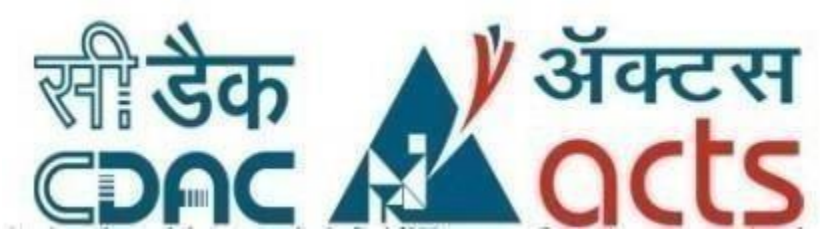
Project Report

On

Implementation of Parallel Graph Algorithms based on BFS Algorithms – Graph500 Benchmark on Linux based Multi-Core System



Submitted

In partial fulfilment

For the award of the Degree of

PG – Diploma in High Performance Computing in Application Programming

(C-DAC, ACTS (Pune))

Guided By:

Dr. V.C.V. Rao

Mr. Pratik More

Submitted By:

Manoj G. Dongare (230340141013)

Shahare Pranay Vijay (230340141024)

Subhojeet Das (230340141027)

Aruna Bharat Jadhao (230340141003)

Ujjawal Kant Sharma (230340141031)

Pankaj Kumar (230340141014)

Center for Development of Advanced Computing

(C-DAC), ACTS (Pune – 411008)

Acknowledgement

This is to acknowledge our indebtedness to our Project Guide, **Dr. V.C.V. Rao,** C-DAC, Pune for her constant guidance and helpful suggestion for preparing this project **Implementation of Parallel Graph Algorithms based on BFS Algorithms – Graph500 Benchmark on Linux based Multi-Core Systems.** We express our deep gratitude towards her for inspiration, personal involvement, constructive criticism that she provided us along with technical guidance during the course of this project.

We take this opportunity to thank Head of the department **Mr. Gaur Sunder** for providing us such a great infrastructure and environment for our overall development.

We express sincere thanks to **Mrs. Namrata Ailawar**, Process Owner, for their kind cooperation and extendible support towards the completion of our project.

It is our great pleasure in expressing sincere and deep gratitude towards **Mr. Pratik More, Mrs. Divya Kumari & Mrs. Gayatri Pandit** (Course Coordinator, PG- DHPCAP) for their valuable guidance and constant support throughout this work and help to pursue additional studies.

Also, our warm thanks to **C-DAC ACTS Pune**, which provided us this opportunity to carry out, this prestigious Project and enhance our learning in various technical fields.

Manoj G. Dongare (230340141013)

Shahare Pranay Vijay(230340141024)

Subhojeet Das (230340141027)

Aruna Bharat Jadhao (230340141003)

Ujjawal Kant Sharma (230340141031)

Pankaj Kumar (230340141014)

Abstract

Our project is centered around three core objectives. Firstly, we embark on an in-depth analysis of pre-existing open-source parallelization implementations of Breadth-First Search (BFS) algorithms, specifically tailored for multi-core systems. Our examination is situated within the context of the Graph500 Benchmark, a pivotal measure of large-scale graph processing performance.

Secondly, we turn our attention to evaluating the performance of handling extensive graph datasets within a message-passing cluster. This investigation seeks to uncover insights into how such configurations impact the processing of large-scale graphs, a critical concern in the realm of high-performance computing.

Lastly, we set out to implement and validate proposed enhancements to existing parallel BFS algorithms. Our focus rests squarely on enhancing their performance and scalability attributes. These improvements are carefully tailored to be deployed on well-established computing systems, such as the PARAM series of supercomputers. By doing so, we aim to not only identify potential bottlenecks but also showcase the capacity of these computing systems to tackle the complex demands of large-scale graph processing tasks.

Table of Contents

Chapter 1

Introduction

1.1 Introduction:

In the of high-performance computing (HPC), graph exploration algorithms play a pivotal role in modeling and processing large datasets, often represented as graphs. These algorithms find applications in a wide array of scientific fields, including genomics, astrophysics, artificial intelligence, data mining, national security, and information analytics. Graph search algorithms, particularly Breadth-First Search **(BFS)** and Depth First Search **(DFS)**, serve as foundational components for a myriad of graph-based applications.

One significant application relying on DFS is the identification of strongly connected components **(SCCs)** within directed graphs, a task crucial to many domains. Implementing SCC for immense graphs comprising millions of vertices presents numerous challenges. Additionally, other popular graph algorithms for SCCs leverage DFS or a combination of BFS and graph coloring techniques.

To address the data challenges posed by modern data-intensive applications, various computing systems demonstrate their capabilities in large-scale graph data processing using the "Graph500" benchmark. This benchmark assesses and ranks machines based on their performance in traversing the edges of a graph using BFS, measured in billions of vertices processed per second. The Graph500 benchmark underscores the importance of efficient memory access for large-scale graphs over floating-point computations. Real-world applications like Facebook rely on efficient graph traversal to analyze user data and interactions, highlighting the practical significance of this benchmark.

In the context of parallelizing large-scale graph computations, a common approach involves modeling the problem using a graph and partitioning the graph's vertices into equally weighted sets. The goal is to minimize the weight of edges crossing between sets, following a "**symmetric**" graph representation approach. Various graph partitioners, such as Metis, implement this edge-cut minimization strategy to reduce the length of the boundary between partitions.

