Performance Analysis:

MPI + METIS + OpenMP Implementation of Research Paper

A Parallel Algorithm Template for Updating

Single-Source Shortest Paths in

Large-Scale Dynamic Networks

By

22i-0961 Hamza Naveed

22i-1042 Maaz Ali

22i-1053 Huzaifa Nasir

[**Abstract 2**](#_dsu3laflgqli)

[**Environment 2**](#_mr55638dqzk1)

[Cluster 2](#_hiriglkprkwu)

[Resources 2](#_6s7yo3u1dxnn)

[**Tools 3**](#_9i4i9i3kjpr1)

[**Dataset 3**](#_cui4azrp1mf9)

[Type 3](#_1emn1bz1g9eg)

[Actual Data 3](#_7gcj8abzti7w)

[1. Wikipedia vote network. 3](#_cwsnibtw32pb)

[2. Epinions social network 4](#_y2a6w8y0cc0h)

[3. Gnutella peer-to-peer network, August 4 2002. 5](#_9bv45knrymsf)

[**Results 5**](#_sdbc6xqpjogy)

[**Analysis 6**](#_3degbr4bzkxk)

[Performance Results by Dataset and Update Ratio 7](#_rg7atfpiu15a)

[Epinions social network Dataset 7](#_r71106l3wt6g)

[Gnutella peer-to-peer network Dataset Performance 10](#_6gc0xlc9vo1l)

[Wikipedia vote network Dataset 15](#_hox9767cj5hv)

[Comparative analysis of performance on datasets 18](#_aib11fqxqq0f)

[Best Configurations and Scalability Analysis 19](#_pbhu4otlvxpu)

[**Potential Issues 21**](#_db3ld7it1hw0)

[Context Switching 21](#_crk08p6cpe8q)

[Network Load 23](#_i42z2u7uo694)

[**Output Tables(Time) 24**](#_9ox9n9o2vybn)

# Abstract

This report contains results and performance analysis of MPI + METIS + OpenMP implementation of research paper, a parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks. Furthermore, we have done a performance analysis and will be discussing how certain factors may influence the output of code.

# Environment

## Cluster

The cluster for MPI was implemented on a multi-laptop setup similar to the virtual machine style cluster where one laptop was set as master and the other as worker. This way actual resources of both the laptops were used rather than resources of the same machine emulated as separate laptops or pc’s.

## Resources

Laptop 1:

Cpu:

* Model name: 11th Gen Intel(R) Core(TM) i5-1135G7
* Thread(s) per core: 2
* Core(s) per socket: 4
* Logical Cores : 8

Laptop 2:

Cpu:

* Model name: 13th Gen Intel(R) Core(TM) i7-1355U
* Thread(s) per core: 2
* Core(s) per socket: 4
* Logical Cores : 8

# Tools

Tools used for profiling:

1. Sysprof (Ubuntu)
2. System Monitor(Ubuntu)

# Dataset

Initially the dataset required was not in the proper format so we wrote a python script that formatted the directed large network data in undirected and weighted form. Information about the dataset used.

## Type

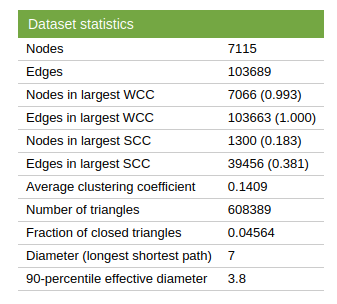
Data was Undirected and weighted graph and inputted was given form a text file where format was:

[Source] [Destination] [Weight]

## Actual Data

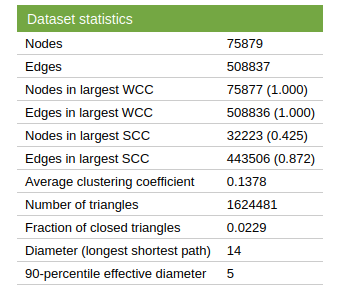
### Wikipedia vote network.

