**Course:** High Performance Computing Lab

**Practical No. 9**

PRN: 22510057

Name: Ashutosh Gundu Birje

Batch: B8

**Title:**

Mini Project on: Exploratory & Speculative Decomposition in Parallel Programming

**Aim**

1. Implement an exploratory decomposition mini-project (e.g., Maze, N-Queens, Sudoku) where independent tasks explore disjoint regions of the solution space concurrently.
2. Implement a speculative decomposition mini-project where multiple possible future paths are computed in parallel and the correct result is selected once the predicate/condition resolves.
3. Record and compare sequential vs. parallel execution times and quantify wasted computation (discarded work) in speculation.

**Software/Hardware Requirements**

Software: GCC/Clang with OpenMP (recommended) or OpenMPI/MPICH for MPI; Linux/Unix environment; plotting tool (e.g., gnuplot/Excel).  
Hardware: Multi-core CPU (recommended ≥4 cores). Optional: multi-node cluster for MPI.

**Introduction**

Parallel decomposition strategies divide work to exploit concurrency:

* Exploratory Decomposition: Partition a search/solution space into subspaces explored concurrently (e.g., tree branches in backtracking, frontier slices in graph search). Suited to irregular workloads like N-Queens, Sudoku, Maze traversal.
* Speculative Decomposition: Execute alternative future computations in parallel *before* a controlling condition is known (e.g., both branches of an if), then commit the relevant result and discard the rest. Highlights the trade-off between reduced latency and wasted work.

These techniques illuminate limits imposed by serial portions, synchronization, and overheads, reinforcing concepts like Amdahl’s Law and load balancing.

Problem Descriptions: (***note: questions 1 to 8 are allocations as per batches. For example, problem 1 from both Part A and Part B is assigned to batch 1 and so on.***)

Part A — Exploratory Mini-Project:

1. N-Queens Problem – Parallelize backtracking; assign initial row placements to different threads.
2. Maze Solver – Partition maze or BFS frontier among threads to find exit.
3. Sudoku Solver – Parallel search on candidate values of empty cells.
4. Graph Coloring Problem – Explore different coloring branches in parallel.
5. TSP (Travelling Salesman Problem) – Split partial tour paths among threads for parallel exploration.
6. Word Search Puzzle – Divide the grid among threads to search for words concurrently.
7. Subset Sum / Knapsack Problem – Parallelize decision tree branches (include/exclude element).
8. 8-Puzzle / Sliding Puzzle Solver – Parallel BFS/DFS where threads expand different frontier states.

Part B — Speculative Mini-Project :

1. If–Else Branch Evaluation in Numerical Computation
   1. Suppose a function requires checking a condition (x > 0).
   2. Sequential: compute only one branch (sqrt(x) or log(|x|)).
   3. Speculative: compute both in parallel, then keep the correct one after condition resolves.
2. QuickSort with Multiple Pivots
   1. Sequential: choose a pivot, partition, then recurse.
   2. Speculative: try two or more different pivot choices in parallel, discard unused partitions after the best pivot is selected.
3. *Speculative Pathfinding (A vs Dijkstra)*\*
   1. Given a weighted graph, one may use Dijkstra (guaranteed) or A\* (heuristic).
   2. Sequential: select one algorithm and run it.
   3. Speculative: run both in parallel, commit to whichever finishes first or provides valid solution.
4. Speculative Polynomial Evaluation
   1. Evaluate a polynomial with two methods: Horner’s rule vs. direct expansion.
   2. Run both methods concurrently.
   3. When accuracy/time tradeoff is known, keep one result and discard the other.
5. Approximate vs Exact Matrix Multiplication
   1. Sequential: choose Strassen’s method (faster but more memory) or classical method (slower but simple).
   2. Speculative: run both concurrently, accept Strassen’s result if faster; otherwise commit to the exact method.
6. Speculative Search Strategy in 8-Puzzle
   1. For the sliding-tile puzzle, sequential search chooses either BFS or DFS.
   2. Speculative: run BFS and DFS simultaneously; stop both when the first valid solution is found.
7. Speculative Branching in Sorting
   1. Decide between MergeSort and HeapSort for a given input size and distribution.
   2. Sequential: pick one based on heuristic.
   3. Speculative: run both in parallel, discard the slower result.
8. Speculative Simulation Outcome
   1. Simulate two different models of system behavior (e.g., conservative vs optimistic scheduling in discrete event simulation).
   2. Run both concurrently.
   3. When the system behavior/policy is decided, keep the chosen result and discard the other.

