Sugarscaping On a Beowulf Ring

New Mexico

Supercomputing Challenge

Final Report

April 7, 2010

Team 11

Artesia High School

Team Members:

Wesley Green

Isaiah Jordan

James McGee

Wen Hai Zheng

Teacher(s):

Randall Gaylor

Mentor(s):

Nick Bennett

Jose Quiroz

TABLE OF CONTENTS

INTRODUCTION

The idea of running Sugarscape in parallel on a Beowulf ring is something that has not been attempted to our knowledge and based on our research. The idea is not only intriguing but it is also a very difficult feat to attempt to run the Epstein model of Sugarscape in parallel on a Beowulf ring. This project is something that the team felt would be a successful project to work on because it combined our interests in parallel computing and agent-based modeling. What exactly is a Beowulf ring? What exactly is a Sugarscape model? How exactly do these two concepts work together? Team 11, comprised of 4 students has worked out one way to run a Sugarscape model parallel with a Beowulf ring. For our team, boundaries for what we know have extended and expanded.

We have converted the Sugarscape model to Python in order to properly run it in parallel on the Beowulf ring. This required breaking down the NetLogo version of Sugarscape into a pseudo-code, [21] then using the pseudo-code, rewriting the program in Python. This activity made the Sugarscape easier for us to understand, and made the conversion to Python much easier.

Python was selected after attempting to use Java as the programming language for use on the Beowulf ring, and discovering (with help from our mentor) that Python would be far less cumbersome.

PROGRESS

At this point in time, we have a modified version of the Sugarscape model that is listed in the Model Library on the NetLogo website. This model, now written in Python, may be viewed in Appendix A [11] The plan is to run the modified model on the Beowulf ring to establish a base line for performance. We then intend to further modify the program to allow Sugarscape to perform in a more “life-like” [18] manner. [22] We will run the modified (more life-like) program on the Beowulf ring to determine whether other information can be acquired. [22]

BEOWULF RING

We have assembled a miniature Beowulf ring [1] composed of 7 Dell PCs with 1Ghz processors and 2 IBM Xeon servers, one being a dual-core. There is a total of 10 cores and about 2.5GB of available RAM. The nodes are connected through a 100Mbps mini-network with each node running ClusterKnoppix v3.6 with OpenMosix[5][6] installed natively. Each node runs Parallel Python. Sugarscape imports the Parallel Python[8] library before it runs and distributes the processing load evenly among the nodes. The idea of the Beowulf ring in this project is to make it so it is possible to run Sugarscape faster and (in theory) get more results than we could on a normal computer with one or two processors. The Beowulf ring also makes it possible to run Sugarscape for more iterations.

EXPECTED OUTCOMES

As explained earlier the goal of the project is to run a program known as a "Sugarscape" parallel with a mini super computer comprised of 7 connected computers. We are also optimizing the Sugarscape model in an attempt to isolate individual societies, one on each node. We anticipate that by running the Sugarscape in parallel on the Beowulf ring, we will get additional results faster than running the Sugarscape on a single computer. We will be able to:

1. Compare how societies evolve differently from the same simple rules on different nodes;
2. View the interaction of differently evolved societies as nodes are combined, and perhaps recombined;
3. Iterate the above to determine common traits within the societies;
4. As well as determine the range of varying traits within the societies.

SUGARSCAPE

What is Sugarscape? Sugarscape is an artificially agent-based social simulation. The first introduction to agent-based simulations since the early 1990's was in form of the Game of Life [13] by Cambridge mathematician John Conway. Conway's work wasn't in agent-based simulation, but in cellular automaton. Conway's work was an early example of what we now call “artificial life” (though he wasn't the first: John Von Neumann and others did some important work in that area before Conway.) Conway's invention was enhanced and applied to the arena of social simulations by Epstein in his book, Growing Artificial Societies. [2] Epstein's implementation came to be known as the Sugarscape.

