Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 9

Title: Exploratory & Speculative Decomposition in Parallel Computing

[GitHub Repository](https://github.com/TerminatorShri/22510025_HPCL)

**Introduction:**

**1. Exploratory Decomposition:**  
Exploratory decomposition divides a **search or solution space** into independent subspaces that can be explored concurrently. Each parallel task works on a **disjoint part of the problem**, allowing multiple possibilities to be checked simultaneously.

* **Use Case:** N-Queens, Sudoku, Maze traversal, or other backtracking/search problems.
* **Benefit:** Reduces total computation time by distributing work across multiple threads/cores.
* **Example:** In the N-Queens problem, each thread can explore different initial placements of queens on the first row.

**2. Speculative Decomposition:**  
Speculative decomposition executes **multiple possible future computations in parallel** before knowing which one will actually be needed. Once the controlling condition or predicate is resolved, the correct result is selected, and the remaining computations are discarded.

* **Use Case:** If–Else branch evaluation, Quicksort with multiple pivot choices, alternative search strategies.
* **Benefit:** Reduces latency when the cost of computing branches in parallel is lower than the delay of waiting for sequential evaluation.
* **Trade-off:** Wasted computation occurs for discarded branches, so careful analysis of efficiency is required.
* **Example:** Evaluating both branches of if (x > 0) in parallel and keeping only the correct result.

**Key Difference:**

* **Exploratory decomposition** explores **different parts of the solution space** simultaneously, with all work contributing to the final result.
* **Speculative decomposition** explores **potential future paths**, but some work may be **discarded** depending on runtime decisions.

**Problem Descriptions:  
  
Part A — Exploratory Decomposition Mini-Project**

**Problem:** **N-Queens Problem**

* **Description:** Place N queens on an N×N chessboard such that no two queens threaten each other (no two queens share the same row, column, or diagonal).
* **Sequential Approach:** Use backtracking to explore one branch at a time, checking valid placements row by row.
* **Parallel Approach (Exploratory Decomposition):**
  + Assign each thread a subset of initial placements in the first row.
  + Each thread explores its subspace independently using backtracking.
  + Combine results from all threads to get the total number of solutions.
* **Objective:** Compare sequential vs parallel execution times, measure speedup, and demonstrate load distribution across threads.

**Part B — Speculative Decomposition Mini-Project**

**Problem:** **Speculative Branch Evaluation in Numerical Computation**

* **Description:** Evaluate a mathematical function based on a condition:

1. if (x > 0)

2. result = sqrt(x);

3. else

4. result = log(|x|);

* **Sequential Approach:** Evaluate only the branch dictated by the condition (x > 0 or x ≤ 0).
* **Speculative Approach:**
  + Execute both branches in parallel using separate threads.
  + Once the condition is resolved, select the correct result and discard the other.
* **Objective:** Compare sequential vs speculative execution times, measure speedup, and quantify **wasted computation** (discarded work).
* **Observation Focus:** Trade-offs between reduced latency and wasted work, highlighting conditions where speculative execution may or may not be efficient.

