OPENMP EXECUTION & ANALYSIS
LORENZ ATTRACTOR

High Performance Computing Project Report

By,

SAI KAUSHIK SUDHAKARAN

CED18I044

Department of Computer Science and Engineering,

Indian Institute of Information Technology Design and Manufacturing

Kancheepuram, Chennai

October 2021

ACKNOWLEDGEMENT

I would like to express my gratitude toward staff of Department of Computer Science and Engineering of Indian Institute of Information Technology Design and Manufacturing Kancheepuram for providing me a great opportunity to complete a project on Parallelizing Lorenz Attractor.

My sincere thanks go to **Dr. Noor Mahammad SK, Assistant Professor, Department of Computer Science and Engineering.** Your valuable guidance and suggestions helped me in various phases of the completion of this project. I will always be thankful to you in this regard.

I am ensuring that this project was done by me and not copied from anywhere.

ABSTRACT

This project is designed to implement parallel programs for the simulation of objects following **Lorenz attractor** satisfying the Lorenz system.

Lorenz System is a system of differential equations, noted to have chaotic solutions for given parameters and the initial conditions. Lorenz attractor is the set of chaotic solution to the above system.

Using multi-dimensional arrays and multiple threads to increase the speed-up of the program. Generating high performance programs for the same of multi-core processor (OpenMP), cluster computers (MPI), and general-purpose graphic processing unit (Cuda C/C++).

Drawback of serial programs is the inability to run multiple objects to visually see the chaos with slight variations in the initial conditions.

TABLE OF CONTENT

INTRODUCTION

The Lorenz System is a system of differential equations, noted to have chaotic solutions for given parameters and the initial conditions, first studied by Edward Lorenz in 1962 [1].

Lorenz attractor is the set of chaotic solutions of the Lorenz System. It is a part of an area of study, Chaos Theory.

The shape of the Lorenz attractor, when plotted graphically, represents that of a butterfly. The butterfly effect is a real-world implication of the Lorenz attractor.

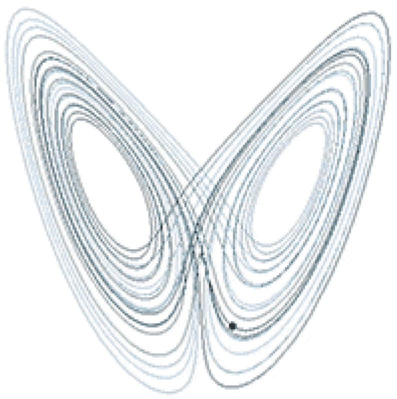


Figure 1: *Lorenz Attractor Animation*

This project is designed to implement parallel programs for the simulation of objects following Lorenz attractor satisfying the Lorenz system.

LITERATURE SURVEY

CHAOS THEORY

Chaos theory is a branch of mathematics focusing on the study of chaos—dynamical systems whose apparently random states of disorder and irregularities are actually governed by underlying patterns and deterministic laws that are highly sensitive to initial conditions [2].

A commonly used definition, to classify a dynamical system as chaotic, it must satisfy three properties:

- It must be sensitive to initial conditions.
- It must be topologically transitive.
- It must have dense periodic orbits.

BUTTERFLY EFFECT

In chaos theory, the butterfly effect is the sensitive dependence on initial conditions in which a small change in one state of a deterministic nonlinear system can result in large differences in a later state. [3]

It is metaphorically derived from the example of a tornado being influenced by minor perturbations such as a distant butterfly flapping its wings several weeks earlier.

ATMOSPHERIC CONVECTION

Atmospheric convection is the result of a temperature difference layer in the atmosphere. Different rate of temperature change with altitude within the air masses lead to instability. This instability with moist air masses causes thunderstorms, which are responsible for severe weather in the world. [4]

HISTORY

Lorenz had been studying the simplified models describing the motion of the atmosphere, in terms of ordinary differential equations depending on few variables. In his 1962 article, he analysed differential equations, for atmospheric convection, in a space of 12 dimensions, in which he numerically detects a sensitive dependence to initial conditions. [5]

In 1963, with the help of Ellen Fetter, Lorenz simplified the above mathematical model to a system of differential equations, now known as the Lorenz equations.