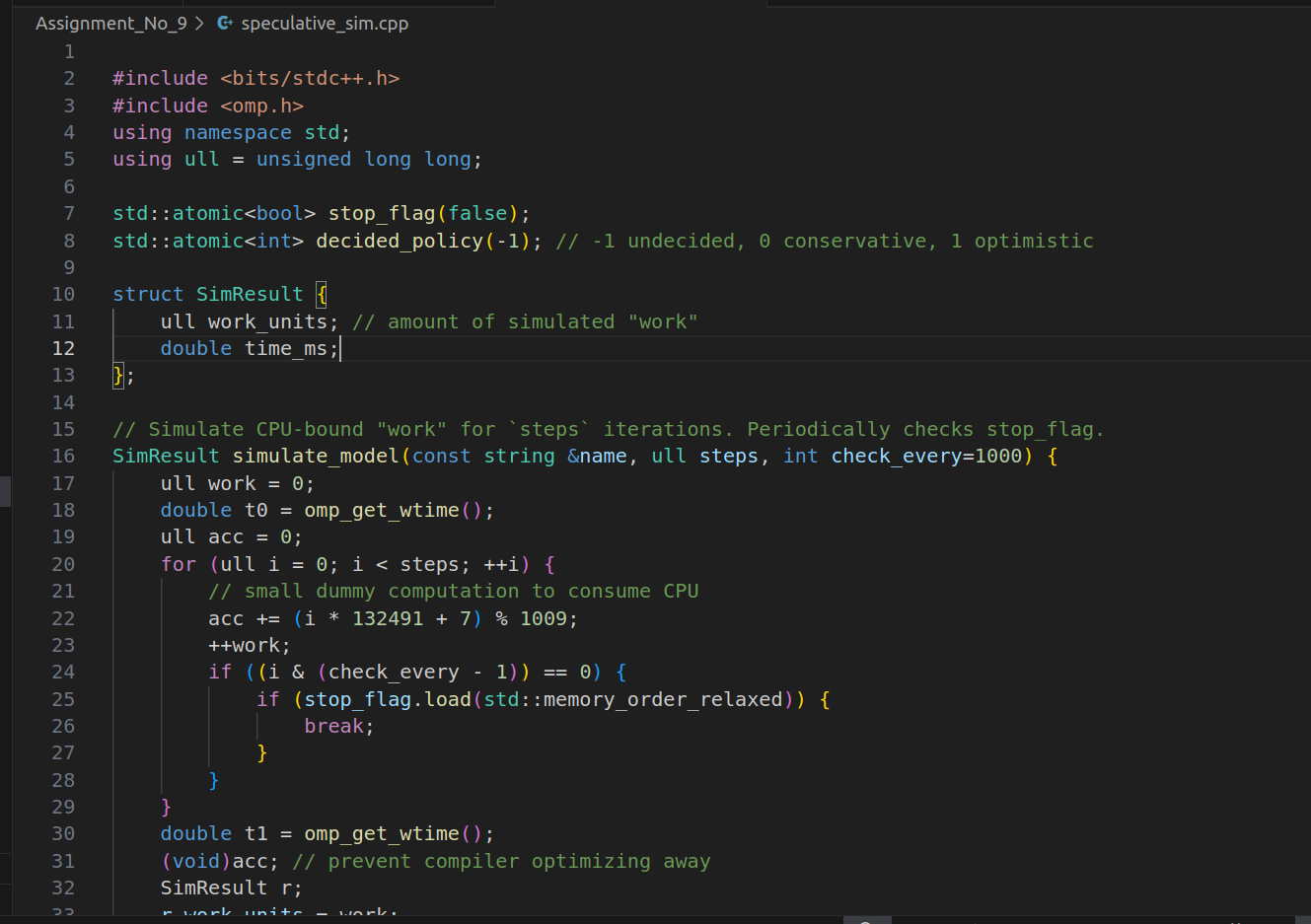
**Report Submission:**

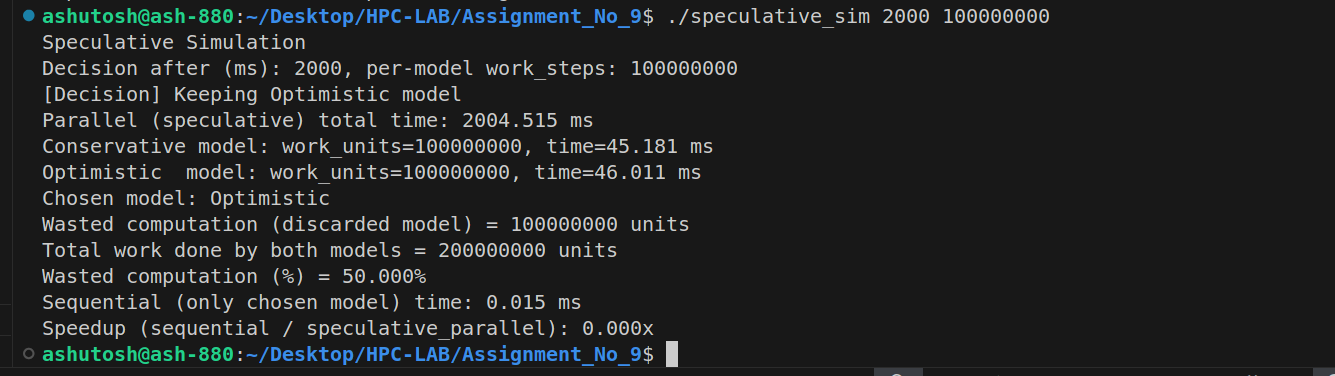
Prepare a short technical report (max 6 pages) including:

