

Kimia Taba and Vincent Hasse

Predator-Prey Relationship in a Closed System

Table of contents

- 1 Our Project
- 2 Sequential Approach
- 3 Parallel Approach
- 4 Benchmarking
- 5 Conclusion

The Idea

- create a simulated environment
- plants, prey, hunters
- apply some machine-learning for prey- and hunter-brains -> Python

Outline

■ World: NumPy-Array, int8

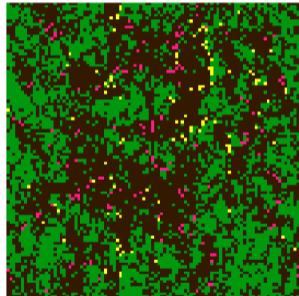
- ▶ 0: Free
- ▶ 1: Plant
- ▶ 2: Prey
- ▶ 3: Hunter
- ▶ ... 255 different agents possible
- ▶ initialized randomly at the beginning of the simulation, based on config-file

■ Agents: Prey and Hunters

- ▶ can move, eat, breed, die

■ Plants: are just dead or alive, spawn randomly every simulation-step

Day 37 of 500
Plants: 4000
Preys: 111
Hunters: 158



Pseudo-Code

Sequential-Simulation in Pseudo-Python

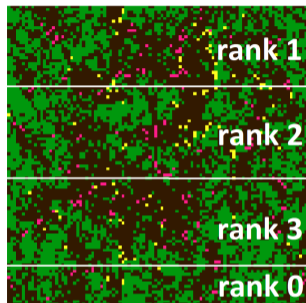
pseudo-python

```
1  for step in range(config.SIMUALTION_STEPS):
2      for everyAgent in dicts:
3          addAge()
4          moveAgent()
5              letItDie(energy, age)
6              letItEat(isFoodAtNewPosition)
7              letItBreed(energy)
8      spawnNewPlants()
```

based on a dict of hunters and a dict of preys: (ID -> Animal)

General Idea for the Parallel Approach

- Split the world in (even) chunks
- Broadcast world state and Hunter / Prey dicts to every process
- Each process calculates changes in the respective part of the world
- Report changes to rank0
- Rank0 gathers changes, creates new world and dicts and broadcasts for next simulation step



Problems and Solutions

- Problem: Pickeling & sending of Hunter/ Prey dictionaries is costly
- Solution: don't broadcast the dicts
 - ▶ Change dicts from (AnimalID -> Animal) to (position -> Animal)
 - ▶ Report back a numpy array with old Positions (as identifier), new Positions
 - ▶ -> Broadcast world-state from rank0
 - ▶ -> Send changes-array back to rank 0
 - ▶ Only communicate NumPy Arrays
- Drawback: rank0 has to do all the updating (world, dicts)

Parallel movement calculation

Movement-Calculation in Python; first part

```
1  #create lists
2  hunters = []
3  preys = []
4  #loop dependent on rank
5  for x in range((rank-1)*numRows, rank*numRows):
6      for y in range(config.WORLD_SIZE):
7          if world[x][y] == config.IDENT_PREY:
8              preys.append((x,y))
9          elif world[x][y] == config.IDENT_HUNTER:
10             hunters.append((x,y))
11
12 #numslice: number of possible agents in slice +1
13 changes = np.zeros((numSlice, 4), dtype=changes_dtype)
14 changeCounter = 0
```


Parallel movement calculation

Movement-Calculation in Python; second part

```
1  for p in preys:
2      #loop through list and calculate new position
3      path = helper.calculateSteps(p, world, config.IDENT_PLANT)
4      if path != None:
5          changeCounter += 1
6          newPos = (path[1][0], path[1][1])
7          changes[changeCounter][0] = p[0]      #save old...
8          changes[changeCounter][1] = p[1]
9          changes[changeCounter][2] = newPos[0] #and new position in array
10         changes[changeCounter][3] = newPos[1]
11     #do the same for hunters
12     #update function only needs to run on the part of the array that...
13     changes[0][0] = changeCounter              #...actually contains changes
14     comm.Send(changes, dest=0)
```

Problems and Solutions

- Problem: Rank0 has to apply all calculated changes
- (Partial) Solution: Already apply changes of fastest process while waiting for callback of the other ranks

rank0 Receive

```
1  if rank == 0:
2      #process changes from other ranks
3      while received_data < numReceives:
4          changes = np.zeros(**_in respective size**)
5          #wait for fastest callback
6          comm.Recv(changes, source=MPI.ANY_SOURCE, status=status)
7          #apply changes from data to world
8          world = sWorld.updateWorld(changes)
9          received_data += 1
10     sWorld.finishStep()
```

Problems and Solutions

- Problem: When the world gets larger, also the broadcasted arrays (world, changes) get larger
- Solution: No solution, it's implicit to the implementation :(
→ performance limits?

