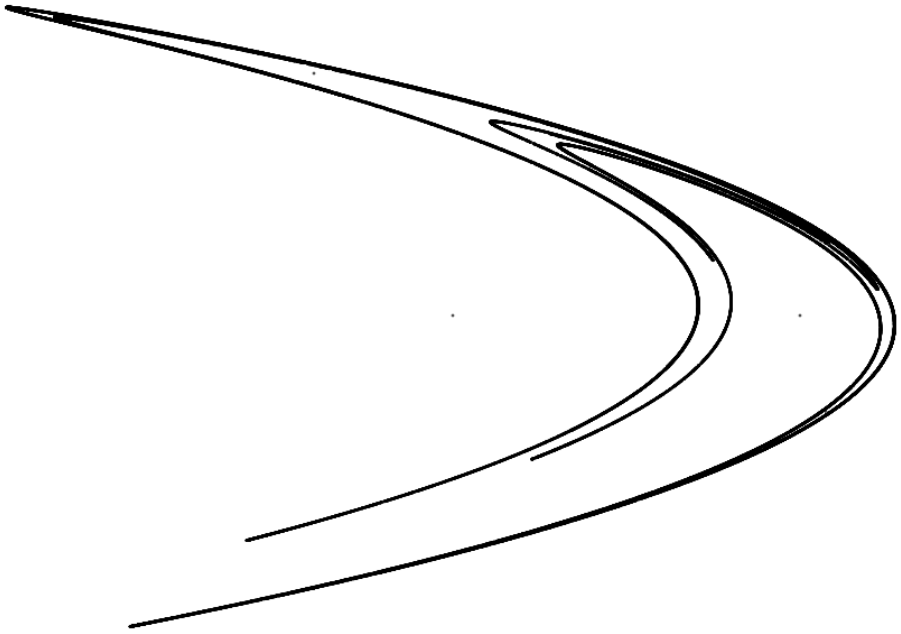


Henon Map

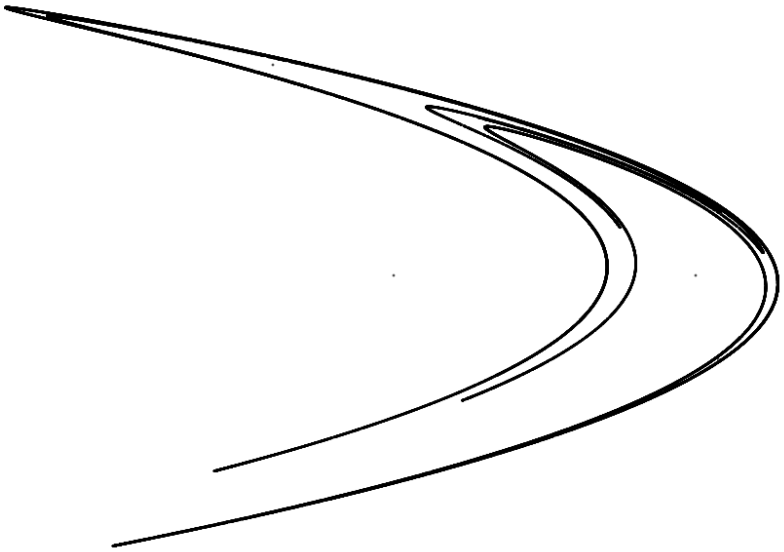
A THEORETICAL ANALYSIS

Samarth Sudarshan Inamdar (CED18I045) | High Performance Computing
August 2021

Department of Computer Science and Engineering,
Indian Institute of Information Technology Design and Manufacturing Kancheepuram, Chennai

Table of Contents

Introduction



The Henon map is an iterated discrete-time dynamical system that exhibits chaotic behavior in two-dimension. It is sometimes called Hénon-Pomeau attractor/map is a discrete-time dynamical system. It is one of the most studied examples of dynamical systems that exhibit chaotic behavior.

What is Henon Map?

The Henon map presents a simple two-dimensional invertible iterated map with quadratic nonlinearity and chaotic solutions called strange attractor. Strange attractors are a link between the chaos and the fractals.

