Department of Computer Science & Engineering

Indian Institute of Technology Delhi

Implementation of Google Pregel in C++ using OpenMPI

COL733-Cloud Computing

November 17, 2023

Students:	Entry No.:
Rishi Jain	2020CS10373
Srijan Gupta	2020CS50444
E.Spandana	2020EE10489

Contents

1 Introduction To Pregel

Google Pregel is a parallel graph processing framework developed by Google for large-scale graph computation. Analyzing and processing large-scale graphs efficiently is a challenging task that requires distributed computing solutions.

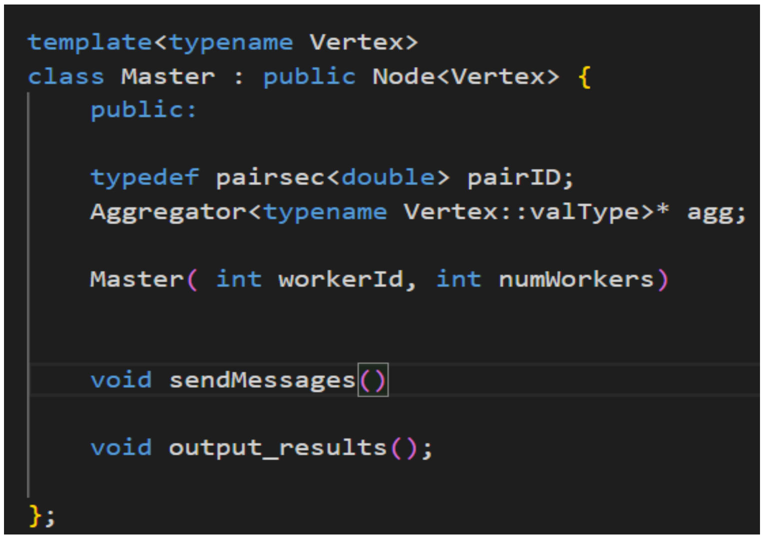
Pregel is designed to address the computational complexities associated with massive graph datasets by providing a scalable and fault-tolerant platform for processing graphs in parallel across multiple computing nodes. Inspired by the Bulk Synchronous Parallel (BSP) model, Pregel simplifies the development of distributed graph algorithms by exposing a programming model based on vertex-centric computation.

2 Implementation Of Pregel

The program initializes the MPI environment and creates a distributed graph.

2.1 Master.hpp

The Master class functions as a central coordinating entity in a distributed graph processing system, adhering to the Google Pregel programming model. Templated on the vertex type (Vertex), it extends the Node class and oversees the execution of the graph algorithm across multiple computing nodes or workers. The sendMessages method in the Master class is implemented to conform to the structure expected by the distributed graph processing framework. However, it essentially remains a no-op for the master node, as it doesn't participate in the exchange of messages. Instead, its primary function is to organize and orchestrate the communication patterns among the worker nodes. During each superstep of the algorithm, the master node ensures the proper distribution of messages across the worker nodes using MPI's Alltoall operation, adhering to the communication requirements of the Pregel model.