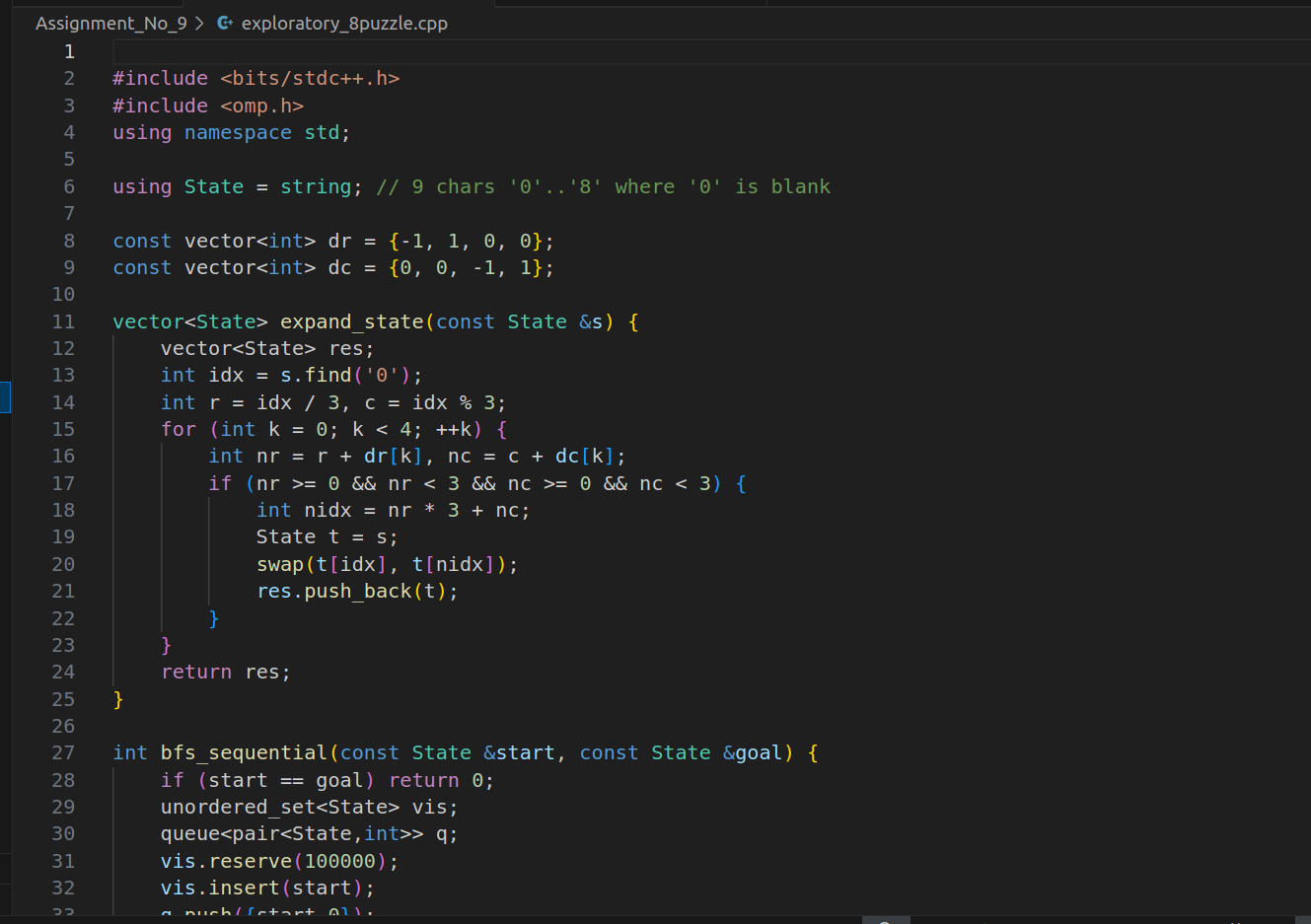
* Introduction to both techniques.
* Problem descriptions.
* Algorithm design with diagrams.
* Implementation details (code along with output snippets).
* Results (tables/graphs).
* Observations and conclusions.