The Henon map takes a point (x_n, y_n) in the plane and maps it to a new point. The below equation helps us to find the next point based on the previous point.

$$x_{n+1} = 1 - ax_n^2 + y_n$$
$$y_{n+1} = bx_n .$$

Code (Henon Map)

SERIAL CODE

```
// Compilation
// g++ main.cpp -o main -lGL -lGLU -lglut -lm -lGLEW
#include <stdio.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
#include <GL/freeglut.h>

typedef struct point
{
    double x;
    double y;
} Point;

int start = 0, end = 100000;
double a = 1.4;
double b = 0.3;
static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;
Point initial = {1.0, 1.0};

Point eq(Point prev, Point pprev)
{
    Point p;
    p.y = b * prev.x;
    p.x = 1 - a * prev.x * prev.x + b * pprev.x;
    return p;
}

void henon_gen()
{
    int n = end - start;
    Point *coord = (Point *)malloc(n * sizeof(Point));
    coord[0] = 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(1.2);
    coord[1].x = 1 - a * coord[0].x * coord[0].x + coord[0].y;
```

```
    coord[1].y = b * coord[0].x;
    for (int i = 1; i < n; i++)
    {
        coord[i] = eq(coord[i - 1], coord[i - 2]);
        glBegin(GL_POINTS);
        glColor3f(1, 1, 1);
        glVertex2f(coord[i].x, coord[i].y);
        glEnd();
        glFlush();
        glutSwapBuffers();
    }
}

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(-1.5, 1.5, -1.5, 1.5, -1.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
}

int main()
{
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Henon Map");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(henon_gen);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
    glutLeaveMainLoop();
}
```

PARALLEL CODE

```
#include <stdio.h>
#include <GL/glut.h>
#include <GL/glu.h>
#include <GL/gl.h>
#include <GL/freeglut.h>
#include <stddef.h>
#include <mpi.h>

typedef struct point
{
    double x;
    double y;
} Point;

int start = 0, end = 1024;
double a = 1.4;
double b = 0.3;
static GLfloat theta[] = {0.0, 0.0, 0.0};
GLint axis = 1;
Point initial = {1.0, 1.0};

Point eq(Point prev, Point pprev)
{

    int myid, numprocs;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Status status;

    MPI_Bcast(&prev, sizeof(Point), MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_Bcast(&pprev, sizeof(Point), MPI_CHAR, 0, MPI_COMM_WORLD);
    Point p;
    double temp1, temp2;
    switch (myid)
    {
    case 1:
        temp1 = a * prev.x * prev.x;
```

```
        MPI_Send(&temp1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        break;
    case 2:
        temp2 = b * pprev.x;
        MPI_Send(&temp2, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        break;
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if (myid == 0)
    {
        MPI_Recv(&temp1, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&temp2, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD, &status);
        p.x = 1 - temp1 + temp2;
        p.y = b * prev.x;
    }

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Bcast(&p, sizeof(Point), MPI_CHAR, 0, MPI_COMM_WORLD);

    return p;
}

void plot(Point coord)
{
    glBegin(GL_POINTS);
    glColor3f(1, 1, 1);
    glVertex2f(coord.x, coord.y);
    glEnd();
    glFlush();
    glutSwapBuffers();
}

void henon_gen()
{
    int n = end - start;
    Point *coord = (Point *)malloc(n * sizeof(Point));
    coord[0] = initial;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glLoadIdentity();
    coord[1].x = 1 - a * coord[0].x * coord[0].x + coord[0].y;
    coord[1].y = b * coord[0].x;
    double start_time = MPI_Wtime();
    for (int i = 1; i < n; i++)
    {
        coord[i] = eq(coord[i - 1], coord[i - 2]);
        plot(coord[i]);
    }
    double end_time = MPI_Wtime();
    printf("%f \n", end_time - start_time);
    MPI_Finalize();
    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)
{
    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(1.2);
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.5, 1.5, -1.5, 1.5, -1.5, 1.5);
    glMatrixMode(GL_MODELVIEW);
}
```

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    glutCreateWindow("Henon Map");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(henon_gen);
    glutIdleFunc(spinCube);
    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
```

Critical Section

```
switch (myid)
    {
    case 1:
        temp1 = a * prev.x * prev.x;
        MPI_Send(&temp1, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        break;
    case 2:
        temp2 = b * pprev.x;
        MPI_Send(&temp2, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
        break;
    }

