

***PARALLEL DISTRIBUTED COMPUTING PROJECT***

***PARALLEL BFS and PAGE RANK Algorithm Implementation***

***Group Members:***

***Fahad Yousuf(21K-4839) BCS-5E***

***Abdul Rafay (21K-4591) BCS-5E***

***Introduction:***

The project focuses on implementing and optimizing two fundamental graph algorithms, Breadth-First Search (BFS) and Page Rank, leveraging parallel computing techniques—Message Passing Interface (MPI) for distributed-memory parallelism and OpenMP for shared-memory parallelism. These algorithms play a critical role in various applications, including network analysis, social network analysis, and web page ranking.

***Methodology:***

The methodology section outlines the steps and approaches taken in the implementation and evaluation of parallel Breadth-First Search (BFS) and Page Rank algorithms using OpenMP and MPI.

***1. Graph Representation and Data Input:***

* Graphs are represented using adjacency matrices for both BFS and Page Rank algorithms.
* Graph data is read from a file for the MPI implementation, while random graphs are generated for OpenMP Page Rank.

***2. Parallel BFS using OpenMP:***

* The BFS algorithm is parallelized using OpenMP to exploit shared-memory parallelism.
* An array 'visited' is used to mark visited nodes, and a queue is employed for BFS traversal.
* OpenMP directives are utilized to parallelize the loop that explores neighboring nodes.
* Time is measured for the execution of the BFS algorithm, and relevant information, including the final visited sequence and execution time, is printed.

***3. Parallel BFS using MPI:***

* MPI is employed for distributed-memory parallelism in BFS.
* Process 0 reads the graph, specifies the start node, and broadcasts necessary information to all processes.
* Memory is allocated for local graph data, and the graph is scattered among processes.
* BFS is executed in parallel across processes, and memory is freed after completion.

***4. Page Rank using OpenMP:***

* The Page Rank algorithm is parallelized with OpenMP to harness shared-memory parallelism.
* Random graphs are generated, and an iterative approach is taken for Page Rank calculation.
* OpenMP directives parallelize the loop that calculates Page Rank scores.
* Convergence is checked, and the results, including iteration-wise runtime, are printed.

***5. Page Rank using MPI:***

* MPI is utilized for distributed-memory parallelism in the Page Rank algorithm.
* Memory is allocated for a global vector, and a new Page Rank vector is created for each iteration.
* Matrix-vector multiplication is performed, and local Page Rank scores are updated in parallel across processes.
* Information exchange is carried out between processes, and convergence is checked globally.
* Local scores are updated after the exchange, and normalization is performed.
* Iteration-wise information is printed, and Page Rank scores are written to an output file.