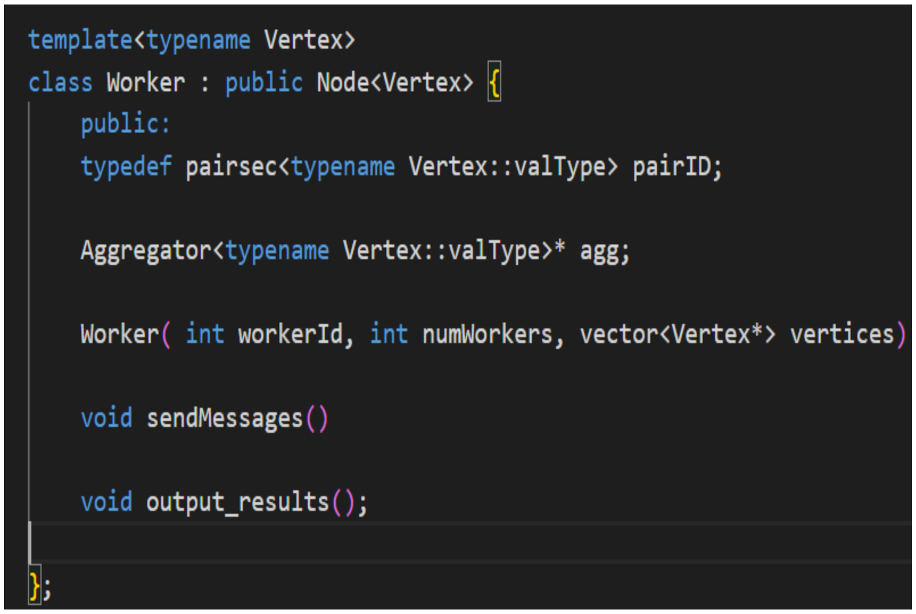


```
template<typename Vertex>
class Master : public Node<Vertex> {
    public:

    typedef pairsec<double> pairID;
    Aggregator<typename Vertex::valType>* agg;

    Master( int workerId, int numWorkers)


    void sendMessages()

    void output_results();

};
```

Figure 1: Master.hpp

2.2 Worker.hpp

The Worker class serves as a fundamental building block in a distributed graph processing framework, designed to execute graph algorithms efficiently across multiple computing nodes. Templated on the vertex type (Vertex), it encapsulates the logic for parallelized computation and communication within the Google Pregel model. Instances of this class operate as individual processing units responsible for a subset of the graph, coordinating with other workers to exchange messages during each iteration (superstep) of the algorithm. The worker manages its local vertices, updates their states, and facilitates communication through the exchange of messages with other workers via MPI collective communication routines. The class is equipped with an aggregator (agg) to handle aggregation tasks across workers. With a flexible template structure, the Worker class can be specialized for different graph structures and algorithms, providing a scalable and adaptable foundation for distributed graph processing applications.



```
template<typename Vertex>
class Worker : public Node<Vertex> {
    public:
    typedef pairsec<typename Vertex::valType> pairID;

    Aggregator<typename Vertex::valType>* agg;

    Worker( int workerId, int numWorkers, vector<Vertex*> vertices)

    void sendMessages()

    void output_results();

};
```