    MPI_Barrier(MPI_COMM_WORLD);
    if (myid == 0)
    {
        MPI_Recv(&temp1, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);
        MPI_Recv(&temp2, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD, &status);
        p.x = 1 - temp1 + temp2;
        p.y = b * prev.x;
    }
```

Above code snippet is the critical section of the code i.e., major computation is being done here as this part runs most frequently.

This part calculates the next point using the previous point using the henon equation stated above. Since there is dependency of the current point with the previous point, this gives us a very small room for parallelization and performance improvement.

We have two cases in the switch case which corresponds to the id of the processor. Hence, we need at least 3 processors (1 master [myid == 0] and 2 slaves [myid == 1 and 2]) as the two slaves need to work on the intermediate operations that are provided in the switch case. After these intermediate operations are done, the slaves send the data to the master.

Now, there is a barrier just before master computes the final values as it needs to wait until all the slaves have completed the computation and have sent all the required data after which the master finally calculates the value of (x,y) that are then plotted on the screen.

Observations

COMPILATION AND EXECUTION

Compiling using mpi compiler with OpenGL essential flags enabled.

```
mpic++ -o mpi mpi.cpp -lm -lGL -lGLU -lglut -lGLEW
```

For execution, use

```
mpirun -n X -f machinefile ./mpi
```

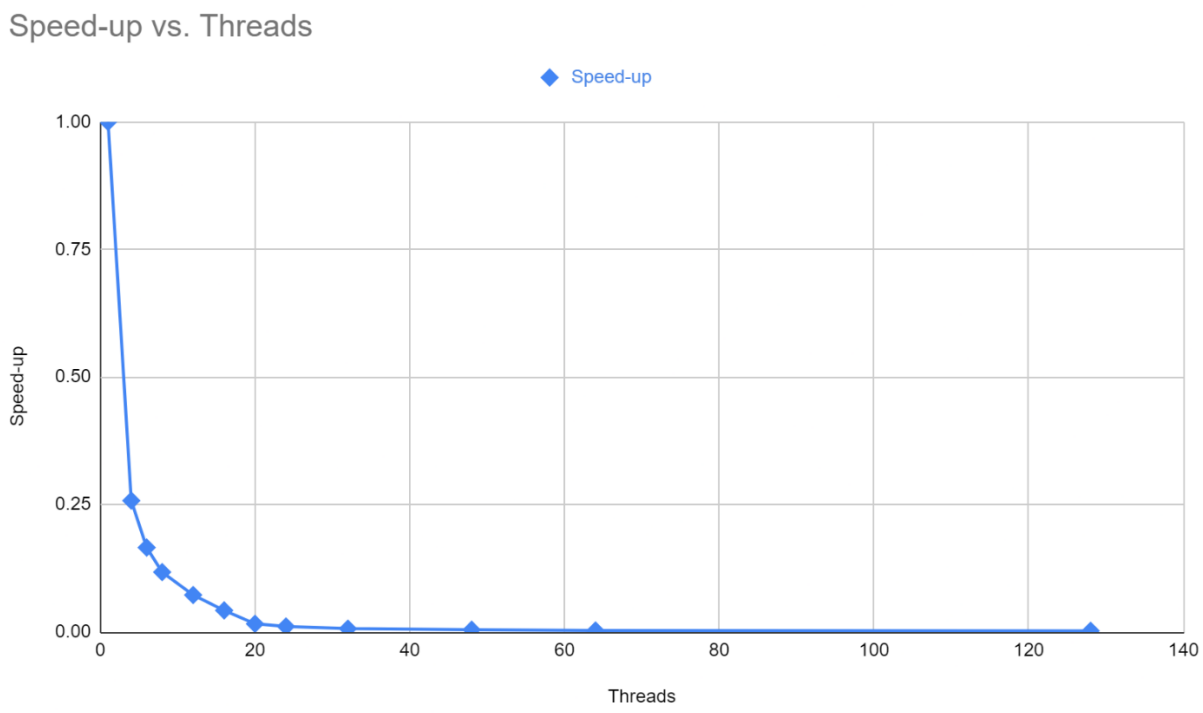
X: Number of processor and the minimum value of X here should be at least 3 as explained in the critical section.

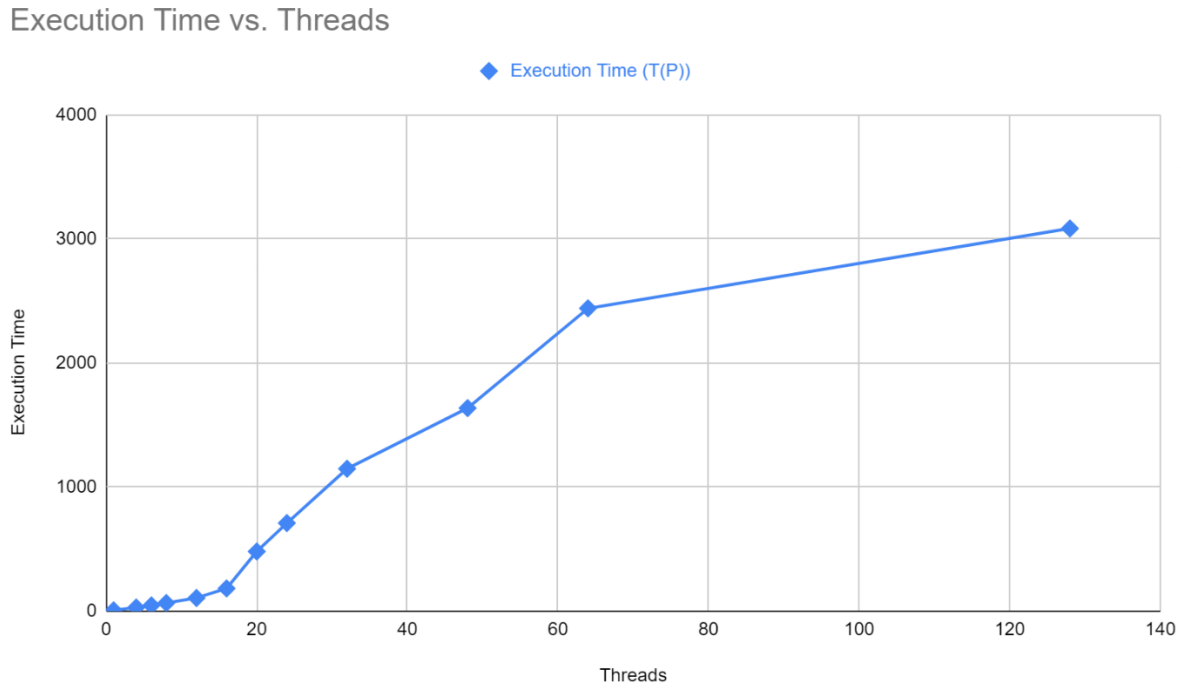
OBSERVATION TABLE

When this program was executed on my system, the system was crashing hence it was executed on my friend's system.

