12-2015

ESTIMATION ON GIBBS ENTROPY FOR AN ENSEMBLE

Lekhya Sai Sake
Lekhya Sai Sake

ESTIMATION ON GIBBS ENTROPY FOR AN ENSEMBLE OF PARALLEL EXECUTIONS

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Lekhya Sai Sake

December 2015

ESTIMATION ON GIBBS ENTROPY FOR AN ENSEMBLE OF PARALLEL EXECUTIONS

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Lekhya Sai Sake

December 2015

Approved by:

Ernesto Gomez, Advisor, School of Computer Science and Engineering

Owen J Murphy, Committee Member

Kerstin Voigt, Committee Member

ABSTRACT

In this world of growing technology, any small improvement in the present scenario would create a revolution. One of the popular revolutions in the computer science field is parallel computing. A single parallel execution is not sufficient to see its non-deterministic features, as same execution with the same data at different time would end up with a different path. In order to see how non deterministic a parallel execution can extend up to, creates the need of the ensemble of executions. This project implements a program to estimate the Gibbs Entropy for an ensemble of parallel executions. The goal is to develop tools for studying the non-deterministic feature of parallel code based on execution entropy and use these developed tools for current and future research.

ACKNOWLEDGEMENTS

I would like to thank all the people with whom I have worked while pursuing my master's degree at California State University, San Bernardino (CSUSB). Studying in the School of Computer Science and Engineering at CSUSB has been a tremendous learning experience, both personally and professionally.

Thank you to the following faculty of the Computer Science and Engineering department for their invaluable guidance, advice, support, help, and patience during this project's long gestation: Dr. Ernesto Gomez, Dr. Owen J.Murphy and Dr. Kerstin Voigt.

Special thanks to my friend Nikhil Dasari in the Department of Computer Science, for his valuable suggestions and support and encouragement throughout my masters.

I would also like to thank my family and friends for their undying support and encouragement for my studies over the years.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

This paper presents an overview of the detailed description of the parallel executions, which are non-deterministic (Conery, 2012). It delves into how a clique of parallel executions can vary out when its executions are represented in a pictorial manner.

Purpose

The purpose of this project is to study the characteristics of a group of non-deterministic parallel executions. The prior work was implemented by Dr. Gomez and Dr. Schubert from California State University, San Bernardino. They speculated about the possibility of applying the concepts of entropy to work to parallel executions, the state space, and visualization techniques. This research will yield interesting results about non-deterministic parallel executions, which will help us to penetrate more into the concept of entropy for a better future.

Project Scope

The scope of this project is to convert the current combination of bash scripts and Scilab programming into an integrated system (probably in C++) to automate the data gathering and generate the parallel (phase space) status file for P processes and N runs of the same parallel task. Also, Design and develop a

program to estimate the Gibbs Entropy-based on the phase space, based on estimating the probability of a given microstate from the frequency of other states from multiple runs in the neighborhood of the microstate, normalized to yield a probability of 1 for the entire ensemble. This may include trials with different definitions of the neighborhood.

Experiments

To test the developed system, we require huge amounts of data. This data is generated by the system. Every time we run the bash script, it produces the huge amount of data, which are later visualized on the phase space. The data represented in the form of a matrix, which includes the time of execution. This data is later sorted using a sorting algorithm, which results in the data with respect to their time execution. Every execution also presents the state they are in, which is later sorted using the merge sort. Experiments will be performed on two selected parallel programs to determine:

1. How big an ensemble do we need (i.e., how many executions with the same data and processes) are required to get a large enough ensemble to get valid statistics
2. Is the number N of executions required vary with the number P of processes

Tests to be carried out on

1. Downhill simplex minimization in D dimensions

2. Standard parallel code (to be selected)

Requirements

Hardware Requirements

- PC for testing
- PC for development

Software Requirements

- Scilab Library
- MPI Library
- Linux Operating System

Limitations of the Study

This project is limited to the programming language that it is developed. We have used Java initially as it is one of the high-level languages with huge and expertise libraries. But it threw an exceptional error and space error. Also, the code was less complicated when we had to deal with huge amount of data. Another programming language considered to be ‘C’. It was also a high level language, but it was not as advanced as C++. Therefore, after using a couple programming languages, we decided C++ will the apt choice for this project.