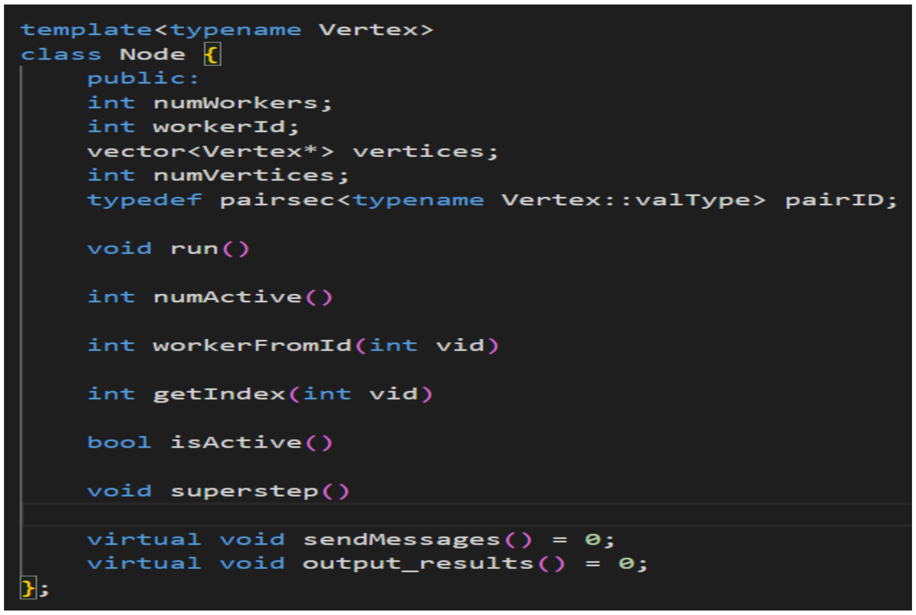
Figure 2: Worker.hpp

2.3 Node.hpp

The Node class serves as an abstract template for the nodes in a distributed graph processing framework, providing a basis for implementing both master and worker nodes. It encapsulates common functionalities and attributes essential for coordinating and executing graph algorithms across multiple workers. The class contains an integer variable numWorkers indicating the total number of worker nodes and an integer workerId representing the unique identifier of the current node. A vector vertices holds pointers to the vertices associated with the node, and a typedef pairID is defined for convenience. The run method orchestrates the execution of the graph algorithm, iterating through supersteps until all vertices become inactive.

The numActive method calculates the total number of active vertices across all nodes using MPI collective communication. The workerFromId method determines the worker node responsible for a given vertex based on its identifier, and getIndex returns the index of the vertex within the node's vertices vector.

The isActive method checks whether there are active vertices within the node, and the superstep method triggers the update logic for each vertex in a superstep. The sendMessages and output_results methods are declared as pure virtual functions, intended to be implemented by derived classes according to their specific role (master or worker) in the distributed graph processing system. The Node class thus establishes a common structure for coordination, iteration, and communication in distributed graph algorithms.

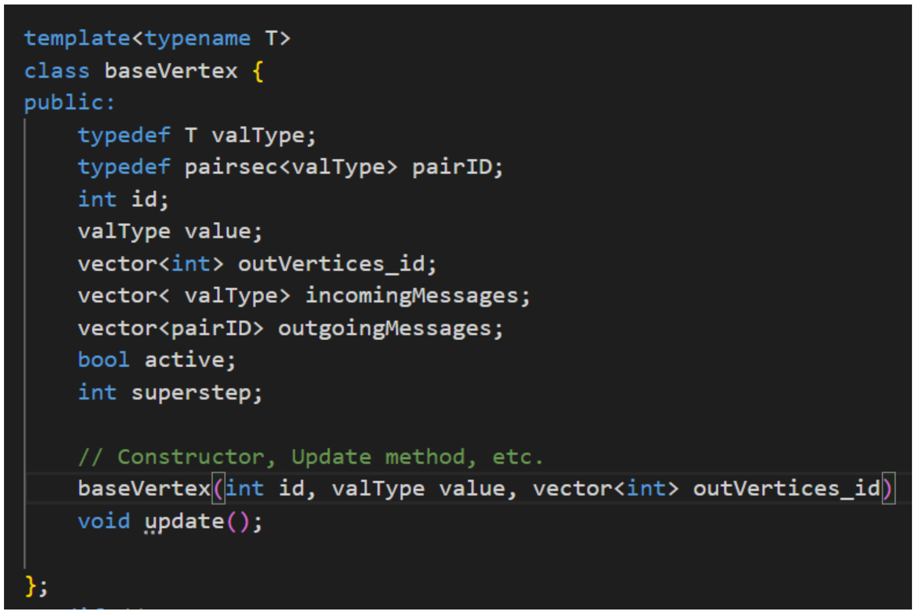


```
template<typename Vertex>
class Node {
    public:
    int numWorkers;
    int workerId;
    vector<Vertex*> vertices;
    int numVertices;
    typedef pairsec<typename Vertex::valType> pairID;

    void run()

    int numActive()

    int workerFromId(int vid)

    int getIndex(int vid)

    bool isActive()

    void superstep()

    virtual void sendMessages() = 0;
    virtual void output_results() = 0;
};
```

