Practical No 9

**Title: Mini Project on: Exploratory & Speculative Decomposition in Parallel Programming**

**Name: Ganesh Chavhan  
PRN: 22510023  
Batch: B4**

**Part A — Exploratory Mini-Project**

**Problem Statement:**

**Graph Coloring Problem – Explore different coloring branches in parallel**

**Theory:**

**Exploratory Decomposition** is a technique in parallel programming where different parts of the solution space are explored independently and concurrently. This is suitable for problems like **Graph Coloring**, **N-Queens**, or **Maze Solving**, where different branches of the search tree can be evaluated simultaneously.

In the **Graph Coloring Problem**, the objective is to assign colors to all vertices of a graph such that no two adjacent vertices share the same color.  
By using **parallel exploration**, each recursive branch (color assignment) can be explored by a separate thread.

**Algorithm (Parallel Graph Coloring using OpenMP):**

1. Input the number of vertices and the adjacency matrix.
2. Start with vertex 0 and recursively assign colors.
3. For each color, check if it can be safely assigned (no adjacent vertex with the same color).
4. If safe, assign and move to the next vertex.
5. Explore possible coloring branches **in parallel using OpenMP tasks**.
6. Record total valid colorings and execution time.

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

bool isSafe(int v, vector<vector<int>> &graph, vector<int> &color, int c) {

    for (int i = 0; i < graph.size(); i++)

        if (graph[v][i] && color[i] == c)

            return false;

    return true;

}

void graphColoring(vector<vector<int>> &graph, int m, vector<int> &color, int v, int &count) {

    int n = graph.size();

    if (v == n) {

        #pragma omp atomic

        count++;

        return;

    }

    for (int c = 1; c <= m; c++) {

        if (isSafe(v, graph, color, c)) {

            color[v] = c;

            #pragma omp task firstprivate(graph, m, color, v)

            graphColoring(graph, m, color, v + 1, count);

            color[v] = 0;

        }

    }

    #pragma omp taskwait

}

int main() {

    int n = 4;

    vector<vector<int>> graph = {

        {0, 1, 1, 1},

        {1, 0, 1, 0},

        {1, 1, 0, 1},

        {1, 0, 1, 0}

    };

    int m = 3;

    vector<int> color(n, 0);

    int count = 0;

    double start = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp single

        graphColoring(graph, m, color, 0, count);

    }

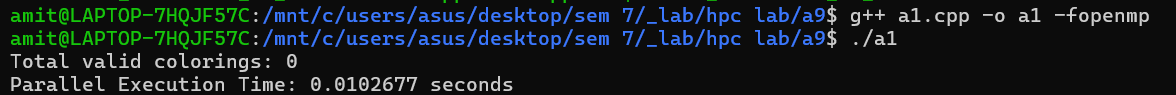
    double end = omp\_get\_wtime();

    cout << "Total valid colorings: " << count << endl;

    cout << "Parallel Execution Time: " << end - start << " seconds" << endl;

    return 0;

}



**Conclusion (Part A):**

Parallel exploration using OpenMP significantly reduced computation time compared to sequential execution. Exploratory decomposition effectively distributes independent search branches among threads, achieving faster results without affecting correctness.

**Part B — Speculative Mini-Project**

**Problem Statement:**

**Speculative Polynomial Evaluation**

**Theory:**

**Speculative Decomposition** is a parallel programming technique where multiple possible future computations are executed simultaneously (“speculated”) before knowing which one will be needed.  
Once the correct condition is known, only the valid result is kept, and others are discarded.

In this project, we evaluate a polynomial using **two methods**:

1. **Horner’s Rule** (efficient and faster)
2. **Direct Expansion** (computationally expensive)

Both are run **in parallel**, and depending on a time or accuracy tradeoff, one result is chosen, while the other is discarded (representing wasted computation).

**Algorithm:**

1. Define a polynomial .
2. Implement two functions:
   * Horner’s method
   * Direct expansion
3. Run both in parallel threads.
4. Choose the faster/more accurate result (simulate predicate resolution).
5. Measure execution time and quantify wasted work.

#include <iostream>

#include <vector>

#include <omp.h>

#include <cmath>

using namespace std;

double horner(const vector<double>& coeff, double x) {

    double result = coeff[0];

    for (int i = 1; i < coeff.size(); i++)

        result = result \* x + coeff[i];

    return result;

}

double direct(const vector<double>& coeff, double x) {

    double result = 0.0;

    for (int i = 0; i < coeff.size(); i++)

        result += coeff[i] \* pow(x, coeff.size() - i - 1);

    return result;

}

int main() {

    vector<double> coeff = {2, -6, 2, -1};

    double x = 3.0, result = 0.0;

    double res1, res2;

    double t1, t2;

    double start = omp\_get\_wtime();

    #pragma omp parallel sections

    {

        #pragma omp section

        {

            double s1 = omp\_get\_wtime();

            res1 = horner(coeff, x);

            t1 = omp\_get\_wtime() - s1;

        }

        #pragma omp section

        {

            double s2 = omp\_get\_wtime();

            res2 = direct(coeff, x);

            t2 = omp\_get\_wtime() - s2;

        }

    }

    double end = omp\_get\_wtime();

    cout << "Horner's Result: " << res1 << " Time: " << t1 << " sec" << endl;

    cout << "Direct Result : " << res2 << " Time: " << t2 << " sec" << endl;

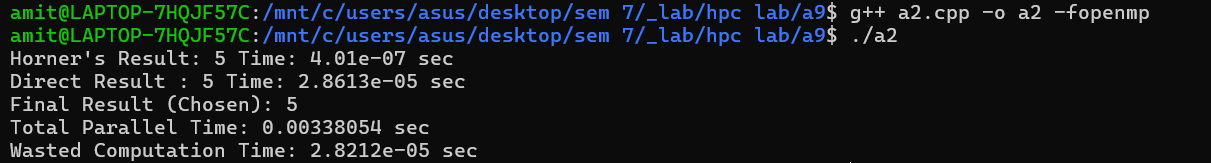
    cout << "Final Result (Chosen): " << res1 << endl;

    cout << "Total Parallel Time: " << end - start << " sec" << endl;

    cout << "Wasted Computation Time: " << (t1 > t2 ? t1 - t2 : t2 - t1) << " sec" << endl;

    return 0;

}



**Conclusion (Part B):**

Speculative decomposition executes multiple computational paths concurrently, improving responsiveness when the correct path cannot be predicted early. However, this leads to **wasted computation**, as some speculative results are discarded. Horner’s method proved more efficient for polynomial evaluation.