**Algorithm Design:**

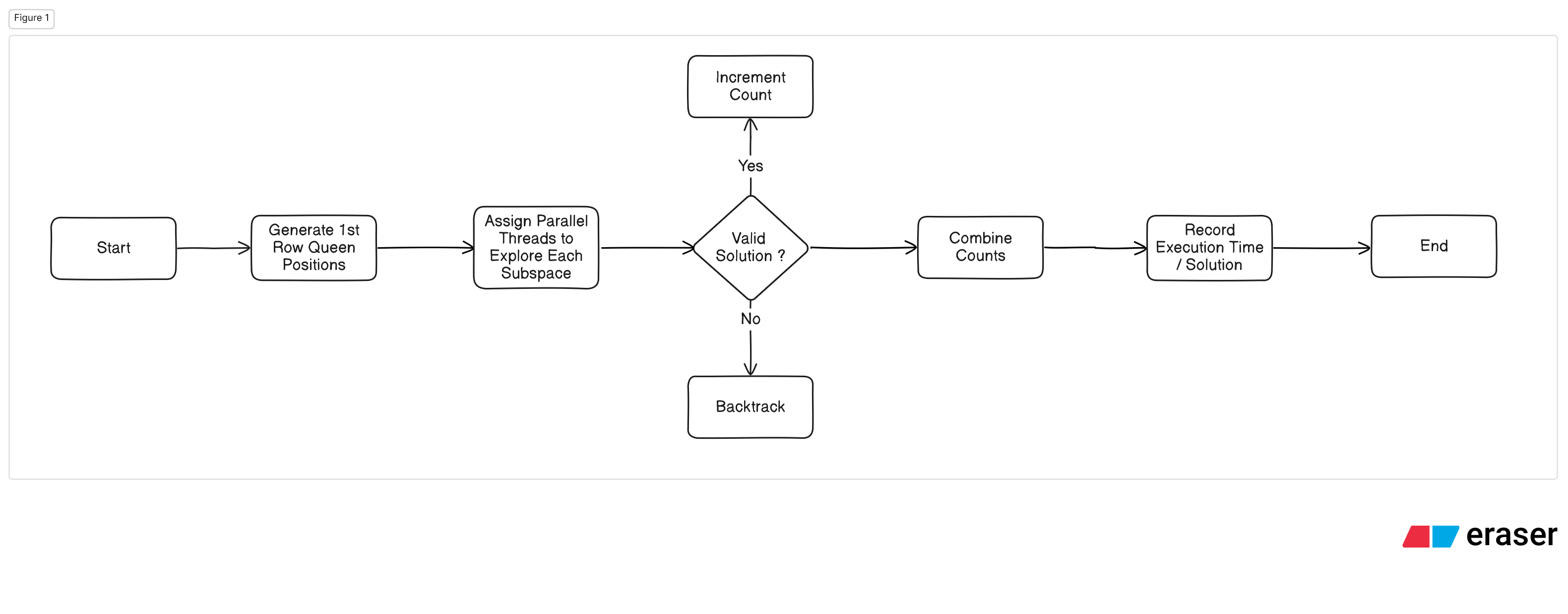
**Part A — Exploratory Decomposition (N-Queens)**

**Algorithm Design (Parallel Backtracking)**

**Input:** Board size N  
**Output:** Total number of valid N-Queens solutions

**Steps:**

1. **Initialize Board:** Create an N×N board.
2. **Generate Initial Row Placements:** For row 0, enumerate all possible column positions for the first queen.
3. **Parallel Exploration:**
   * Assign each initial placement to a separate thread.
   * Each thread executes backtracking independently for remaining rows.
4. **Backtracking Procedure:**
   * For each row, try placing a queen in a valid column.
   * If placement conflicts (same column or diagonal), backtrack.
   * If row N is reached, increment solution count.
5. **Combine Results:** Aggregate solution counts from all threads.
6. **Record Execution Time:** Measure sequential vs parallel time.



