VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

BCSE303L

OPERATING SYSTEMS

TITLE

Enhancing Predictive Model Performance through Multi-Processor Scheduling and Parallel Processing

Submitted By

Dakshin Ghai 22BRS1073 , Pranav Parab 22BPS1026
Shaivya Agrawal 22BAI1151

Submitted To

VALLIDEVI K

Introduction:

Predictive modeling is a fundamental part of data science, helping us create smart algorithms for making predictions and decisions based on data. But as our datasets get bigger and more complex, our predictive models need to work harder. To tackle this challenge, we've come up with a smart strategy: using multiple processors and running tasks at the same time, which makes our models work faster and handle tons of data.

In this context, we'll explore how using multiple processors and parallel processing can supercharge our predictive models. It's like giving them a speed boost, making them more efficient and capable of handling a lot of data.

Normal Distribution :

The normal distribution, often referred to as the Gaussian distribution, is a fundamental concept in statistics and probability theory. It describes the way data tends to be distributed in many natural and man-made processes.

The normal distribution represents a common pattern in data where most values cluster around a central point, forming a symmetric, bell-shaped curve. This central point is the mean (average) of the data. In a normal distribution:

1. The data is balanced, with roughly equal numbers of values on either side of the mean.

2. The curve is symmetrical, meaning it looks the same on both sides of the mean.

3. The majority of data points are close to the mean, with fewer data points as you move away from it.

4. It's continuous, which means it can take any real value (decimal numbers) within a range.

5. It's characterized by two parameters: the mean (μ), which defines the central

point, and the standard deviation (σ), which measures the spread or dispersion of the data.

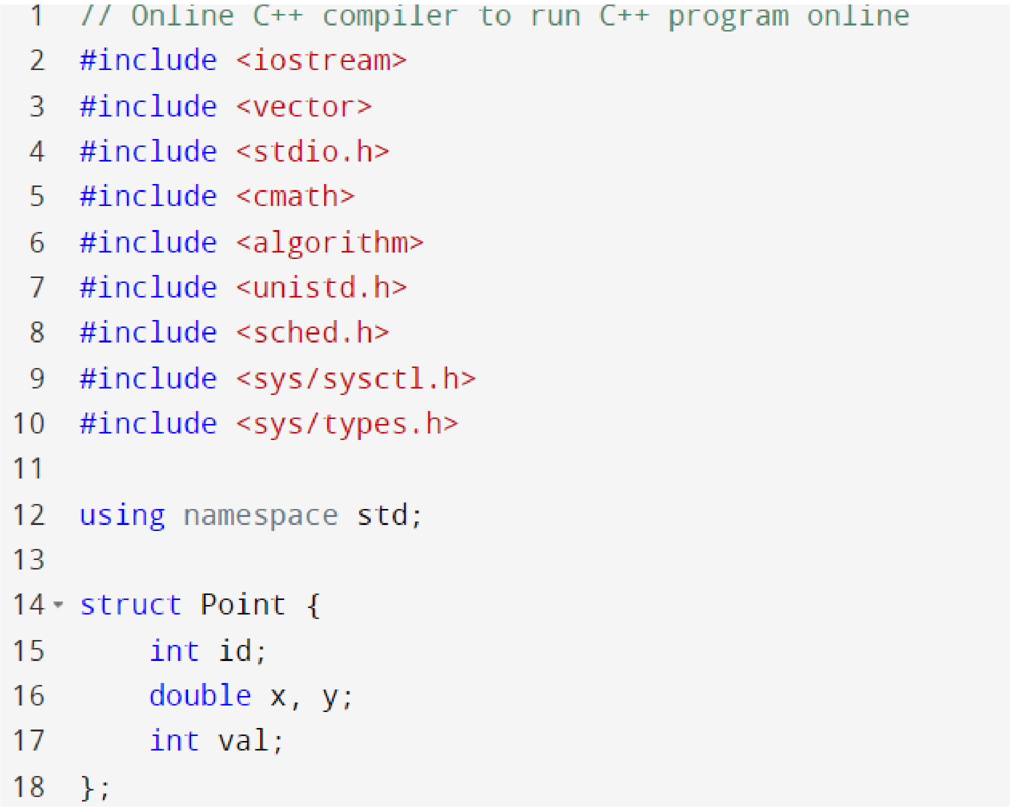
Normal distributions are crucial in various fields, including statistics, science, finance, and quality control, because many real-world phenomena tend to follow this pattern. They allow us to make predictions and understand the likelihood of events occurring within a certain range of values. In summary, the normal distribution helps us understand how data tends to be distributed and make informed decisions based on that understanding.

Code:

This code is a C++ program that performs some data classification using a basic form of the Gaussian Naive Bayes algorithm. It also demonstrates how to use multiple CPU cores by forking child processes to perform the classification task in parallel.

- Header Includes:

- The code includes several standard C and C++ libraries, including `<iostream>`, `<vector>`, `<stdio.h>`, `<cmath>`, `<algorithm>`, `<unistd.h>`, `<sched.h>`, `<sys/sysctl.h>`, and `<sys/types.h>`. These libraries provide functions and data types for working with processes, CPU affinity, and mathematical calculations.