CHAPTER TWO

INFORMATION THEORY

Entropy

Entropy can be defined as the measure of the number of specific ways in a system can be arranged (Nave, 2015). In general, it is usually considered as the degree of disorder. By the laws of the thermodynamics, when we consider the entropy of an isolated system never decrements and also will spontaneously evolve towards thermodynamic equilibrium, the configuration the maximum entropy. There are some systems that are not isolated may decrement, but they increment the entropy of their environment by at least the same amount. The change in the entropy of a system is the same for any process that goes from a given initial state to a given final state, whether the process is reversible or irreversible because entropy is a state function. However, an irreversible process increases the combined entropy of the system and its environment.

The first law of thermodynamics was derived by James Joule in 1843. He conducted many experiments on heat, friction and expressed the concept of energy and its conservation. Unfortunately, this law was unable to quantify the effects of friction and dissipation.

Rudolf classes, a German physicist, gave the “change” a mathematical interpretation by question the nature of the inherent loss of usable heat when work is done. He described entropy as dissipative energy use, by a

thermodynamic system during s change of state. This was quite opposite to the primitive views by the theories of Isaac Newton which described heat was an indestructible particle that has mass. After a few years, in 1877, scientists such as Ludwig Boltzmann, Josiah Willard Gibbs, and James Clerk Maxwell derived entropy that was based on statistics

Boltzmann Entropy

Boltzmann derived an equation that is a probability relating to the entropy of an ideal gas to its quantity *W.* This equation is

$$S = K_B \ln W$$

Where K_B is the Boltzmann constant that is equal to 10^{-23} J/K

The Boltzmann equation gives us the relationship between the entropy and the number of ways the atoms or molecules of a thermodynamic system can be arranged (Fraydoun Rezakhanlou, 2001). In 1934, Swiss physical chemist Werner Kuhn successfully derived a thermal equation of state for rubber molecules using Boltzmann's formula, which has since come to be known as the entropy model of rubber.

For example, let's consider four playing cards. The possible ways of arranging these cards are 4x3x2x1 = 24.

Therefore,

W = 24, Boltzmann Constant: 1.4×10^{-23} J/k

By using the Boltzmann constant, after all the calculations, $S = 4.4 \times 10^{-23}$ J/K.

Shannon Entropy

A way to estimate the minimum average number of bits needed to encode a string of symbols based on its respective frequencies is provided by Shannon entropy. The probability of the events is coupled with the information about every event and form a random variable. The average of this random variable is considered as the average amount of the information. Shannon's entropy units are measured as bits.

Shannon's entropy estimates the minimum average number of bits needed to encode a string of symbols (Coifman, 2002). This process of encoding depends on the frequency of the symbols. This is given

$$H = -\sum_i p_i \log_b p_i$$

In the Shannon entropy equation, p_i is the probability of a given symbol. Shannon entropy conveys what is the minimal number of bits per symbol needed to encode the information in binary form.

Gibb's Entropy

J. Willard Gibbs defined a formula for entropy. The macroscopic state of the system is defined by distribution on the microstates that are accessible to a system in the course of its thermal fluctuations (Dyre 2009). Henceforth, the entropy is defined for two different levels of description of the given system. At one of these levels, the entropy of the formula is given by the Gibbs entropy.

Gibbs versus Boltzmann entropies. With equality if and only if the distribution on is canonical. On the other hand, the expression for the change of the Boltzmann entropy shows that it ignores both the internal energy and the effect of the inter-particle forces on the pressure.

CHAPTER THREE

NOTES ON VISUALIZATION

Introduction

This chapter explains the prior work done on this project. It includes a detailed explanation of how did this project entails.