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

Here, x is proportional to the rate on convection

y is proportional to the horizontal temperature variation

z is proportional to the vertical temperature variation

σ is proportional to the Prandtl number, ratio of momentum diffusivity to thermal diffusivity

ρ is proportional to the Rayleigh number, describes the behaviour of fluids when the mass density is non-uniform

β is proportional to certain physical dimensions

CODE (LORENZ ATTRACTOR)

SERIAL CODE

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <GL/freeglut.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>

typedef struct point
{
    double x;
    double y;
    double z;
    float t;
} Point;

double sigma = 10.0;
double rho = 28.0;
double beta = 8.0 / 3.0;

Point initial = {1.0, 1.0, 1.0, 0.0};
double dt = 0.01;

unsigned long int start = 0;
unsigned long int end = 1000000000000;

static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;

unsigned long int op = 0;

Point differential(Point curr)
{
    Point diff;
    // 4 artih ops, 5 reads, 1 write
    diff.x = (sigma * (curr.y - curr.x)) * dt + curr.x;
    op += 4;
    // 5 arith ops, 6 reads, 1 write
    diff.y = (curr.x * (rho - curr.z) - curr.y) * dt + curr.y;
    op += 5;
    // 5 artih ops, 6 reads, 1 write
    diff.z = (curr.x * curr.y - beta * curr.z) * dt + curr.z;
    op += 5;
    // 1 arith op, 2 reads, 1 write
    diff.t = curr.t + dt;
    op += 1;
    return diff;
}

void lorenzGenerator()
{
    Point coord = initial;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    glPointSize(1.0);
    FILE *fp = fopen("new.txt", "w");
    for (unsigned long int i = start; i < end; i++)
    {
        coord = differential(coord);
        fprintf(fp, "%f, %f, %f %f\n", coord.t, coord.x, coord.y, coord.z);
        glBegin(GL_POINTS);
        glColor3f(1, 1, 1);
        glVertex3f(coord.x, coord.y, coord.z);
        glEnd();
        glFlush();
        glutSwapBuffers();
    }
    fclose(fp);
    printf("Ops = %ld\n", op);
    glutLeaveMainLoop();
}

void spinCube()
{
    if (theta[axis] > 360.0)
        theta[axis] -= 360.0;
    else if (theta[axis] < 0)
        theta[axis] += 360.0;
    glutPostRedisplay();
}

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(1280, 720);

    glutCreateWindow("Lorenz");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(lorenzGenerator);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

PARALLEL CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <GL/freeglut.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
#include <omp.h>

typedef struct point
{
    double x;
    double y;
    double z;
    float t;
} Point;

double sigma = 10.0;
double rho = 28.0;
double beta = 8.0 / 3.0;
Point initial = {1.0, 1.0, 1.0, 0.0};
double dt = 0.01;
unsigned long int start = 0;
unsigned long int end = 100000;
static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;

unsigned long int op = 0;

Point differential(Point curr)
{
    Point diff;
    double temp0, temp1, temp2, temp3;
#pragma omp parallel sections
    {
#pragma omp section
        temp0 = sigma * (curr.y - curr.x);
#pragma omp section
        temp1 = curr.x * (rho - curr.z) - curr.y;
#pragma omp section
        temp2 = curr.x * curr.y;
#pragma omp section
        temp3 = beta * curr.z;
#pragma omp section
        diff.t = curr.t + dt;
    }
#pragma omp barrier
    {
        diff.x = temp0 * dt + curr.x;
        diff.y = temp1 * dt + curr.y;
        diff.z = (temp2 - temp3) * dt + curr.z;
    }
    return diff;
}
```

```
void draw(Point coord)
{
    glBegin(GL_POINTS);
    glColor3f(1, 1, 1);
    glVertex3f(coord.x, coord.y, coord.z);
    glEnd();
    glFlush();
    glutSwapBuffers();
}

void lorenzGenerator()
{
    Point coord = initial;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    double startTime = omp_get_wtime();
    glLoadIdentity();
    glPointSize(1.0);
    for (unsigned long int i = start; i < end; i++)
    {
        coord = differential(coord);
        draw(coord);
    }
    double endTime = omp_get_wtime();
    printf("\n%f \n", endTime - startTime);
    glutLeaveMainLoop();
}

void myReshape(int w, int h)
{
    glRotatef(theta[0], 1.0, 0.0, 0.0);
    glRotatef(theta[1], 0.0, 1.0, 0.0);
    glRotatef(theta[2], 0.0, 0.0, 1.0);
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
    glMatrixMode(GL_MODELVIEW);
}

void spinCube()
{
    if (theta[axis] > 360.0)
        theta[axis] -= 360.0;
    else if (theta[axis] < 0)
        theta[axis] += 360.0;
    glutPostRedisplay();
}

int main(int argc, char **argv)
{
    if (argc == 2)
        omp_set_num_threads(atoi(argv[1]));
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(1280, 720);
    glutCreateWindow("Lorenz");
    glutReshapeFunc(myReshape);
    glutIdleFunc(spinCube);
    glutDisplayFunc(lorenzGenerator);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