Figure 3: Node.hpp

2.4 Vertex.hpp

Each vertex is uniquely identified by an integer id and holds a templated value associated with the vertex. The outVertices_id vector maintains the identifiers of vertices connected to the current vertex. Incoming messages received by the vertex are stored in the incomingMessages vector, while outgoing messages to be sent are organized as pairs of int and the templated valType in the outgoingMessages vector. The boolean attribute active signifies the activity status of the vertex, and the integer superstep denotes the current step or iteration in a graph computation. The vertex class establishes a standardized representation, allowing algorithm-specific logic to be seamlessly incorporated into the graph computation framework.



```
template<typename T>
class baseVertex {
public:
    typedef T valType;
    typedef pairsec<valType> pairID;
    int id;
    valType value;
    vector<int> outVertices_id;
    vector< valType> incomingMessages;
    vector<pairID> outgoingMessages;
    bool active;
    int superstep;

    // Constructor, Update method, etc.
    baseVertex(int id, valType value, vector<int> outVertices_id)
    void update();

};
```

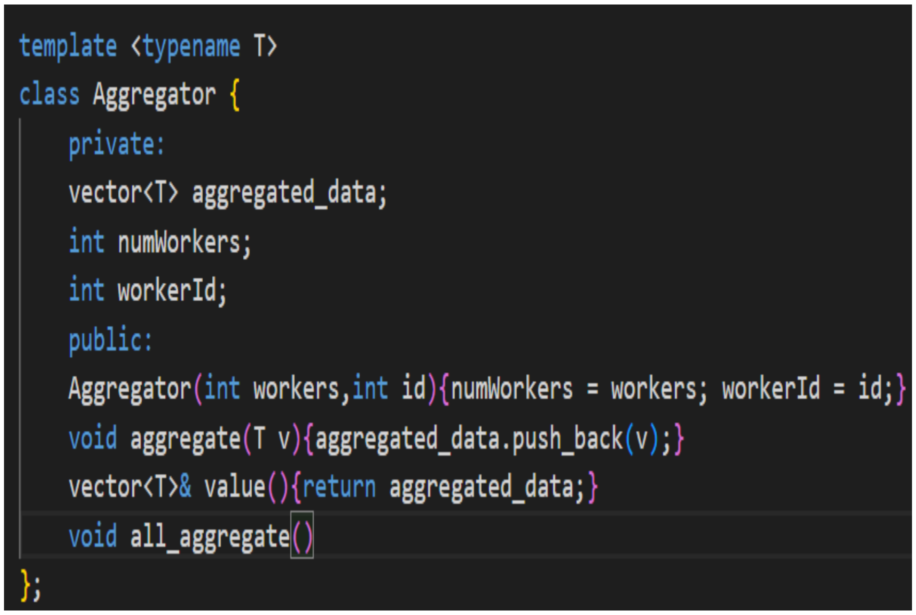
Figure 4: vertex.hpp

2.5 MPIType.hpp

This contains a function that takes a template parameter T representing a data type and returns the corresponding MPI datatype using the MPI library. The function employs a series of type checks using std::is_same to determine the type of T and selects the appropriate MPI datatype accordingly. The supported types include fundamental data types (int, float, double, char, etc.), as well as more complex types such as std::complex, std::pair, and std::string. If the provided type T does not match any of the supported types, the function returns MPI_DATATYPE_NULL. This template function is designed to facilitate the dynamic determination of MPI datatypes for various C++ data types, aiding in the development of MPI-based parallel and distributed applications.

2.6 Aggregator.hpp

Instantiated with parameters indicating the total number of workers (numWorkers) and the identifier of the current worker (workerId), the class employs a vector (aggregated_data) to consolidate aggregated values of a generic type T. The aggregate method allows worker nodes to contribute local results to the global aggregation, while the all_aggregate method orchestrates the exchange of aggregated data between worker nodes through MPI communication operations. Employing MPI's Alltoall operation, the class ensures that each worker node shares its aggregated data with others, receiving contributions from them in return. The process concludes with the accumulation of unique values received from other nodes into the local aggregated data, effectively avoiding duplications. In essence, the Aggregator class facilitates the cohesive aggregation of distributed results, fostering collaboration and information sharing among worker nodes in executing graph algorithms.



```
template <typename T>
class Aggregator {
    private:
    vector<T> aggregated_data;
    int numWorkers;
    int workerId;
    public:
    Aggregator(int workers,int id){numWorkers = workers; workerId = id;}
    void aggregate(T v){aggregated_data.push_back(v);}
    vector<T>& value(){return aggregated_data;}
    void all_aggregate()
};
```

