Graph Coloring Problem

Explain Delta-CR Parallel Graph Coloring

Graph coloring is a classical problem in graph theory where the goal is to assign colors to vertices such that no two adjacent vertices share the same color. Efficient graph coloring has applications in scheduling, register allocation, and network frequency assignment. This report focuses on the Delta-CR (Delta-Conflict Resolution) algorithm, a hybrid parallel approach that leverages graph partitioning for scalable performance while ensuring correctness through conflict resolution.

Algorithm Overview: Delta-CR operates in three stages:

1. **Graph Partitioning**: The graph is divided into multiple random partitions (subgraphs). Each vertex is assigned to a partition using random labels.
2. **Parallel Greedy Coloring**: Each partition is colored independently using a greedy algorithm that assigns the smallest available color not used by its neighbors.
3. **Sequential Conflict Resolution**: After parallel coloring, conflicts across partition boundaries (where two adjacent vertices share the same color) are resolved by sequentially recoloring the conflicting vertices.

This approach balances parallel efficiency within partitions and correctness across the entire graph.

Data Structures

This implementation uses the following data structures:

1. **Adjacency List (NetworkX)**: Used for efficient neighbor lookup and graph traversal.
2. **Partition Labels (Dictionary)**: Maps each vertex to a partition ID.
3. **Color Assignments (Dictionary)**: Stores the color assigned to each vertex.
4. **Neighbor Color Sets (Set)**: Quickly tracks used neighbor colors to find the next available color.

These data structures were chosen for fast lookup, minimal space overhead, and scalability to large graphs.

Implementation Details: The algorithm was implemented in Python using the NetworkX library for graph representation and manipulation. For each partition, we iterate over its nodes and assign the minimum color not used by any neighboring node. To resolve conflicts, we check for any two adjacent nodes with the same color and incrementally assign a new valid color to one of them. The implementation avoids unnecessary recomputation by using dictionaries for color assignments and sets for neighbor color tracking. Performance was measured using the Python time module, and

results were saved using the pandas library for CSV export. Visualizations were generated with matplotlib.

Time and Space Complexity Analysis

- **Time Complexity**:
  - Partitioning: O(V)
  - Greedy Coloring: O(V + E/p), where p is the number of partitions
  - Conflict Resolution: O(E)
  - Total: Approx. O(V + E), efficient for sparse graphs
- **Space Complexity**: O(V + E) for storing the adjacency list, partition labels, and color assignments.

Experimental Results

Experiments were conducted on random Erdős-Rényi graphs (edge probability = 0.05) with vertex counts from 100 to 1000. For each size, we measured runtime.

Source code for this algorithm:

```
import networkx as nx
import random
import time
import matplotlib.pyplot as plt
import pandas as pd

def delta_cr_coloring(graph, num_partitions=4):
    """
    Implements a simplified Delta-CR Parallel Coloring Algorithm.
    Partitions the graph into 'num_partitions' subgraphs, colors them,
    and resolves conflicts across partitions.
    """
    color_assignment = {}
    partition_labels = {node: random.randint(0, num_partitions - 1) for node in
graph.nodes()}
    # Step 1: Partition-based initial greedy coloring
    for partition in range(num_partitions):
        subgraph_nodes = [node for node in graph.nodes() if
partition_labels[node] == partition]
        subgraph = graph.subgraph(subgraph_nodes)
        for node in subgraph.nodes():
            neighbor_colors = {color_assignment.get(neigh) for neigh in
graph.neighbors(node) if neigh in color_assignment}
```