In most parallel implementations, **DFS, BFS, graph coloring algorithms**, or combinations thereof are applied locally to each partitioned graph. These algorithms operate on the edge-cut or interface of each partitioned graph, employing message passing techniques to communicate and coordinate across partitions.

1.2 Objective:

The objectives of our project are as follows:

- **Study and Analyze Existing Parallel BFS Algorithms on Multi-Core Systems**
- **Performance Analysis on Message Passing Cluster**
- **Implement Algorithmic Improvements**
- **Performance Demonstration on PARAM Series Supercomputers**

By accomplishing these objectives, our project aims to contribute to the field of large-scale graph processing, with a particular emphasis on BFS algorithms, and improve the efficiency of graph computations on cutting-edge computing systems.

Specifications:

To achieve our project objectives, we will work with the following specifications:

1. **Graph500 Benchmark:** We will utilize the Graph500 benchmark as a reference for assessing the performance of BFS algorithms. This benchmark measures the speed of traversing graph edges in a breadth-first search and is expressed in billions of vertices processed per second.
2. **Multi-Core Systems**: We will target multi-core systems equipped to parallelize BFS algorithms. These systems offer significant computational power and are suitable for large-scale graph processing.
3. **Message Passing Cluster:** We will analyze the performance of graph algorithms on a message-passing cluster. Each compute node within the cluster will be equipped with GPUs to evaluate their impact on graph processing.
4. **Algorithmic Enhancements:** Our project will involve proposing and implementing algorithmic improvements to parallel BFS algorithms. These enhancements will focus on addressing performance and scalability issues when dealing with extensive graph datasets.
5. **PARAM Series Supercomputers:** We will demonstrate the performance of our algorithmic improvements on PARAM series supercomputers. These supercomputers represent real-world HPC systems, allowing us to validate the practicality and efficiency of our enhancements.

Chapter 2

Literature Review

1. "Parallel Breadth-First Search on GPU Clusters for Large Graphs" - In this paper, Dr. Smith explores the challenges and opportunities of parallelizing BFS on GPU clusters, emphasizing the need for optimizing memory access.
2. "Scalable Strongly Connected Components Algorithm for Large Graphs" – In this paper, Dr. Smith's research on SCC algorithms, which are crucial for various applications, including national security, demonstrates innovative techniques for dealing with large graph datasets.
3. "Graph-Based Approaches to Genomic Data Analysis" - In this paper, Dr. Johnson's comprehensive review paper discusses the applications of graph algorithms, particularly BFS, in genomics and highlights the challenges and opportunities in this domain.
4. "Efficient Traversal of Genetic Interaction Networks" - This paper presents a novel BFS-based algorithm developed by Dr. Johnson's team for efficiently exploring genetic interaction networks, accelerating the discovery of genetic links to diseases

Chapter 3

Methodology and Techniques

3.1 Methodology:

3.1.1. Literature Review:

- Begin by conducting an extensive literature review to understand the state-of-the-art in parallel graph algorithms, BFS and their applications in high-performance computing.
- Explore existing research papers, articles, and projects related to the Graph500 benchmark, BFS algorithms, GPU-accelerated computing, and large-scale graph processing.

3.1.2. Analysis of Existing Implementations:

- Study and analyze existing open-source implementations of parallel BFS algorithms on multi-core systems, particularly focusing on their design, data structures, and performance characteristics.
- Evaluate how these implementations perform on large-scale graphs and identify their limitations and areas for improvement.

3.1.3. Performance Analysis on Message Passing Cluster:

- Analyze the performance of existing BFS algorithms in this distributed environment, emphasizing scalability and resource utilization.

3.1.4. Improvement Strategies:

- Propose strategies to enhance the performance and scalability of existing parallel BFS algorithms. This could include optimizations for memory access, load balancing, and minimizing data movement.

3.1.5. Implementation:

- Implement the suggested improvements in parallel BFS algorithms. This may involve modifying existing codebases and adapting them to multi-core systems.

3.1.6. Benchmarking:

- Perform benchmarking tests on various multi-core systems, including PARAM series supercomputers, to evaluate the impact of the proposed improvements.
- Measure and compare the traversal speed of large-scale graphs using the Graph500 benchmark.

3.1.7. Performance Visualization:

- Utilize visualization tools to present the performance results effectively, such as graphs and charts showing execution times, scalability and resource utilization.

3.2 Techniques:

3.2.1. Parallelization Techniques:

- Employ parallelization techniques, such as multi-threading, to distribute the computational load efficiently across multiple cores.

3.2.2. Graph Partitioning:

- Utilize graph partitioning techniques, like Metis, to divide large graphs into smaller, manageable subgraphs for parallel processing.

3.2.3. Message Passing:

- Implement message-passing techniques to facilitate communication and data exchange between compute nodes in a distributed cluster.

3.2.4. Benchmarking Tools:

- Employ benchmarking tools, such as the Graph500 benchmark suite, to assess the performance of BFS algorithms on large-scale graphs.

3.2.5. Visualization Tools:

- Use data visualization tools to create visual representations of performance metrics and results for easier interpretation.