Parallel Execution

The definition of the execution is given by K.R, Apt, and Olderog in 1997, their main idea is to represent a parallel program as a collection of sequential programs, where the execution of the sequential program is the sequence of the basic block. Therefore, sequential executions are deterministic, and parallel executions are not deterministic (Kosta, March, 2012). Apt Olderog et al., also states that the collections of the basic blocks are running any given time unless the time is synchronized. But different threads are processed at different rates, these results in running the same execution with the same data at different times give us different results. Finally, after years of research on parallel executions being non-deterministic, Gomez, Schubert & Cai in 2012 conducted an experiment to see how non-deterministic can a parallel execution be. They instrumented the executions and realized that the executions turn out to be way more non-deterministic than it seems to be.[8][7]

History

Gomez et al. conducted an experiment where he had a system that had only the parallel executions running, and each thread was connected to one CPU. They recorded the execution time of each execution and plotted on the graph to expecting to see a cluster of executions. But the results the results turn out to be shocking. All the executions were shattered all around the graph. This paper led me to study more about non-deterministic features of parallel executions.

A single parallel execution is not sufficient to see its non-deterministic features as same execution with the same data at a different time would end up with a different path. To see how non-deterministic a parallel execution can extend creates the need for the ensemble of executions. To define the ensemble, we need the phase space and entropy.

Notes on Visualization on Parallel Execution

A practical method for instrumenting parallel execution to extract parallel state as a function of time and to generate the path of an execution to extract in phase space was developed. Another method for displaying P and P+1 dimensional paths and sets of states in two and three dimensions that will give a reasonable intuitive grasp of the properties of the P-dimensional original was developed. We applied the methods described by a particular algorithm, which

we implemented using different types of communication as well as no communication.

Assumptions

We consider parallel executions in the static Single Program Multiple Data (SPMD) model. A set of P processes, all of which start and end in a basic block label. They follow a control flow graph set of direct arcs between the nodes. A transition from one basic block to another to be a step in the execution is numbered in sequence as the process follows a path (Pi). We denote the state of a process 'I' by σi, j = (Xj, Mj) where j denotes the step in the execution, X the block being executed, and M the contents of memory at that process. Knowing M and X we may determine the next step process 'I' will take and the contents of memory at the start of the next block. By induction, we can determine the rest of the execution of i from σi, j.

This is a total state of all processes in Γ, replacing Γ with I c Γ gives us a partial state involving a subset of processes. We can then describe a parallel execution in total state and note that an execution in partial states must be a DAG such that any slice that partitions the DAG by cutting concurrent paths must be consistent with a state in the total execution.

Instrumenting the Code

It is not practical to save M_j for each process at each step. We use a basic block $X_j$as our state, proxy, and the path P_i as the proxy for the process execution. It would be possible to record a total state by interrupting all

processes at a specific time and inspecting memory, but this would be too intrusive for use in recording an execution. We choose instead to record the sequence of basic blocks and times (X_j, T_j) at each process and use these to reconstruct a total execution after the fact.

In the code, we number basic blocks sequentially from s to e. We define an *out (b)* command, where *b* is block number; *out* prints the triple “t b PID \n” to file “name+pid. Out”, creating one. Out file per process. Here “name” is the program name, “PID” is the process id from 0 to P-1, b is block number and t is time in microseconds to 10 μs resolutions. These files are then concatenated and piped to the UNIX “sort” command, which creates a single file “name+ident. Srt”, sorted on time and with “time block PID” on each line.

The available hardware constrains the time resolution. Block identification and instrumentation is presently done by hand, so is only approximately related to actual basic blocks. It is recognized that some states, including blocks that take very little execution time will not be resolved by the present system, so it will generate an approximation view that is consistent with states.

Reconstruction of Parallel State

For a P process execution, we represent each state as an ordered P+1 tuple, where the first P places are the basic block executed by the process with PID corresponding to position in the tuple and the P+1 place holds the time. For example “0 0 1 2 50” would be in a state of a 4 processor execution, in which processes 0 and 1 are each in block 0, a process is in block 1 and process 2 is in

block 2, 50 microseconds after the start. The execution is placed in an M* (P+1) matrix, where each row represents a state and there is one row for each of the M total states identified in the execution.

We assume the start time is zero. If two entries in raw data have the same PID and different times, the data from the earlier time will be overwritten – this is not a frequent issue in current data, but with faster systems or to capture detail on small basic blocks we will have to increase time resolution. If a process does not have an entry to match a particular time, the block from the previous state will be copied. This captures the possibility that some blocks will take significantly more time to execute than others, so it is possible for a particular process to take consecutive steps, making new parallel, particularly relevant for partial state execution, in which concurrent process subsets are executing different tasks.

