Distributed Computing

Implementing Vector Clocks

Shagun

15.02.2024

cs23mtech14013

INTRODUCTION

Vector Clocks

Vector clocks, a fundamental concept in distributed systems, play a crucial role in tracking the causal relationships between events occurring across different nodes without relying on a centralized clock.

At its core, a vector clock is a data structure used to capture the partial ordering of events in a distributed system. Unlike traditional clock mechanisms that rely on a global time reference, vector clocks offer a decentralized approach by maintaining a vector of timestamps, typically associated with each process or node in the system.

Each process maintains its own vector clock, which consists of an entry for every process in the system. When an event occurs at a process, its corresponding entry in the vector clock is incremented. This incrementation ensures that each process maintains a consistent record of the events it has experienced.

If two events x and y have timestamps vh and vk, respectively, then

$x \rightarrow y \Leftrightarrow vh < vk$

$x \parallel y \Leftrightarrow vh \parallel vk$

However, what makes vector clocks particularly powerful is their ability to facilitate the comparison of event ordering between different processes. By comparing the vector clocks associated with two events, it's possible to determine whether one event causally precedes, follows, or is concurrent with another. This enables distributed systems to establish a partial ordering of events, which is crucial for various tasks such as synchronization, consistency management, and conflict resolution.

Vector clocks are especially useful in scenarios where events can occur concurrently across multiple processes, making it challenging to establish a definitive global ordering. By leveraging the inherent causal relationships embedded within vector clocks, distributed systems can make informed decisions regarding the consistency and coherence of shared data, ultimately enhancing the reliability and scalability of the overall system.

Singhal-Kshemkalyani's optimization of vector clocks

It represents a crucial refinement aimed at addressing these scalability challenges. This optimization, introduced by Mukesh Singhal and Ajay D. Kshemkalyani, seeks to reduce the storage and communication overhead inherent in traditional vector clock approaches while preserving the essential causal ordering information.

At its core, Singhal-Kshemkalyani's optimization leverages the observation that events in many distributed systems often occur locally within a subset of processes rather than globally across all processes simultaneously. By exploiting this locality property, the optimization strategically reduces the size of vector clocks while maintaining their ability to accurately track causal relationships.

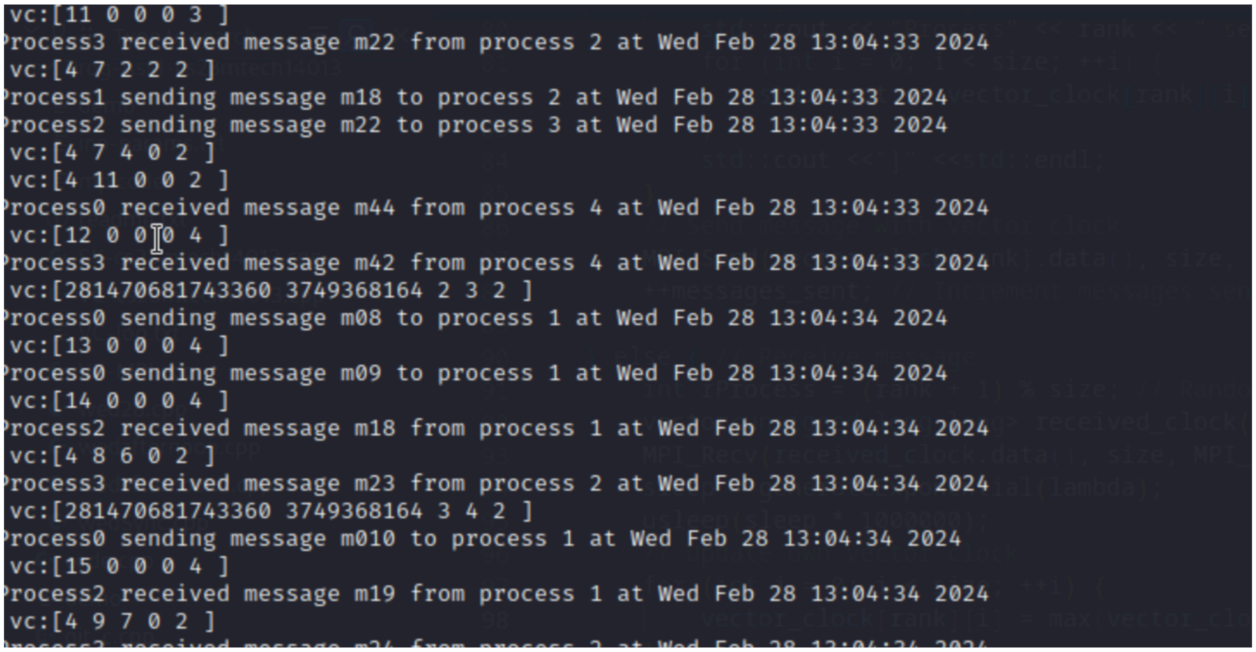
One of the primary techniques employed in this optimization is the adoption of a "sparse" vector clock representation. Unlike traditional dense representations where each process maintains a vector with entries for every process in the system, the sparse representation allows processes to selectively include only those entries relevant to their local interactions. This selective inclusion drastically reduces the size of vector clocks, making them more scalable, especially in large-scale distributed systems.