***Parallel BFS using OpenMP (Pseudocode):***

***// Pseudocode for main OpenMP BFS functionality***

***function parallel\_BFS(graph, numNodes, startNode):***

***// Initialize data structures***

***visited = initialize\_array(numNodes, false)***

***queue = createQueue()***

***visitedSequence = initialize\_array(numNodes, 0)***

***visitedIndex = 0***

***// Set the start node as visited and enqueue***

***visited[startNode] = true***

***enqueue(queue, startNode)***

***// Start measuring time***

***startTime = getCurrentTime()***

***print("Breadth-First Search starting from node", startNode)***

***// Main BFS loop***

***while queue is not empty:***

***currentVertex = dequeue(queue)***

***visitedSequence[visitedIndex++] = currentVertex***

***// Parallel loop for exploring neighbors***

***#pragma omp parallel for***

***for i from 0 to numNodes - 1:***

***if graph[currentVertex][i] and not visited[i]:***

***// Critical section to update visited and enqueue***

***#pragma omp critical***

***visited[i] = true***

***enqueue(queue, i)***

***// Print the contents of the queue***

***#pragma omp barrier***

***#pragma omp single***

***print("Queue:", queue)***

***#pragma omp barrier***

***#pragma omp single***

***print("Visited:", visited)***

***#pragma omp barrier***

***#pragma omp single***

***print("Current Vertex:", currentVertex)***

***// Print the final sequence of visited vertices***

***print("Final Visited Sequence:", visitedSequence)***

***// Stop measuring time***

***endTime = getCurrentTime()***

***// Print execution time***

***print("Time taken:", endTime - startTime, "seconds";***

***// Free allocated memory***

***free(visited)***

***free(visitedSequence)***

***free(queue->items)***

***free(queue)***

***Parallel BFS using MPI (Pseudocode):***

***def main():***

***MPI\_Init()***

***myRank, numProcesses = MPI\_Comm\_rank(MPI\_COMM\_WORLD), MPI\_Comm\_size(MPI\_COMM\_WORLD)***

***if myRank == 0:***

***# Read graph from file in process 0***

***graph, numNodes = read\_graph\_from\_file()***

***startNode = specify\_start\_node()***

***# Broadcast number of nodes to all processes***

***MPI\_Bcast(numNodes, root=0)***

***# Scatter graph data to all processes***

***scatter\_graph\_data(graph, numNodes)***

***# Broadcast start node to all processes***

***MPI\_Bcast(startNode, root=0)***

***else:***

***# Receive number of nodes from process 0***

***numNodes = MPI\_Bcast(None, root=0)***

***# Allocate memory for local graph data***

***graph = allocate\_memory\_for\_local\_graph(numNodes)***

***# Scatter graph data to all processes***

***scatter\_graph\_data(graph, numNodes)***

***# Receive start node from process 0***

***startNode = MPI\_Bcast(None, root=0)***

***# Execute BFS in all processes***

***BFS(graph, numNodes, startNode, myRank, numProcesses)***

***# Free allocated memory***

***free\_memory(graph, numNodes)***

***MPI\_Finalize()***

***Page Rank using OpenMP (Pseudocode):***

***// Pseudocode for OpenMP Page Rank***

***function generate\_random\_graph(n, adj\_matrix):***

***// Generate a random directed graph (sparse)***

***for i in range(n):***

***for j in range(n):***

***if i != j and random() % 4 == 0:***

***adj\_matrix[i \* n + j] = 1***

***function pagerank(n, pr, adj\_matrix, num\_threads, max\_iterations, output\_filename):***

***new\_pr = [0.0] \* n***

***converged = False***

***// Iterative PageRank calculation***

***for iter in range(max\_iterations):***

***start\_time = get\_current\_time()***

***// Parallel PageRank calculation using OpenMP***

***#pragma omp parallel for***

***for i in range(n):***

***new\_pr[i] = calculate\_page\_rank(adj\_matrix, pr, i, n, DAMPING\_FACTOR)***

***// Check for convergence***

***converged = check\_convergence(new\_pr, pr, EPSILON)***

***end\_time = get\_current\_time()***

***runtime = end\_time - start\_time***

***print(f"Iter: {iter + 1}, Runtime: {runtime}, Converged: {converged}")***

***if output\_filename is not None:***

***write\_page\_rank\_to\_file(output\_filename, iter + 1, pr)***

***if converged:***

***break // Exit the loop when