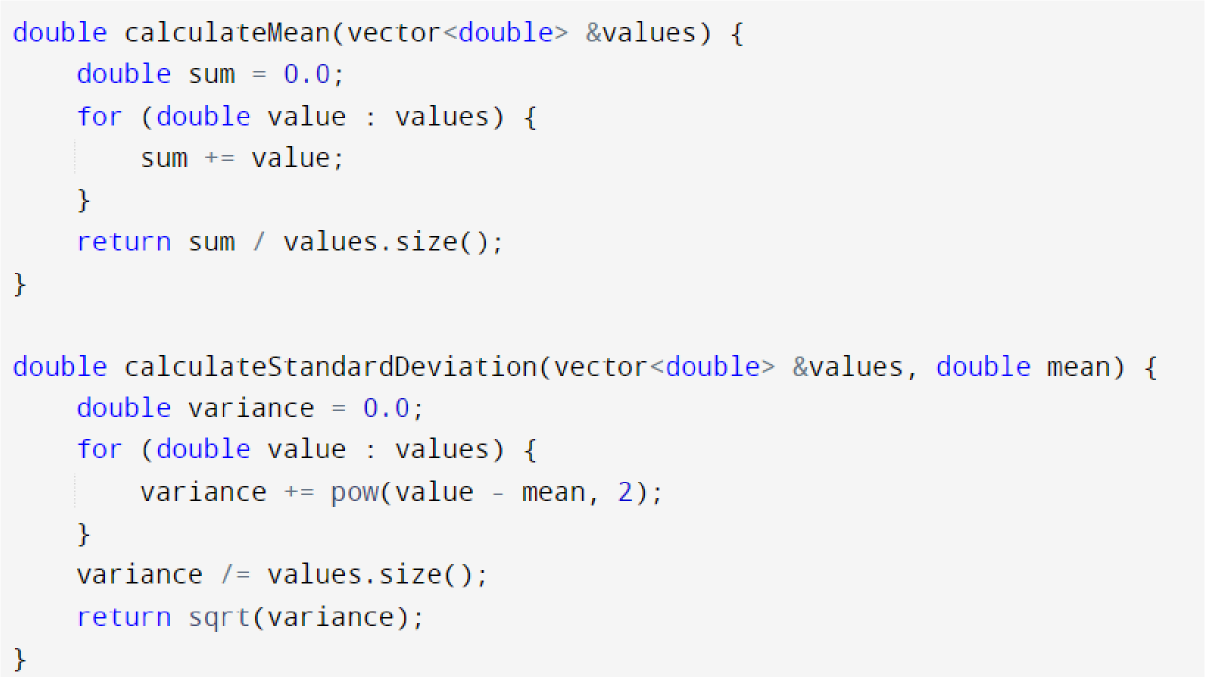
```cpp
// Online C++ compiler to run C++ program online
#include <iostream>
#include <vector>
#include <stdio.h>
#include <cmath>
#include <algorithm>
#include <unistd.h>
#include <sched.h>
#include <sys/sysctl.h>
#include <sys/types.h>

using namespace std;

struct Point {
    int id;
    double x, y;
    int val;
};
```

- Using Namespace

- The code uses the `using namespace std;` statement, which allows using standard C++ features without the `std::` prefix.

- Point Struct:

- The `Point` struct represents a data point with four attributes: `id` (an identifier), `x` and `y` (numerical features), and `val` (the class label, either 0 or 1).