Figure 5: vertex.hpp

3 Implementation Example (BFS)

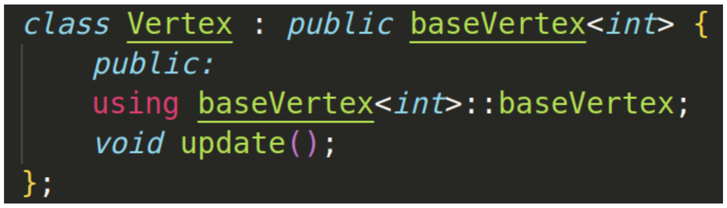
We have implemented the BFS algorithm on our implementation of pregel. The algorithm returns the distance of each vertex from the 0^{th} vertex. Here, each vertex has a value of type *int*, which is set to -1 by default.

At each superstep, if the vertex has not been visited, i.e., its value is still -1, and it has received a message, then it sets its value to the current superstep number (which will be the same as the distance from vertex 0) and sends a message to each of its out-neighbours. To start the algorithm, at the 0^{th} superstep, vertex 0 sends a message to itself. The rest of the 0^{th} superstep precedes, as mentioned earlier.

Should a vertex not receive any messages or be already visited, it will set its active attribute to false - which is the same as voting to halt. As per pregel standards, the algorithm terminates when all vertices vote to halt in the same superstep.

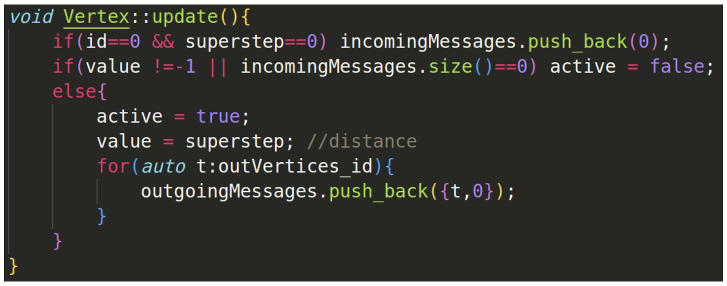
4 Implementation Instructions

A class which inherits from the `BaseVertex<T>` class needs to be defined, which will act as the vertex class. In this class, the update method which will be executed every superstep will need to be defined.



```
class Vertex : public baseVertex<int> {
    public:
    using baseVertex<int>::baseVertex;
    void update();
};
```

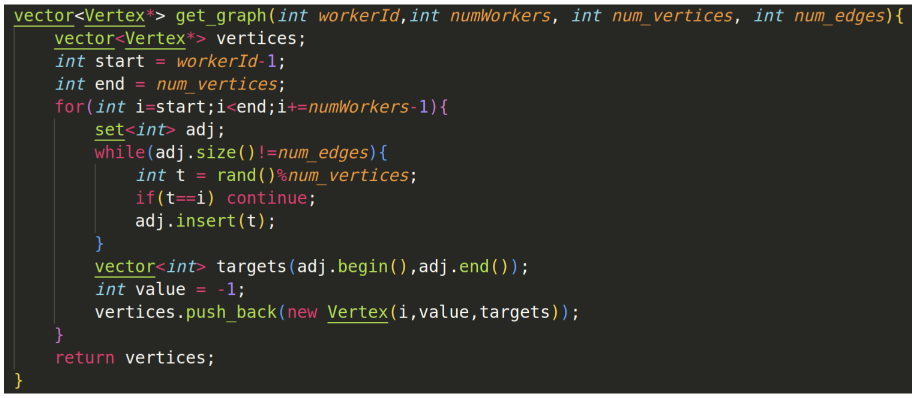
Figure 6: Vertex class for BFS



```
void Vertex::update(){
    if(id==0 && superstep==0) incomingMessages.push_back(0);
    if(value !=-1 || incomingMessages.size()==0) active = false;
    else{
        active = true;
        value = superstep; //distance
        for(auto t:outVertices_id){
            outgoingMessages.push_back({t,0});
        }
    }
}
```