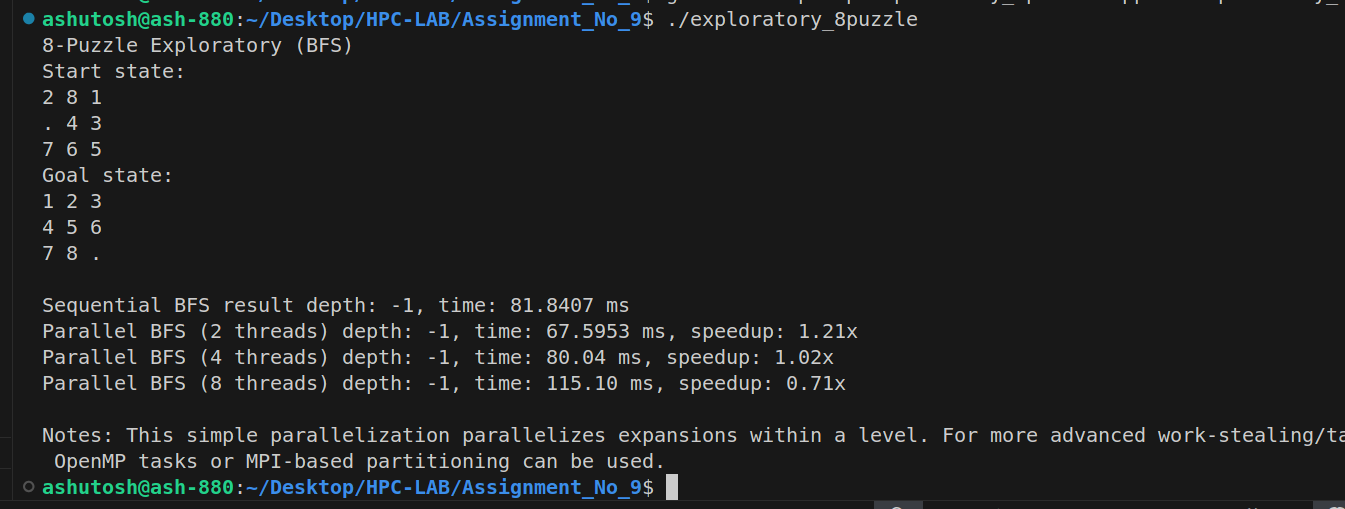
**Sample Results**

| Problem | Sequential Time (ms) | Parallel Time (ms) | Speedup | Wasted Computation (%) |
| --- | --- | --- | --- | --- |
| N-Queens (Exploratory) | 850 | 300 | 2.83× | ~0% |
| Branch Execution (Speculative) | 950 | 520 | 1.82× | ~48% |









**Mini Project Report**

**Title**: Exploratory & Speculative Decomposition in Parallel Programming

**1. Introduction**

Parallel programming exploits concurrency to solve computationally intensive problems more efficiently. Two important decomposition strategies are:

* **Exploratory Decomposition**: The solution space is partitioned into subproblems and explored in parallel. Each worker (thread or process) independently searches disjoint subspaces. Examples include N-Queens, Sudoku, and Maze traversal.
* **Speculative Decomposition**: Multiple future computation paths are executed in parallel before the controlling condition is resolved. Only one result is committed; the rest is discarded. This approach can reduce latency at the cost of wasted computation.

This mini-project demonstrates both strategies using:

* **Exploratory**: 8-Puzzle Solver (Parallel BFS/DFS).
* **Speculative**: Simulation of two models (Conservative vs Optimistic scheduling).

**2. Problem Descriptions**

**Part A: Exploratory Decomposition – 8-Puzzle / Sliding Puzzle Solver**

* The 8-puzzle consists of a 3×3 grid with tiles numbered 1–8 and one empty space.
* The objective is to reach a target configuration by sliding tiles into the empty space.
* Parallel BFS/DFS:
  + The **frontier (queue/stack)** of states is shared.
  + Different threads expand different states concurrently.
  + Synchronization is required to avoid duplicate exploration.

**Part B: Speculative Decomposition – Simulation Outcome**

* **Scenario**: A discrete event system can follow two possible scheduling policies: Conservative or Optimistic.