converged***

***// Update PageRank scores***

***for i in range(n):***

***pr[i] = new\_pr[i]***

***function calculate\_page\_rank(adj\_matrix, pr, i, n, damping\_factor):***

***// Perform matrix-vector multiplication***

***new\_pr\_i = (1 - damping\_factor) / n***

***for j in range(n):***

***if adj\_matrix[i \* n + j] == 1:***

***new\_pr\_i += damping\_factor \* pr[j] / out\_degree(j, adj\_matrix, n)***

***return new\_pr\_i***

***function out\_degree(node, adj\_matrix, n):***

***// Calculate the out-degree of a node in the graph***

***return sum(adj\_matrix[node \* n + j] for j in range(n))***

***function check\_convergence(new\_pr, pr, epsilon):***

***// Check if the difference between new and current scores is within epsilon for all nodes***

***for i in range(len(new\_pr)):***

***if abs(new\_pr[i] - pr[i]) > epsilon:***

***return False***

***return True***

***Page Rank using MPI (Pseudocode):***

***# Pseudocode for MPI Page Rank***

***function pagerank(n, pr, adj\_matrix, num\_processes, max\_iterations, output\_filename, comm):***

***rank = get\_rank(comm)***

***global\_vector = allocate\_memory\_for\_global\_vector(n)***

***new\_pr = allocate\_memory\_for\_new\_pr(n)***

***converged = False***

***output\_file = open\_output\_file(output\_filename)***

***for iter in range(max\_iterations):***

***start\_time = get\_current\_time()***

***matrix\_vector\_multiply(n, pr, new\_pr, adj\_matrix)***

***update\_local\_pagerank\_scores(n, pr, new\_pr, adj\_matrix, DAMPING\_FACTOR)***

***exchange\_information\_between\_processes(comm, new\_pr, global\_vector, n)***

***check\_convergence\_and\_update\_converged\_status(global\_vector, pr, n, EPSILON, converged)***

***update\_local\_pagerank\_scores\_after\_exchange(n, pr, global\_vector)***

***normalize\_local\_pagerank\_scores(n, pr)***

***end\_time = get\_current\_time()***

***runtime = end\_time - start\_time***

***output\_iteration\_info(iter, rank, converged, runtime)***

***write\_pagerank\_to\_output\_file(output\_file, iter, rank, pr)***

***if converged:***

***break***

***close\_output\_file(output\_file)***

***free\_memory\_for\_global\_vector(global\_vector)***

***free\_memory\_for\_new\_pr(new\_pr)***

***Performance Comparison: OpenMP vs. MPI***

The project team executed both OpenMP and MPI versions of BFS and Page Rank algorithms on various data sizes. The computation time, communication time, and total time were recorded for different test cases, and the results were compiled in an Excel sheet.

***Dataset:***

*No of Vertices* *Algorithm* *Computation Time* *Communication Time* *Total Time* *Type*

7 BFS 0.373 0.287 0.661 OpenMP

7 BFS 0.002 83.994 83.997 MPI

7 Page Rank 2.7 0.0000 2.7 OpenMP

7 Page Rank 0.1 15041.0000 74624.1 MPI

5 BFS 1.453 0.189 1.643 OpenMP

5 BFS 0.001 0.090000 0.092 MPI

5 Page Rank 1.0000 0.0000 1.0000 OpenMP

5 Page Rank 0.0000 28395.4 148452.8 MPI

8 BFS 29.811 0.787 30.599 OpenMP

8 BFS 0.003 8755.826 8755.829 MPI

8 Page Rank 2.4 0.0000 2.4 OpenMP

8 Page Rank 0.1 91993.1 131421.6 MPI

***Conclusion:***

***Page Rank OpenMP vs. Page Rank MPI:***

* OpenMP shows varying performance, with a maximum total time of 2.7 seconds.
* MPI exhibits significantly higher total times, peaking at 74624.1 seconds.
* Despite the challenges, OpenMP remains more efficient than MPI for Page Rank.

***BFS MPI vs. BFS OpenMP:***

* OpenMP consistently outperforms MPI in terms of total time across different datasets.
* The computation time and communication time for MPI in some cases contribute to higher total times.
* OpenMP is recommended for better BFS performance in the given workload.

***Overall Project Conclusion:***

The project successfully implemented and compared parallel BFS and Page Rank algorithms.

OpenMP consistently outperforms MPI in various scenarios. The dataset provides insights into the strengths and weaknesses of each parallelization strategy. Recommendations for algorithm-specific choices are based on observed results. Further analysis and testing on a broader range of graphs are suggested for a comprehensive understanding of parallelization efficiency.