Sugarscape refers to the silicon-based society that Epstein created. It included the agents which are the inhabitants, the environment and the rules governing the interaction of the agents with each other and the environment. This project is an adoption of their ideas with some modifications. The Sugarscape provides accurate data to Social Science researchers through its model of an artificial society. Agents or citizens move about the environment gathering food, mating with suitable partners, bearing offspring, dying, and leaving an inheritance for their survivors. Researchers can use the simulation as a test for basic social rules. The patterns resulting from the execution of the simulation can be used to confirm or revise their claims.

PYTHON

Python is one of the more successful programming language since it was created in 1991. It helped our team write the code for our project. Since Python is open-source, people within the Python community can write add-on libraries for it, allowing greater flexibility in programming and coding. One example of this is the Parallel Python [8] library our project uses. We had originally planned on using Java, but Python was selected after discovering (with help from our mentor) that Python would be far less cumbersome because Java couldn't be run in parallel very easily.

REFERENCES

[1] "Beowulf (computing) -." *Wikipedia, the free encyclopedia*. Web. 15 Sept. 2009. <http://en.wikipedia.org/wiki/Beowulf_(computing)>.

[2] Epstein, Joshua M. *Growing artificial societies: Social science from the bottom up*. Washington, D.C: Brookings Institution, 1996. Print.

[3] *The Linux Documentation Project*. Web. 15 Sept. 2009. <http://tldp.org/HOWTO/openMosix-HOWTO/>.

[4] "NetLogo User Community Models: Sugarscape." *The Center for Connected Learning and Computer-Based Modeling*. Web. 15 Sept. 2009. <http://ccl.northwestern.edu/netlogo/models/community/Sugarscape>.

[5] "OpenMosix -." *Wikipedia, the free encyclopedia*. Web. 15 Sept. 2009. <http://en.wikipedia.org/wiki/OpenMosix>.

[6] *OpenMosix, an Open Source Linux Cluster Project*. Web. 15 Sept. 2009. <http://openmosix.sourceforge.net/>.

[7] "The openMosix Stress-Test." *OpenMosixview cluster-management GUI*. Web. 15 Sept. 2009. <http://www.openmosixview.com/omtest/>.

[8] Parallel Python. N.p., n.d. Web. 5 Apr 2010. <http://www.parallelpython.com/>.

[9] Random stuff from the past. Web. 17 Oct. 2009. <http://wspinell.altervista.org/index.php?section=20_sugarscape>.

[10] "StarLogo Sample Projects - Sugarscape." Web. 20 Sept. 2009. <http://education.mit.edu/starlogo/samples/sugarscape.htm>.

[11] Sugarscape - Growing Agent-based Artificial Societies. Web. 18 Sept. 2009. <http://sugarscape.sourceforge.net/>.

[12] "SugarScape." Ressources naturelles et simulations multi-agents. Web. 22 Sept. 2009. <http://cormas.cirad.fr/en/applica/sugarScape.htm>

[13] Gardner, M. "Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life" - M. Gardner – 1970." *Mathematical Games*. Scientific American, Oct 1970. Web. 5 Apr 2010. <http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm>.