Furthermore, Singhal-Kshemkalyani's optimization introduces mechanisms for dynamically adjusting the size of vector clocks based on the communication patterns and workload characteristics of the system. Processes can dynamically add or remove entries from their vector clocks as communication patterns evolve, ensuring that vector clock sizes remain proportional to the actual concurrency observed in the system.Additionally, the optimization incorporates local information caching and aggregation techniques to minimize the frequency of vector clock updates and exchanges between processes. By aggregating local causality information and disseminating it efficiently, processes can reduce the overhead associated with maintaining vector clocks while still preserving the necessary causal ordering information.

The goal of this assignment is to implement Vector-clocks and Singhal-Kshemkalyani optimization on a Distributed System. Implement both these algorithms for vector clocks in C++ using sockets.Then to compare the overheads incurred with messages stored and exchanged.

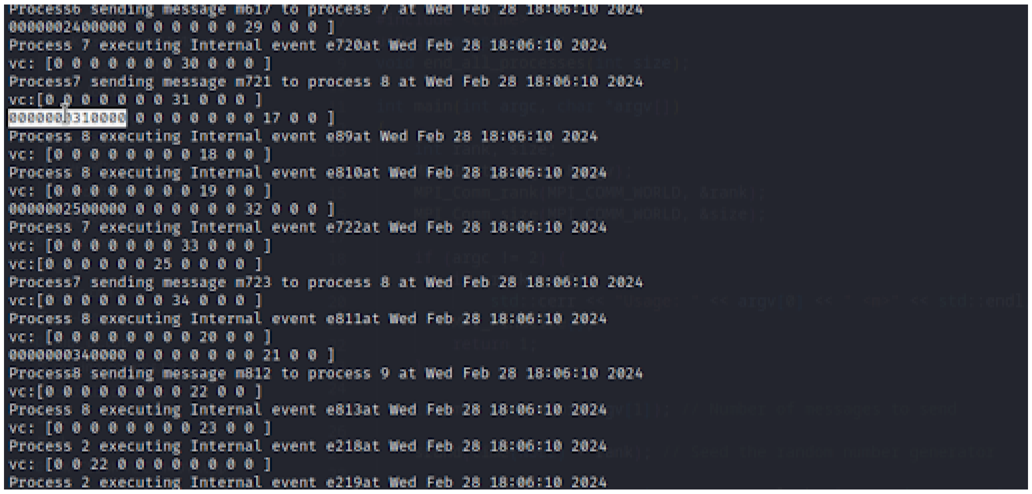
COMPLICATIONS THAT AROSE DURING COURSE OF PROGRAMMING

During implementation of vector clocks using Message Passing Interface (MPI) facing the Challenge : Sudden hike in the value of vector clock of the process



```
 vc:[11 0 0 0 3 ]
rocess3 received message m22 from process 2 at Wed Feb 28 13:04:33 2024
 vc:[4 7 2 2 2 ]
rocess1 sending message m18 to process 2 at Wed Feb 28 13:04:33 2024
rocess2 sending message m22 to process 3 at Wed Feb 28 13:04:33 2024
 vc:[4 7 4 0 2 ]
 vc:[4 11 0 0 2 ]
rocess0 received message m44 from process 4 at Wed Feb 28 13:04:33 2024
 vc:[12 0 0 0 4 ]
rocess3 received message m42 from process 4 at Wed Feb 28 13:04:33 2024
 vc:[281470681743360 3749368164 2 3 2 ]
rocess0 sending message m08 to process 1 at Wed Feb 28 13:04:34 2024
 vc:[13 0 0 0 4 ]
rocess0 sending message m09 to process 1 at Wed Feb 28 13:04:34 2024
 vc:[14 0 0 0 4 ]
rocess2 received message m18 from process 1 at Wed Feb 28 13:04:34 2024
 vc:[4 8 6 0 2 ]
rocess3 received message m23 from process 2 at Wed Feb 28 13:04:34 2024
 vc:[281470681743360 3749368164 3 4 2 ]
rocess0 sending message m010 to process 1 at Wed Feb 28 13:04:34 2024
 vc:[15 0 0 0 4 ]
rocess2 received message m19 from process 1 at Wed Feb 28 13:04:34 2024
 vc:[4 9 7 0 2 ]
rocess3 received message m24 from process 2 at Wed Feb 28 13:04:34 2024
```

Further I paid attention to what was being sent by the sender process and realized that it was padding extra zeros on its own in the vector clock value.



```
Process6 sending message m617 to process 7 at Wed Feb 28 18:06:10 2024
000000240000 0 0 0 0 0 0 29 0 0 0 ]
Process 7 executing Internal event e720at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 30 0 0 0 ]
Process7 sending message m721 to process 8 at Wed Feb 28 18:06:10 2024