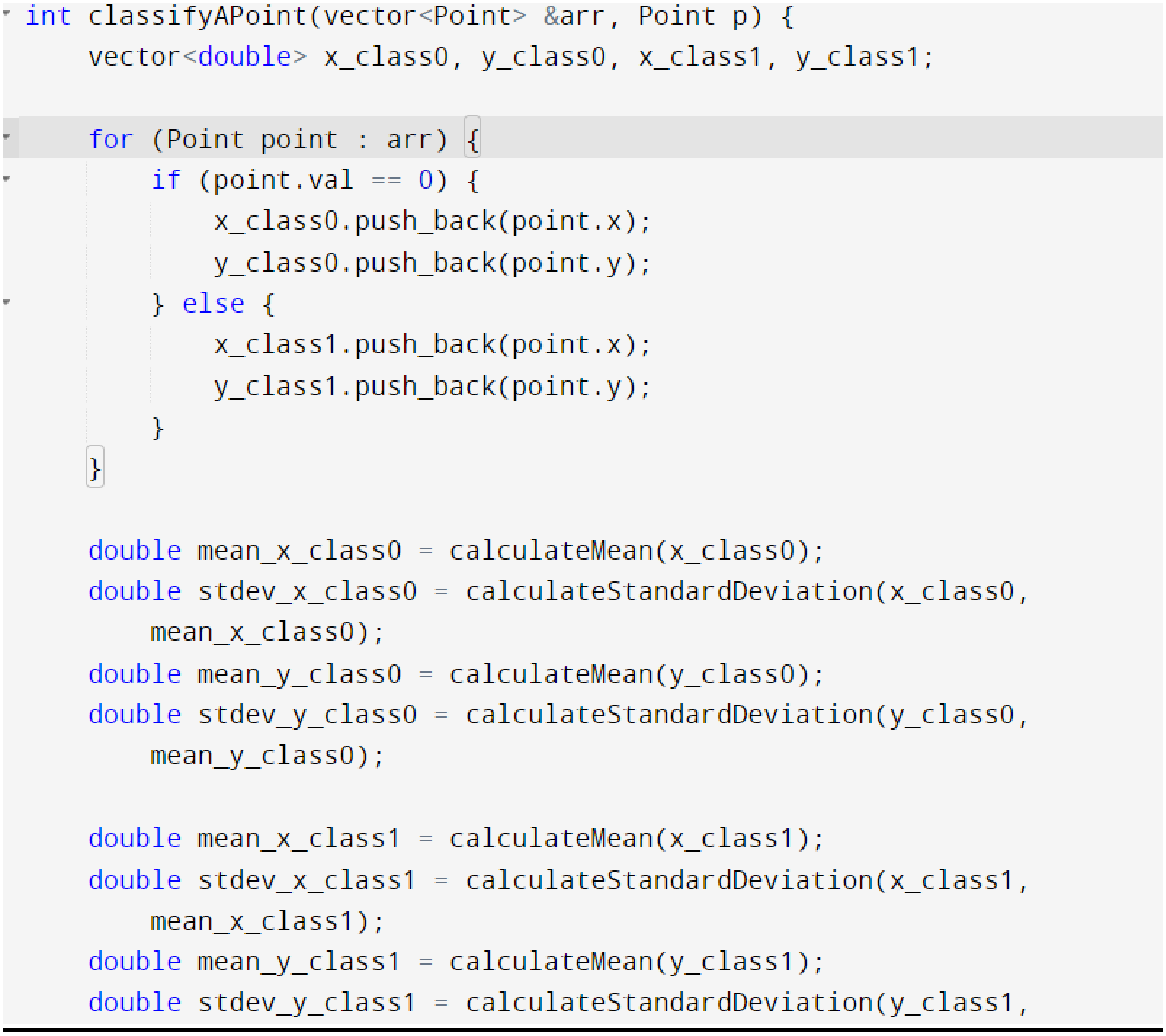
```
double calculateMean(vector<double> &values) {
    double sum = 0.0;
    for (double value : values) {
        sum += value;
    }
    return sum / values.size();
}

double calculateStandardDeviation(vector<double> &values, double mean) {
    double variance = 0.0;
    for (double value : values) {
        variance += pow(value - mean, 2);
    }
    variance /= values.size();
    return sqrt(variance);
}
```

- calculateMean Function:

- This function calculates the mean (average) of a vector of double values.

- calculateStandardDeviation Function:

- This function calculates the standard deviation of a vector of double values, given the mean.



```
int classifyAPoint(vector<Point> &arr, Point p) {
    vector<double> x_class0, y_class0, x_class1, y_class1;

    for (Point point : arr) {
        if (point.val == 0) {
            x_class0.push_back(point.x);
            y_class0.push_back(point.y);
        } else {
            x_class1.push_back(point.x);
            y_class1.push_back(point.y);
        }
    }

    double mean_x_class0 = calculateMean(x_class0);
    double stdev_x_class0 = calculateStandardDeviation(x_class0,
        mean_x_class0);
    double mean_y_class0 = calculateMean(y_class0);
    double stdev_y_class0 = calculateStandardDeviation(y_class0,
        mean_y_class0);

    double mean_x_class1 = calculateMean(x_class1);
    double stdev_x_class1 = calculateStandardDeviation(x_class1,
        mean_x_class1);
    double mean_y_class1 = calculateMean(y_class1);
    double stdev_y_class1 = calculateStandardDeviation(y_class1,
```

- classifyAPoint(): This function performs the actual classification. It takes a vector of points (`arr`) and a point `p` as input. It separates the data points into two classes (0 and 1), calculates mean and standard deviation for each feature (x and y) in both classes, and then uses these statistics to calculate the likelihood for each class and make a prediction.

```
    double mean_y_class1 = calculateMean(y_class1);
    double stdev_y_class1 = calculateStandardDeviation(y_class1,
        mean_y_class1);

    double likelihood_class0 = exp(-0.5 * (pow((p.x - mean_x_class0) /
        stdev_x_class0, 2) + pow((p.y - mean_y_class0) / stdev_y_class0, 2
        )));
    double likelihood_class1 = exp(-0.5 * (pow((p.x - mean_x_class1) /
        stdev_x_class1, 2) + pow((p.y - mean_y_class1) / stdev_y_class1, 2
        )));

    // Prior probabilities (assumed equal in this basic example)
    double prior_class0 = static_cast<double>(x_class0.size()) / arr.size
        ();
    double prior_class1 = static_cast<double>(x_class1.size()) / arr.size
        ();

    // Posterior probabilities
    double posterior_class0 = likelihood_class0 * prior_class0;
    double posterior_class1 = likelihood_class1 * prior_class1;

    return (posterior_class0 > posterior_class1) ? 0 : 1;
}
```

- main Function:

- The `main` function is the entry point of the program.