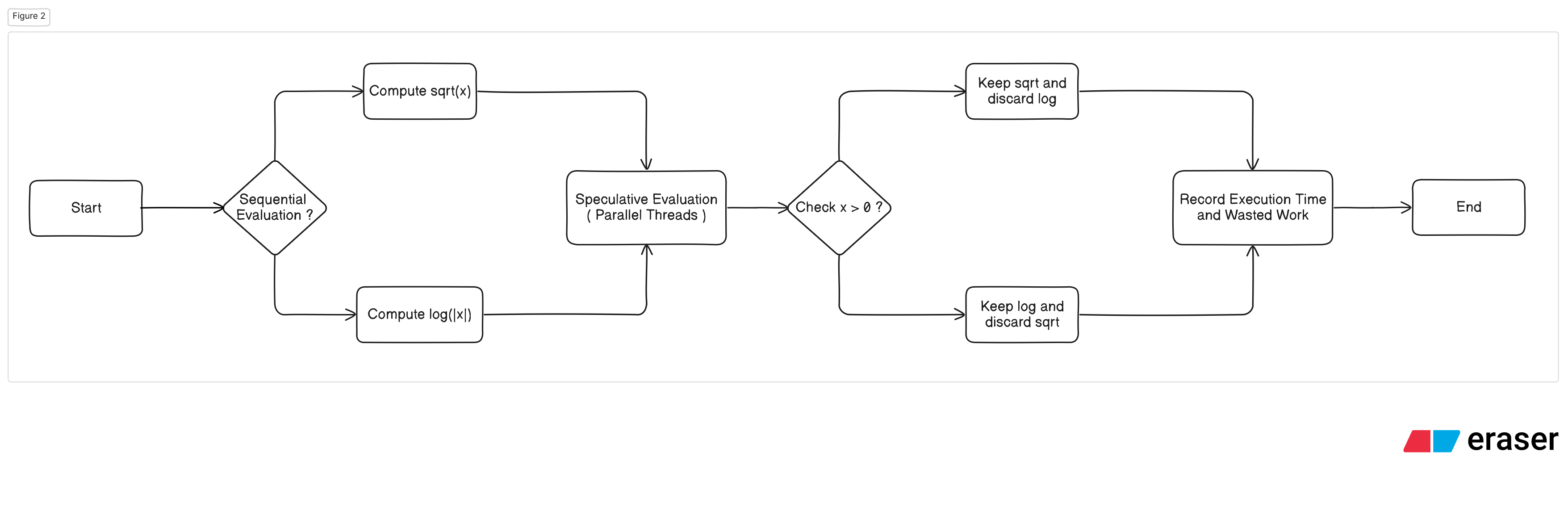
**Part B — Speculative Decomposition (Branch Evaluation)**

**Algorithm Design**

**Input:** Value x  
**Output:** Correct result (sqrt(x) or log(|x|))

**Steps:**

1. **Sequential Evaluation:**
   * Check if x > 0
     + True → compute sqrt(x)
     + False → compute log(|x|)
2. **Speculative Evaluation (Parallel):**
   * Launch two threads:
     + Thread 1 → compute sqrt(|x|)
     + Thread 2 → compute log(|x|)
   * Wait for both threads to finish
3. **Select Result:**
   * Check original condition (x > 0)
   * Choose correct thread result, discard the other (wasted work)
4. **Record Metrics:**
   * Execution time, speedup, wasted work percentage



**Implementation Details:**

**Part A – Exploratory Decomposition Mini – Project**

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. #define MAX\_THREADS 16

6. #define MAX\_SIZES 3

7.

8. int is\_safe(int \*board, int row, int col) {

9.     for (int i = 0; i < row; i++) {

10.         if (board[i] == col || abs(board[i] - col) == abs(i - row)) {

11.             return 0;

12.         }

13.     }

14.     return 1;

15. }

16.

17. void solve\_nqueens(int \*board, int row, int N, long \*solutions) {

18.     if (row == N) {

19.         #pragma omp atomic

20.         (\*solutions)++;

21.         return;

22.     }

23.

24.     for (int col = 0; col < N; col++) {

25.         if (is\_safe(board, row, col)) {

26.             board[row] = col;

27.             solve\_nqueens(board, row + 1, N, solutions);

28.         }

29.     }

30. }

31.

32. long parallel\_nqueens(int N, int threads) {

33.     long solutions = 0;

34.

35.     #pragma omp parallel num\_threads(threads)

36.     {

37.         int \*board = (int \*)malloc(N \* sizeof(int));

38.         int tid = omp\_get\_thread\_num();

39.

40.         #pragma omp for schedule(dynamic)

41.         for (int col = 0; col < N; col++) {

42.             board[0] = col;

43.             solve\_nqueens(board, 1, N, &solutions);

44.         }

45.         free(board);

46.     }

47.     return solutions;

48. }

49.

50. int main() {

51.     int sizes[MAX\_SIZES] = {8, 10, 12};

52.     FILE \*fp = fopen("PS\_1\_A.csv", "a");

53.

54.     if (fp == NULL) {

55.         printf("Error opening PS\_1\_A.csv!\n");

56.         return 1;

57.     }

58.

