Implementing and Comparing Vector Clocks

Goal: The goal here is to implement Vector-clocks and Singhal-Kshemkalyani optimization on a Distributed System. We have to implement both these algorithms for vector clocks in C++ using MPI.

Vector clocks are a mechanism for determining the partial ordering of events in a distributed system. Each process Pi maintains a vector clock VCi of size n (number of processes).
The Singhal-Kshemkalyani optimization technique reduces communication overhead by sending only the vector clock entries that have been updated since the last message to a specific destination process.

In the implementations for the vector clocks, we have:
Parameters:

- λ (lambda): Mean inter-event time (milliseconds)
- α (alpha): Ratio of internal events to send events
- Probability of send: P(send) = 1 / (1 + α)

Event Types:

- Internal Event: Increment own clock entry
- Send Event: Increment clock, send message with timestamp
- Receive Event: Update clock using element-wise maximum

Communication Pattern:

- Topology: Fully connected graph (all-to-all communication)
- Message Selection: Random neighbour selection from adjacency list
- Termination: Each process sends exactly m messages
- Synchronization: MPI barriers ensure proper termination

We have the following total files:

```
├── vector_clock.cpp          # Standard Vector Clock implementation
├── sk_vector_clock.cpp       # Singhal-Kshemkalyani optimization
├── Makefile                  # Build script
├── quick_test.sh             # Automation script for a quick test
├── run_experiments.sh        # Automation script for the required experiments
├── verify_consistency.py     # Verification script
├── plot_results.py           # Plotting script
├── inp-params.txt            # Input parameters file
```

Firstly, we set up the ‘inp-params.txt’ file in the following format:

n λ α m
node1 neighbor1 neighbor2 ...
node2 neighbor1 neighbor2 ...
...

We have the following input file:

3 5 1.5 40
1 2 3
2 1 3
3 1 2

Then we ‘make’ all the files up and start running the programs. We firstly run the ‘quick_test.sh’ with n = 3 (and m=40), which automatically runs both the normal vector and the SK vector clocks programs.
With the standard vector clock’s average entries per message set at 3, the SK technique’s average entries per message hovered around 2.0-2.1, depicting a ~28% saving on average.

After this, we run the ‘verify_consistency.py’, to check and verify the causality, concurrency and monotonicity.