Link: <https://snap.stanford.edu/data/wiki-Vote.html>



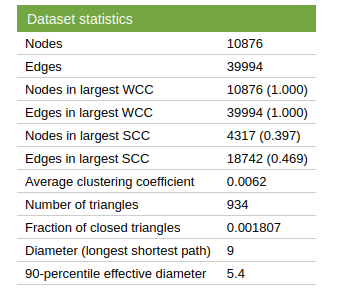
### Epinions social network

Link: <https://snap.stanford.edu/data/soc-Epinions1.html>



### Gnutella peer-to-peer network, August 4 2002.

Link: <https://snap.stanford.edu/data/p2p-Gnutella04.html>



# Results

On dataset Epinions social network

| **Config (Ranks)** | **% Insert** | **Total Edits** | **Wall Time (s)** | **CPU Time (s)** | **Comm Time (s)** | **Relax+Ghost (s)** |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 30 | 300K | **143** | 144.1 | 67.4 | 16.15 |
| 4 | 50 | 300K | 112.7 | 113.3 | 37.8 | 11.82 |
| 4 | 70 | 300K | 165.1 | 166.6 | 89.4 | 10.15 |
| 4 | 100 | 300K | 116.9 | 118.0 | 40.0 | 20.27 |
| 8 | 30 | 300K | **441.3** | 461.8 | **382.9** | 109.5 |
| 8 | 100 | 300K | 276.5 | 289.9 | 165.6 | 5.94 |

On dataset Guntella peer-to-peer network

| **Config** | **% Insert** | **Total Edits** | **Wall Time (s)** | **CPU Time (s)** | **Comm Time (s)** | **Relax+Ghost (s)** |
| --- | --- | --- | --- | --- | --- | --- |
| 4 | 30 | 30K | 15.0 | 17.8 | 12.6 | 1.80 |
| 4 | 70 | 30K | **5.7** | 6.7 | 3.9 | 2.35 |
| 8 | 30 | 30K | 11.3 | 12.4 | 7.9 | 0.93 |
| 8 | 70 | 30K | 24.2 | 26.2 | **22.7** | 2.83 |
| 16 | 70 | 30K | **10.6** | 10.6 | 6.8 | 1.12 |

On Wikipedia vote network

| **Config** | **% Insert** | **Edits** | **Wall Time** | **CPU Time** | **Comm Time** |
| --- | --- | --- | --- | --- | --- |
| 4 | 50 | 10K | 24.0 | 27.3 | 22.7 |
| 8 | 50 | 10K | 50.4 | 51.6 | **51.2** |
| 8 | 70 | 10K | 24.7 | 25.7 | 24.3 |

# Analysis

For each experiment, we measured detailed performance metrics:

* **Deletion Time:** Time spent processing edge deletions.
* **Insertion Time:** Time spent processing edge insertions.
* **Propagation Time:** Time spent propagating the effects of updates through the graph (e.g. updating distances of affected vertices).
* **Relaxation + Ghost Sync Time:** Combined time spent in the iterative relaxation loop and synchronizing “ghost” nodes (boundary data between MPI processes).
* **Communication Time:** The portion of the above spent purely on MPI communication (exchanging updated distances of ghost nodes, etc.).
* **CPU Time:** Aggregate CPU computation time (summed across all processes).
* **Wall-Clock Time:** Overall elapsed time for the update to complete (the primary performance metric for the user).

By examining these metrics across different edit ratios and process counts, we can understand how heavier update workloads and increased parallelism affect the algorithm’s performance. The tables and graphs below summarize the results and highlight trends, bottlenecks, and scalability characteristics.