59.     fseek(fp, 0, SEEK\_END);

60.     if (ftell(fp) == 0) {

61.         fprintf(fp, "BoardSize,Threads,ExecutionTime,Solutions\n");

62.     }

63.

64.     printf("%-10s %-7s %-12s %-12s\n", "BoardSize", "Threads", "Time(s)", "Solutions");

65.     printf("------------------------------------------------------\n");

66.

67.     for (int s = 0; s < MAX\_SIZES; s++) {

68.         int N = sizes[s];

69.

70.         for (int t = 1; t <= MAX\_THREADS; t \*= 2) {

71.             double start\_time = omp\_get\_wtime();

72.             long solutions = parallel\_nqueens(N, t);

73.             double end\_time = omp\_get\_wtime();

74.

75.             double exec\_time = end\_time - start\_time;

76.

77.             printf("%-10d %-7d %-12.6f %-12ld\n", N, t, exec\_time, solutions);

78.             fprintf(fp, "%d,%d,%f,%ld\n", N, t, exec\_time, solutions);

79.         }

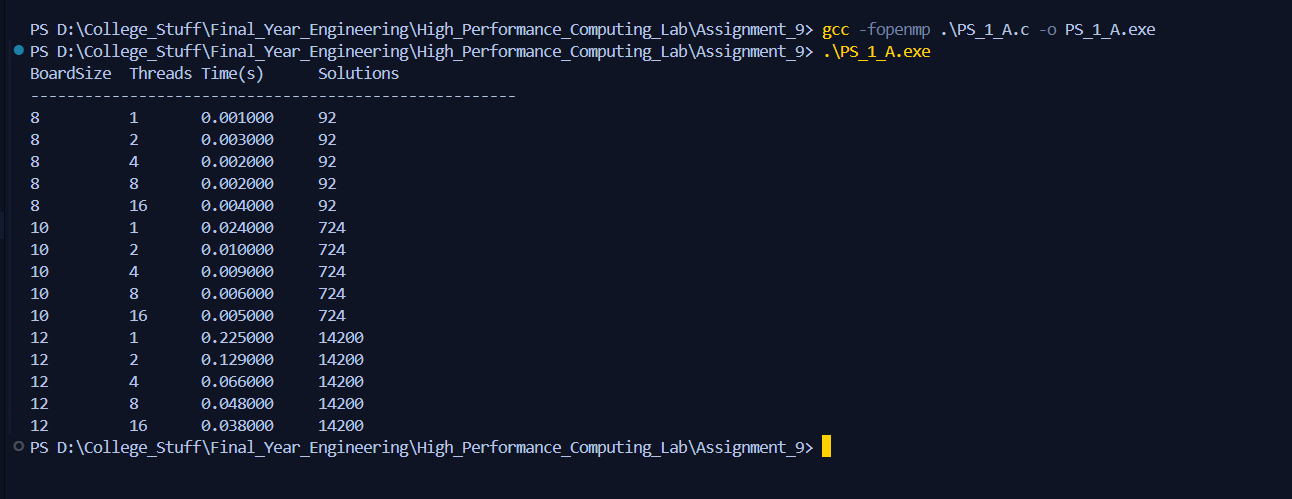
80.     }

81.

82.     fclose(fp);

83.     return 0;

84. }



**Part B – Speculative Decomposition Mini – Project**

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <math.h>

4. #include <omp.h>

5.

6. #define MAX\_THREADS 2

7. #define NUM\_INPUTS 10

8. #define REPETITIONS 100000

9.

10. double sequential\_eval(double x) {

11.     double res = 0.0;

12.     for(int rep = 0; rep < REPETITIONS; rep++) {

13.         if (x > 0)

14.             res = sqrt(x);

15.         else

16.             res = log(fabs(x));

17.     }

18.     return res;

19. }

20.

21. double speculative\_eval(double x) {

22.     double result1 = 0.0, result2 = 0.0, final\_result = 0.0;

23.