Phase Space

Our definition of the state as a P-tuple of basic block numbers leads to a phase space of all possible states as a P-dimensional hypercube of side *e – s.* The state of any particular process is recorded as the coordinate value for dimension *j*, where *j* is the process number; the tuple P of all the process block values is a point in of the hypercube. Our base case is P processes without interacting, each, therefore, independent of the other processes. Each process can be in the set of states denoted by integers 0…… me where 0 is taken as the number if start states and *e* is the number of the end state.

Each process can transition independently from all others, so processes can be considered orthogonal. Therefore, we represent a phase space for a parallel execution as a hypervolume of a P-dimensional hyper-cube., with sides of length *e.* A point inside of the hyper-volume is's' P-tuple; if we restrict the elements of the tuple to integers, we get all possible states.

States of execution with the interaction between processes are certainly a subset of the states of non-interacting executions, so the same phase space works. In either case an execution may be represented as a poly linear path in the hypercube, constrained by the requirement that each vertex of the execution that terminates normally, the path begins on the corner labeled (0, 0… 0) of the hypercube and ends at (e, e, e… e), which is the opposite corner.

We note that the states P represented in the phase space are not in fact the complete parallel state $S_{r,j}$. In particular, we have not found a practical way of representing the contents of memory, and the phase space represents only total states of processes in Γ, but does not represent partial states of G c Γ; furthermore, it needs to be extended to allow the possibility that an initial set of P processes may create or destroy processes during an execution.

We further note that the phase space allows only states with integer values of all coordinates; and that the control flow graph of the code restricts the possible successor states for any given point 'P'. An implication of the control flow graph is that the successor states for a given process 'P' represented in a state 'P' does not have to be adjacent to 'P' in the phase space, since it might

represent a jump to a block of code that is not the immediate successor of the block P is in.

Evolution in Time. The path in phase space is a parametric graph of the progress of individual processes with time as the parameter.

We may include time explicitly in the representation by adding a dimension *t* orthogonal to the hypercube, in which case the path would advance in time at each step and would have no cycles since the time coordinate is monotonically increasing. In this case we would have a polyline transiting through a set of phase space cubes labeled t_0 to it starting at (0,0,0…. 0) on the first cube, a and ending at (e, e, e,…. e) on the last cube.

Displaying P/P+1 Dimensional Data. We display the phase space as a (hyper) cube standing on a diagonal with the line from (0, 0, 0…… 0) to (e, e, e… e) displayed vertically from the center. For better visualization of multidimensional data, we use a 3D perspective graph displayed on a plane – we use Scilab for display, so we can rotate the 3D graph and view it from different angles.

This scheme has the advantage of being fairly intuitive. Specifically, if processes advance in lockstep through the same code, then every state should look like (k, k, k… k) where k is a constant block id, and the execution plot should display all points on a vertical line. Barrier synchronizations involving all processes should also display on the center line. Excursions away from the

center line should reflect processes, advancing at different rates or subsets of processes doing different tasks concurrently.

It has the disadvantage of introducing an artificial sense of proximity between consecutively numbered dimensions – for example (2,0,0,2,0,0) would be plotted on the vertical, whereas in 6D it would be as far from the vertical as (2,2,0,0,0) – which would appear at a large angle from vertical. The 3D plot makes some points appear closer to the diagonal than they are; it further distorts the distance from the origin, making them appear larger because the unit vectors used for display are not orthogonal.

No attempt has been made to compensate for distortion, so scales on phase space graphs are only used for comparison between graphs of the same type and do not give a true value of the P-dimensional distances and positions that the graphs are generated from graphs onto the XY plane (view from the top) or onto the XZ place (side view).

Ensemble of Executions. Parallel transition $S_{\Gamma, I}$ ($S_{\Gamma, J}$ is non-deterministic, because absent synchronization any combination of processes in Γ may transition, giving a new state. As a result, each parallel execution may follow a different path through phase space, even for the same code, number of processes, data, and machine.

Therefore, to study the behavior of the parallel code, we need an ensemble of executions of the same code, a number of processes, machinery,

and data to allow us to distinguish features intrinsic to the algorithm and code from features that may vary randomly from one execution to another.