- It defines a vector `arr` containing 17 data points with features and class labels.

- CPU Core Affinity:

- The code uses system calls and functions to manage CPU core affinity.

- It determines the number of available CPU cores using `sysconf(_SC_NPROCESSORS_ONLN)`.

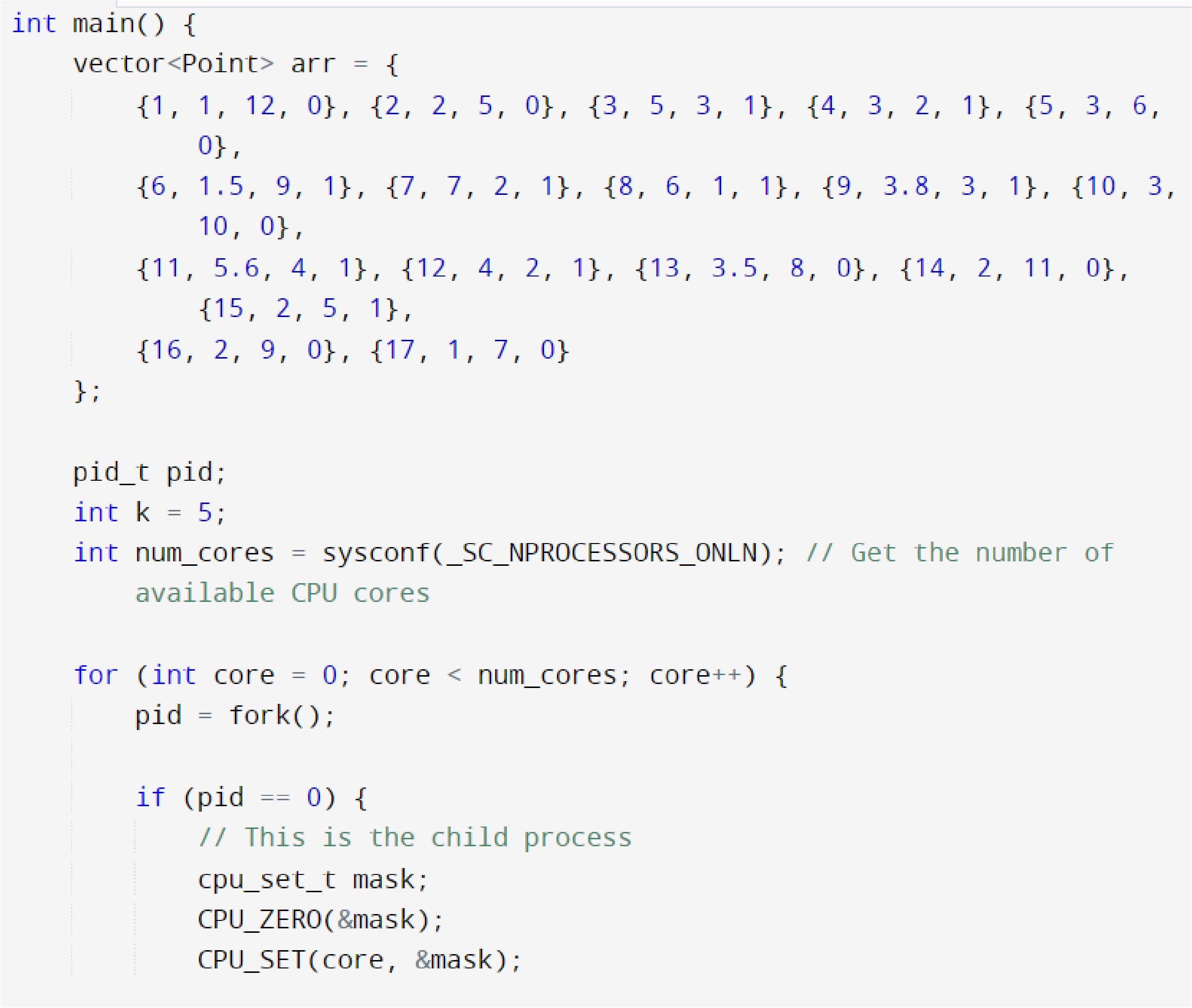
- It then enters a loop to create child processes, each assigned to a different CPU core using `fork`. This enables parallel execution of the classification task on multiple CPU cores.

- Child Process Configuration:

- Inside the loop, each child process sets its CPU affinity to a specific core.

- It prints the process ID (PID) and the core number it's running on.

- It then calls the `classifyAPoint` function to classify a test point `(2.5, 4)` and prints the predicted class.

