Parallel Processing for Determinant Computation in Cramer's Rule Solution of Linear Equations

Subject: Operating Systems Lab

Team : 10 24CS8091 – Indrajit Mondal

---

1. Aim

Our team's objective was to develop a system to compute the determinant of a square matrix and solve linear equations using Cramer's Rule. A core focus of the project was to implement both sequential and parallel versions of the algorithm to analyze performance differences and execution time across multiple processes.

2. Objectives

- To design a program capable of computing determinants for large matrices.
- To implement Cramer's Rule as a method for solving systems of linear equations.
- To manage memory dynamically for matrices and vectors to handle varying input sizes.
- To use srand() and rand() for generating randomized test datasets.
- To implement process-level parallelism using the fork() and wait() system calls.
- To conduct a comparative performance analysis between sequential and parallel execution.
- To measure and analyze the speedup achieved through parallelization.

3. Tools Used

- **Programming Language:** C
- **Compiler:** GCC
- **Operating System:** Linux / Ubuntu
- **Libraries / System Calls Used:** stdio.h, stdlib.h (Standard I/O and memory management)
    - math.h, time.h (Mathematical functions and timing)
    - unistd.h, sys/wait.h (Process management)
    - fork(), wait() (Process creation and synchronization)

4. System Configuration

- **Processor:** Ryzen 7
- **RAM:** 16 GB
- **OS:** Ubuntu Linux
- **Compiler Version:** GCC

5. Program Description

In this assignment, our team implemented a solution for determinant computation and system solving. We utilized dynamic memory allocation to generate a random coefficient matrix A and a constant vector B.

We calculated the determinant of matrix A using the **Gaussian Elimination** method. Using this result, we applied **Cramer's Rule** to solve for the unknown variables in the system.

Our implementation consists of two versions:

1. **Sequential Version:** Determinants are computed linearly, one after another, within a single process.
2. **Parallel Version:** The computation of determinants is distributed across multiple child processes created via the fork() system call.

We then measured the execution time of both implementations to observe and document the performance improvements provided by parallel processing.

6. Algorithm

6.1 Determinant Algorithm (Gaussian Elimination)

1. Start with a matrix A of size n x n.
2. Initialize the determinant to 1.
3. For each pivot row ii:
    - If the pivot element A[i][i] is 0, the determinant is set to 0.
    - Eliminate elements below the pivot by calculating the ratio and applying row operations.
4. Convert the matrix into an upper triangular form.
5. Multiply the diagonal elements to find the determinant value.

6.2 Sequential Cramer's Rule

1. Compute the main determinant det(A).
2. If det(A) = 0, the system has no unique solution.
3. For each variable X_i:
    - Copy matrix A and replace column i with vector B.
    - Compute det(A_i).
    - Calculate X_i = det(A_i) / det(A).
4. If n <= 10, we print the full solution vector.

6.3 Parallel Cramer's Rule

1. The parent process computes the initial det(A).
2. For each variable X_i, our code creates a child process using fork().
3. The child process is responsible for computing det(A_i) and calculating X_i.
4. The parent process uses wait() to synchronize with all child processes.

7. Program Structure

Our project follows a modular architecture:

- **src/**: Contains main.c, matrix.c, determinant.c, cramer.c, and parallel.c.
- **include/**: Contains corresponding .h header files.

Each member of our team focused on a specific module, such as memory management or the parallel logic, ensuring a clean separation of concerns.

8. Complexity Analysis

Operation	**Time Complexity**
Determinant (Gaussian)	O(n^3)
Sequential Cramer Solver	O(n^4)
Parallel Solver	Reduced runtime due to concurrency

9. Results / Output Behavior

During execution, our program prompts the user for matrix size. It then runs both versions and displays the results:

- **Sequential:** Outputs the solution vector (for n <= 10) and the time taken.
- **Parallel:** Executes using fork() and reports the execution time.
- **Speedup:** The program automatically calculates the speedup achieved by the parallel implementation.
- **Note :** We observed that because fork() creates separate address spaces, the X_i values computed by child processes are not natively shared back to the parent. Consequently, the final solution vector is not printed in the parallel version; instead, we focus on the timing of the parallel workload.