CHAPTER FOUR

ESTIMATION ON THE ENSEMBLE OF PARALLEL EXECUTIONS USING GIBB'S ENTROPY

Introduction

The main motive of this project is to study the degree of randomness observed in the result of the prior work. The prior work describes the randomness of the parallel executions. A set of parallel executions is executed in parallel to each other, resulting in a sequence of numbers which had no meaning. To understand this problem in a better way, these points were plotted on a graph to study the occurrence of each instruction on its time of execution. It was expected to see a big cluster of points as all the instructions were executed in parallel to each other. To the surprise, all the points were spread across the graph. It is an interesting situation to study the randomness of the parallel executions.

The prior work has a collection of data points which are represented in the form of a matrix. The size of this matrix is quite interesting, as it is 6000 X 18, where the rows represent each point in the maze of points and columns represent dimensions of each point. Technically, we have 6000 points with 17 dimensions each. The last column in the matrix represents the time taken to execute that particular instruction.

The next big decision to make was which programming language to use. As this program is more related to the executions of instructions and its

respective calculations, we had few programming languages. They were Java, C, C++, Python and UNIX. Also, shell programming was suggested.

There were a couple of approaches to solving this mystery of numbers and the gist behind them. To acquire the clusters, we need to the point and its neighboring points in the given radius.

Approach

In mathematics, to find the distance between one points to another point we use a formula called the Euclidean distance. In geometry, to find the length of a line or to find the distance between two points we use Euclidean distance. For example, consider two points A, B. The Cartesian coordinates of these points will be (a_1, a_2, a_3… a) and (b_1, b_2, b_3 … b_n). To calculate the distance between A and B using Euclidean distance will be the following

$$D(A, B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2 + \cdots\cdots + (a - b_n)^2}$$

Initially, we wanted to find the Euclidean distance from a given point to any point in the graph. This idea would tell us how far each point is away from each other. It was implemented in the Java programming language. After implementing the program, we had difficulties understanding the resulting matrix. Then we modified the approach by calculating the distance between each point to every point in the graph. The result of this approach will give the information about the position of any point in the graph. This program took 26minutes to calculate the matrix. It resulted in a matrix of 6000X6000.

The algorithm of the Java code is as follows

```
/* Code begins*/
Public class EcluDist
                {
                        HashMap mymap = new HashMap ();
//Hash map code that will help us to store and retrieve the data easily
                        Public static void main (String [] args)
                        {
                        Try {
                        Long starts = System. CurrentTimeMillis ();
// calculating the start time
                        String filename = "C:\\techstack\\lekhya\\mini.amoeba";
// Read file into another hash
// after reading the file, this for loop is to retrieve the data from the hasp map
                        For (int in=1; in<16; in++) {
                        String key =out+"."+in;
                        Sb. append (String. format ("%. 2f", my map. Gets (key)));
                        Sb. append ("                    ");
                        //System. Out. Println (key+ ": "+mymap. Get (key));
                        System. Out. Println (Sb. ToString ());
                Long endts = System. CurrentTimeMillis ();
 // calculating the end time
```

```
Public void readElement (String file) throws Exception
// reading the element
                {
                BufferedReader br = new BufferedReader (new FileReader (file));
                        Int lnNo = 1;
                 String line="";
                        While (St. HasMoreTokens ()) {
                                        String s = St. next Token ();
                                        Double t = Double. Parse Double (s);
                                        String elemKey = lnNo+"."+i;
                                        Elements. Put (elemKey, t) ;}}
                Public void calcElementEclu () throws Exception
                {
                Euclidean Distance Ed = new Euclidean Distance ();
// Loop thru the total number of Lines = outer loop
                        //For (int ln=1; ln< totalLines; ln++) {
                        Int ln=1;
                        While (Ln<totalLines-1) {
// Loop thru the total number of elements in each line (17 fixed) = inner loop int
elemCounter=1;
                        While (elemCounter<17) {
// get X1 and X2 from hash
```

```
String x1 = ln+"."+elemCounter;
String x2 = ln+"."+ (elemCounter+1);
Double [] a = {(double) elementHash.get(x1), (double)
elementHash.get(x2)};
// Printing all the calculated values}}}
```

Lately, this program started throwing errors as it could not accommodate 36,000,000 elements in its console. Also, Java did not have many advantages like C++ programming. Hence, we changed the code from Java to C++ for a better result.