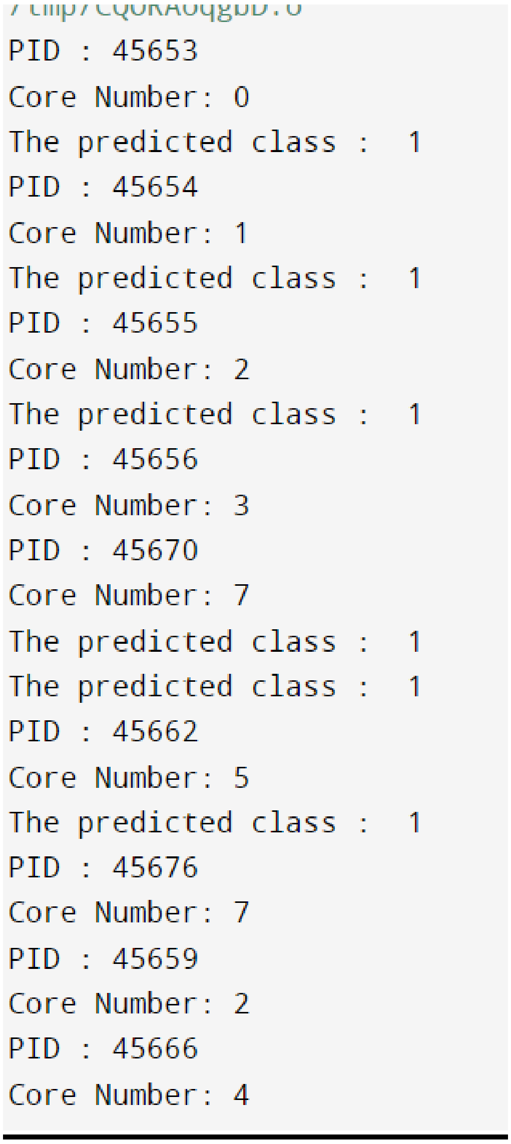


```
int main() {
    vector<Point> arr = {
        {1, 1, 12, 0}, {2, 2, 5, 0}, {3, 5, 3, 1}, {4, 3, 2, 1}, {5, 3, 6,
            0},
        {6, 1.5, 9, 1}, {7, 7, 2, 1}, {8, 6, 1, 1}, {9, 3.8, 3, 1}, {10, 3,
            10, 0},
        {11, 5.6, 4, 1}, {12, 4, 2, 1}, {13, 3.5, 8, 0}, {14, 2, 11, 0},
            {15, 2, 5, 1},
        {16, 2, 9, 0}, {17, 1, 7, 0}
    };

    pid_t pid;
    int k = 5;
    int num_cores = sysconf(_SC_NPROCESSORS_ONLN); // Get the number of
        available CPU cores

    for (int core = 0; core < num_cores; core++) {
        pid = fork();

        if (pid == 0) {
            // This is the child process
            cpu_set_t mask;
            CPU_ZERO(&mask);
            CPU_SET(core, &mask);
        if (sched_setaffinity(0, sizeof(mask), &mask) == -1) {
            perror("sched_setaffinity");
        }

        // Now each child process can work on a specific core
        printf("PID : %d\n", getpid());
        printf("Core Number: %d\n", sched_getcpu());

        // You can call classifyAPoint here with the data you want to
            classify

Point p;
p.x = 2.5;
p.y = 4;

int predicted_class = classifyAPoint(arr, p);
cout << "The predicted class :  " << predicted_class << endl;
    } else if (pid < 0) {
        perror("fork");
    }
}
```

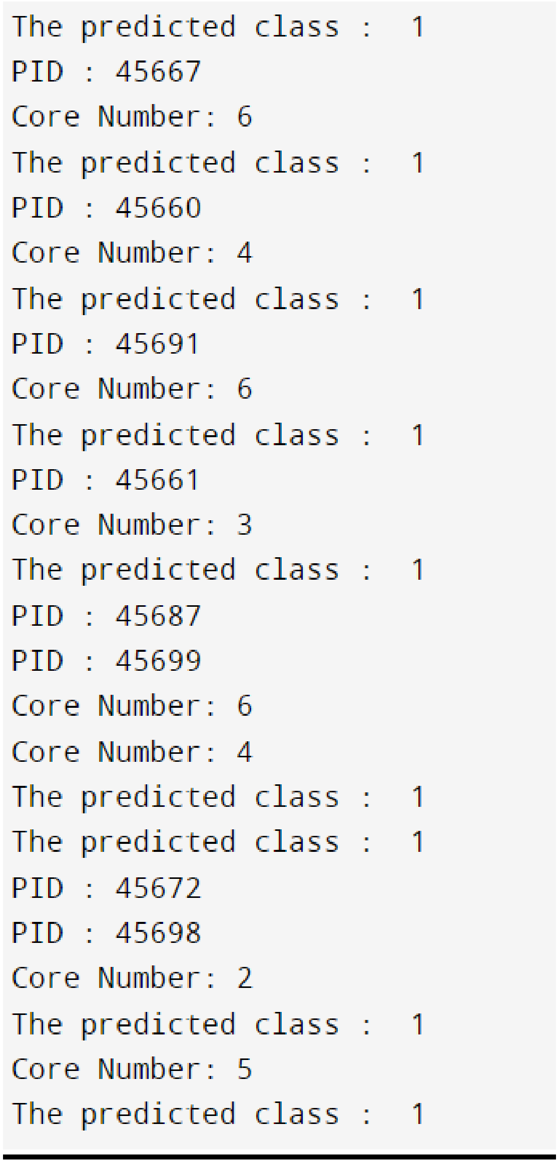
```
    // The parent process doesn't do anything, just waits for children to
        finish


    return 0;
}
```

- The parent process (the original program) simply waits for all child processes to finish.