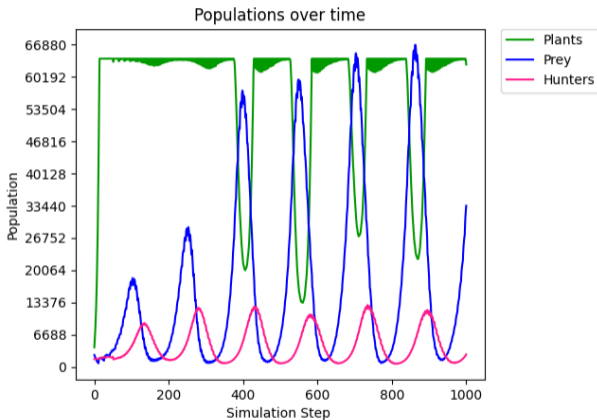
However: keep the memory size of communicated arrays as small as possible:

rank0 Receive

```
1  # smallest possible datatype for changes array (can be further optimized)
2  if config.WORLD_SIZE * numRows > 255:
3      changes_dtype = np.uint16
4      if config.WORLD_SIZE * numRows > 65535:
5          changes_dtype = np.uint32
6          if config.WORLD_SIZE * numRows > 4294967295:
7              changes_dtype = np.uint64
8  else: changes_dtype = np.uint8
9  #world:10.000 x 10.000; uint8 ~ 95,4MB; changes with n = 50; uint ~7,6MB
```

Does it work

Yes. Populations are as to expect according to the Lotka-Volterra Laws:

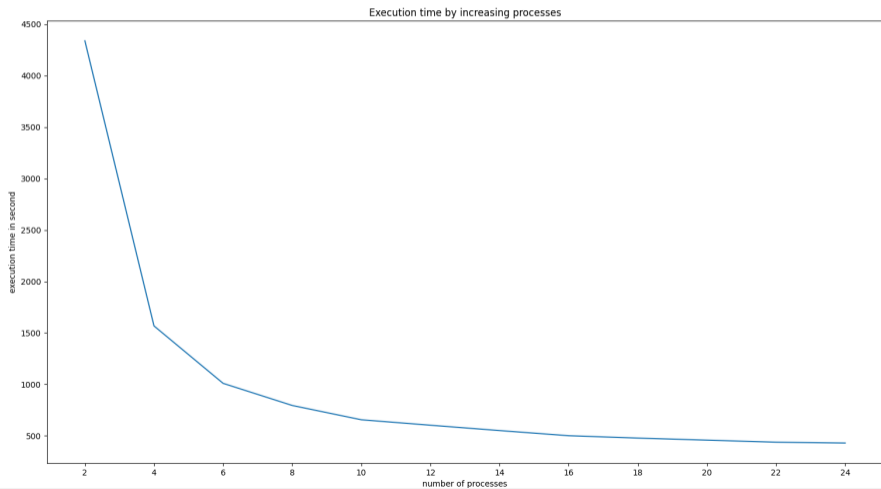


Base Config

- World Size=1000, Simulation Steps=100
- Without Paralleling => World Size = 500 , 10 simulation steps => 47 minutes