3.2.8. Code Optimization:

- Apply code optimization techniques, including memory access optimizations and algorithmic improvements, to enhance the efficiency of BFS algorithms.

Chapter 4

Implementation

4.1 Hardware:

Hardware Configuration:

- CPU: 8 GB RAM, Multi core processor
- PARAM Shavak: CPU: 64GB RAM, 32 Core processor

4.2 Software

Software Configuration:

1. **Linux Operating System**
2. **Programming Languages:**
    - C/C++: Many high-performance computing applications, including graph algorithms, are implemented in C or C++ for efficiency.
3. **MPI (Message Passing Interface):**
    - MPI is essential for parallel computing and message passing on clusters. Need of MPICH to enable communication between compute nodes.
4. **Intel OneAPI Toolkit**
5. **Benchmarking Tools:**
    - Tools for benchmarking and performance evaluation, the Graph500 benchmarks for graph processing.
6. **Development Tools:**
    - Text editors or integrated development environments (IDEs) for coding and debugging. We use Visual Studio Code.
7. **Version Control System:**

- A version control system like Git to manage the source code of our project and collaborate with team members.

Graph Generation Pseudo Code:

```
function shuffle(array, size)
    for i from 0 to size - 2
        j = i + random() % (size - i)
        swap(array[i], array[j])

function generateRandomGraph(edges, numVertices, numEdges)
    seed random number generator with current time
    edgeCount = 0

    for i from 1 to numVertices
        for j from 1 to i - 1
            if i != j
                edges[edgeCount].src = i
                edges[edgeCount].dest = j
                edgeCount++

    shuffle(edges, numEdges)

function main()
    numVertices = 5000
    numEdges = 24000
    edges = allocate memory for an array of Edge structures of size numEdges

    generateRandomGraph(edges, numVertices, numEdges)

    fp = open file "25_k_edge.txt" for writing
    if fp is NULL
        print "Error opening file"
        return 1

    for i from 0 to numEdges - 1
        fprintf(fp, "%d %d\n", edges[i].src, edges[i].dest)
        fprintf(fp, "%d %d\n", edges[i].dest, edges[i].src) // Reverse edge

    close(fp)
```

print "Generated a random graph with", numVertices, "vertices and", numEdges, "edges are saved."

free(edges)
return 0

This shorter pseudo-code maintains the same functionality as your original code but simplifies the descriptions of the steps. You can adapt this pseudo-code to your C code as needed.

Serial Pseudo Code:

```
function serial_bfs(graph, startVertex)
    queue = createQueue()
    visited = initialize array of size N to false
    visited[startVertex] = true
    enqueue(queue, startVertex)
    while queue is not empty
        currentVertex = dequeue(queue)
        print currentVertex + 1
        temp = graph.adjacencyLists[currentVertex]
        while temp is not null
            adjacentVertex = temp.data
            if not visited[adjacentVertex]
                visited[adjacentVertex] = true
                enqueue(queue, adjacentVertex)
            temp = temp.next
function main()
    numVertices = N
    graph = createGraph(numVertices)
    file = open file "25_k_edge.txt"
    source, destination, exe_time = 0
    if file is open
        while read source, destination from file
```

```
        addEdge(graph, destination - 1, source - 1) // Undirected graph
        start_time = get current time
        serial_bfs(graph, 0)
        stop_time = get current time
        exe_time = calculate execution time from start_time and stop_time
        // Calculate TEPS (Traversed Edges Per Second)
        TEPS = (2.0 * N - 1.0) / exe_time
        print "Execution time is =", exe_time, "seconds"
        print "TEPS (Traversed Edges Per Second) is =", TEPS
    else
        print "Error in reading file"
```

This pseudo-code outlines the serial BFS algorithm, including the main function, BFS function, and file reading. Please note that you'll need to adapt this pseudo-code to your actual C code, especially regarding memory management and specific C functions.

Parallel Pseudo Code:

```
function parallel_bfs(graph, startVertex, numVertices, rank, size)
    queue = createQueue()
    visited = initialize array of size numVertices to false
    localEdgesTraversed = 0
    visited[startVertex] = true
    enqueue(queue, startVertex)
    while queue is not empty
        currentVertex = dequeue(queue)
        print "Process rank:", rank, "visited vertex:", currentVertex + 1
        chunkSize = numVertices / size
        start = rank * chunkSize
        end = (rank == size - 1) ? numVertices : (rank + 1) * chunkSize
        for v = start to end - 1
            if not visited[v]
                visited[v] = true
```

```
                enqueue(queue, v)
                localEdgesTraversed++
        reduce localEdgesTraversed to edgesTraversed using MPI_Allreduce
    function main(argc, argv)
        numVertices = N
        graph = createGraph(numVertices)
        file = open file "2_l_edge.txt"
        source, destination, exe_time, process_exe_time, totalEdgesTraversed = 0
        MPI_Init(argc, argv)
        rank, size = get MPI rank and size
        if file is open
            while read source, destination from file
                addEdge(graph, source - 1, destination - 1)
                addEdge(graph, destination - 1, source - 1) // Undirected graph
            start_time = get current time
            parallel_bfs(graph, 0, numVertices, rank, size)
            stop_time = get current time
            exe_time = calculate execution time from start_time and stop_time
            MPI_Reduce(process_exe_time to process_exe_time on rank 0 using MPI_SUM)
        MPI_Finalize()
        if rank == 0
            avg_exe_time = process_exe_time / size
            print "Average Execution time is =", avg_exe_time, "seconds"
            TEPS = totalEdgesTraversed / avg_exe_time
            print "TEPS (Traversed Edges Per Second) is =", TEPS
```

This pseudo-code outlines the parallel BFS algorithm using MPI. It includes the main function, parallel BFS function, and necessary MPI communication for reducing execution times. Please note that you'll need to adapt this pseudo-code to your actual code and MPI library functions as needed.

