# COL819 A2 Report

Shashwat Shivam (2016CS10328)
April 12, 2020

## 1 Introduction to GHS

Gallager, Humblet and Spira Algorithm (or GHS Algorithm for short) is used to find the Minimum Spanning Tree of a graph in a Distributed setting. The graph is assumed to be connected, undirected and having unique edge weights (and therefor unique MST).

Each vertex of the graph is simulated as a single node in a distributed system. Each node in the system runs the same copy of the algorithm. The algorithm works by combining vertices into fragments using confirmed edges in the MST. The nodes communicate by using messages.

## 2 Implementation Details

I have implemented the algorithm using Golang. Golang provides easy parallel computing as it is built especially for this purpose. This can be used easily to simulate distributed systems with very efficient code.

We start with a single thread which does the following :-

1. Reads the graph input file and creates graph object
2. Creates message channels for each node and one for result accumulation
3. Creates new threads for each node while providing it with required parameters which include neighbours, edge weights, neighbour channels and self channel.
4. Accumulates results, prints results and terminates other threads.

### 2.1 Reading Input

Graph data structure is defined using Adjacency List as follows :-

```
type graph struct {
    numNodes  int
    edges     map[int][]int
    distances map[int]map[int]int
}
```

I read the input and store it in a graph object using the createGraph function.

### 2.2 Dedicated message channels

Next i create dedicated message receiving channels for each node of the graph. Also i create a receiving channel for results read by the main thread. This can be very easily achieved in Golang. The channels receive message type objects which are described as follows :-

```
type msgStruct struct {
    msgType, state, level, sender, weight, fragmentName int
}
```

### 2.3 Spinning off threads

Next i create a node object every vertices in the graph. These defined as follows :-

```
type node struct {
    selfID, state, level, rec, parentInd,
    bestWt, bestNodeInd, testNodeInd, name int
    selfChannel                    chan msgStruct
    mainChannel                    chan edge
    neighbours, distances, status  []int
    channels                       []chan msgStruct
}
```

These node objects contain all the initialization data to work as a node in distributed right after starting. These include node number (selfID) , the initial state,level,rec,parentInd,bestWt, bestNode,testNode,name variables and channels, distances and ids of the neighbours as well as channel of result thread and self.

These node objects have an associated function with them to run the node defined as :-

```
func (n *node) runNode() {
    ...
}
```

Using these function and the go syntax the threads can be launched easily using the following command :-

```
go n.runNode()
```

### 2.4 RESULTS

After launching threads, each threads runs code defined in the GHS algo pseudo code which is straightforward. In addition to this the result node keeps listening to any results that any of the distributed system nodes might send. After finding any MST branch, one of the nodes of the branch sends result to main thread.

Once the main thread receives the number of branches required to form the MST, it sorts the branches in increasing order of their edge weights, prints results and terminates all other threads by simply exiting.

```
for i := 0; i < expectedMsgs; i++ {
        e := <-mainChannel
        finalEdges = append(finalEdges, e)
}

sort.Sort(byLength(finalEdges))

for i := 0; i < expectedMsgs; i++ {
        e := finalEdges[i]
        finalEdges = append(finalEdges, e)
        fmt.Printf("(%d, %d, %d)\n", e.v1, e.v2, e.e)
}
```

## 3 RUNTIME ANALYSIS

### 3.1 MESSAGE ANALYSIS

The graph for number of messages vs. number of nodes (input graph is fully connected) is shown in Fig:3.1. We can see that the trend is roughly quadratic which is consistent to the expected number of messages 2E + 5Nlog(N) (Note:- E is O($N^2$)).

### 3.2 TIME ANALYSIS

The graph for run time vs. number of nodes (input graph is fully connected) for varying number of threads on system is shown in Fig:3.2. The important thing to be noted here is that we don't get a two fold speedup on doubling the number of cores because of the overhead involved in message passing between all the node threads. Still the speedup increases significantly as the number of cores reaches the number of nodes in the network as context switching is reduced by a large number.

## 4 INSTRUCTIONS TO RUN CODE

It is dead simple to compile and run the code if Golang is setup on the system. Just go inside the code directory and run the command '**go build src/main.go**'. This will create an executable