APPENDIX A: PYTHON SUGARSCAPE CODE

```
# Sugarscape1

# sugarscape: (two-peak sugarscape, rule: Ginf)

# agents: (moving sequence: random, view: four directions, rule: M)

# agents are characterized by their 'ifAlive', 'vision', 'metabolism', 'wealth'

# Change: modified how agents are identified, and neighbor fn

    so agents are able to share a location


from numpy import *

import matplotlib.pyplot as plt

import pp


# Initialize two-peak sugarscape and display

def initsugarscape(nruns, size, maxsugar):


    # Generate sugarscape with one south west peak

    x = range(-int(math.ceil(0.75*size)), size - int(math.ceil(0.75*size)))

    y = range(-int(math.ceil(0.25*size)), size - int(math.ceil(0.25*size)))

    s1 = zeros((size,size))


    for i in range(size) :

        for j in range(size) :

            if x[i] == 0 and y[j] == 0:

                s1[i,j] = maxsugar

            else:

                s1[i,j] = maxsugar/(abs(x[i]) + abs(y[j]))
```

```
 # Generate two-peak sugarscape (south west and north east peak)
   sugarscape = s1 + s1.T


   # Print maximum effective sugar level
   print('Maximum effective sugar level: ', sugarscape.max())


   return(sugarscape)


# Initialize agents population including age range
def initAgents(size, sugarscape, visionv, metabolismv) :


   agent = {}


   numAgents = int(ceil(random.rand()*0.2*size**2))


   location = random.randint(size, size=(numAgents, 2)) # generate the random locations for agents


   numAgentSpot = zeros((size, size))


   for i in range(size):
      for j in range(size):
         dupRow = nonzero(location[:,0] == i)
         dupCol = nonzero(location[:,1] == j)
         for k in dupRow[0]:
            if k in dupCol[0]: numAgentSpot[i,j] += 1
```

```
for i in range(numAgents):
        agent[i,'location']   = location[i,:]  # set the agent i's location
        agent[i,'ifAlive']    = 1  # agent is alive
        agent[i,'metabolism'] = int(math.ceil(random.rand() * metabolismv))
        agent[i,'vision']     = int(math.ceil(random.rand() * visionv))
        agent[i,'wealth']     = sugarscape[location[i,0],location[i,1]]/numAgentSpot[location[i,0], location[i,1]]    #
agents share sugar if they share a spot



    return(agent, numAgents, numAgentSpot)


# Transform field "agent" from data structure into matrix and display agents locations
def dispAgentLoc(numAgentSpot, numAgents,size, nruns, runs):
    a = zeros((size, size))
    av, am = a, a


    plt.figure(2)
    plt.subplot(math.ceil((nruns+1)/3), 3, runs)
    plt.spy(numAgentSpot)


    plt.title('# of Agents = ' + str(numAgents))


    print('Number of runs', runs)
    print('Average Vision:', sum(sum(av))/sum(sum(a)))
    print('Average Metabolism:', sum(sum(am))/sum(sum(a)))
```

```
# Initialize model parameters

nruns = 10

size = 50   # even number

metabolismv = 4

visionv = 6  # set always smaller than size

maxsugar = 20.0


keys = ['location', 'ifAlive', 'vision', 'metabolism', 'wealth'] # global key index


# Initialize sugarscape and display


sugarscape = initsugarscape( nruns, size, maxsugar )


# Initialize agents population

agent, numAgents, numAgentSpot = initAgents( size, sugarscape, visionv, metabolismv )


# Main loop (runs)

for runs in range(nruns):


    # Display agents locations

    #dispAgentLoc( numAgentSpot, numAgents, size, nruns, runs + 1 )


    # Select agents in a random order and move around the sugarscape following rule M
```

```
 #for i in range(1):

  for i in random.permutation(numAgents):

      if agent[i, 'ifAlive']: # is the agent alive?


          vision = agent[i, 'vision']

          loc = agent[i, 'location']


          NS = range(max(loc[0]-vision, 0),min(loc[0]+vision+1, size))    # North-South location index vector

          WE = range(max(loc[1]-vision,0), min(loc[1]+vision+1, size))    # West-East location index vector


          NSwealth = sugarscape[NS, loc[1]]/(numAgentSpot[NS, loc[1]]+1)  # calculate the expected sugar: NS dir-
ection

          WEwealth = sugarscape[loc[0], WE]/(numAgentSpot[loc[0], WE]+1)  # calculate the expected sugar: WE
direction


          bestsugar = max(max(NSwealth), max(WEwealth))                  # get the best location


##          print('number of runs', runs)

##          print('before loc', loc)


          if bestsugar > sugarscape[loc[0], loc[1]]/(numAgentSpot[loc[0], loc[1]]): # if sugar outlook is better in the
new location:


              # update the agent's location:

              if NSwealth[nonzero(NSwealth == bestsugar)].shape[0]:

                  agent[i, 'location'][0] += max(nonzero(NSwealth == bestsugar)[0]) - vision

              else:

                  agent[i, 'location'][1] += max(nonzero(WEwealth == bestsugar)[0]) - vision
```

```
##            print('after loc', loc)


          newLoc = agent[i, 'location']   # can be the same as the original location


          numAgentSpot[loc[0], loc[1]] -= 1       # reduce the number of agents in the original location

          numAgentSpot[newLoc[0], newLoc[1]] += 1 # increase the number of agents in the new location



          spotSugar = sugarscape[newLoc[0], newLoc[1]]/(numAgentSpot[newLoc[0], newLoc[1]])

          agent[i, 'wealth'] = agent[i, 'wealth'] + spotSugar - agent[i, 'metabolism']


      # If wealth is less than zero set location to unoccupied

      if agent[i, 'wealth'] <= 0:

          agent[i, 'ifAlive'] = 0


   print (runs)
```