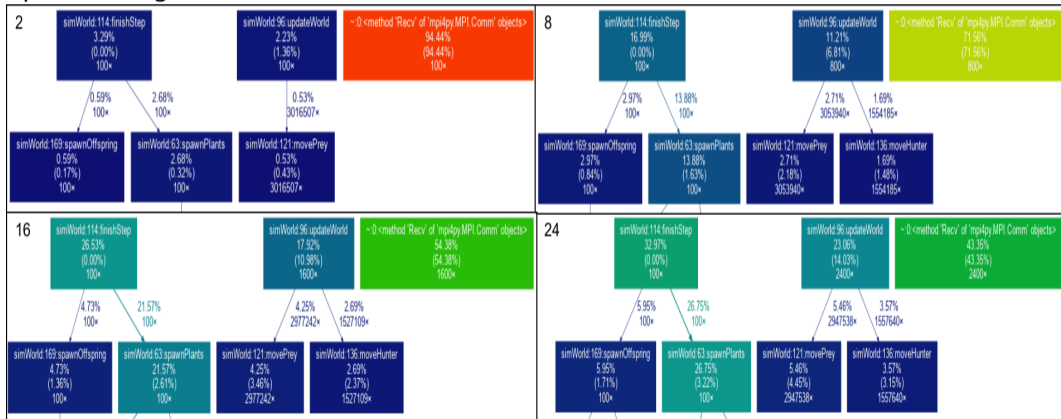
Strong Scaling

Keeping the problem size, increase parallelism



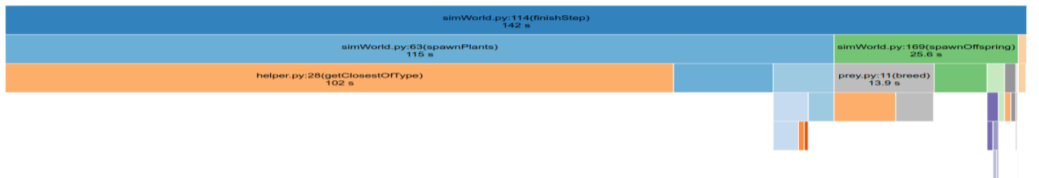
Application Performance

Cpu Profiling - rank 0



Application Performance

Visualizing trace - Snakeviz

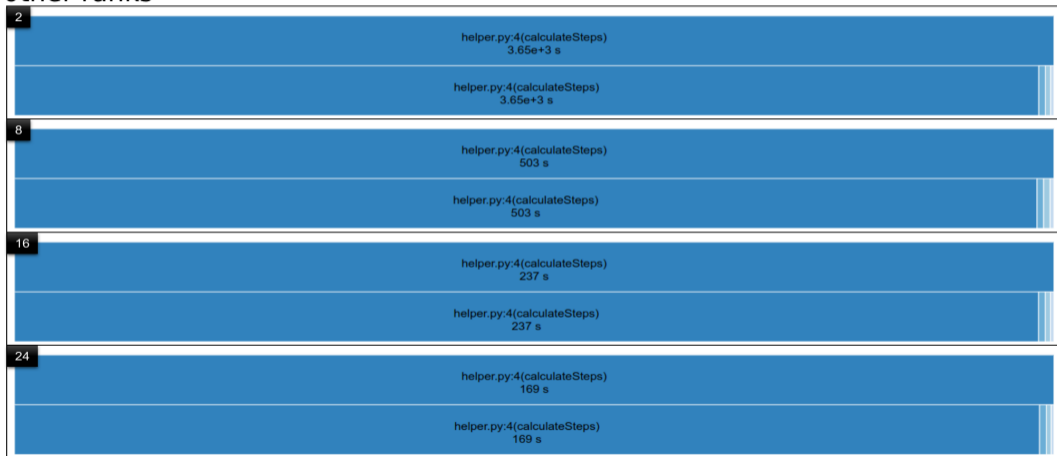


Search:

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
2400	186.7	0.07778	186.7	0.07778	~0(<method 'Recv' of 'mpi4py.MPL.Comm' objects>)
2385217	102.3	4.29e-05	102.3	4.29e-05	helper.py:28(getCloseOfType)
2400	60.41	0.02517	99.31	0.04138	simWorld.py:96(updateWorld)
2947538	19.17	6.503e-06	23.52	7.979e-06	simWorld.py:121(movePrey)
100	13.85	0.1385	115.2	1.152	simWorld.py:63(spawnPlants)
1557640	13.57	8.71e-06	15.38	9.872e-06	simWorld.py:136(moveHunter)
100	7.346	0.07346	25.61	0.2561	simWorld.py:169(spawnOffspring)
8599856	5.251	6.106e-07	13.93	1.62e-06	prey.py:11(breed)
6487818	3.575	5.511e-07	8.448	1.302e-06	random.py:290(randrange)
6487818	3.504	5.401e-07	4.872	7.51e-07	random.py:237(_randbelow_with_getrandbits)
4973284	2.077	4.176e-07	2.077	4.176e-07	animal.py:12(setPosition)

Application Performance

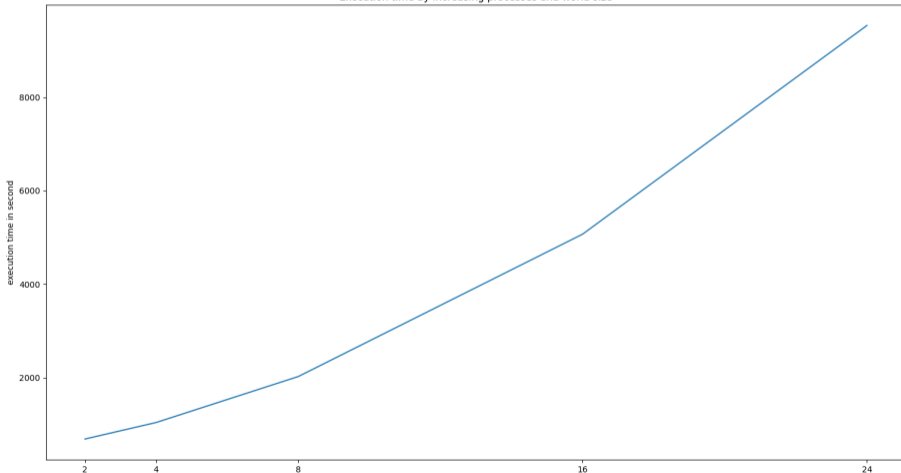
Other ranks



Weak Scaling

Increase the problem size with parallelism

Execution time by increasing processes and world size



Future Work

- Writing report
- Memory Profiling
- Change path finding algorithm

Conclusion

- Goal: Simulating a predator-prey relationship
- Achieve lots of Performance improvements by using mpi