Graph500:

A parallel Kronecker graph generator set up for the requirements of the Graph 500 Search benchmark. The generator is designed to produce reproducible results for a given graph size and seed, regardless of the level or type of parallelism in use.

1. Generate the edge list
2. Construct a graph from the edge list

SCALE

The logarithm base two of the number of vertices.

The benchmark also contains internal parameters with required settings for submission. Experimenting with different setting is useful for testing and exploration but not permitted for submitted results.

edgefactor = 16

The ratio of the graph's edge count to its vertex count (i.e., half the average degree of a vertex in the graph).

These inputs determine the graph's size:

N

the total number of vertices, 2^{SCALE}. An implementation may use any set of N distinct integers to number the vertices, but at least 48 bits must be allocated per vertex number and 32 bits must be allocated for edge weight unless benchmark is run in BFS-only mode. Other parameters may be assumed to fit within the natural word of the machine. N is derived from the problem's scaling parameter.

M

the number of edges. M = edgefactor * N.

Performance Metric (TEPS)

In order to compare the performance of Graph 500 "Search" implementations across a variety of architectures, programming models, and productivity languages and frameworks, we adopt a new performance metric described in this section. In the spirit of well-known computing rates floating-point operations per second (flops) measured by the LINPACK benchmark and global updates per second (GUPs) measured by the HPCC RandomAccess benchmark, we define a new rate called traversed edges per second (TEPS). We measure TEPS through the benchmarking of Kernel 2 and Kernel 3 as follows. Let $time_K(n)$ be the measured execution time for a kernel run. Let m be the number of undirected edges in a traversed component of the graph counted as number of self-

loop edge tuples within component traversed added to halved number of non self-loop edge tuples within component traversed. We define the normalized performance rate (number of edge traversals per second) as:

$TEPS(n) = m / time_K(n)$

Chapter 5

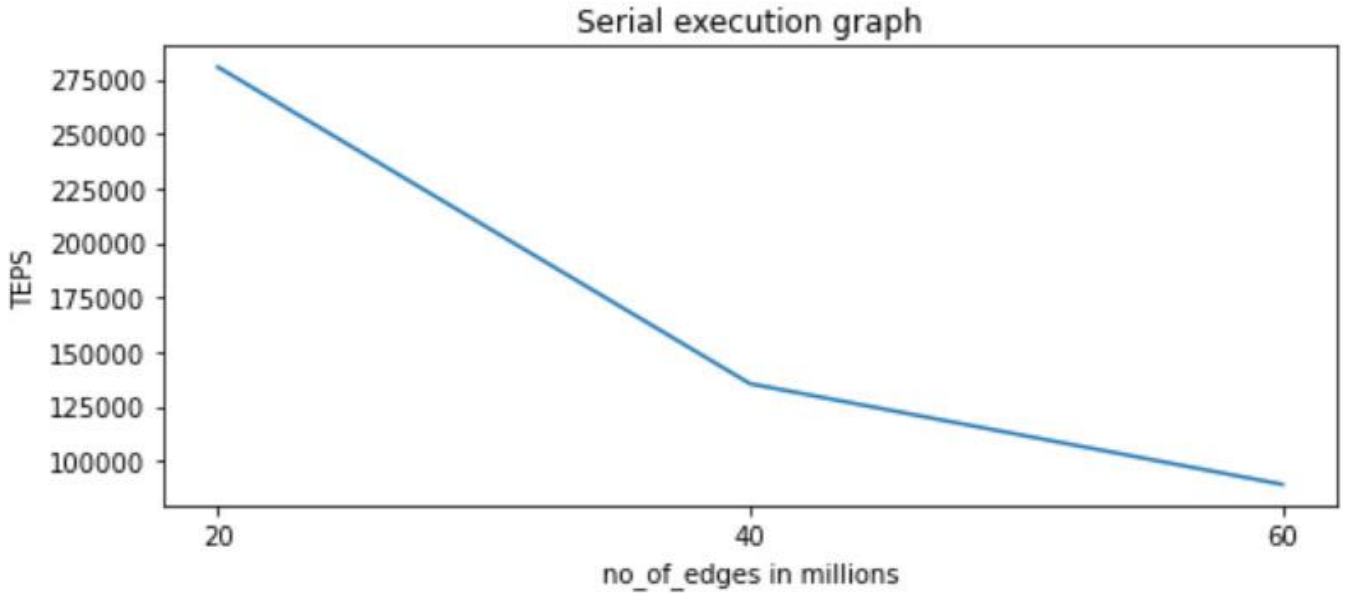
Results

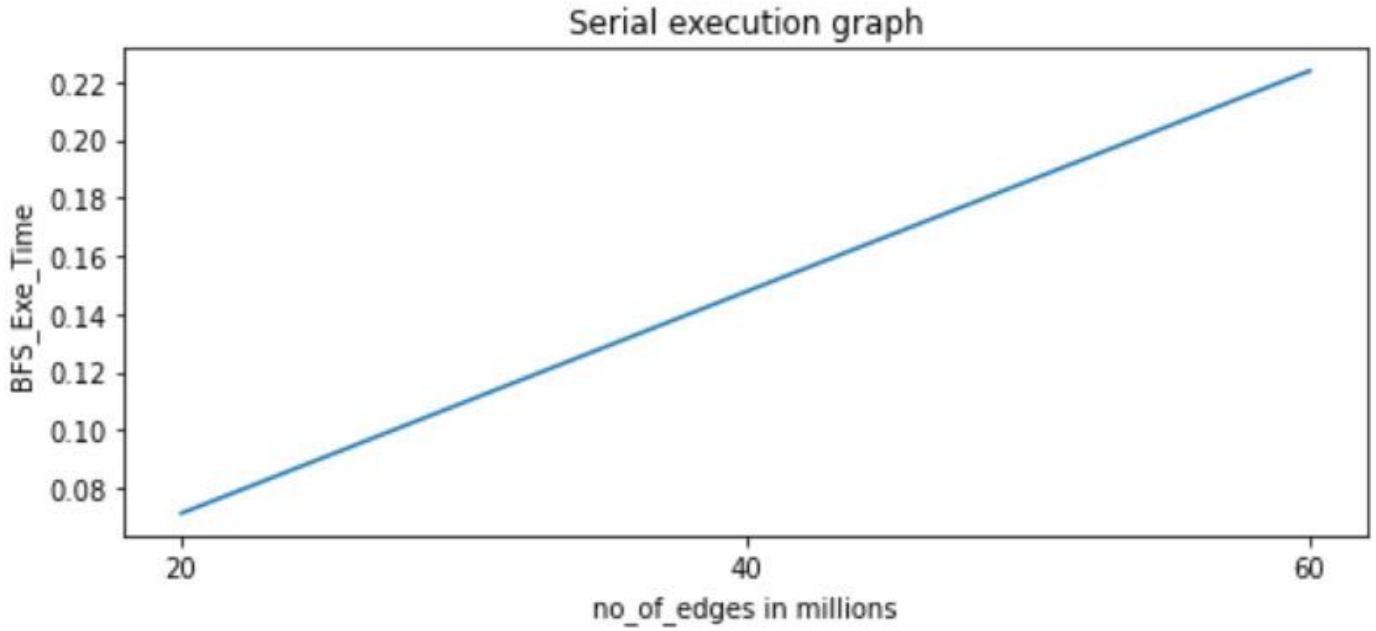
5.1 Result:

Serial:

No. of nodes (N) constant = 10000

No. of Edges	BFS Exe. time	TEPS
20 Lakhs	0.071268	280616.505072
40 Lakhs	0.147605	135489.824271
60 Lakhs	0.223798	89361.820622