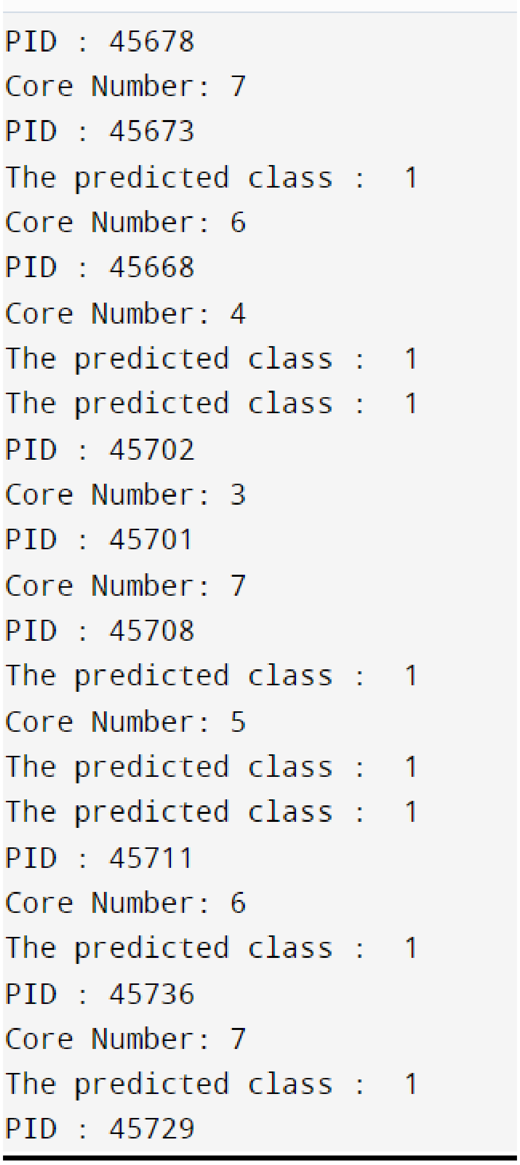
Output:



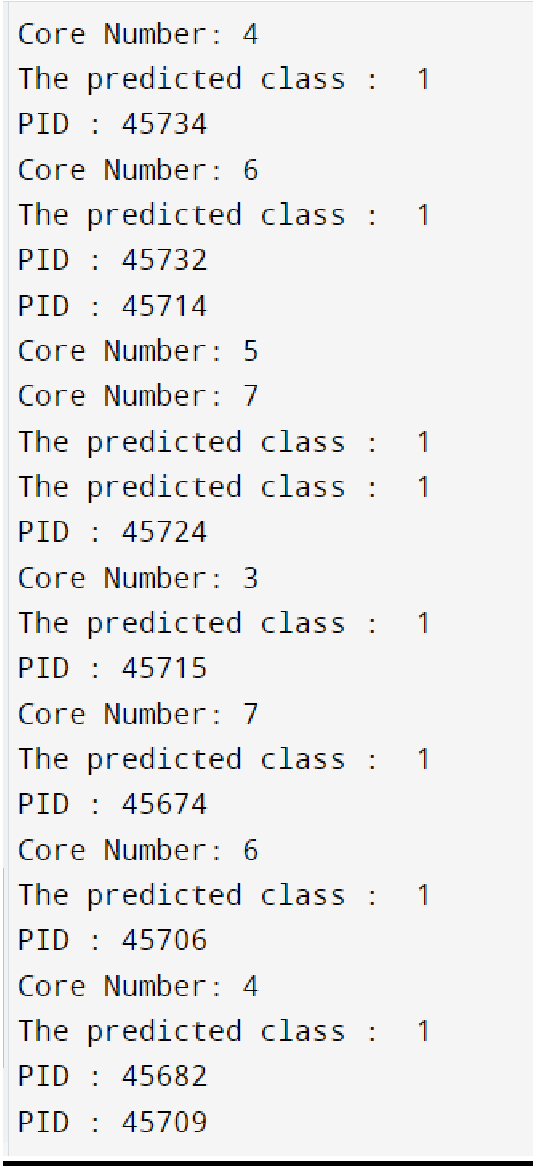
```
/tmp/CQoRAoqgbD.o
PID : 45653
Core Number: 0
The predicted class :  1
PID : 45654
Core Number: 1
The predicted class :  1
PID : 45655
Core Number: 2
The predicted class :  1
PID : 45656
Core Number: 3
PID : 45670
Core Number: 7
The predicted class :  1
The predicted class :  1
PID : 45662
Core Number: 5
The predicted class :  1
PID : 45676
Core Number: 7
PID : 45659
Core Number: 2
PID : 45666
Core Number: 4
Core Number: 4
The predicted class :  1
The predicted class :  1
The predicted class :  1
PID : 45671
Core Number: 5
The predicted class :  1
PID : 45677
Core Number: 7
The predicted class :  1
PID : 45663
Core Number: 3
The predicted class :  1
PID : 45665
Core Number: 4
The predicted class :  1
PID : 45669
PID : 45657
Core Number: 1
Core Number: 5
The predicted class :  1
The predicted class :  1
PID : 45658
Core Number: 2
The predicted class :  1
```