C++ Programming

We chose this programming language as its library has the most feasible and accurate libraries to handle huge amounts of data. Also, we decided to split the whole task into bits and pieces of code. This will help us to read and understand the process in a detailed way. The primitive step is to generate the distances between all the elements.

Adjacency Matrix

The primitive step was to find the distance between all the points. This will give us the information how far is a point from another point. The distance between two points is calculated by using the Euclidean's Distance formula. The input file for this program is the 'Datasos. Amoeba' file, which has the data of

different executions with its time of executions. It is in the form of a matrix of size 18 X 6000. The 18th column is the time column, which is ignored.

This program finds the distance from each point to every point in the matrix, and it results in an adjacency matrix. The result of this program looks as follows.

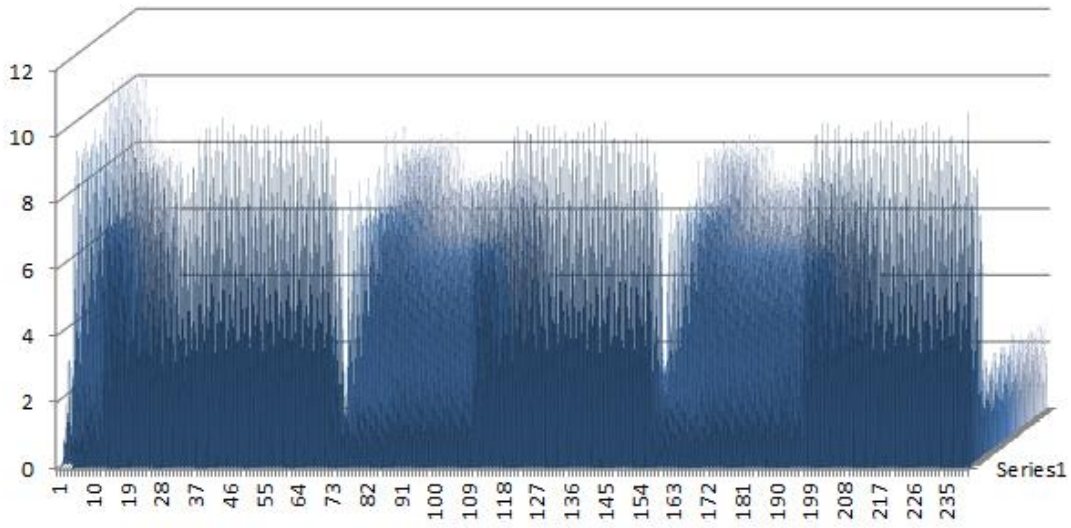

Figure 1. Adjacency Matrix on 3d- Graph.

Sorting the Adjacency Matrix

The next step was to sort the elements generated which are in the form of the adjacency matrix. I used merge sort technique to sort this matrix. The input to this code is the output of the adjacency matrix. This code outputs another adjacency matrix that has sorted elements in ascending order. It also generates

another file, which says ‘Console Output.txt’ which displays all the elements that are generated in the console.

Radius

This is the third part of the code. In this code, we input the radius value, and we generate all the elements within that radius value. The input to this code is adjacency matrix that is generated in the sorted matrix code. The output of this code is the set of elements that lie within the range of the given radius. Also, this code generates a text file “Radius Console Output.txt’ which contains all the code that is processed in the console.