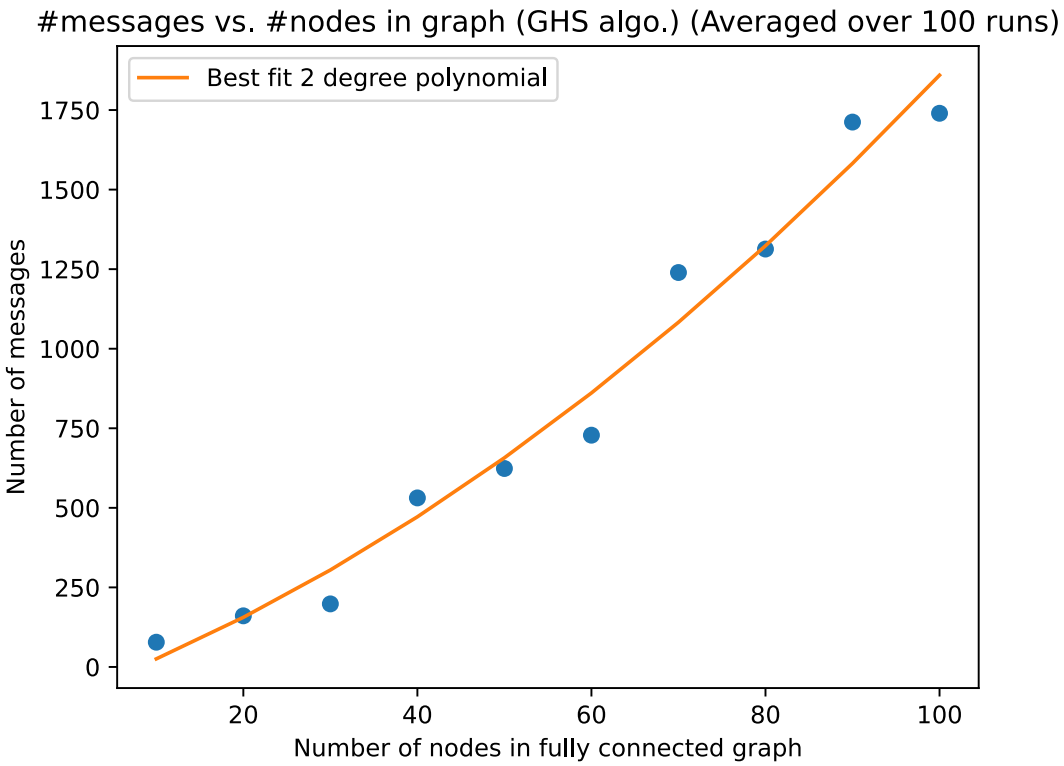


Figure 3.1: Number of messages passed vs. number of nodes in fully connected graph for GHS algorithm (Averaged over 100 runs)

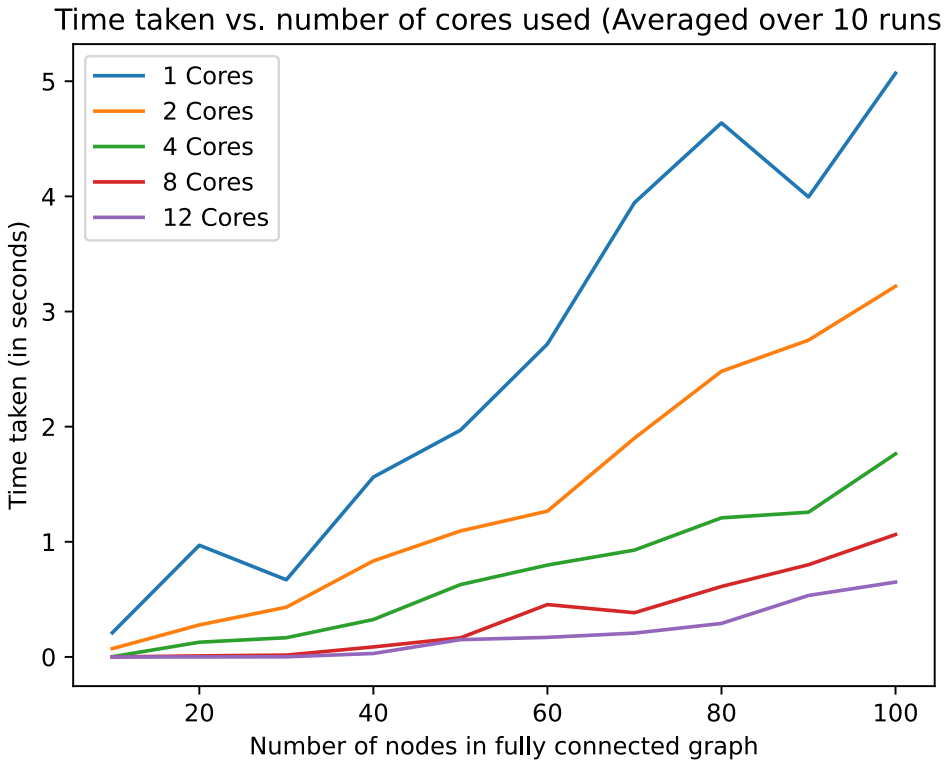


Figure 3.2: Time taken to find MST vs. number of cores used (Averaged over 10 runs)

in the current directory. For convenience a pre-compiled version is already included in the submission.

To run the executable run the command '**./main <INPUT FILE PATH>**' and the results of the computation will be printed on the screen.