**University of Washington - Bothell**

**CSS 600 – Independent Study.**

***SugarScape – Dynamic Multi-Threaded Load Balancing***

***Experiment Report***

**Submitted by:**

**Bhargav Mistry**

Contents

[Introduction: 3](#_Toc332755684)

[Platform: 3](#_Toc332755685)

[Test Machine configuration: 3](#_Toc332755686)

[SugarScape dynamic load balancing overview: 4](#_Toc332755687)

[OS level load detection per core - Research and findings 5](#_Toc332755688)

[Sar 6](#_Toc332755689)

[Mpstat 6](#_Toc332755690)

[Ps 7](#_Toc332755691)

[SugarScape load balancing source code: 8](#_Toc332755692)

[Test Results : 9](#_Toc332755693)

[Conclusion: 9](#_Toc332755694)

[Future Work: 10](#_Toc332755695)

# Introduction:

SugarScape is an application based on artificial intelligent agent-based social simulation.

# Platform:

The application is implemented using Java language and is compiled and executed on Linux platform.

# Test Machine configuration:

The configuration of the test machine used is:

**Operating system:** GNU/Linux

**Processor:** i686 athlon.

**Hardware platform:** i386

**Processor details:**

3.6 GHz

Quad core

1024 \* 2 KB cache.

**RAM:** 2GB

**Test execution clock time:**

Following are the test execution details and the machine activity. From the below command details we understand that there was some activity going on, on the test machine so the results might show the extra noise:

[bhargavm@uw1-320-19 CSS600\_12]$ date;who

Tue Aug 14 22:12:47 PDT 2012

bhargavm pts/3 2012-08-14 20:28 (50-47-27-83.evrt.wa.frontiernet.net)

sanusha pts/7 2012-07-23 09:24 (:20)

sanusha pts/8 2012-07-23 09:24 (:20)

sanusha pts/9 2012-07-23 09:24 (:20)

# SugarScape dynamic load balancing overview:

The application has a simulation space which is a 2 dimensional array. The simulation space is equal in size (a perfect square). Each cell of the array is termed as a space. Each space has properties like Sugar and pollution. A set of agents are initialized when the application starts, the agents have metabolism and they consume the sugar as they more around in the simulation space. When sugar is consumed, the agent metabolism decreases and the pollution increases. The agents die out when the metabolism reaches zero.

The application main method accepts the simulation space size, number of threads, simulation cycles, number of agents and a flag called “show”. The “show” flag, when specified, prints out the values of agents array and sugar present in the simulation space after each cycle. SugarScape program accepts a simulation space size and divides the space equally among given total number of threads. These threads perform the computation of sugar consumption and pollution increase on respective data slices. During execution, a method called “barrier” synchronizes all the threads after a cycle of computation is completed. At this point one threads enters a critical section and mobilizes the agents so that agents can consume the sugar present in the cell spaces.

The current SugarScape load balancing algorithm checks the total time used by a particular thread, in the processor and determines the highest time consuming thread. The algorithm then dynamically re-computes the boundary of that particular slice and at the same time corrects the boundaries of adjacent threads. This way when the slice size of the highest time consuming thread is reduced, the total time taken by that thread would be less in the next cycle and the extra load would be shared among adjacent threads.

Hence dynamic load balancing is achieved by changing the boundaries dynamically during runtime.

**Boundary change decision making:**

Currently the boundaries will be change based on the thread that displays highest CPU time. If a particular thread shows the highest CPU time then that time will be compared against a pre-determined threshold value plus the tolerance. If the CPU time is higher than the threshold + tolerance then the thread will be marked to adjust. The simulation slice will be determined and the boundaries will be reduced so that the threads get lesser computation space. The reduction in the slice will be compensated by adding that much space to the thread that attributed least about of CPU time. The application will not load balance in every cycle but will have a timer scheduled, which will trigger the load balancing algorithm.