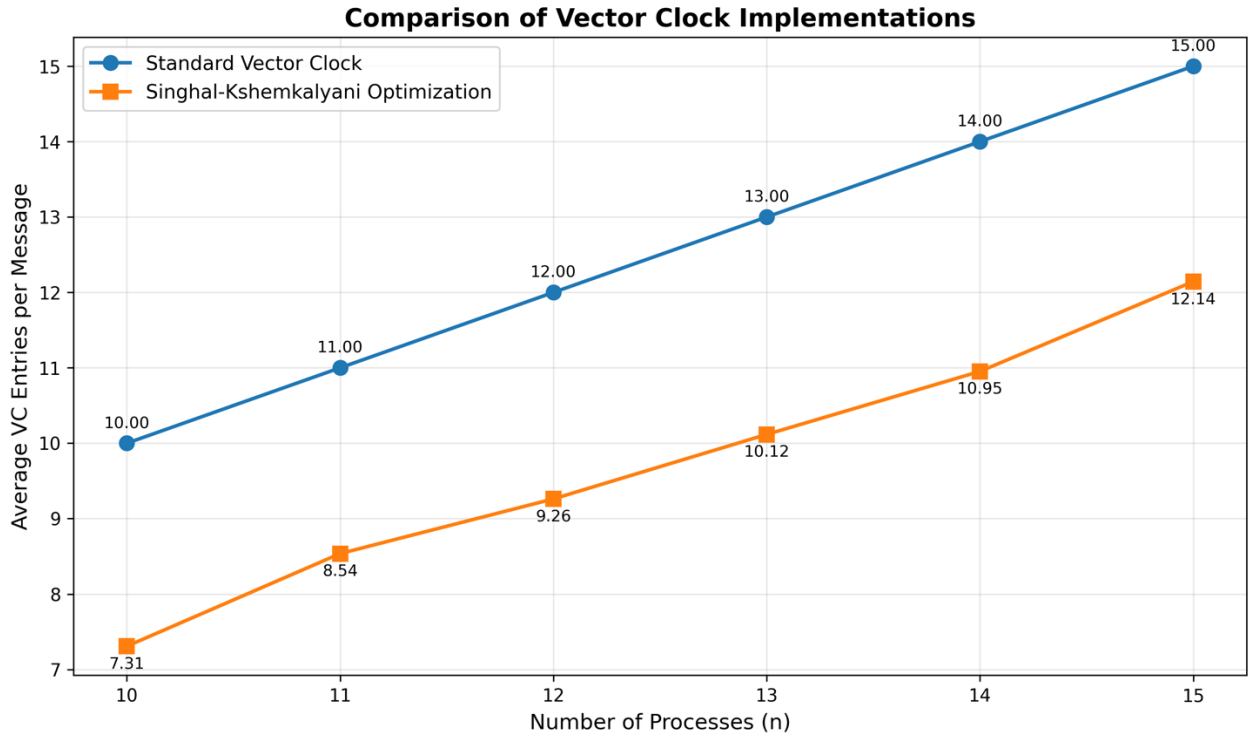
With the system set in place, and correctly running, we finally run the ‘run_experiments.sh’ with n=10 to n=15 (and m=50). This script automatically edits the input text file accordingly, runs the vector clock programs, and the summary and the results are saved.
We then run the final file ‘plot_results.py’ to plot the results obtained.

So, final results obtained for n=10 to n=15:

No. of Processes (n)	Standard VC	SK VC	Entries Reduced	Savings (%)
10	10.00	7.31	2.69	26.92%
11	11.00	8.54	2.46	22.40%
12	12.00	9.26	2.74	22.82%
13	13.00	10.12	2.88	22.18%
14	14.00	10.95	3.05	21.77%
15	15.00	12.14	2.86	19.05%

Average Vector Clock Entries per Message ; Avg: 2.78 22.52%

A Graph for the same:



The files were run using the Linux Terminal, on a Linux Virtual Machine, on a M1 Mac.
More detailed and complete instructions about how to set up and run all the programs are given in the 'README.txt' file.

Key Observations:

1. The SK optimization technique achieved savings ranging from 19.05% to 26.92%, with an average of 22.52% reduction in communication overhead compared to standard vector clocks, consistently performing better than them.

2. An interesting pattern emerged where savings decrease as the number of processes increases:

- n=10: 26.92% savings (highest)
- n=15: 19.05% savings (lowest)

This trend occurs because in a fully connected topology, as n increases, more processes are communicating frequently. This means more vector clock entries are being updated between successive messages to the same destination, reducing the effectiveness of the differential update strategy.

Other related anomalies/explanations:

1. When running ‘quick_test.sh’ multiple times, savings varied between 26-31%. This is not an error but expected behavior due to:
   - Random event generation
   - Random neighbour selection for message destinations
   - Non-deterministic message ordering in MPI

2. The SK optimization technique achieved 19-27% savings instead of the theoretical 50-60% maximum because:
   - Fully connected topology: Every process communicates with every other process, causing frequent updates to all vector clock positions
   - High message rate (m=50): Frequent messages mean less time for entries to remain unchanged between communications to the same destination
   - Random communication pattern: Messages distributed uniformly across all processes, preventing locality that would benefit optimization

   In sparse topologies (ring, star) or with lower message rates, savings would be higher.

3. Despite decreasing percentage savings, the absolute number of entries saved actually increases with n (2.69 to 3.05 entries on average). This shows that SK optimization provides growing benefits in absolute terms even as relative benefits decrease.