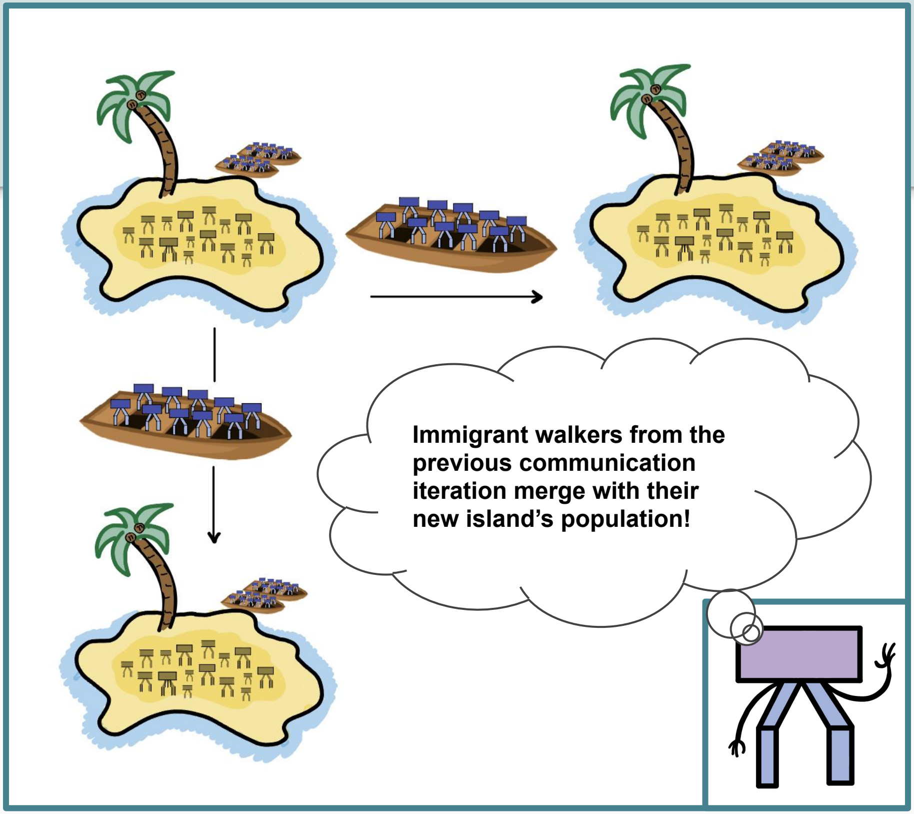




```cpp
void IslandMigration::initial_migration_()
{
    int dest_, sorc_;

    // Find send/receiver pairs in cartesian grid, then implement skew
    MPI_Cart_shift(cart_comm_, 1, -coords_[0], &sorc_, &dest_);
    MPI_Sendrecv_replace(Arrive1_, 1, MPI_Vessel, dest_, 0, sorc_, 0,
                         cart_comm_, MPI_STATUS_IGNORE);

    MPI_Cart_shift(cart_comm_, 0, -coords_[1], &sorc_, &dest_);
    MPI_Sendrecv_replace(Arrive2_, 1, MPI_Vessel, dest_, 0, sorc_, 0,
                         cart_comm_, MPI_STATUS_IGNORE);

}
```



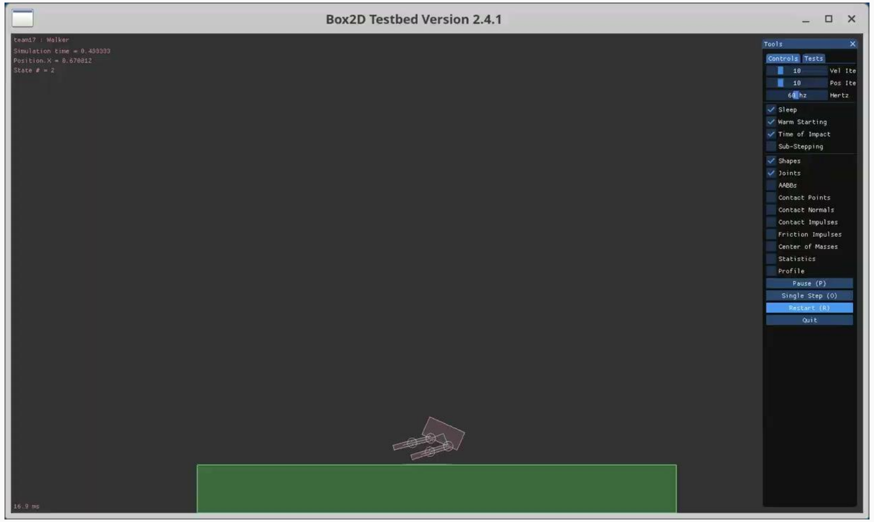
```cpp
void IslandMigration::Migrate()
{

    // 1. Initialize the population of walkers on your island.
    initialize_();

    // 2. Prepare first set of migrations
    initial_migration_();

    // Asynchronous C/T overlapped cyclic shifts
    for (int k = 1; k < proc_per_dim_; ++k)
    {
        MPI_Request request_[4];
        // Send/recieve requests, using non-blocking calls
        MPI_Irecv(Next1_, 1, MPI_Vessel, neighbor_[Left], 0, cart_comm_,
                    &request_[0]);
        MPI_Irecv(Next2_, 1, MPI_Vessel, neighbor_[Top], 0, cart_comm_,
                    &request_[1]);
        // Pick immigrants to send away and store that in Depart1_ and Depart2_
        // select first MIGRATION_RATE_1 walkers from CurrPopulation_ and store
        // them in Depart1_ and the next MIGRATION_RATE_2 walkers in Depart2_
        for (int i = 0; i < int(MIGRATION_RATE_1 * NUM_WALKERS); ++i)
        {
            Depart1_[i] = CurrPopulation_[i];
        }
        for (int i = 0; i < int(MIGRATION_RATE_2 * NUM_WALKERS); ++i)
        {
            Depart2_[i] =
                CurrPopulation_[int(MIGRATION_RATE_1 * NUM_WALKERS) + i];
        }

        // Send the selected walkers to your neighbors
        MPI_Isend(Depart1_, 1, MPI_Vessel, neighbor_[Right], 0, cart_comm_,
                    &request_[2]);
        MPI_Isend(Depart2_, 1, MPI_Vessel, neighbor_[Bottom], 0, cart_comm_,
                    &request_[3]);
```


Immigrant walkers from the previous communication iteration merge with their new island's population!

Non-Blocking Communication

We create two buffers **Next1_** and **Next2_** into which other islands send walkers. While we wait on the arrival of new walkers, we merge the old immigrants in **Arrive1_** and **Arrive2_** into our general population and go through a preselected number of generations. This is how we hide the communicated overhead.

Visualizations



```
team17 : Walker
Simulation time = 0.433333
Position.X = 0.670812
State # = 2
```

```
10      Vel Ite
10      Pos Ite
60 hz   Hertz
```

- ✓ Sleep
- ✓ Warm Starting
- ✓ Time of Impact
- Sub-Stepping
- ✓ Shapes
- ✓ Joints
- AABBs
- Contact Points
- Contact Normals
- Contact Impulses
- Friction Impulses
- Center of Masses
- Statistics
- Profile

100 walkers; 150 gen. (75s); fittest 2 per gen.

4000 walkers; 150 gen. (75s); fittest 2 per gen.