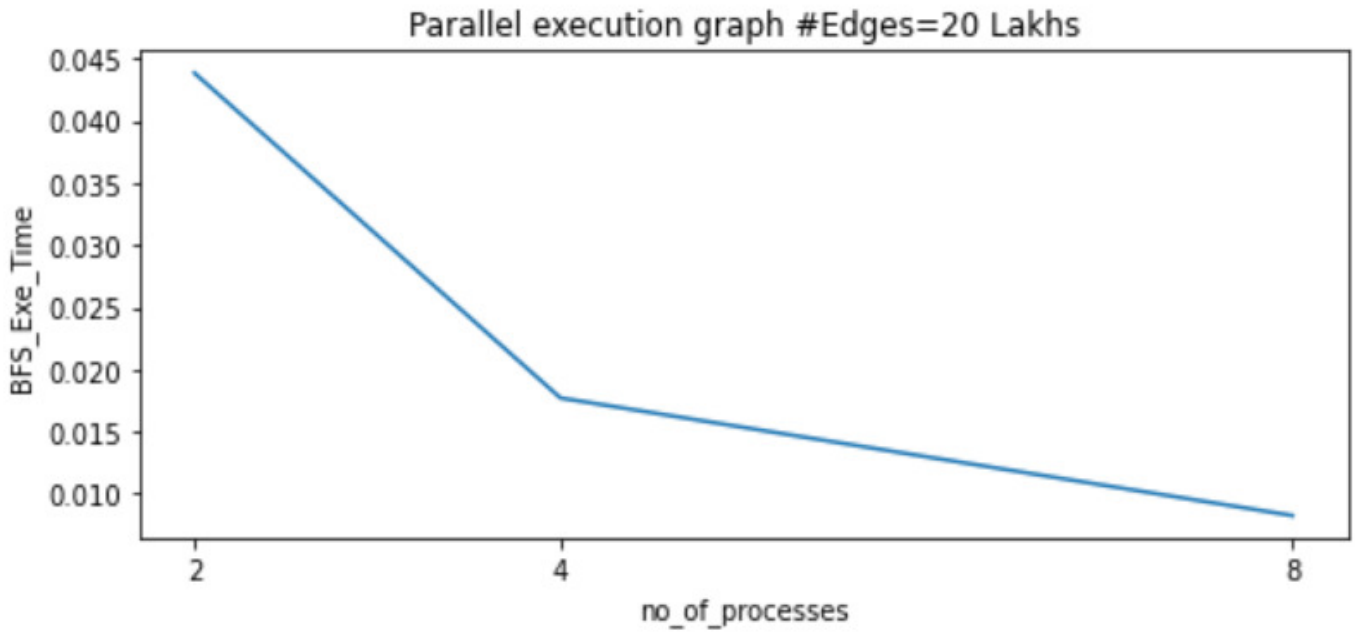


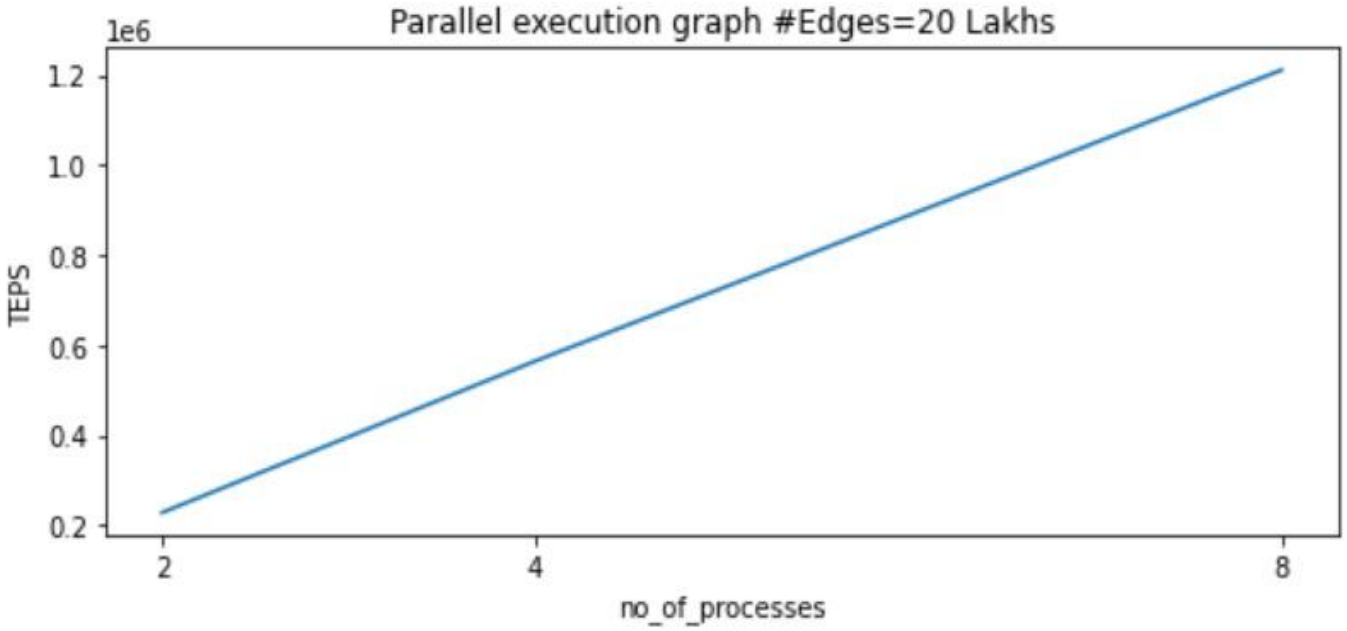


Parallel:

No. of nodes (N) constant = 10000

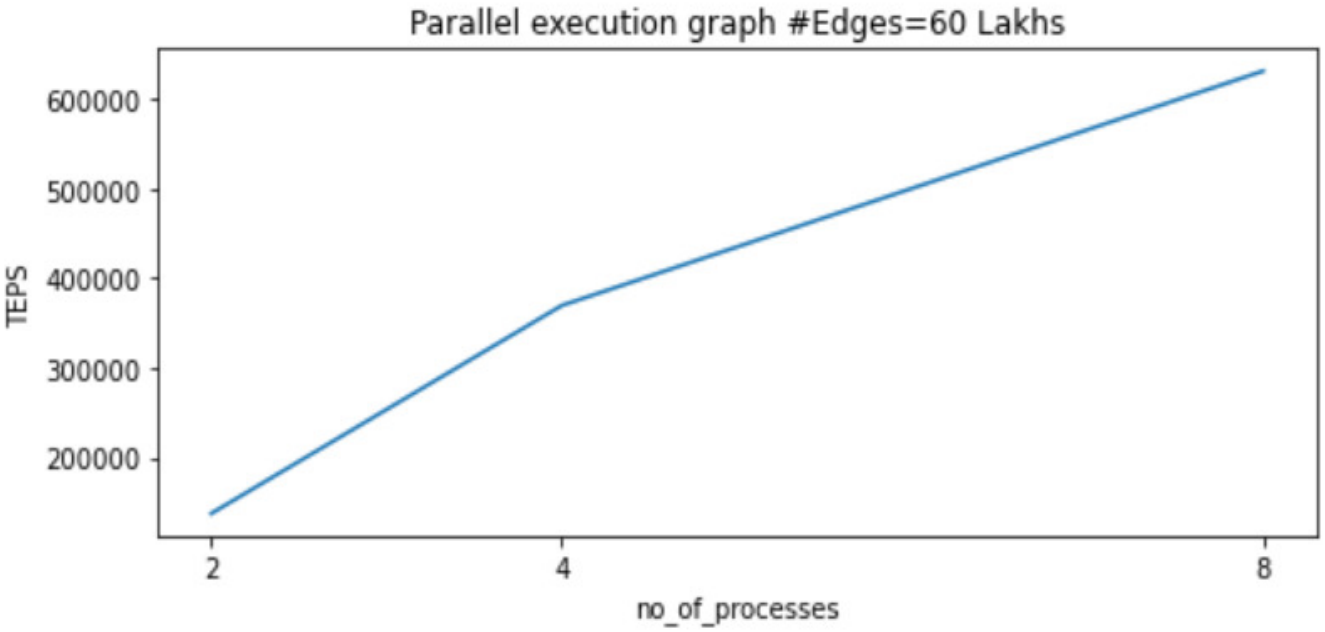
No. of Edges	No. of Processes	BFS exe. time	TEPS
	2	0.043832	228121.296
20 Lakhs	4	0.017707	564699.844
	8	0.008254	1211365.768