CRITICAL SECTION OF THE CODE

```
Point differential(Point curr)
{
    Point diff;
    double temp0, temp1, temp2, temp3;
#pragma omp parallel sections
    {
#pragma omp section
        temp0 = sigma * (curr.y - curr.x);
#pragma omp section
        temp1 = curr.x * (rho - curr.z) - curr.y;
#pragma omp section
        temp2 = curr.x * curr.y;
#pragma omp section
        temp3 = beta * curr.z;
#pragma omp section
        diff.t = curr.t + dt;
    }
#pragma omp barrier
    {
        diff.x = temp0 * dt + curr.x;
        diff.y = temp1 * dt + curr.y;
        diff.z = (temp2 - temp3) * dt + curr.z;
    }
    return diff;
}
```

The above snippet is the critical section of the program, i.e., the most frequently run section in the program. This section calculates the new point with the help of the Lorenz equations mentioned above [link].

Due to the dependency of the current point with the previous point, gives us a small room for parallelization.

Here, pragma sections are used to do the computations that are independent of each other in parallel by multiple threads, which theoretically increase the efficiency of the program.

The final calculation requires the values calculated in the sections. Therefore, we add a barrier that waits until all the threads complete the execution the assigned sections.

Here, we added 5 different sections. Theoretically, the performance should remain the same for thread count of 5 or above. But practically, the performance will degrade due to the communication overhead and thread scheduling.

OBSERVATION

COMPILATION AND EXECUTION

Enabling OpenMP environment use -fopenmp flag while compiling using gcc.

```
gcc -fopenmp -O0 vectorSum.c -o vectorSum
```

For execution, use

```
./vectorSum NUMBER_OF_THREADS
```

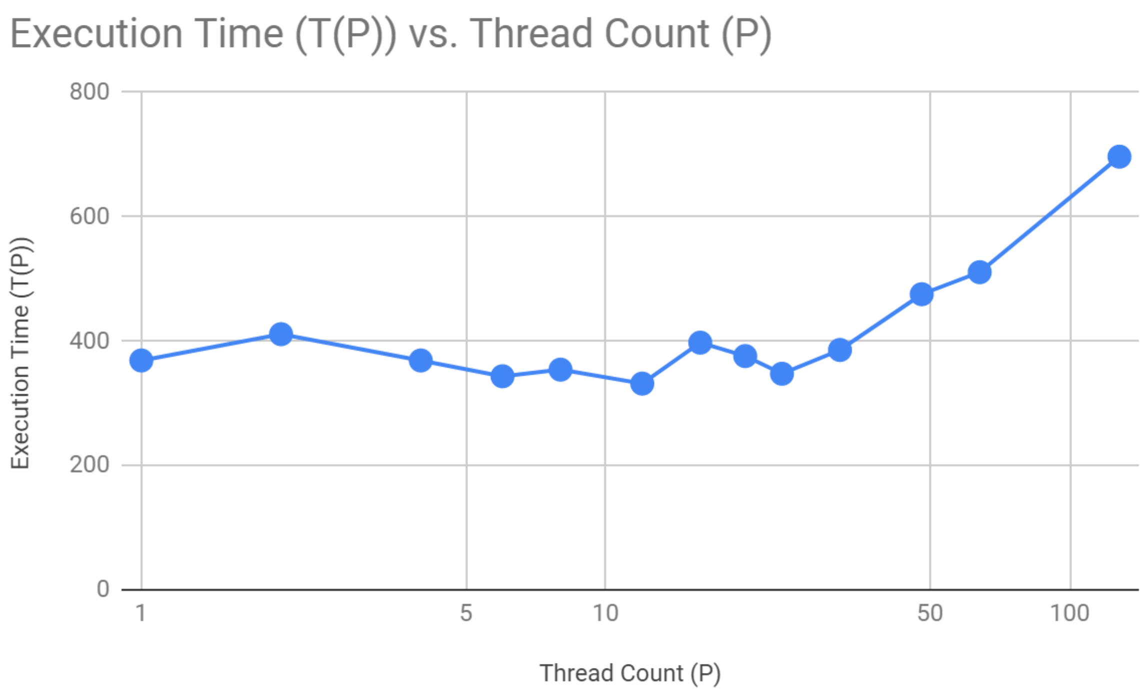
Or to change the number of threads in the environment variables,