24.     for(int rep = 0; rep < REPETITIONS; rep++) {

25.         #pragma omp parallel sections num\_threads(2)

26.         {

27.             #pragma omp section

28.             { result1 = sqrt(fabs(x)); }

29.             #pragma omp section

30.             { result2 = log(fabs(x)); }

31.         }

32.

33.         if (x > 0)

34.             final\_result = result1;

35.         else

36.             final\_result = result2;

37.     }

38.

39.     return final\_result;

40. }

41.

42. int main() {

43.     FILE \*fp = fopen("PS\_1\_B.csv", "w");

44.     if (!fp) {

45.         printf("Error opening file!\n");

46.         return 1;

47.     }

48.

49.     fprintf(fp, "InputX,Mode,Threads,ExecutionTime,Result,WastedWorkPercent\n");

50.

51.     double start\_val = -50.0;

52.     double end\_val = 50.0;

53.     double step = (end\_val - start\_val) / NUM\_INPUTS;

54.

55.     printf("%-10s %-12s %-7s %-12s %-12s %-10s\n",

56.            "InputX", "Mode", "Threads", "Time(s)", "Result", "Wasted(%)");

57.     printf("---------------------------------------------------------------\n");

58.

59.     for (int i = 0; i < NUM\_INPUTS; i++) {

60.         double x = start\_val + i\*step;

61.

62.         double start = omp\_get\_wtime();

63.         double res\_seq = sequential\_eval(x);

64.         double end = omp\_get\_wtime();

65.         double time\_seq = end - start;

66.

67.         printf("%-10.2f %-12s %-7d %-12.6f %-12.6f %-10d\n",

68.                x, "Sequential", 1, time\_seq, res\_seq, 0);

69.         fprintf(fp, "%f,Sequential,1,%f,%f,0\n", x, time\_seq, res\_seq);

70.

71.         start = omp\_get\_wtime();

72.         double res\_spec = speculative\_eval(x);

73.         end = omp\_get\_wtime();

74.         double time\_spec = end - start;

75.

76.         double wasted = 50.0;

77.         printf("%-10.2f %-12s %-7d %-12.6f %-12.6f %-10.2f\n",

78.                x, "Speculative", 2, time\_spec, res\_spec, wasted);

79.         fprintf(fp, "%f,Speculative,2,%f,%f,%f\n", x, time\_spec, res\_spec, wasted);

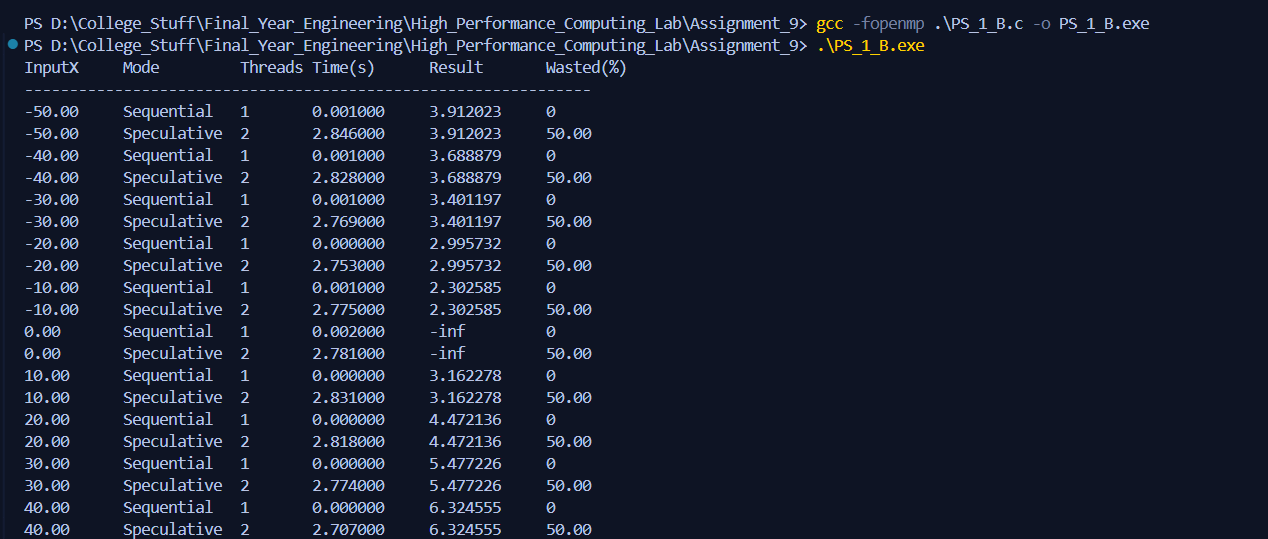
80.     }

81.