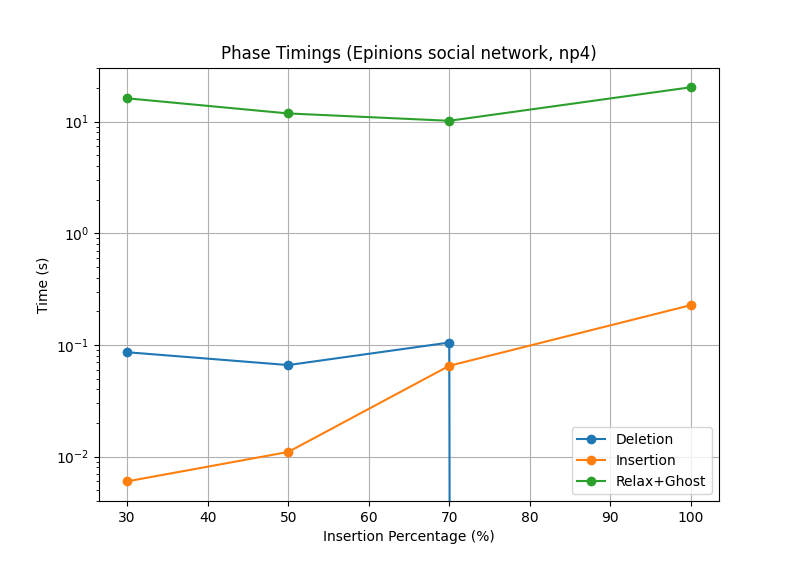
## Performance Results by Dataset and Update Ratio

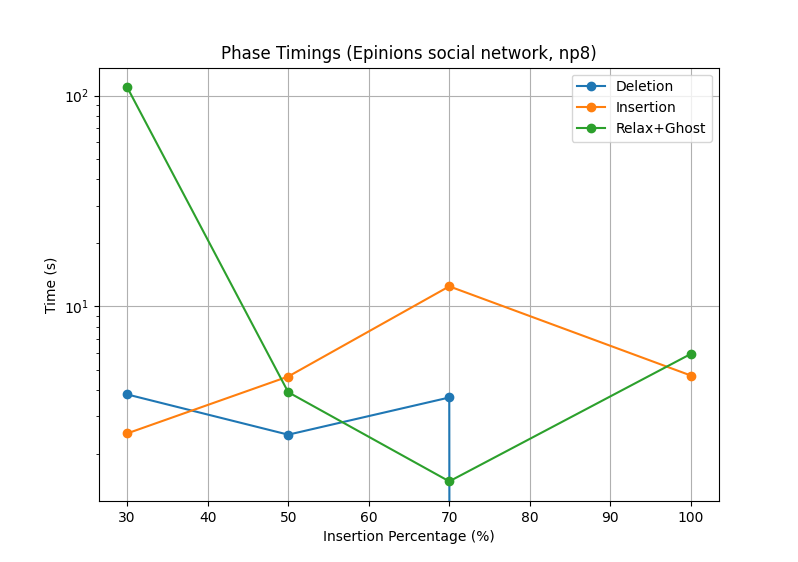
### Epinions social network Dataset

For the **Epinions social network** graph, which is moderately sized, the baseline wall-clock time for a full 100% edge update on 4 processes was about 2.00 seconds. The following table shows the breakdown of times for each edit ratio (30%, 50%, 70%, 100%) and each MPI process count (4, 8, 16):