Observations			
Thread Count (P)	**Execution Time (T(P))**	**Speedup (S(P))**	**Parallelization Fraction (f(P))**
1	7.835281	1	N/A
4	30.406821	0.2576816892	-384.1009233
6	47.299155	0.1656537205	-604.4026857
8	66.642128	0.1175724911	-857.7589641
12	107.569953	0.07283893672	-1388.609552
16	183.970923	0.04258977926	-2397.846588
20	481.591165	0.01626956965	-6364.677977
24	709.813905	0.0110385003	-9348.732148
32	1147.673782	0.006827097667	-15016.78734
48	1636.280732	0.00478846988	-21225.69984
64	2439.732692	0.003211532569	-31530.44488
128	3083.543478	0.002540999034	-39563.6911

GRAPH


Speed-up vs. Threads
Speed-up
1.00
0.75
0.50
0.25
0.00
Speed-up
0
20
40
60
80
100
120
140
Threads


Execution Time vs. Threads
Execution Time (T(P))
4000
3000
2000
1000
0
Execution Time
0
20
40
60
80
100
120
140
Threads

Conclusion

The performance of the program becomes worse when using a mpi cluster as we are dividing the same resources between different virtual machines.

At any given instance of the program's runtime, there are 9 OS (1 Windows 11 and 8 Ubuntu 20.10) running, using the common resources from the laptop, this bottlenecks the program. Also, there are multiple other programs running in background which consume resources hence reducing the performance.

Also, since for computation of one point we need the previous points, we need to have multiple sends and receives which increases the time of execution. So, major part of the program is the communication overhead rather than the program execution which can be clearly observed from the graphs above as when we increase the number of threads the execution time increases drastically.

References

Wikipedia | Henon Map

Wikipedia |Dynamical System

Wikipedia | Attractor

Dynamical Properties of the Hénon Mapping