Figure 7: Update method for BFS

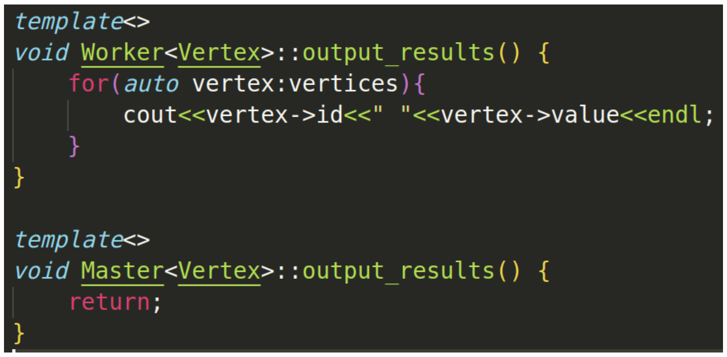
A `get_graph` function, which returns a vector of vertices, will need to be defined in which the id, initial value, and edge set will be set. Each worker receives vertices whose id is the modulo of the number of workers (i.e., total processes minus one, as there is a master as well).



```
vector<Vertex*> get_graph(int workerId,int numWorkers, int num_vertices, int num_edges){
    vector<Vertex*> vertices;
    int start = workerId-1;
    int end = num_vertices;
    for(int i=start;i<end;i+=numWorkers-1){
        set<int> adj;
        while(adj.size()!=num_edges){
            int t = rand()%num_vertices;
            if(t==i) continue;
            adj.insert(t);
        }
        vector<int> targets(adj.begin(),adj.end());
        int value = -1;
        vertices.push_back(new Vertex(i,value,targets));
    }
    return vertices;
}
```

Figure 8: get_graph for BFS

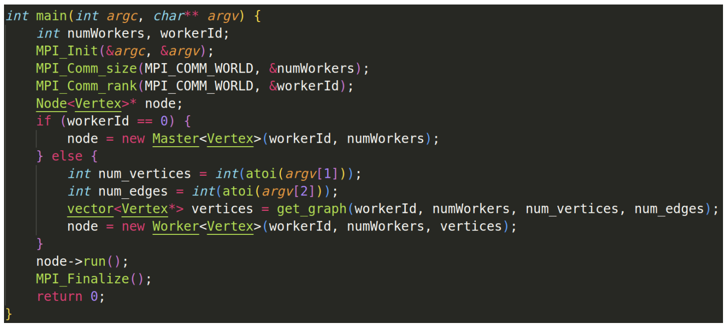
Functions for both the master and worker to output results need to be defined. In case no output is needed - can be defined to be empty as well.



```
template<>
void Worker<Vertex>::output_results() {
    for(auto vertex:vertices){
        cout<<vertex->id<<" "<<vertex->value<<endl;
    }
}

template<>
void Master<Vertex>::output_results() {
    return;
}
```

Figure 9: output_results for BFS

The `main` function does not need to be changed from the one given in BFS unless a user explicitly wishes to - which may lead to unforeseen issues so one must be careful. We advise sticking to the template `main` function provided in the BFS and max examples.



```
int main(int argc, char** argv) {
    int numWorkers, workerId;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numWorkers);
    MPI_Comm_rank(MPI_COMM_WORLD, &workerId);
    Node<Vertex>* node;
    if (workerId == 0) {
        node = new Master<Vertex>(workerId, numWorkers);
    } else {
        int num_vertices = int(atoi(argv[1]));
        int num_edges = int(atoi(argv[2]));
        vector<Vertex*> vertices = get_graph(workerId, numWorkers, num_vertices, num_edges);
        node = new Worker<Vertex>(workerId, numWorkers, vertices);
    }
    node->run();
    MPI_Finalize();
    return 0;
}
```

Figure 10: main for BFS

5 Results

We have compared our code with two other implementations of BFS - one using OpenMPI without any supersteps or classes and the other using parallel BGL. Testing has been done on IIT Delhi's HPC platform.

Due to certain library version mismatches, we were unable to run the parallel BGL implementation on HPC, and the size of the data used to check proper scalability is too large to run on our local machines. Therefore, we have compared only our custom implementation and the pregel implementation.

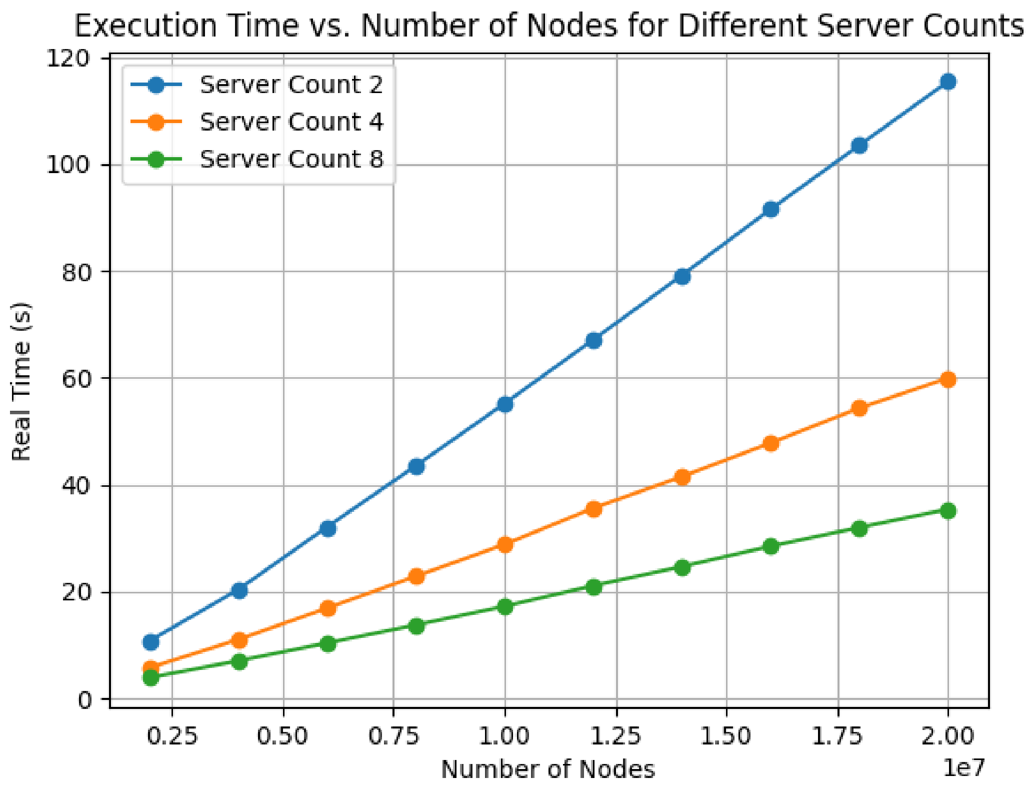


Figure 11: Runtime for Custom BFS

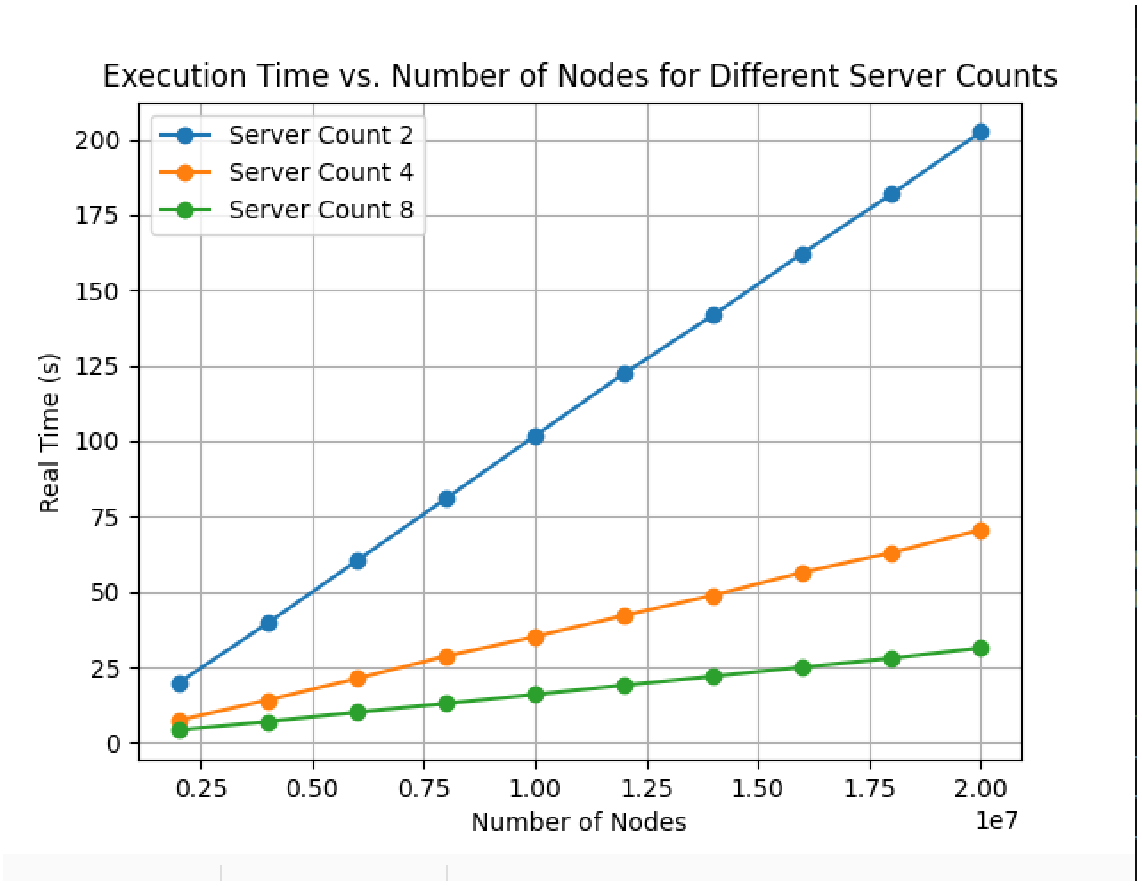


Figure 12: Runtime for Pregel BFS

6 Observations

It is clear that both the custom BFS and pregel BFS scale linearly as the number of nodes increases - which shows the scalability of both approaches.

In both cases, we see a sharp drop in runtime when the number of workers goes from 2 to 4, with a smaller drop when going from 4 to 8. This is because communication overheads become more significant as the number of workers increases.

We notice that pregel performs more overall communication compared to our custom implementation, which is the likely reason behind the higher runtime of pregel BFS. However, the pregel code is modular and can be easily modified to run other graph algorithms.

7 Future Work

At the moment, the master has no role and is just an extra process that is doing nothing. As per the paper on pregel, the main role of the master is to assist in fault tolerance.

Despite our best efforts, we could not make our implementation fault-tolerant. The major issue we faced was when workers crashed with inflight messages, which resulted in other workers stalling due to the blocking nature of MPI functions.

8 Code

https://github.com/rishijain55/733-project