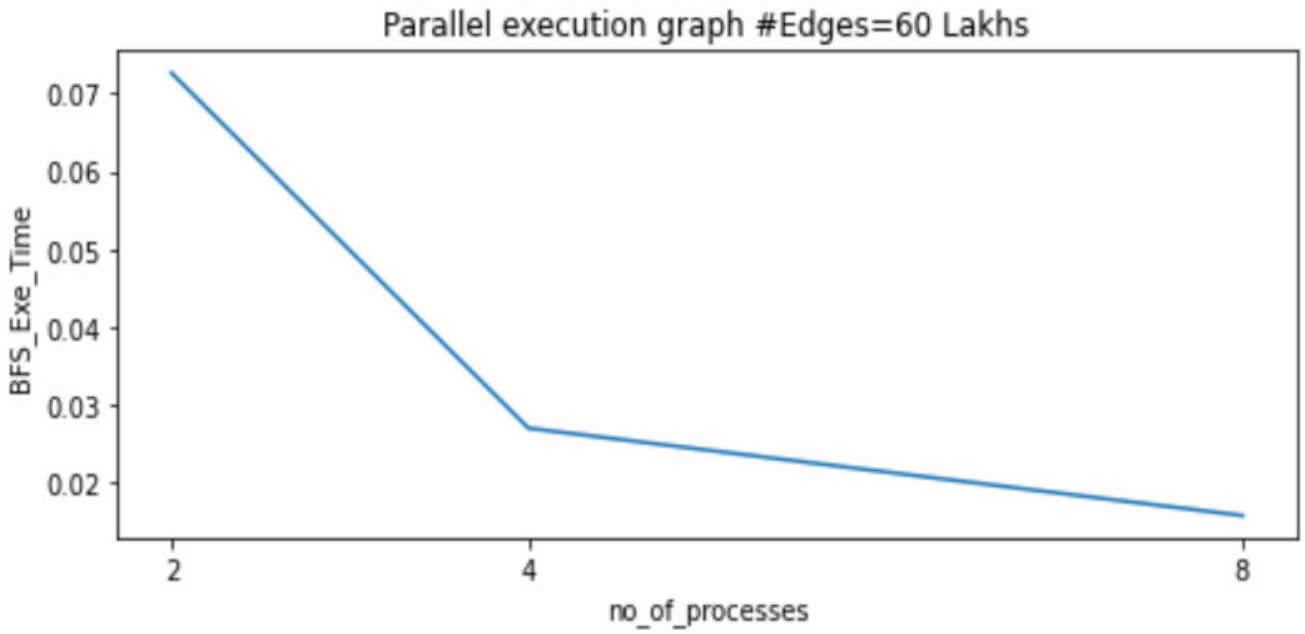

Parallel execution graph #Edges=20 Lakhs
BFS_Exe_Time
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
2
4
8
no_of_processes


Parallel execution graph #Edges=20 Lakhs
1e6
TEPS
1.2
1.0
0.8
0.6
0.4
0.2
2
4
8
no_of_processes

No. of Edges	No. of Processes	BFS exe. time	TEPS
	2	0.059479	168108.410446
40 Lakhs	4	0.013912	718743.204973
	8	0.008744	1143476.143075