# OS level load detection per core - Research and findings

Since a thread runs in the core of a processor, it would be very accurate to find how much load a thread exerts on the core. An attempt was made to find out a way to detect the load exerted by a thread on a single core of the processor, at the operating system level. This was tried using 3 commands:

## Sar

Sar command writes the activity counters to the standard output. Sar command can be configured to write the activity to a file periodically. Sar command was examined to find out if it can be useful to detect load per core of a multicore processor. The command arguments were as follows:

*Sar -u 1 3*

This command displays the cumulative CPU usage every 1 sec for total of 3 times. The output looks like:

[bhargavm@uw1-320-19 CSS600\_12]$ sar -u 1 3

Linux 2.6.18-308.11.1.el5 (uw1-320-19) 08/14/2012

10:58:55 PM CPU %user %nice %system %iowait %steal %idle

10:58:56 PM all 0.00 0.00 0.50 0.00 0.00 99.50

10:58:57 PM all 0.25 0.00 0.25 0.00 0.00 99.50

10:58:58 PM all 0.00 0.00 0.00 0.25 0.00 99.75

Average: all 0.08 0.00 0.25 0.08 0.00 99.58

The command was not useful to detect the load per core because it only displayed the cumulative load and I did not find any way to configure it to display per core load.

## Mpstat

The command displays the cumulative CPU loads but displayed the individual loads on a particular CPU. The command displayed the results in detail as compared to the sar command but was not useful since it was not configurable to display load per core.

mpstat -P ALL

[bhargavm@uw1-320-19 CSS600\_12]$ mpstat -P ALL

Linux 2.6.18-308.11.1.el5 (uw1-320-19) 08/14/2012

11:03:49 PM CPU %user %nice %sys %iowait %irq %soft %steal %idle intr/s

11:03:49 PM all 13.78 0.09 1.21 0.12 0.01 0.06 0.00 84.73 1158.32

11:03:49 PM 0 11.75 0.07 0.91 0.05 0.00 0.01 0.00 87.21 1000.19

11:03:49 PM 1 14.98 0.14 1.44 0.33 0.03 0.09 0.00 83.01 88.35

11:03:49 PM 2 12.09 0.08 1.02 0.07 0.00 0.02 0.00 86.73 0.44

11:03:49 PM 3 16.31 0.09 1.47 0.04 0.00 0.11 0.00 81.97 69.34

## Ps

Ps command was also examined to see if it was helpful to display the load per core:

ps -eo pcpu,pid,user | sort -k 1 -r | head -10

The command displays CPU, processed and user id. Sorted in decending order on the CPU load. This command was examined and again it cannot be configured to display load per core.

Following is the output:

[bhargavm@uw1-320-19 CSS600\_12]$ ps -eo pcpu,pid,user | sort -k 1 -r | head -10

%CPU PID USER

81.4 9224 preethik

 1.5 6336 root

 1.5 6268 root

 0.9 23241 dslab

 0.5 9223 preethik

 0.4 9218 root

 0.1 28298 bhargavm

 0.1 16931 nani0410

 0.0 9 root

# SugarScape load balancing source code:

Pls refer next page.

# Test Results :

1. Execution using 1 thread:

*java Sugar 10 2000 20 1*

 *[dslab@hercules CSS600\_12]$ ./run.sh*

*nAgents = 10 maxSimTime = 2000 simSize = 20 nThreads = 1 show = false*

*Elapsed time : 74*

1. Execution using 2 threads:

*java Sugar 10 2000 20 2*

 *[dslab@hercules CSS600\_12]$ ./run.sh*

*nAgents = 10 maxSimTime = 2000 simSize = 20 nThreads = 2 show = false*

*Elapsed time : 167*

*Elapsed time : 167*

1. Execution using 3 threads:

*java Sugar 10 2000 20 3*

*[dslab@hercules CSS600\_12]$ ./run.sh*

*nAgents = 10 maxSimTime = 2000 simSize = 20 nThreads = 3 show = false*

*Elapsed time : 170*

*Elapsed time : 170*

*Elapsed time : 170*

1. Execution using 4 priority load balancing:

*java Sugar 10 2000 20 4*

*[dslab@hercules CSS600\_12]$ ./run.sh*

*nAgents = 10 maxSimTime = 2000 simSize = 20 nThreads = 4 show = false*

*Elapsed time : 256*

*Elapsed time : 256*

*Elapsed time : 256*

*Elapsed time : 256*

# Conclusion:

Execution time with 1 thread: 74ms

Execution time with 2 threads: 167ms

Execution time with 3 threads: 170ms

Execution time with 4 threads: 250ms

Original version execution time: 112ms

From the above tests executed on Hercules, performance degrades as the number of threads increases. Further refining of the application is required at this point.

# Future Work: