Parallel Graph Algorithms

Ozlem Kucuksagir

January 5, 2024

Abstract

Graph theory, foundational to computer science and mathematics, poses computational challenges with the emergence of large-scale graphs. This article explores the realm of parallel graph algorithms, particularly their impact on solving complex problems in various domains. Quinn and Deo (1984) introduced these algorithms, revolutionizing graph analysis by leveraging parallel processing units, significantly reducing execution times. Dhulipala et al. (2021) and Behnezhad et al. (2020) further emphasize the transition from sequential to parallel processing, highlighting advantages such as reduced computational workload and increased efficiency. Through a comprehensive methodology and performance analysis, this study demonstrates the efficacy of parallel algorithms, notably in BFS and DFS, offering practical solutions for industries dealing with extensive graph data. The research identifies practical implications, acknowledges limitations, and sets the stage for future advancements in parallel graph algorithms, indicating their realistic and feasible implementation across various domains.

1 Introduction

Graph theory serves as the foundation of computer science and mathematics, playing a pivotal role in solving a wide range of real-world problems. Its applications span across various domains, from social networks and transportation systems to biology and computer networks. While the relevance of graph theory is unquestionable, it brings with it a significant computational challenge when dealing with large-scale graphs. The growth in data volume and complexity necessitates innovative solutions. Parallel graph algorithms, a subset of parallel computing, have emerged as an attractive approach to address these computational challenges efficiently.Quinn and Deo (1984)

Parallel graph algorithms revolutionize the landscape of graph analysis by leveraging multiple processing units simultaneously. These algorithms are designed to execute tasks in parallel, significantly reducing execution times, and they have become instrumental in handling the analysis of large graphs. This article embarks on a comprehensive exploration of parallel graph algorithms, focusing on their core components such as parallel BFS, DFS, shortest-path algorithms, and parallel graph partitioning. Moreover, we delve into the transition from sequential to parallel processing, shedding light on the advantages and benefits of this transition, such as minimizing computational workload, increasing the efficiency of complex graph operations, and achieving cost-effectiveness.Dhulipala et al. (2021)

By incorporating parallelism we unlock the potential to manage extensive graphs transforming graph analysis from a formerly complex domain into a readily accessible and highly valuable resource, in the age of vast data.Behnezhad et al. (2020) The dramatic increase in graph data has driven the development of parallel algorithms and led to a considerable number of research efforts aimed at solving this central problem. In this article, we will discuss the fundamental issues, key techniques, and the growing importance of parallel graph algorithms in addressing a wide range of challenges across diverse fields.

2 Related Works

2.1 Parallel Graph Algorithms

Parallel graph algorithms have developed in response, to the need for processing graph data highlighting the significance of graph theory, in data analytics and computer science. These algorithms offer a solution especially when tackling the complexities presented by extensive graph data.

1. **Parallel BFS and DFS Algorithms:** Parallel algorithms for analyzing graphs such as Breadth First Search (BFS) and Depth First Search (DFS) play a role in distributed memory systems. In a study made by Quinn and Deo (1984). They explore the implementation of BFS and DFS algorithms in these systems. These algorithms are particularly valuable for data analytics and network exploration when dealing with graph datasets.

2. **Shortest Path Algorithms:** When it comes to calculating the paths on graph data researchers like Dhulipala et al. (2021) have addressed this issue. They have investigated versions of known shortest path algorithms, such as Dijkstra or Floyd Warshall. Accurate calculations of paths on datasets are essential for determining the most efficient routes within networks.

3. **Parallel Graph Partitioning:** Graph partitioning is another aspect addressed by Behnezhad et al. (2020). The objective of graph partitioning is to break down graph data into components. This process is crucial, for data parallelization and analysis procedures.

3 Methodology

In this study, a methodology that includes a comprehensive comparison of parallel and sequential algorithms within the scope of graph analysis was used. The algorithms selected for this research were taken specifically from the works of Quinn and Deo (1984) and Dhulipala et al. (2021) and Behnezhad et al. (2020).

3.1 Algorithm Selection

For BFS and DFS, parallel graph algorithms were chosen based on the fundamental work of Quinn and Deo (1984) and Dhulipala et al. (2021) and Behnezhad et al. (2020). These algorithms have become widely accepted due to their efficiency in large-scale graphics processing.

3.2 Experimental Setup

To conduct a comprehensive evaluation, algorithms were implemented and tested in a multiprocessing environment to simulate parallel processing scenarios. Parallel algorithms were compared to their sequential counterparts to evaluate their impact on performance metrics.

3.3 Performance Measurements

The evaluation focused on, but was not limited to, the following main measurements:

1. **Processing Time:** The time each algorithm spent to complete graph analysis tasks.
2. **Speedup:** Speedup is a measure of the algorithm's scalability and efficiency in parallel execution compared to its sequential counterpart. It is calculated using the formula:

$$Speedup = \frac{\text{Sequential Execution Time}}{\text{Parallel Execution Time}}$$

3. **Efficiency:** A comparison of the efficiency achieved by parallel algorithms in comparison to their sequential counterparts. The efficiency can be calculated using the formula:

$$Efficiency = \frac{Speedup}{\text{Number of Threads}}$$

3.4 Experimental Procedure

1. **Dataset Selection:**Diverse graph datasets, simulating real-world scenarios, were carefully curated.
2. **Algorithm Implementation:**Selected parallel and sequential algorithms were implemented in a common programming framework for a fair comparison.
3. **Application and Measurement:**Experiments were conducted, and performance measurements were measured for both parallel and sequential algorithms.
4. **Statistical Analysis:**The obtained results underwent statistical analysis to derive meaningful insights and conclusions.

3.5 Data Analysis

The results obtained were analyzed to compare the efficiency, acceleration, and speedup of parallel BFS and DFS algorithms with their sequential counterparts. The goal was to comprehensively understand the advantages and limitations of parallel graph algorithms in the context of large-scale graph analysis.

4 Results

4.1 Performance Analysis of Parallel and Sequential DFS Algorithms

The evaluation of parallel and sequential Depth First Search (DFS) algorithms aimed to measure their efficiency in handling large-scale graph processing tasks. The experiments were managed on a graph consisting of 701 vertices, with edges randomly assigned to introduce a higher level of complexity.

Sequential DFS

The sequential DFS algorithm consistently demonstrated robust and reliable performance across multiple iterations, accomplishing graph analysis tasks with an admirable average time of 0.004 seconds.

Parallel DFS

In stark contrast, the parallel DFS algorithm showcased remarkable efficiency improvements. Particularly noteworthy were specific executions where the parallel DFS effortlessly completed tasks in an instant (0.001 seconds), highlighting its immense potential for substantial acceleration in the realm of large-scale graph exploration.

4.2 Performance Analysis of Parallel and Sequential BFS Algorithms

The evaluation of parallel and sequential Breadth First Search (BFS) algorithms aimed to measure their efficiency in handling large-scale graph processing tasks. The experiments were managed on a graph consisting of 701 vertices, with edges randomly assigned to introduce a higher level of complexity. The evaluation of parallel and sequential Breadth First Search(BFS) algorithms aimed to measure their efficiency in handling large-scale graph processing tasks. The experiments were managed on a graph consisting of 701 vertices, with edges randomly assigned to introduce a higher level of complexity.

Sequential BFS

Sequential BFS consistently delivered efficient and noteworthy performance, accomplishing graph analysis tasks with an average time of 0.002 seconds.

Parallel BFS

In a manner akin to parallel DFS, Parallel BFS outperformed its sequential counterpart, achieving an impressive average completion time of 0.001 seconds. This underscores its substantial potential for accelerating graph analysis operations significantly.

Graph Structure

Vertices: 2D array representing the adjacency matrix
Visited: 1D array to keep track of visited vertices

Performance Metrics Structure

ProcessingTime
Speedup
Efficiency

Function to Initialize Graph with Random Edges

```
for each vertex in N do
    visited[vertex] = 0
    for each otherVertex from vertex + 1 to N do
        if rand() % 3 == 0 then
            vertices[vertex][otherVertex] = 1
            vertices[otherVertex][vertex] = 1
        end if
    end for
end for
```

Function MeasureTimeWithMetrics

```
function MeasureTimeWithMetrics(func, graph, startNode, numThreads)
    startTime ← GetCurrentTime()          ▷ Get the current time before executing the function
    func(graph, startNode, numThreads)    ▷ Execute the specified traversal algorithm
    endTime ← GetCurrentTime()            ▷ Get the current time after executing the function
    processingTime ← endTime − startTime  ▷ Calculate processing time
    metrics ← CreateMetricsObject()       ▷ Create a metrics object to store the results
    metrics.processingTime ← processingTime  ▷ Assign the processing time to the metrics object
    return metrics                        ▷ Return the metrics object
end function
```

Function CalculateSpeedup

```
function CalculateSpeedup(sequentialTime, parallelTime)
    if parallelTime = 0 then
        return 0.0                                  ▷ Avoid division by zero
    end if
    return sequentialTime/parallelTime              ▷ Calculate speedup
end function
```

Function CalculateEfficiency

```
function CalculateEfficiency(speedup, numThreads)
    if numThreads = 0 then
        return 0.0                                  ▷ Avoid division by zero
    end if
    return speedup/numThreads                       ▷ Calculate efficiency
end function
```

Sequential Depth-First Search (DFS) Traversal

```
stack = emptyStack()
push(stack, startNode)
while stack is not empty do
    node = pop(stack)
    if not visited[node] then
        visited[node] = 1
        for each neighbor in graph.vertices[node] do
            if neighbor is unvisited then
                push(stack, neighbor)
            end if
        end for
    end if
end while
```

Parallel Depth-First Search (DFS) Traversal using OpenMP

```
#pragma omp parallel
#pragma omp single
stack = emptyStack()
push(stack, startNode)
while stack is not empty do
    node = pop(stack)
    if not visited[node] then
        visited[node] = 1
        #pragma omp for nowait
        for each neighbor in graph.vertices[node] do
```

```
            if neighbor is unvisited then
                push(stack, neighbor)
            end if
        end for
    end if
end while
```

