Prije modifikacije:

```
grouped_by_candidate_color = nodes_with_candidate_color.groupByKey().mapValues(list)

colored_nodes_touple = grouped_by_candidate_color.flatMap(resolve_collisions)
```

```
def resolve_collisions(color_and_nodes):
    color, nodes = color_and_nodes
    if not nodes:
        return []
    # Use a set for O(1) lookups instead of a list
    colored_ids = set()
    colored_nodes = []
    # Sort nodes by ID (or degree) to minimize conflicts
    sorted_nodes = sorted(nodes, key=lambda x: len(x.neighbors))
    for node in sorted_nodes:
        # Check if any neighbor is already colored
        if not any(neighbor.id in colored_ids for neighbor in node.neighbors):
            colored_ids.add(node.id)
            colored_nodes.append(node)
    return [(node, color) for node in colored_nodes]
```

Poslije modifikacije:

```
# Use aggregateByKey to group nodes by color AND resolve conflicts in one operation
# This significantly reduces shuffling by combining operations
colored_nodes = color_assignment.aggregateByKey(
    [],  # Initial empty accumulator
    # Combine function within partitions
    lambda acc, node_info: resolve_conflicts_within_partition(acc, node_info),
    # Merge function across partitions
    lambda acc1, acc2: resolve_conflicts_across_partitions(acc1, acc2)
)

# Flatten the results into (node_id, color) pairs
color_updates = colored_nodes.flatMap(
    lambda color_nodes: [(node["node"].id, color_nodes[0]) for node in color_nodes[1]]
)
```

```
def resolve_conflicts_within_partition(accumulated_nodes, node_info):
    # Sort accumulated nodes by degree (descending)
    sorted_nodes = sorted(accumulated_nodes + [node_info],
                          key=lambda x: len(x["neighbor_ids"]),
                          reverse=True)

    # Keep track of colored node IDs to check for conflicts
    colored_ids = set()
    result = []

    for node_data in sorted_nodes:
        # Check if this node conflicts with already colored nodes
        if not any(nid in colored_ids for nid in node_data["neighbor_ids"]):
            colored_ids.add(node_data["node"].id)
            result.append(node_data)

    return result
```

```
def resolve_conflicts_across_partitions(nodes1, nodes2):
    # Combine both lists and sort by degree
    combined = sorted(nodes1 + nodes2,
                      key=lambda x: len(x["neighbor_ids"]),
                      reverse=True)

    colored_ids = set()
    result = []

    for node_data in combined:
        if not any(nid in colored_ids for nid in node_data["neighbor_ids"]):
            colored_ids.add(node_data["node"].id)
            result.append(node_data)

    return result
```

Teoretsko obrazlozenje:

aggregateByKey je efikasniji od groupByKey jer - vrsi agregaciju na nivou particije prije shuffle-a, sto znacajno smanjuje kolicinu podataka koji se prenose

Hijerarhijsko razrjesavanje konflikata - prvo se maksimalno iskoristi lokalnost podataka unutar svake particije, a zatim se minimizuje komunikacija izmedju particija.

Poredjenje vremena izvrsavanja:

Node-count	Max-degree	Neoptimizovano	Optimizovano
10	3	107	100
10	5	210	139
20	3	154	64
20	5	140	135
50	3	160	97
50	5	221	181
100	5	193	180
100	10	320	296
200	5	271	179
200	10	405	374