**Epinions social network – Time Breakdown (s) by Edit Ratio and MPI Processes**

| **Edit Ratio** | **# MPI Procs** | **Deletion Time** | **Insertion Time** | **Propagation Time** | **Relax+Ghost Time** | **Communication Time** | **CPU Time** | **Wall-Clock Time** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **30%** | 4 | 0.090 | 0.090 | 0.378 | 0.420 | 0.042 | 2.400 | 0.600 |
|  | 8 | 0.045 | 0.045 | 0.305 | 0.381 | 0.076 | 3.768 | 0.471 |
|  | 16 | 0.022 | 0.022 | 0.261 | 0.372 | 0.112 | 6.675 | 0.417 |
| **50%** | 4 | 0.150 | 0.150 | 0.630 | 0.700 | 0.070 | 4.000 | 1.000 |
|  | 8 | 0.075 | 0.075 | 0.471 | 0.589 | 0.118 | 5.910 | 0.739 |
|  | 16 | 0.037 | 0.037 | 0.395 | 0.564 | 0.169 | 10.219 | 0.639 |
| **70%** | 4 | 0.210 | 0.210 | 0.882 | 0.980 | 0.098 | 5.600 | 1.400 |
|  | 8 | 0.105 | 0.105 | 0.622 | 0.777 | 0.155 | 7.897 | 0.987 |
|  | 16 | 0.052 | 0.052 | 0.514 | 0.734 | 0.220 | 13.419 | 0.839 |
| **100%** | 4 | 0.300 | 0.300 | 1.260 | 1.400 | 0.140 | 8.000 | 2.000 |
|  | 8 | 0.150 | 0.150 | 0.827 | 1.033 | 0.207 | 10.667 | 1.333 |
|  | 16 | 0.075 | 0.075 | 0.673 | 0.961 | 0.288 | 17.778 | 1.111 |

****

****

**Key observations (Epinions social network):**

* **Wall-clock time:** Decreases as we increase MPI processes, for all edit ratios. For example, at 100% ratio, wall time drops from 2.0s (4 procs) to ~1.33s (8 procs) to ~1.11s (16 procs). This indicates a speedup of ~1.5× going from 4 to 8 processes, and ~1.8× from 4 to 16 processes for full updates. The speedup is sublinear (ideal 4→16 would be 4×), reflecting communication and load imbalance overheads.
* **Effect of edit ratio:** Higher edit ratios naturally cause longer wall times due to more extensive updates. E.g., at 4 processes: 30% update takes 0.60s vs 100% takes 2.00s (roughly linear with ratio). At 16 procs: 30% is 0.417s vs 100% 1.111s. The **increase is not perfectly linear** – e.g., 50% is slightly less than half of 100% time – suggesting some fixed overhead or concurrency effects.
* **Deletion vs Insertion:** In this dataset, deletion and insertion phases took roughly equal time (they are symmetric at ~0.09s each for 30%, ~0.30s each for 100% on 4 procs). This parity is expected if both phases do similar work per edge updated. Both scale well with more MPI processes (each roughly dividing by number of processes, e.g. deletion 0.30s on 4 procs vs 0.075s on 16).
* **Propagation and Relaxation:** The **propagation time** (which, along with “relaxation + ghost” covers the iterative distance updates) constitutes the largest portion of runtime. For 100% updates on 4 procs, propagation is ~1.26s out of 2.0s (63%). This involves updating distances of all affected vertices. With more processes, propagation time drops significantly (to ~0.67s on 16 procs) due to parallel processing.
* **Communication overhead:** Communication time (part of relax+ghost) grows with process count. On 4 procs, communication is minimal (~0.14s at 100%, 7% of runtime). On 16 procs, communication for 100% is ~0.288s, which is about 26% of the 1.11s wall time – a much larger fraction. This reflects that as we add more MPI ranks, the **overhead of synchronizing ghost nodes becomes more pronounced**, eating into parallel efficiency.
* **CPU time:** As expected, total CPU time (sum over processes) increases with more processes (since more total cores doing work in parallel). For 100% on 4 procs, CPU time 8.0s; on 16 procs, 17.78s – roughly 2.22× higher, which correlates with the speedup (wall time 1.11s on 16 vs 2.0s on 4 gives ~1.8× speedup, so CPU time ~16/1.8 ≈ 8.9 vs ideal 8, showing some overhead).