82.     fclose(fp);

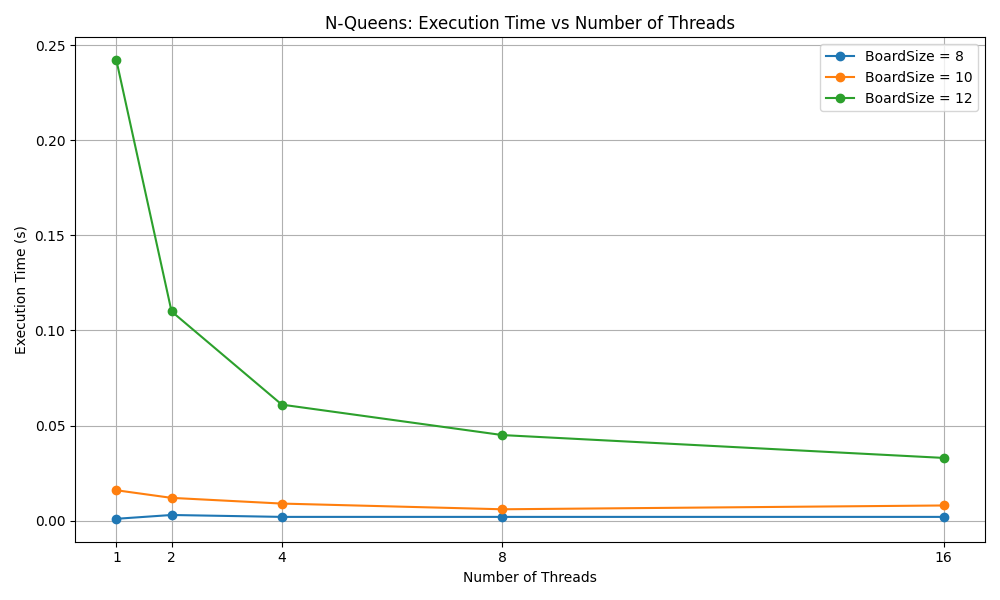
83.     return 0;