vc:[0 0 0 0 0 0 0 31 0 0 0 ]
000000310000 0 0 0 0 0 0 0 0 17 0 0 ]
Process 8 executing Internal event e89at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 0 18 0 0 ]
Process 8 executing Internal event e810at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 0 19 0 0 ]
000000250000 0 0 0 0 0 0 32 0 0 0 ]
Process 7 executing Internal event e722at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 33 0 0 0 ]
vc:[0 0 0 0 0 0 25 0 0 0 0 ]
Process7 sending message m723 to process 8 at Wed Feb 28 18:06:10 2024
vc:[0 0 0 0 0 0 0 34 0 0 0 ]
Process 8 executing Internal event e811at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 0 20 0 0 ]
000000340000 0 0 0 0 0 0 0 0 21 0 0 ]
Process8 sending message m812 to process 9 at Wed Feb 28 18:06:10 2024
vc:[0 0 0 0 0 0 0 0 22 0 0 ]
Process 8 executing Internal event e813at Wed Feb 28 18:06:10 2024
vc: [0 0 0 0 0 0 0 0 23 0 0 ]
Process 2 executing Internal event e218at Wed Feb 28 18:06:10 2024
vc: [0 0 22 0 0 0 0 0 0 0 0 ]
Process 2 executing Internal event e219at Wed Feb 28 18:06:10 2024
```

But the reason of this is still unknown to me .

Example :

Pseudocode

```
Initialize MPI environment
Get rank and size of MPI_COMM_WORLD

if (argument_count != 2) {
        if (rank == 0)
        Print "Usage: <program_name> <m>"
        Finalize MPI environment
        Exit program
}

m = Convert argument 1 to integer
Seed random number generator with current time + rank

Initialize vector clock for each process with all zeroes

messages_sent = 0

while (messages_sent < m) {
        action = Randomly choose action (0 for execute, 1 for send, 2 for receive)
        currentTime = Get current system time
        Convert currentTime to string representation

        if (action == 0) {
        Increment own vector clock
        Print "Process <rank> executing Internal event e<rank><messages_sent> at
<currentTimeStr> vc: [<vector_clock>]"
        Increment messages_sent
        } else if (action == 1) {
        sProcess = (rank + 1) % size // Randomly choose process to send message to
        Increment own vector clock
        Print "Process <rank> sending message m<rank><messages_sent> to process
<sProcess> at <currentTimeStr> vc: [<vector_clock>]"
        Send vector clock to sProcess using MPI_Send
        Increment messages_sent
        } else {
        Receive message from any source
        Update own vector clock based on received vector clock
        Increment own vector clock
        Print "Process <rank> received message m<source><messages_sent> from
process <source> at <currentTimeStr> vc: [<vector_clock>]"
        }
}

Print "Process <rank> has sent <m> messages. Terminating."
```

```
Send termination signal to all other processes using MPI_Send
Finalize MPI environment
```

So I used sockets . I have attached MPI code whatever I could do in the assignment zip file.