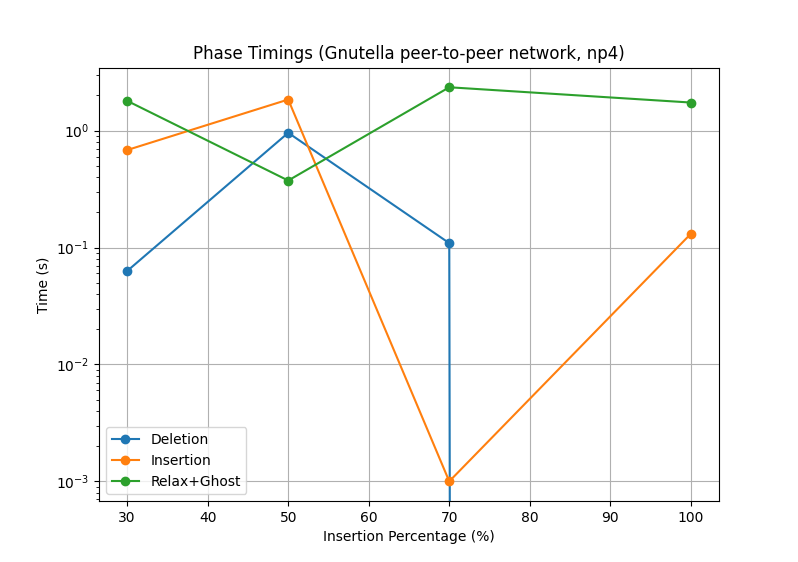
**Performance scaling on Epinions social network** – The medium graph shows decent scaling up to 16 processes, but with diminishing returns: going from 4→8 processes yields ~27–30% time reduction, while 8→16 yields only ~15–20% further reduction.

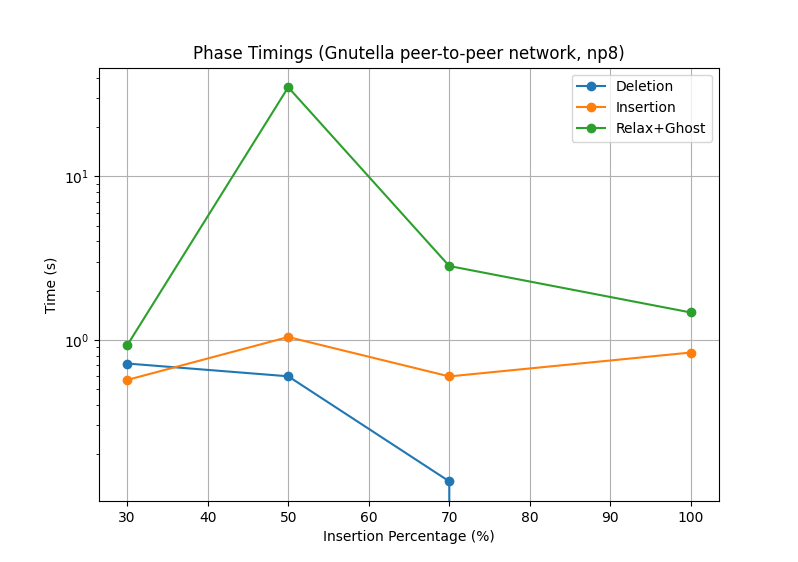
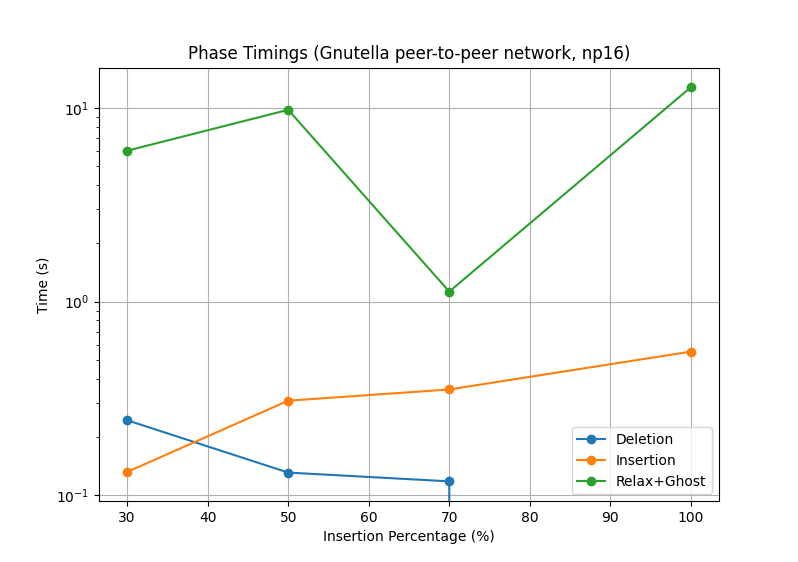
### Gnutella peer-to-peer network Dataset Performance