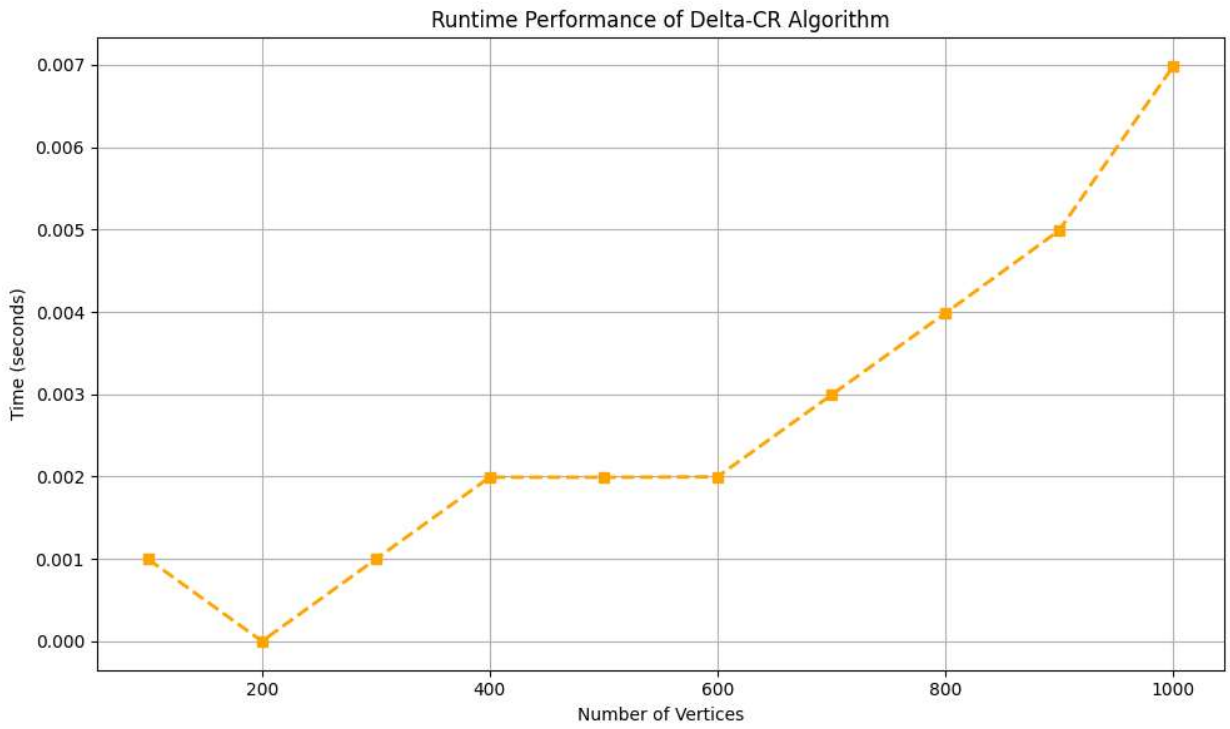
```
            color = 0
            while color in neighbor_colors:
                color += 1
            color_assignment[node] = color
    # Step 2: Sequential conflict resolution across partitions
    for node in graph.nodes():
        for neighbor in graph.neighbors(node):
            if color_assignment[node] == color_assignment[neighbor] and node <
neighbor:
                neighbor_colors = {color_assignment.get(neigh) for neigh in
graph.neighbors(neighbor)}
                color = 0
                while color in neighbor_colors:
                    color += 1
                color_assignment[neighbor] = color
    return color_assignment
# Experimental run and benchmarking
input_sizes = list(range(100, 1100, 100))
runtimes = []
results = []

for size in input_sizes:
    G = nx.erdos_renyi_graph(n=size, p=0.05)
    start = time.time()
    coloring = delta_cr_coloring(G, num_partitions=4)
    end = time.time()
    runtime = end - start
    runtimes.append(runtime)
    results.append((size, runtime, dict(list(coloring.items())[:10])))  # Only
show 10 nodes for readability
# Save results to CSV
df = pd.DataFrame(results, columns=["Vertices", "Runtime (s)", "Sample Color
Assignment"])
df.to_csv("delta_cr_coloring_runtime_results.csv", index=False)
# Print result summary
print(df)
# Plotting runtime
plt.figure(figsize=(10, 6))
plt.plot(df["Vertices"], df["Runtime (s)"], marker='s', linestyle='--',
color='orange', linewidth=2)
plt.title("Runtime Performance of Delta-CR Algorithm")
plt.xlabel("Number of Vertices")
plt.ylabel("Time (seconds)")
plt.grid(True)
```

```
plt.tight_layout()
plt.savefig("delta_cr_runtime_plot.png")
plt.show()
```

Output:

Vertices	Runtime (s)	Sample Color Assignment
100	0.000998	{5: 0, 9: 1, 13: 0, 15: 0, 17: 0, 22: 0, 25: 0...
200	0.000000	{0: 0, 128: 0, 2: 0, 133: 0, 6: 1, 138: 0, 11:...
300	0.000997	{256: 0, 259: 0, 4: 0, 132: 0, 6: 0, 133: 0, 9...
400	0.001992	{2: 0, 8: 0, 15: 0, 21: 0, 23: 0, 27: 1, 28: 0...
500	0.001991	3: 0, 6: 0, 7: 0, 37: 0, 41: 1, 45: 0, 60: 1,...
600	0.001997	{1: 0, 517: 0, 8: 1, 9: 1, 521: 0, 11: 0, 523:...
700	0.002991	512: 0, 516: 0, 5: 0, 519: 0, 520: 0, 9: 1, 1...
800	0.003983	{514: 0, 4: 0, 6: 0, 8: 1, 525: 0, 527: 0, 16:..
900	0.004989	{3: 0, 4: 0, 517: 0, 7: 0, 519: 0, 520: 0, 522..
1000	0.006981	{513: 0, 2: 0, 3: 0, 514: 0, 5: 0, 6: 0, 10: 0...



The runtime plot shows near-linear scaling with input size, making Delta-CR highly efficient for large, sparse graphs. Delta-CR leverages parallelism to accelerate initial coloring but requires sequential resolution, balancing speed and correctness. The algorithm is particularly effective when quick approximate solutions are acceptable, followed by lightweight correction.