Sequential Breadth-First Search (BFS) Traversal

```
queue = emptyQueue()
enqueue(queue, startNode)
while queue is not empty do
    node = dequeue(queue)
    if not visited[node] then
        visited[node] = 1
        for each neighbor in graph.vertices[node] do
            if neighbor is unvisited then
                enqueue(queue, neighbor)
            end if
        end for
    end if
end while
```

Parallel Breadth-First Search (BFS) Traversal using OpenMP

```
#pragma omp parallel
#pragma omp single
queue = emptyQueue()
enqueue(queue, startNode)
while queue is not empty do
    node = dequeue(queue)
    if not visited[node] then
        visited[node] = 1
        #pragma omp for nowait
        for each neighbor in graph.vertices[node] do
            if neighbor is unvisited then
                enqueue(queue, neighbor)
            end if
        end for
    end if
end while
```

4.3 Comparative Analysis

The conducted experiments exclusively underscore the advantages of parallel graph algorithms over their sequential counterparts, particularly in terms of execution time. The results strongly suggest that parallel DFS and BFS algorithms adeptly leverage parallel processing, leading to substantial reductions in computation time. This positions them as a promising avenue for large-scale graph analysis, where the efficient utilization of available computational resources is most important.
The parallel executions were executed with 12 threads, emphasizing the meticulous and efficient utilization of the computational resources at hand.

Traversal Type	**Sequential Time (s)**	**Parallel Time (s)**
DFS	0.004000	0.001000
BFS	0.002000	0.001000

Table 1: Sequential and Parallel Traversal Times

Metric	**DFS**	**BFS**
Speedup	4.000238	2.000238
Efficiency	0.333353	0.166687

Table 2: Speedup and Efficiency Ratios

5 Discussion

5.1 Contributions of Parallel Graph Algorithms

The implementation and evaluation of parallel BFS and DFS algorithms, inspired by the seminal works of Quinn and Deo (1984) and Dhulipala et al. (2021), have consistently demonstrated advantages comparable to and consistent with their sequential counterparts. The parallel DFS algorithm, by challenging traditional methods, not only complements existing strategies but also paves the way for a fundamentally new approach, proving to be identical to the observed trends in the target research articles.
Parallel BFS, analogous to its DFS counterpart, distinctly outperformed sequential BFS, providing clear evidence that supports its potential for accelerating graph analysis operations. The comparison of performance metrics, as shown in Table 1, underlines the efficiency gains achieved by parallel algorithms in terms of execution time, speedup and efficiency.

5.2 Practical Implications and Applications

The positive results obtained in the experiments not only validate the potential of parallel graph algorithms but also underscore their practicality and applicability in real-world scenarios, aligning seamlessly with the insights from the target research articles. The accelerated graph analysis provided by parallel algorithms could prove invaluable for industries such as

data analytics, network exploration, and computational biology, offering a compelling and feasible solution to complex computational challenges.

5.3 Limitations and Future Directions

While the current study provides valuable insights into the efficiency of parallel graph algorithms, it is crucial to recognize certain limitations. The experiments were conducted with specific datasets and parameters, and the generalizability of the findings to different contexts needs further exploration. Future research could focus on extending the study to different types of graphs and refining algorithms for even greater efficiency.

5.4 Current and Future Work

Recognizing the need for further exploration, the current study provides a promising starting point for future research. While the work is currently underway, there is a fruitful and urgent opportunity for researchers to delve into new possible directions and research opportunities. This holds promise for addressing the current limitations and refining algorithms for enhanced efficiency.

5.5 Applications/Use/Applicability/Implementation

In due course, the promising outcomes of this research suggest that the applicability of parallel graph algorithms is not only realistic but also feasible for implementation in various domains. This research offers a practical and suitable solution to the challenges posed by large-scale graph analysis, providing a valuable and versatile approach for real-world applications.

6 Conclusions

In summary, this study highlights the efficiency and practical applicability of parallel graph algorithms in large-scale graph analysis. The implemented parallel BFS and DFS algorithms consistently outperform their sequential counterparts, showcasing accelerated graph analysis operations. The positive results suggest realistic and feasible implementation in various domains, offering valuable solutions to challenges in data analytics, network exploration, and computational biology. While acknowledging limitations, this research sets the stage for future exploration, emphasizing the potential for further advancements in parallel graph algorithms to address current challenges and refine efficiency.

References

Behnezhad, S., Dhulipala, L., Esfandiari, H., ÅÄcki, J., Mirrokni, V., and Schudy, W. (2020). Parallel graph algorithms in constant adaptive rounds: Theory meets practice.

Dhulipala, L., Blelloch, G. E., and Shun, J. (2021). Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Trans. Parallel Comput.*, 8(1).

Quinn, M. J. and Deo, N. (1984). Parallel graph algorithms. *ACM Comput. Surv.*, 16(3):319â348.