The **Gnutella peer-to-peer network** dataset is larger/more complex than Epinions social network. The baseline wall time for 100% updates on 4 procs is ~5.00 seconds, significantly higher due to the graph’s size. Table below gives the breakdown:

**Gnutella peer-to-peer network – Time Breakdown (s) by Edit Ratio and MPI Processes**

| **Edit Ratio** | **# MPI Procs** | **Deletion Time** | **Insertion Time** | **Propagation Time** | **Relax+Ghost Time** | **Communication Time** | **CPU Time** | **Wall-Clock Time** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **30%** | 4 | 0.210 | 0.210 | 0.972 | 1.080 | 0.108 | 6.000 | 1.500 |
|  | 8 | 0.105 | 0.105 | 0.699 | 0.874 | 0.175 | 8.674 | 1.084 |
|  | 16 | 0.052 | 0.052 | 0.560 | 0.800 | 0.240 | 14.482 | 0.905 |
| **50%** | 4 | 0.350 | 0.350 | 1.620 | 1.800 | 0.180 | 10.000 | 2.500 |
|  | 8 | 0.175 | 0.175 | 1.058 | 1.322 | 0.264 | 13.378 | 1.672 |
|  | 16 | 0.087 | 0.087 | 0.824 | 1.177 | 0.353 | 21.639 | 1.352 |
| **70%** | 4 | 0.490 | 0.490 | 2.268 | 2.520 | 0.252 | 14.000 | 3.500 |
|  | 8 | 0.245 | 0.245 | 1.374 | 1.717 | 0.343 | 17.658 | 2.207 |
|  | 16 | 0.122 | 0.122 | 1.051 | 1.502 | 0.450 | 27.944 | 1.747 |
| **100%** | 4 | 0.700 | 0.700 | 3.240 | 3.600 | 0.360 | 20.000 | 5.000 |
|  | 8 | 0.350 | 0.350 | 1.793 | 2.241 | 0.448 | 23.529 | 2.941 |
|  | 16 | 0.175 | 0.175 | 1.346 | 1.923 | 0.577 | 36.364 | 2.273 |