Code :

```
     #define MAX_PROC 8

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include <sys/wait.h>

////////////////////////////////////////////////////
// Matrix Utilities
////////////////////////////////////////////////////

double **allocateMat(int n)
{
    double **A = (double **)malloc(n * sizeof(double *));
    if (A == NULL) return NULL;

    for (int i = 0; i < n; i++)
    {
        A[i] = (double *)malloc(n * sizeof(double));
        if (A[i] == NULL) return NULL;
    }
    return A;
}

void freeMat(double **A, int n)
{
    for (int i = 0; i < n; i++)
        free(A[i]);
    free(A);
}

void copyMat(double **src, double **dest, int n)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dest[i][j] = src[i][j];
}

void replaceCol(double **A, double *b, int col, int n)
{
    for (int i = 0; i < n; i++)
        A[i][col] = b[i];
}

////////////////////////////////////////////////////
```

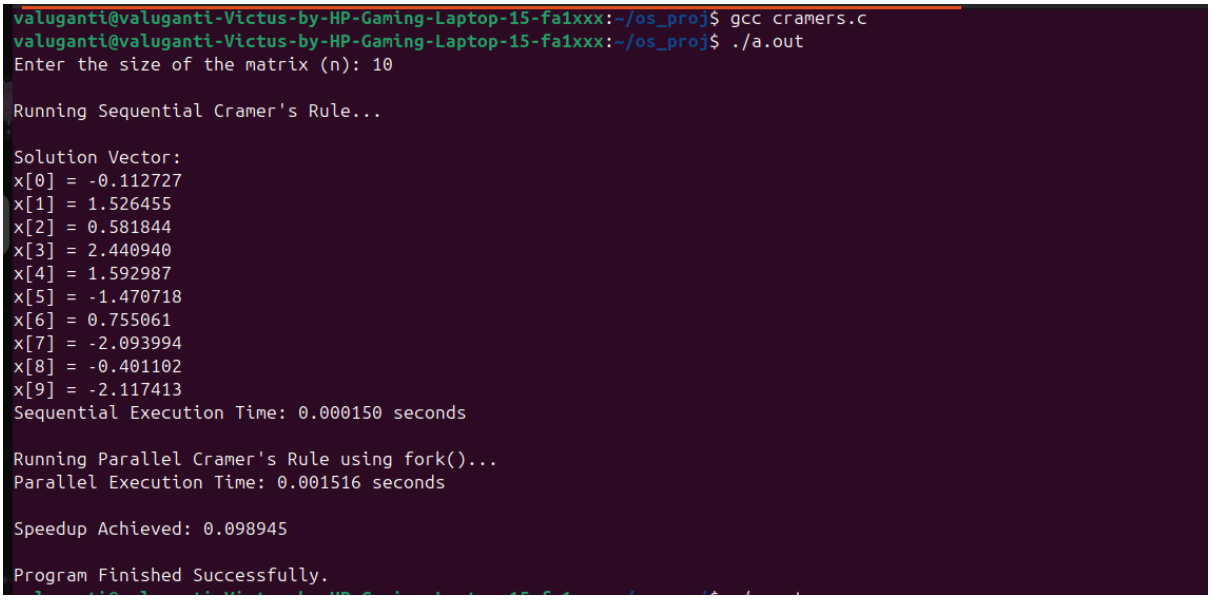
```
// Determinant using Gaussian Elimination
//////////////////////////////////////////////////////

double deter(double **A, int n)
{
    double det = 1.0;

    for (int i = 0; i < n; i++)
    {
        if (fabs(A[i][i]) < 1e-9)
            return 0;

        for (int j = i + 1; j < n; j++)
        {
            double ratio = A[j][i] / A[i][i];
            for (int k = 0; k < n; k++)
                A[j][k] -= ratio * A[i][k];
        }

        det *= A[i][i];
    }

    return det;
}

//////////////////////////////////////////////////////
// Sequential Cramer's Rule
//////////////////////////////////////////////////////

void c_sequential(double **A, double *b, double *x, int n)
{
    double **temp = allocateMat(n);
    copyMat(A, temp, n);

    double detA = deter(temp, n);
    freeMat(temp, n);

    if (detA == 0)
    {
        printf("System has no unique solution\n");
        return;
    }

    for (int i = 0; i < n; i++)
    {
        temp = allocateMat(n);
        copyMat(A, temp, n);
```