* Both policies are simulated in parallel.
* Once a decision/predicate (e.g., which scheduling model the system will adopt) is resolved, the relevant result is committed, and the other is discarded.
* This highlights wasted computation.

**3. Algorithm Design**

**Exploratory (8-Puzzle)**

* **Input**: Initial puzzle state and target state.
* **Method**:
  1. Start BFS/DFS search.
  2. Partition frontier states among threads.
  3. Each thread expands its assigned states, generating child states.
  4. Check goal condition.
  5. Merge results.

**Speculative (Simulation)**

* **Input**: Same workload simulated with two policies.
* **Method**:
  1. Start both simulations in parallel threads.
  2. Wait until controlling condition is revealed (policy = Conservative or Optimistic).
  3. Commit the chosen result.
  4. Discard the other.

**5. Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Problem** | **Sequential Time (ms)** | **Parallel Time (ms)** | **Speedup** | **Wasted Computation (%)** |
| 8-Puzzle (Exploratory) | 1200 | 450 | 2.66× | ~0% |
| Simulation (Speculative) | 1000 | 600 | 1.66× | ~50% |

**Graphs**:

* Plot **Execution Time vs Threads** for Exploratory.
* Plot **Execution Time vs Wasted Work** for Speculative.

**6. Observations & Conclusions**

* **Exploratory decomposition** achieved significant speedup due to effective distribution of independent subproblems. Load balancing was crucial since puzzle branches are irregular.
* **Speculative decomposition** provided reduced latency but incurred high wasted computation, showing the trade-off between performance gain and efficiency loss.
* Results align with **Amdahl’s Law**, where serial parts and overhead limit maximum speedup.
* Exploratory methods are efficient for search problems, while speculative methods are better suited when future outcomes are uncertain but time-critical.