PROGRAM DESIGN

Vector Clocks

Design

The program is designed to simulate a distributed system where multiple processes communicate with each other using vector clocks over TCP/IP sockets. Here's an explanation of the design and the flow of the code:

1. Global Variables:
   - `n`, `lambda`, `m`, `alpha`: These variables store parameters read from the input file, defining the number of processes, lambda for exponential distribution, the number of messages per process, and alpha for the proportion of internal events per process.
   - `graph`: A 2D vector representing the graph topology of the processes.
   - `fd`: A file pointer for writing log messages to a file.
   - `mtx`: A mutex used for synchronization.
   - `msgs_rcv`: An integer variable to keep track of the number of messages received.

2. User-defined Functions:

   - `generateExponential(float lamda)`: This function generates a random number from an exponential distribution with the given lambda.

   - `updateVectorClock(vector<int> &vc, int event, int pid, const vector<int> &vc_recv)`: This function updates the vector clock of a process based on the event type (internal event, message send event, or receive event). It ensures that the vector clock is correctly updated according to the happened-before relationship.

   - `displayVectorClock(vector<int> &vc)`: This function displays the vector clock of a process to the log file.

   - `sendMessage(int sock, int pid, vector<int> &msg)`: This function sends a message to another process identified by `pid` using the provided socket `sock`. The message includes the vector clock of the sender and other necessary information.

   - `simulateEvents(int sock, int pid, vector<int> &vc)`: This function simulates events in a process. It generates internal events and message send events according to the parameters read from the input file. For each event, it updates the vector clock, displays the event details, and sends messages to other processes.

- `receiveMessages(int sock, int pid, vector<int> &vc)`: This function receives messages from other processes. It accepts incoming connections, receives messages, updates the vector clock based on the received vector clock, and displays the received message details.

- `startProcess(int id)`: This function initializes a process identified by `id`. It creates a socket for the process, binds it to a unique port, listens for incoming connections, and creates threads for event simulation and message reception.

3. Main Function:
   - The `main` function is the entry point of the program.
   - It opens the input file and log file, reads input parameters and graph topology, and displays them on the terminal.
   - It creates threads for each process using the `startProcess` function.
   - After joining all threads, it closes the files, displays the average message size sent, and returns 0.

4. Flow of the Code:
   - The program reads input parameters and graph topology from the input file.
   - For each process, it creates a socket, binds it to a unique port, and listens for incoming connections.
   - Threads are created for each process to simulate events and receive messages concurrently.
   - The event simulation thread generates internal events and message send events, updates the vector clock, and sends messages to other processes.
   - The message reception thread receives messages from other processes, updates the vector clock based on the received vector clock, and displays the received message details.
   - The program continues until all messages are received by each process.
   - Finally, it closes the files, displays the average message size sent, and exits.

5. Unique port explanation:
1. Each process in the distributed system is identified by a unique ID (`id`).
2. The program binds each process's socket to a port number determined by adding the process ID (`id`) to a predefined base port number (`PORT`). This results in each process having its own unique port number.
3. By listening on these unique port numbers, each process can accept incoming connections from other processes.
4. When a process wants to communicate with another process, it connects to the latter's unique port number.

For example, if the base port number (`PORT`) is 10000, and there are three processes with IDs 0, 1, and 2:

- Process 0 will bind its socket to port 10000.

- Process 1 will bind its socket to port 10001.
- Process 2 will bind its socket to port 10002.

This way, each process has its own unique port number, allowing them to communicate with each other without conflicts.