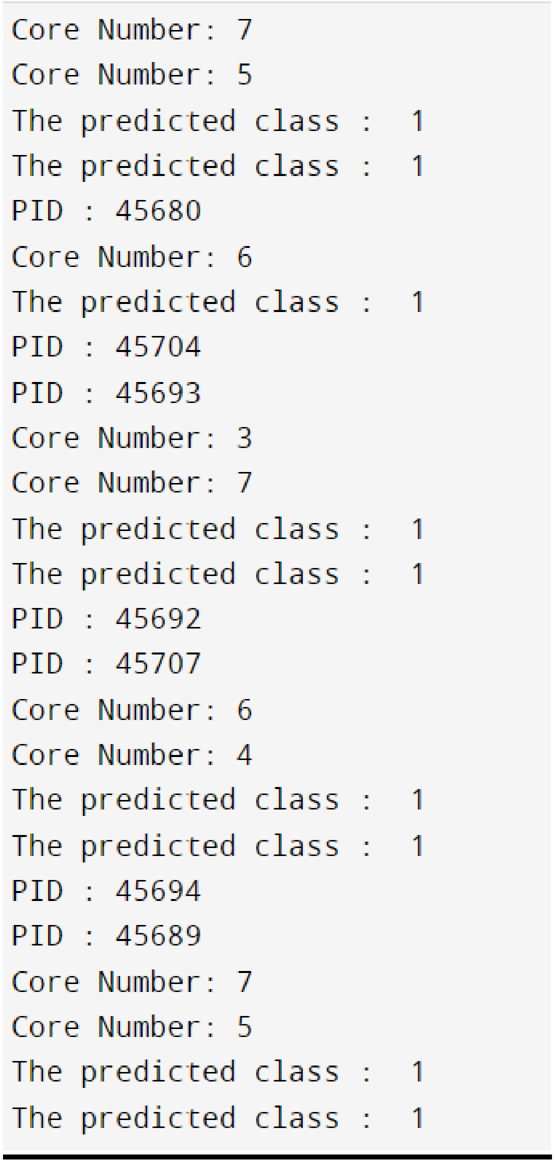
```
The predicted class :  1
PID : 45667
Core Number: 6
The predicted class :  1
PID : 45660
Core Number: 4
The predicted class :  1
PID : 45691
Core Number: 6
The predicted class :  1
PID : 45661
Core Number: 3
The predicted class :  1
PID : 45687
PID : 45699
Core Number: 6
Core Number: 4
The predicted class :  1
The predicted class :  1
PID : 45672
PID : 45698
Core Number: 2
The predicted class :  1
Core Number: 5
The predicted class :  1
```



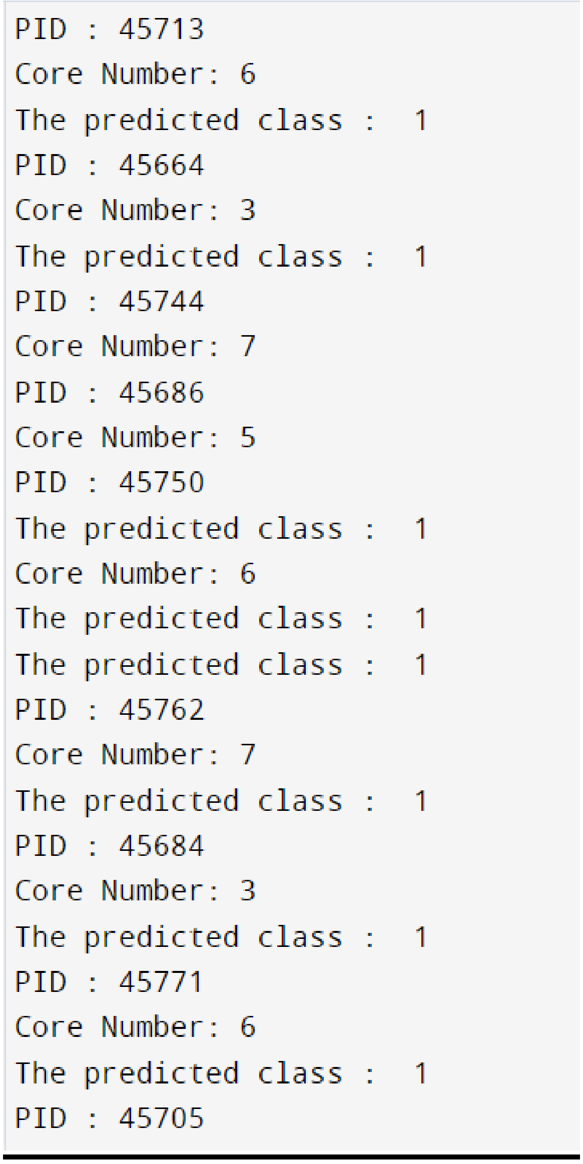
```
PID : 45678
Core Number: 7
PID : 45673
The predicted class :  1
Core Number: 6
PID : 45668
Core Number: 4
The predicted class :  1
The predicted class :  1
PID : 45702
Core Number: 3
PID : 45701
Core Number: 7
PID : 45708
The predicted class :  1
Core Number: 5
The predicted class :  1
The predicted class :  1
PID : 45711
Core Number: 6
The predicted class :  1
PID : 45736
Core Number: 7
The predicted class :  1
PID : 45729
```