84. }

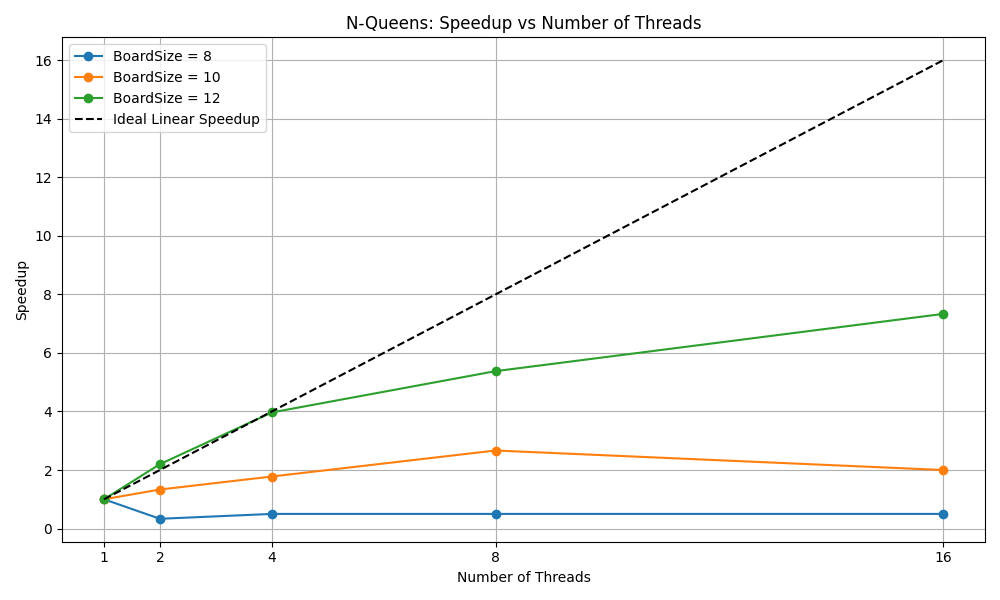


**Results (Observations & Conclusions):**

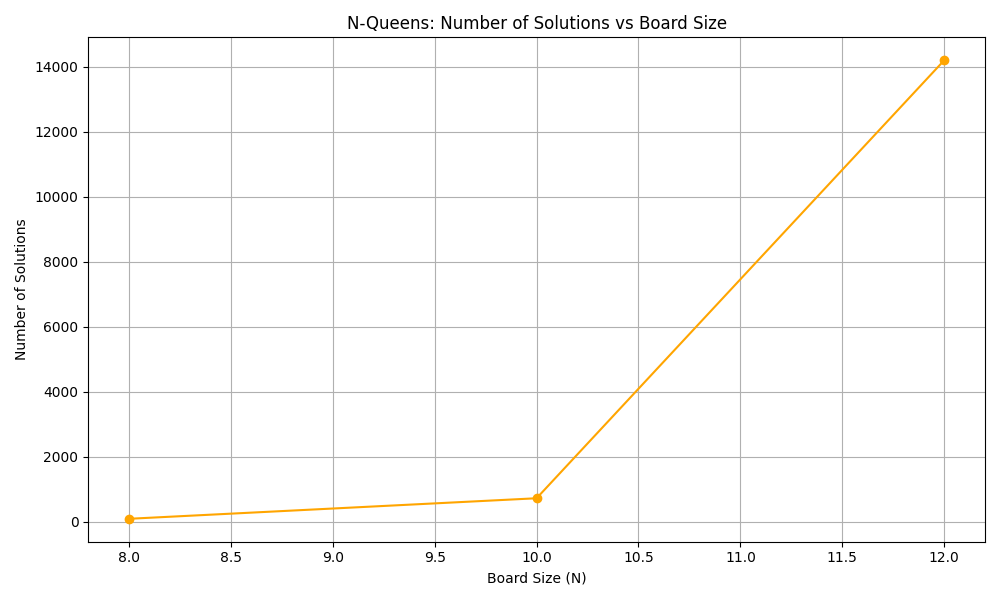
**Part A – Exploratory Decomposition Mini – Project**