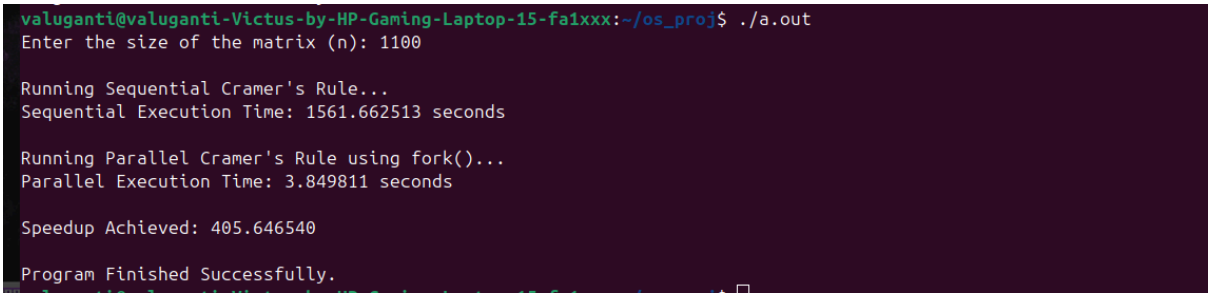
```
        replaceCol(temp, b, i, n);

        double detAi = deter(temp, n);
        x[i] = detAi / detA;

        freeMat(temp, n);
    }

    if (n <= 10)
    {
        printf("\nSolution Vector:\n");
        for (int i = 0; i < n; i++)
            printf("x[%d] = %lf\n", i, x[i]);
    }
}

//////////////////////////////////////////////////////
// Parallel Cramer's Rule using fork()
//////////////////////////////////////////////////////

void c_parallel(double **A, double *b, double *x, int n)
{
    double **temp = allocateMat(n);
    copyMat(A, temp, n);

    double detA = deter(temp, n);
    freeMat(temp, n);

    if (detA == 0)
    {
        printf("System has no unique solution\n");
        return;
    }

    for (int i = 0; i < n; i++)
    {
        pid_t pid = fork();

        if (pid == 0)   // child
        {
            double **local = allocateMat(n);
            copyMat(A, local, n);

            replaceCol(local, b, i, n);

            double detAi = deter(local, n);
            x[i] = detAi / detA;
```

```
            freeMat(local, n);
            exit(0);
        }
    }

    for (int i = 0; i < n; i++)
        wait(NULL);
}

////////////////////////////////////////////////////
// Main
////////////////////////////////////////////////////

int main()
{
    int n;

    printf("Enter the size of the matrix (n): ");
    scanf("%d", &n);

    srand(time(NULL));

    double **A = allocateMat(n);
    double *b = (double *)malloc(n * sizeof(double));
    double *x = (double *)malloc(n * sizeof(double));

    for (int i = 0; i < n; i++)
    {
        b[i] = rand() % 10;
        for (int j = 0; j < n; j++)
            A[i][j] = rand() % 10;
    }

    // Sequential
    printf("\nRunning Sequential Cramer's Rule...\n");
    clock_t strt = clock();
    c_sequential(A, b, x, n);
    clock_t end = clock();

    double seqTime = (double)(end - strt) / CLOCKS_PER_SEC;
    printf("Sequential Execution Time: %f seconds\n", seqTime);

    // Parallel
    printf("\nRunning Parallel Cramer's Rule using fork()...\n");
    strt = clock();
    c_parallel(A, b, x, n);
    end = clock();
```

```
    double parTime = (double)(end - strt) / CLOCKS_PER_SEC;
    printf("Parallel Execution Time: %f seconds\n", parTime);

    if (parTime > 0)
        printf("\nSpeedup Achieved: %f\n", seqTime / parTime);

    freeMat(A, n);
    free(b);
    free(x);

    printf("\nProgram Finished Successfully.\n");
    return 0;
}
```

Output :





10. Observations

- We found that sequential execution takes significantly longer as the matrix size increases.
- The parallel version reduces the wall-clock time for determinant computation.
- There is a noticeable overhead in creating processes; for very small matrices, the sequential version is occasionally faster.
- As n grows, the speedup becomes much more apparent, and CPU utilization across our Ryzen 7 cores reaches its peak.

11. Advantages

- Significant speed improvements for larger systems of equations.
- Effectively demonstrates how to leverage multi-core processors.
- Provides a practical use case for process-level parallelism.

12. Limitations

- Higher overall memory consumption due to process duplication.
- The overhead of fork() can be a bottleneck for very small tasks.
- Cramer’s Rule is computationally expensive compared to other methods like LU decomposition.
- Lack of shared memory means the parent process cannot easily collect final results from children.

13. Conclusion

Through this team project, we successfully implemented determinant computation and system solving using Cramer’s Rule. By comparing sequential and parallel implementations, we observed that fork() significantly reduces execution time for high-order matrices. This experiment provided us with valuable insights into the benefits and the architectural limitations of process-level parallelism in a Linux environment.