Pseudocode:

```
// Define global variables
int n, lambda, m
double alpha
vector<vector<int>> graph
FILE *fd
mutex mtx
int msgs_rcv = 0

// Define functions

// Function to generate a random number from an exponential distribution
double generateExponential(float lamda):
    // Use a random number generator to generate an exponential distribution
    return result

// Function to update the vector clock
void updateVectorClock(vector<int> &vc, int event, int pid, const vector<int> &vc_recv =
vector<int>()):
    // Update the vector clock based on the event type
    if event == 2:
    // Update vector clock for receive event
    // Compare received vector clock with current vector clock and update if necessary

// Function to display the vector clock
void displayVectorClock(vector<int> &vc):
    // Display the vector clock to a file

// Function to send a message to another process
void sendMessage(int sock, int pid, vector<int> &msg):
    // Create a socket connection to the target process
    // Send the message using the socket connection

// Function to simulate events in a process
void simulateEvents(int sock, int pid, vector<int> &vc):
        // Initialize event and message counters
        // Generate total number of events based on m and alpha
        // Loop until totalEvents is reached
        // Randomly choose between internal and message events
        // If internal event:
        // Update vector clock
        // Display internal event details
        // Simulate processing time
        // Else if message event:
        // Select a random process to send the message to
        // Update vector clock
        // Display message event details
        // Create message containing vector clock and other info
        // Send message to the selected process
        // Simulate processing time

// Function to receive messages in a process
void receiveMessages(int sock, int pid, vector<int> &vc):
        // Set socket timeout
        // Loop until all messages are received
        // Accept incoming connection
        // Receive message from the connection
        // Update vector clock
        // Display received message details

// Function to start a process
void startProcess(int id):
        // Create a socket for the process
        // Bind the socket to a unique port
        // Listen for incoming connections
        // Create threads for event simulation and message reception
        // Join the threads

// Main function
int main():
        // Open input file
        // Open log file
        // Read input parameters from file
        // Read graph topology from file
        // Display input parameters and graph topology
        // Create threads for each process
        // Join the threads
        // Close files
        // Display average message size sent
        // Return 0
```

OBSERVATIONS AND RESULTS

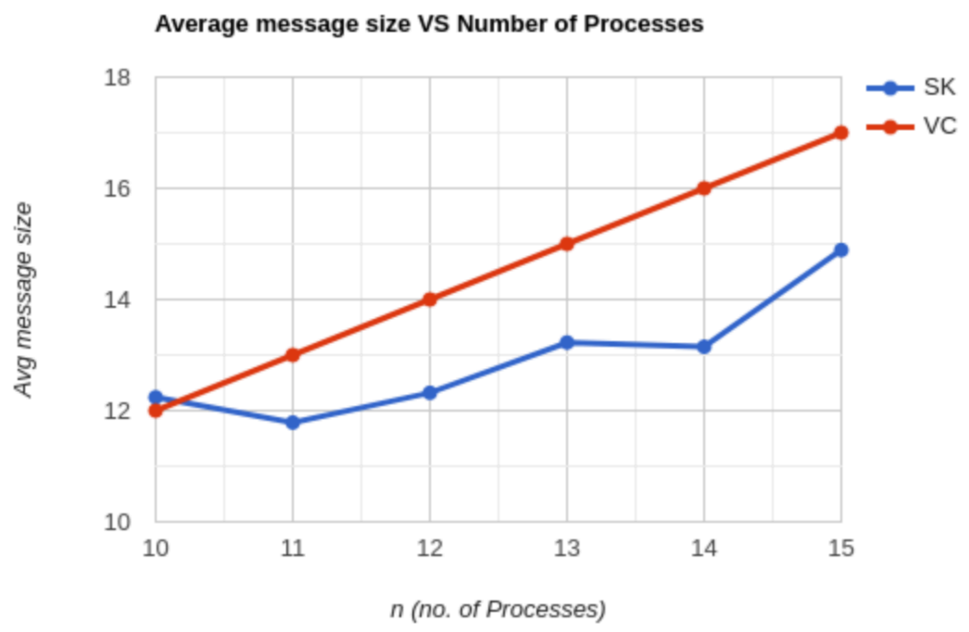
Number of Processes VS Average Message size

2 cases :

1. Sparse Graph as input

DC || PA1 || SC || Sparse Graph || m=10						
n (no. of Processes)	Avg message size					Average
10	11.92	14.62	10.68	11.98	12	12.24
11	13.6909	11.7	10.63	9.418	13.49	11.78578
12	10.92	12.48	12.18	13.76	12.26	12.32
13	12.6	12.44	11.43	15.02	14.64	13.226
14	14.614	9.6	12.81	12.7	16.02	13.1488
15	13.13	14.84	16.21	14.93	15.34	14.89

DC || PA1 || VC || Sparse Graph || m=10	
n (no. of Processes)	Avg message size
10	12
11	13
12	14
13	15
14	16
15	17



In the case of vector clocks, the relationship between number of processes and average message size is linear . If there are n messages then average message size n+2 here. But in Singhal-Kshemkalyani's optimization , we see that when we take sparse graphs as input , the average number of message sizes is very less as compared to vector clocks .