APPENDIX B: EXPECTED RESULTS

We expect that by running Sugarscape in parallel on the Beowulf Ring that the outcome will output results faster than running Sugarscape on a single computer which is only a single- or dual-processor. The Beowulf ring distributes the processing load evenly across each node, allowing the program to run faster than it normally would.

APPENDIX C: GLOSSARY OF TERMS

1. **Life-like – A**gents will behaving more like modern humans; forming societies, trading, etc.
2. **Iteration** – One run of the program.
3. **Node –** One computer on the Beowulf ring
4. **Agents –** The name for the “people” in the model
5. **Sugar –** The program name for the agents.
6. **Increment Time –** Increments a time-step
7. **Libraries –** Libraries that add additional functions to Python

APPENDIX D: PYTHON PSEUDO-CODE

```
1. # Sugarscape

2. # sugarscape: (two-peak sugarscape, rule: Ginf)

3. # agents: (moving sequence: random, view: four directions, rule: M)

4. # agents are characterized by their 'ifAlive', 'vision', 'metabolism', 'wealth'

5. # Change: modified how agents are identified,

   and neighbor fn so agents are able to share a location.

6. Import the Numpy library to do complex mathematical equations.

7. Import the Math Plot library as the “plt” request (for sake of simplicity)

       •  Allows us to plot graphs

8. Import Parallel Python to run Sugarscape on the Beowulf ring in parallel.

9. # Initialize two-peak Sugarscape and display

10. Define the “initsugarscape” command and variables.

11. # Generate sugarscape with one south west peak

12. # Generate two-peak sugarscape (south west and north east peak)

13. # Initialize agents population including age range
```

14. # Transform field "agent" from data structure into matrix and display agents locations

15. (Commented out)   Create a graph

16. Show number of runs

17. # Initialize model parameters

18. # Initialize Sugarscape and display

19. # Initialize agents population

20. # Main loop (runs)

    - Loop command for Sugarscape.

21. (Commented out) # Display agents locations

22. # Select agents in a random order and move around Sugarscape following rule M

APPENDIX E: NETLOGO PSEUDO-CODE

```
to go:
        1. If any agents are in the world, stop.
        2. Turtles move
                -check rules/algorithm
                -if there is no sugar or agents are older than the preset age, agents die
                -check rules/algorithm
                -set color based on rules
        3. set color
        4. set reproduction rate
        5. increment time
        6. update graphs
```

APPENDIX F: POSSIBLE FUTURE MODIFICATIONS

Since this project first began, our ultimate goal has been to eventually add the “spice” function into our Sugarscape model and make it so the Sugarscape could span multiple societies across multiple nodes and slowly merge them together or “tear down the walls,” if you will. This would allow multiple societies to grow and eventually “trade” with one another. To our knowledge, this also has not been accomplished.

APPENDIX G: AKNOWLEGEMENTS

First of all, we would like to thank Nicholas Bennett for making the trip to Artesia on multiple occasions throughout the year to help us with our project and code in person and for always being just an email away when we needed help with anything.

Next, we would like to thank Randall Gaylor. He has been an invaluable asset throughout this Challenge year. He has kept us on our toes this entire year, and for that we are grateful.