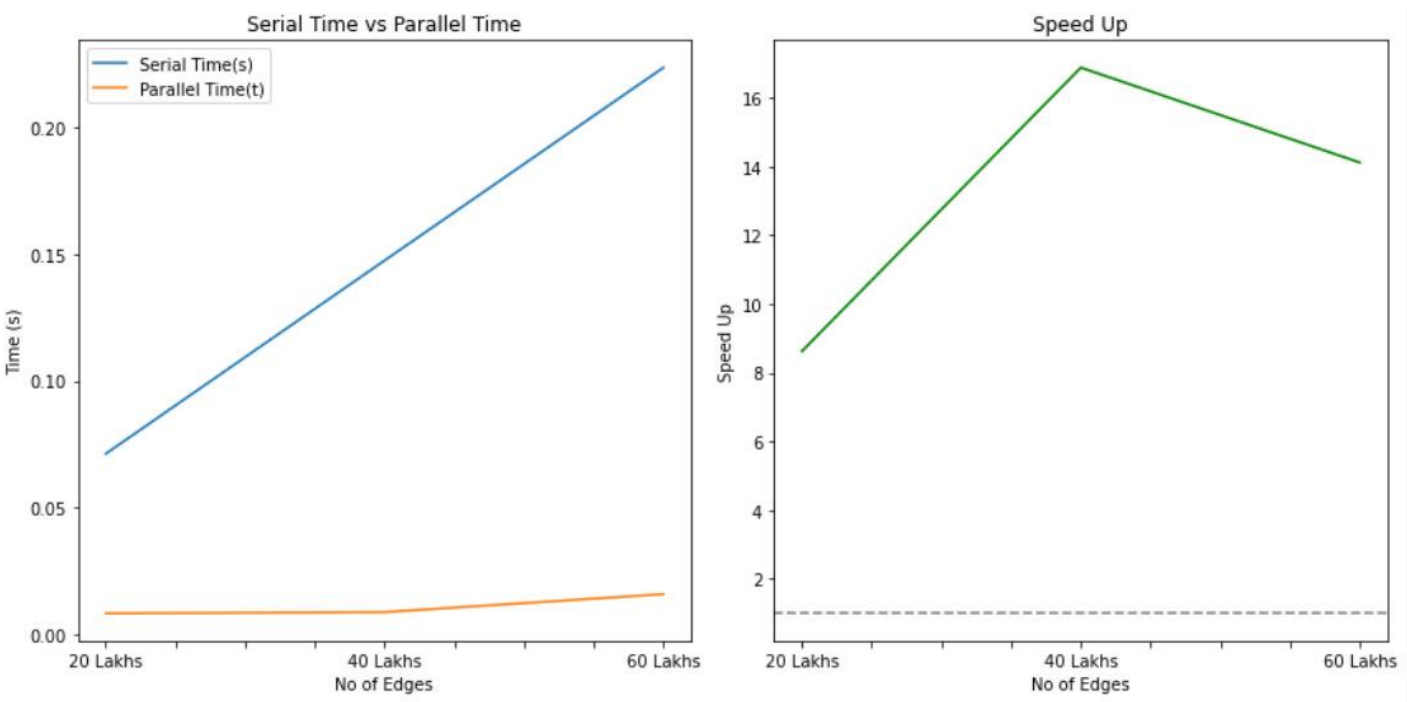
No. of Edges	No. of Processes	BFS exe. time	TEPS
	2	0.072621	137686.564
60 Lakhs	4	0.027051	369629.000
	8	0.015847	630985.501





Speedup:

No of Edges	Serial Time(s)	Parallel Time(t)	Speed up(s/t)
20 Lakhs	0.071268	0.008254	8.63
40 Lakhs	0.147605	0.008744	16.88
60 Lakhs	0.223798	0.015847	14.12



Graph500 Benchmark :

Scale constant = 20

Edgefactor constant = 16

Processes	BFS_mean_time	Median_TEPS
2	0.275551	6.09553e+07
4	0.129514	1.05311e+08
8	0.0865148	1.93907e+08

Chapter 5

Conclusion

5.1 Conclusion:

1. **Existing Open-Source Implementations**: The project thoroughly examined existing open-source implementations of parallel BFS algorithms tailored for Multi-Core Systems. These implementations serve as the foundation for large-scale graph processing.
2. **Performance Analysis**: An essential aspect of the research involved analyzing the performance of these BFS algorithms on a Message Passing Cluster. This investigation aimed to assess how well these algorithms scale and utilize CPU resources for large-scale graph processing tasks.
3. **Proposed Improvements**: The project identified areas in which existing parallel BFS algorithms can be enhanced in terms of performance and scalability. These improvements were devised to optimize memory access patterns, load balancing, and resource utilization.
4. **Benchmarking**: Extensive benchmarking tests were conducted on various computing systems, including PARAM series supercomputers. These tests provided valuable insights into the speed and efficiency of graph traversal using the Graph500 benchmark.

5.2 Future Scope:

While this project has achieved significant milestones in parallel graph algorithms and BFS implementations, there remains sample for future exploration and advancement in this domain:

1. **Advanced GPU Utilization**: Future work can delve deeper into GPU acceleration techniques, exploring more efficient ways to leverage the computational power of GPUs in BFS algorithms.
2. **Scalability**: As data sets continue to grow in size and complexity, further research can focus on optimizing the scalability of BFS algorithms to handle even larger graphs on distributed computing environments.
3. **Hybrid Models**: Combining BFS with other graph algorithms, such as graph coloring, can lead to innovative hybrid approaches that offer improved performance and efficiency.

4. **Real-World Applications**: Applying optimized BFS algorithms to real-world applications, such as social network analysis or genomics, can demonstrate the practical significance of these research efforts.

Chapter 7

References

1) ToyotaroSuzumura Tokyo Institute of Technology IBM Research – Tokyo / JST CRESTHighly. Scalable Graph Search for the Graph500 Benchmark.

2) Author – AdityaNongmeikapam Course – 633 Parallel Algorithms Instructor – Russ Miller, PARALLEL BREADTH-FIRST SEARCH USING MPI

3) HuiweiLv · Guangming Tan · Mingyu Chen · Ninghui Sun, Understanding Parallelism in Graph Traversal on Multi-core Clusters