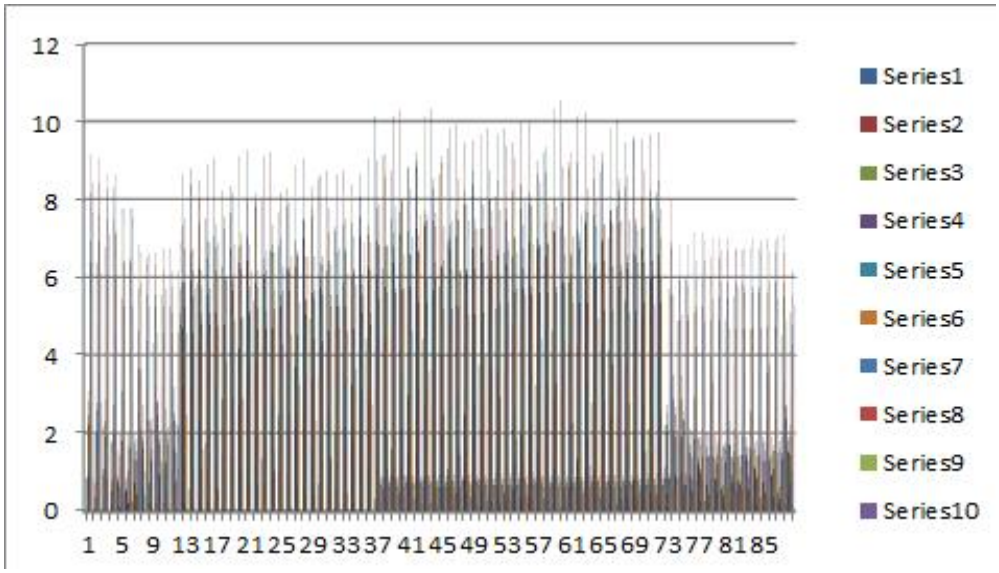

Figure 2.Elements with Radius

Normalization

This is the fifth step in the code. In this step, we study each value to the average of all the data elements. To implement this process, we need to calculate for a formula, that is a = N/r^D

Where a is the normalization value

N is the number of elements within the given radius

R is the radius

D is the density value calculated

The output of this file is the probability of elements with some elements in the whole data; that is, 6000. In this code, the output file is saved as ‘Density2’.

Gibb’s Entropy

This is the final step in the code. In this part of the code, we calculate the relation between the probabilities that we generated using Gibbs Entropy. The input to this code is the output of Normalization code, i.e., ‘Density2’ file. The output of this file is a single value which gives the Gibbs Entropy, the result of this project.

CHAPTER FIVE

CONCLUSION

The results presented from the above discussion and research yield interesting results. The results are in agreement with earlier experiments and works of research carried out before date, showing that that parallel execution proves to be highly non-deterministic and hence the single value produced by the output of the normalization code. The estimation of Gibbs entropy here can be considered a success and as such proving the earlier premises that of non-determinism of parallel execution. Parallel execution has revolutionized how computer systems operate and in their handling of information.

REFERENCES

[1] C. F. Rezakhanlou, “Entropy methods for Boltzmann equation”, *Paris: Springer Science and Business Media*, 2001.

[2] E. Gomez, *Scientific Parallel Computing,CSE 624 Textbook, CSUSB, 2011.*

[3] E. Gomez, L. Ridway Scott, *Overlapping and shot-cutting Techniques on Loosely Synchronous Irregular Problems,* in Solving Irregular Structured Problems in Parallel, volume LNCS 1457, p.p. 116-27, Springer, 1998.

[4] S. F. Goldman, *Information Theory*, New Jersey: Prentice- hall, 1953.

[5] I. Csiszar, “Information Theory: coding theory for discrete memoryless systems”, Cambridge University Press, 2011.

[6] J. H. Dyre, “A brief critique of the Adam-Gibbs entropy model”, *Journal of Non-Crystalline Solids*, 624-627, 2009.

[7] H. F. Jordan and Alaghband, *Fundamentals of Parallel Processing, Prentice-Hall 2003.*

[8] J. S. Conery, “Parallel execution of logic programs”, *Springer Science and Business Media,* 2012.

[9] R. Coifman, “Entropy based algorithms for best basis selection”, *Information Theory*, IEEE Transactions on 713-718, 2002.

[10] R. Nave, “Entropy as Times Arrow”, http://hyperphysics.phy-astr.gsu.edu/hbase/therm/entrop.html, October 2015.

[11] S. A. Kosta, “Hinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading in INFOCOM”, 2012 proceedings IEEE, 945-953, March 2012.

[12] K. Apt, F. S. De Boer, E. R. Olderog *Verification of Sequential and Concurrent Programs, Springer-Verlag London Limited, 2009, London.*

[13] Unpublished Manuscript, Department of Computer Engineering, CSUSB, California. Baylor University, Texas.