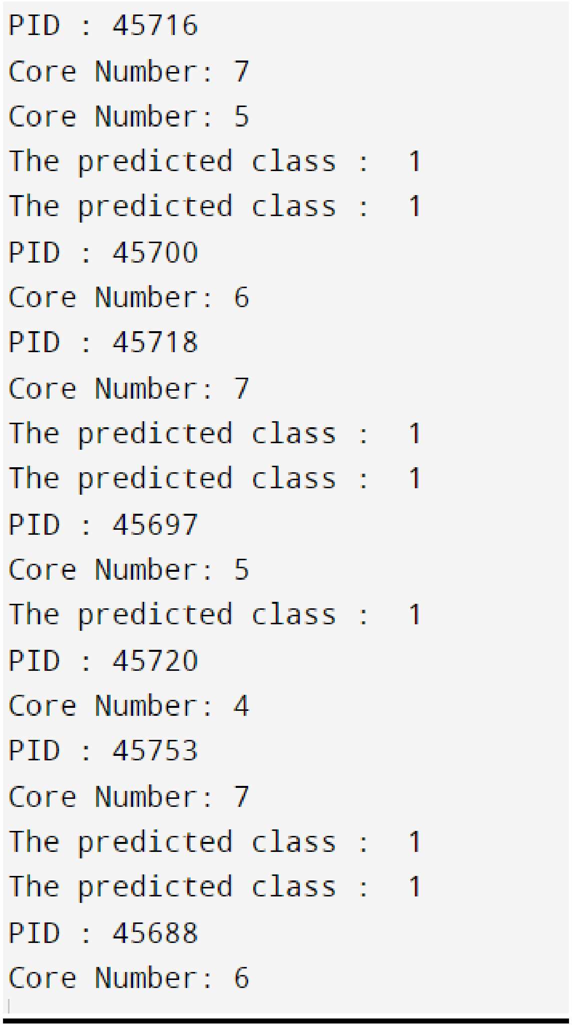
```
Core Number: 4
The predicted class :  1
PID : 45734
Core Number: 6
The predicted class :  1
PID : 45732
PID : 45714
Core Number: 5
Core Number: 7
The predicted class :  1
The predicted class :  1
PID : 45724
Core Number: 3
The predicted class :  1
PID : 45715
Core Number: 7
The predicted class :  1
PID : 45674
Core Number: 6
The predicted class :  1
PID : 45706
Core Number: 4
The predicted class :  1
PID : 45682
PID : 45709
```



```
Core Number: 7
Core Number: 5
The predicted class :  1
The predicted class :  1
PID : 45680
Core Number: 6
The predicted class :  1
PID : 45704
PID : 45693
Core Number: 3
Core Number: 7
The predicted class :  1
The predicted class :  1
PID : 45692
PID : 45707
Core Number: 6
Core Number: 4
The predicted class :  1
The predicted class :  1
PID : 45694
PID : 45689
Core Number: 7
Core Number: 5
The predicted class :  1
The predicted class :  1
```



```
PID : 45713
Core Number: 6
The predicted class :  1
PID : 45664
Core Number: 3
The predicted class :  1
PID : 45744
Core Number: 7
PID : 45686
Core Number: 5
PID : 45750
The predicted class :  1
Core Number: 6
The predicted class :  1
The predicted class :  1
PID : 45762
Core Number: 7
The predicted class :  1
PID : 45684
Core Number: 3
The predicted class :  1
PID : 45771
Core Number: 6
The predicted class :  1
PID : 45705
```



```
PID : 45716
Core Number: 7
Core Number: 5
The predicted class :  1
The predicted class :  1
PID : 45700
Core Number: 6
PID : 45718
Core Number: 7
The predicted class :  1
The predicted class :  1
PID : 45697
Core Number: 5
The predicted class :  1
PID : 45720
Core Number: 4
PID : 45753
Core Number: 7
The predicted class :  1
The predicted class :  1
PID : 45688
Core Number: 6
```

Conclusion:

Overall, this code demonstrates a basic example of parallelization and classification using Gaussian Naive Bayes, where different child processes run on separate CPU cores to perform classification tasks.