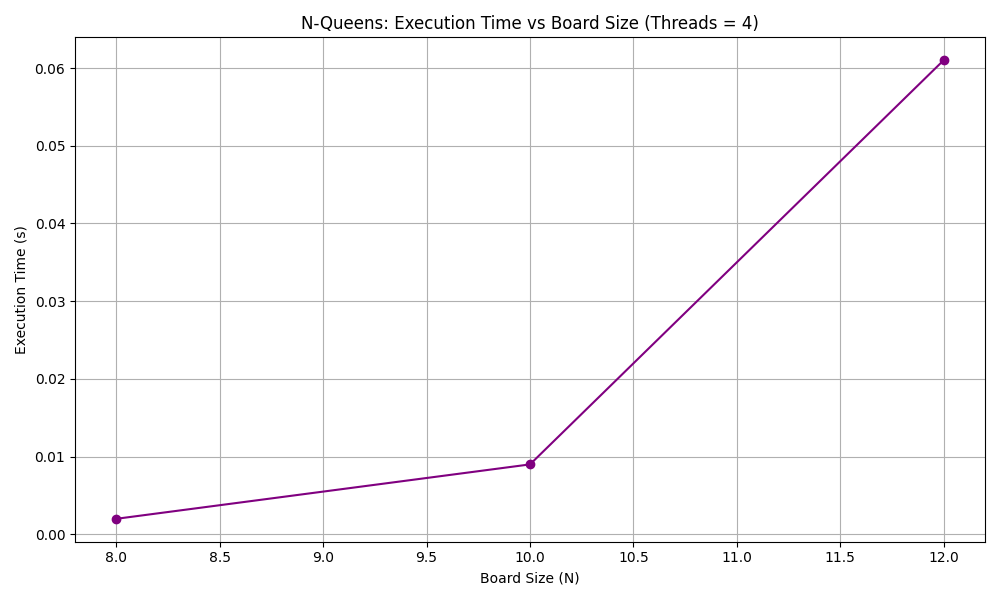
****

We can see for larger board sizes there is significant decrease in execution time which shows benefits from parallelism but for smaller sizes there is no benefit or sometimes even require more time which shows that overhead for synchronization dominates.

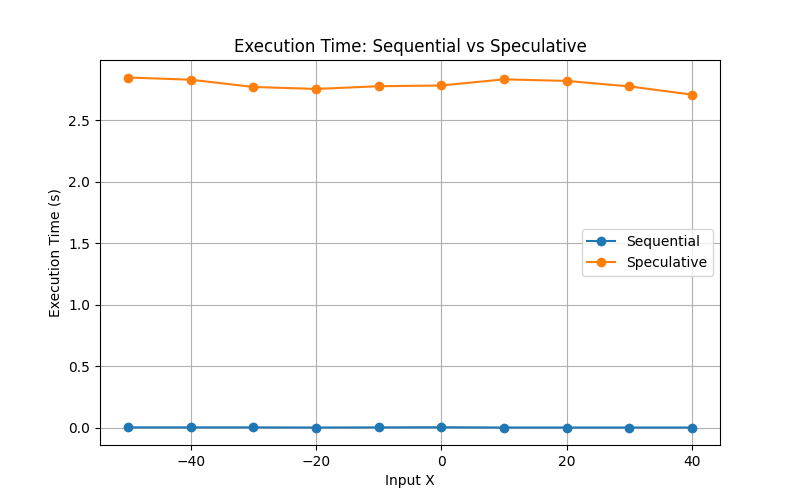


We can see for larger board sizes significant speedup but is not close to ideal which can be due to overheads incurred but for smaller speedup is negligible or none.



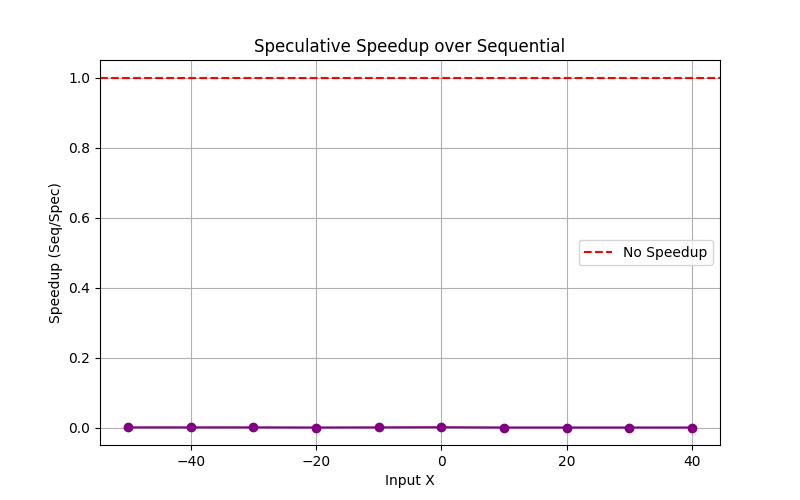


**Part B – Speculative Decomposition Mini – Project**

****

* Sequential code runs extremely fast — practically negligible time.
* Speculative version takes about 2.8 seconds because it:
  + Spawns parallel threads every iteration.
  + Does *both* computations (sqrt + log).
  + Incurs large OpenMP overhead for thread creation and synchronization.

So, **speculative execution here is slower**, not faster — the overhead dominates the tiny computation.



This plot confirms that **speculative execution here provides no benefit** — it introduces overhead instead of improving performance.