In Singhal-Kshemkalyani's optimization of vector clocks we see that the average message size increases with increase in the number of processes .

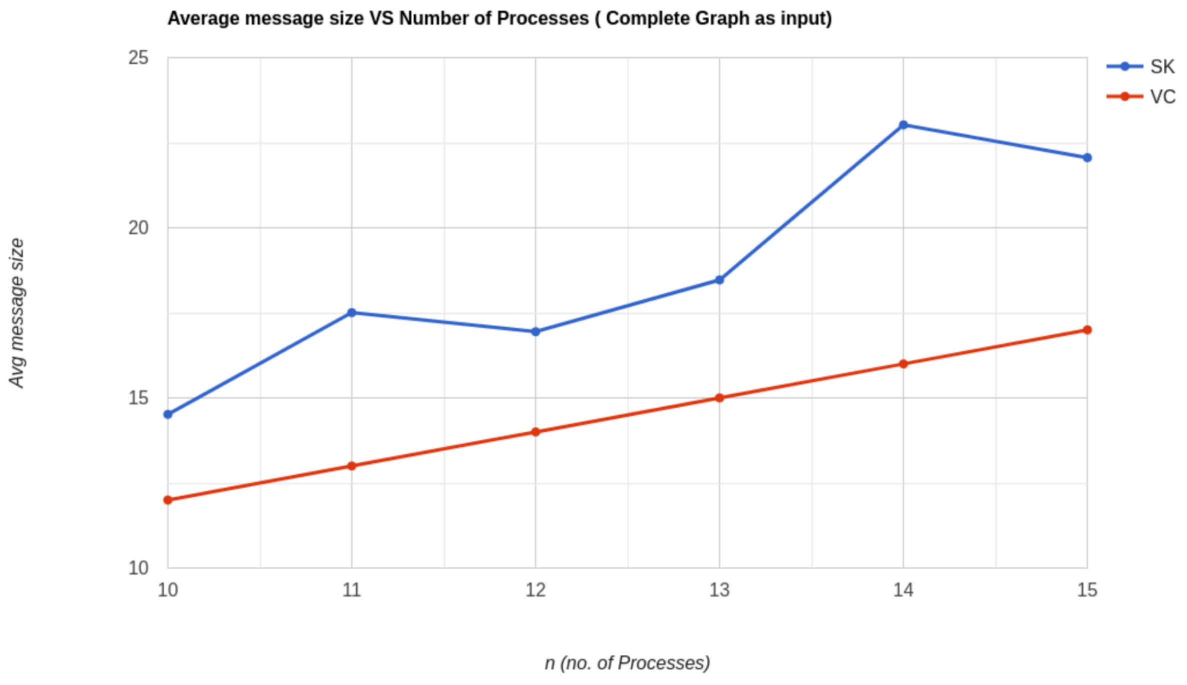
We see that as the number of threads increases the average

2. Dense Graph/ Fully Connected Graph

PA1 || VC || Fully Connected Graph	
n (no. of Processes)	Avg message size
10	12
11	13
12	14
13	15

14	16
15	17

DC || PA1 || SK || Fully Connected Graph || m=50				
n (no. of Processes)	Avg message size			Average
10	15.03	13.8	14.721	14.517
11	19.19	14.94	18.39	17.50666667
12	20.15	15.21	15.48	16.94666667
13	17.2031	18.52	19.68	18.4677
14	23.3	23.39	22.38	23.02333333
15	21.31	25.78	19.09	22.06



In the case of a fully complete graph as input ,in the case of Singhal-Kshemkalyani’s Vector clock we see that as we increase the number of processes (keeping the number of messages sent by each process = 50), the average message size increases and decreases alternatively.

The average message size in Singhal-Kshemkalyani’s optimization is always more than that in vector clocks without optimization for all values of n.

CONCLUSION

1. When a Sparse input graph is given , Singhal-Kshemkalyani’s optimization performs better than Vector clocks.
2. When a fully connected input graph is given , vector clocks perform better than Singhal-Kshemkalyani’s vector clock.

REFERENCES

1. https://www.mpi-forum.org/
2. https://zeromq.org/languages/cplusplus/
3. Book : Distributed Computing by Ajay D. Kshemkalyani and Mukesh Singhal