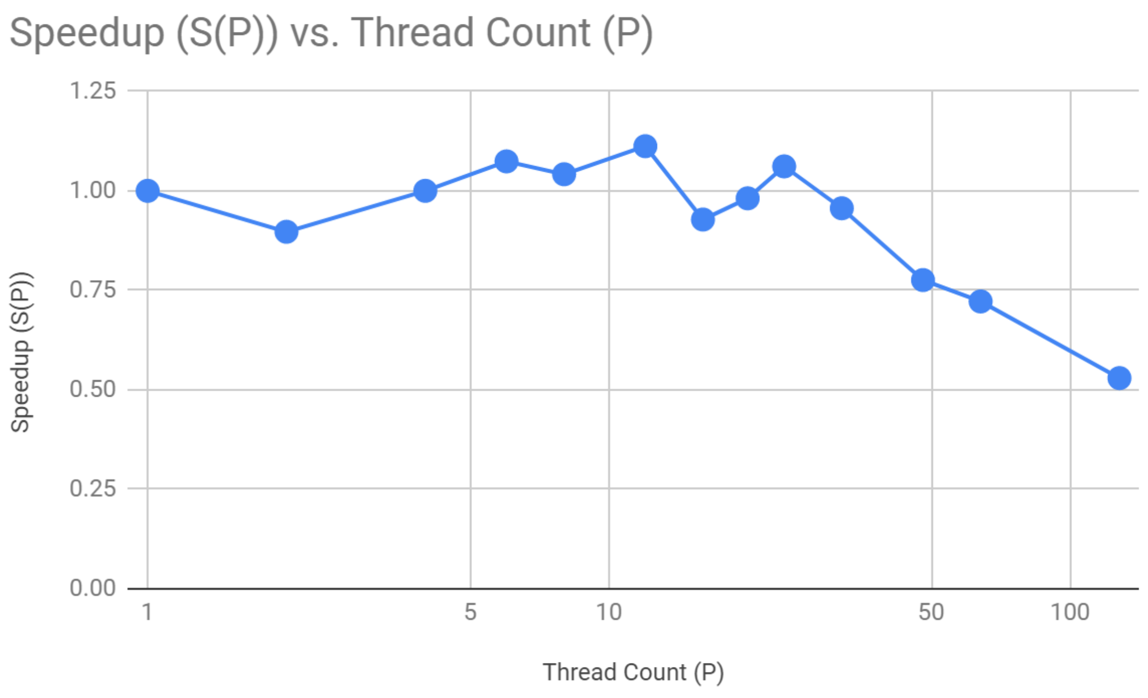
```
export OMP_NUM_THREADS=1
```

TABLE AND GRAPHS

Thread Count (P)	Execution Time (T(P))	Speedup (S(P))	Parallelization Fraction (f(P))
1	368.773989	1	N/A
2	411.063063	0.8971226612	-22.93495488
4	368.771708	1.000006185	0.0008247147098
6	343.365947	1.073996977	8.267841906
8	354.015732	1.041688139	4.573690101
12	331.581223	1.11216789	11.00238297
16	397.436363	0.9278818531	-8.290497661
20	375.846401	0.9811827066	-2.018755236
24	347.42177	1.06145907	6.041797147
32	385.661776	0.9562108872	-4.727164833
48	475.343543	0.7758051928	-29.51319631
64	510.750549	0.7220236762	-39.11071835
128	696.555429	0.5294251881	-89.58397449

GRAPHS


Execution Time (T(P)) vs. Thread Count (P)
800
600
400
200
0
Execution Time (T(P))
1
5
10
50
100
Thread Count (P)


Speedup (S(P)) vs. Thread Count (P)
1.25
1.00
0.75
0.50
0.25
0.00
Speedup (S(P))
1
5
10
50
100
Thread Count (P)

CONCLUSION

We see the **highest speedup of 1.11x at thread count of 12**, where the execution time is reduced by 37.22 seconds to generate and visualize the Lorenz attractor compared to the serial code.

Also, at thread count of 12, we observe a **parallelization fraction of 11%.**

The performance reduces on thread counts of 32 and above.

REFERENCES

[1] Wikipedia, “Lorenz system - Wikipedia,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Lorenz_system.

[2] Wikipedia, “Chaos theory - Wikipedia,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Chaos_theory.

[3] Wikipedia, “Butterfly effect - Wikipedia,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Butterfly_effect.

[4] Wikipedia, “Atmospheric convection - Wikipedia,” Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Atmospheric_convection.

[5] E. Lorenz, “The statistical prediction of solutions of dynamic equations.,” Meteorological Society of Japan, Tokyo, 1962.

[6] J. Burkardt, “LORENZ_ODE - The Lorenz System,” Florida State University, [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/lorenz_ode/lorenz_ode.html.