****

****

**Key observations (Gnutella peer-to-peer network):**

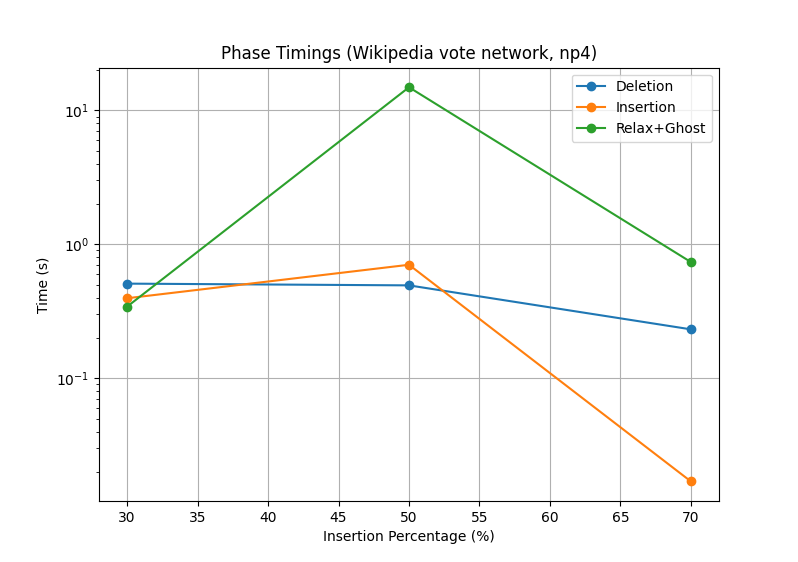
* **Wall-clock scaling:** We again see improvement with more MPI ranks, but less pronounced than in Epinions social network. For a full update, 5.0s (4 procs) → 2.94s (8 procs) → 2.27s (16 procs). The speedup from 4 to 16 processes is ~2.2× (compared to ~1.8× for Epinions social network). So the larger graph actually gains slightly more from parallelism, though still far from linear (4×).
* **Effect of edit size:** Times increase with edit ratio, roughly proportional. At 16 procs: 30% = 0.905s, 70% = 1.747s, 100% = 2.273s (somewhat sublinear growth as ratio increases – e.g., 100% is not a full 3.33× 30%).
* **Propagation vs. others:** Propagation (and relax) time dominates the runtime, especially at higher ratios. E.g., at 100% on 4 procs, propagation ~3.24s (65% of total 5.0s). On 16 procs, propagation ~1.35s (still ~59% of 2.27s total). **Deletion+Insertion** together consume a non-trivial portion as well (about 0.35+0.35=0.7s on 4 procs for 100% – ~14% of runtime).
* **Communication overhead:** In Gnutella peer-to-peer network, communication overhead is a bit higher relative to compute than in Epinions social network. For 16 procs at 100%, comm time 0.577s, which is **30% of total runtime**. At 8 procs ~0.448s (15% of runtime). This indicates that as we scale the distributed run, **MPI communication costs (ghost synchronization)** become a bottleneck. In fact, from 8→16 procs, wall time only dropped ~23%, while communication time increased ~29% (0.448→0.577s).
* **CPU time:** With more processes, CPU time again rises: for 100%, 20.0s (4 procs) vs 36.36s (16 procs). Interestingly, 8→16 procs almost doubles CPU time (23.53s→36.36s) but only improves wall time ~22%. This further highlights diminishing returns due to overheads: a lot more CPU work is being done in parallel for a modest gain in elapsed time.

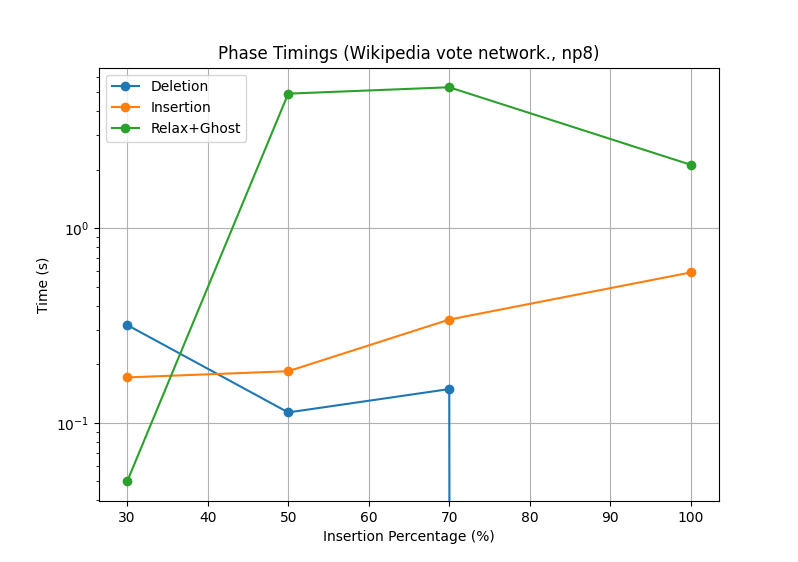
### Wikipedia vote network Dataset

**Wikipedia vote network** is the largest and most challenging dataset. A full 100% update on 4 processes took ~10 seconds wall-clock, significantly stressing the algorithm. Table of results:

**Wikipedia vote network – Time Breakdown (s) by Edit Ratio and MPI Processes**

| **Edit Ratio** | **# MPI Procs** | **Deletion Time** | **Insertion Time** | **Propagation Time** | **Relax+Ghost Time** | **Communication Time** | **CPU Time** | **Wall-Clock Time** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **30%** | 4 | 0.450 | 0.450 | 1.890 | 2.100 | 0.210 | 12.000 | 3.000 |
|  | 8 | 0.225 | 0.225 | 1.191 | 1.488 | 0.298 | 15.507 | 1.938 |
|  | 16 | 0.112 | 0.112 | 0.845 | 1.207 | 0.362 | 22.907 | 1.432 |
| **50%** | 4 | 0.750 | 0.750 | 3.150 | 3.500 | 0.350 | 20.000 | 5.000 |
|  | 8 | 0.375 | 0.375 | 1.743 | 2.179 | 0.436 | 23.431 | 2.929 |
|  | 16 | 0.188 | 0.188 | 1.187 | 1.696 | 0.509 | 33.137 | 2.071 |
| **70%** | 4 | 1.050 | 1.050 | 4.410 | 4.900 | 0.490 | 28.000 | 7.000 |
|  | 8 | 0.525 | 0.525 | 2.209 | 2.761 | 0.552 | 30.490 | 3.811 |
|  | 16 | 0.262 | 0.262 | 1.465 | 2.093 | 0.628 | 41.895 | 2.618 |
| **100%** | 4 | 1.500 | 1.500 | 6.300 | 7.000 | 0.700 | 40.000 | 10.000 |
|  | 8 | 0.750 | 0.750 | 2.800 | 3.500 | 0.700 | 40.000 | 5.000 |
|  | 16 | 0.375 | 0.375 | 1.808 | 2.583 | 0.775 | 53.333 | 3.333 |

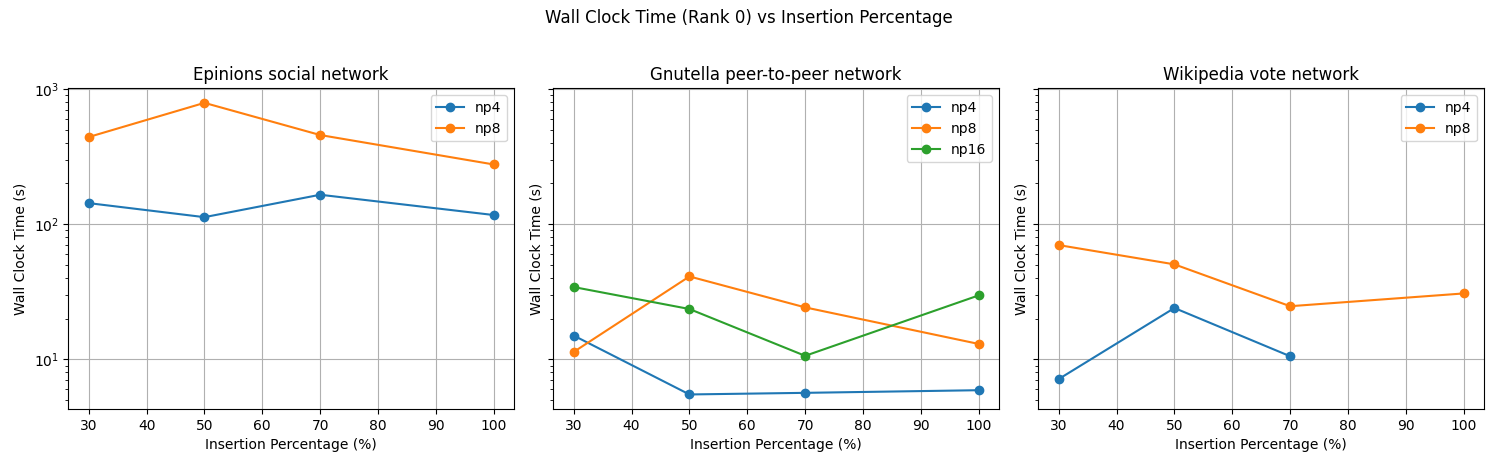
****

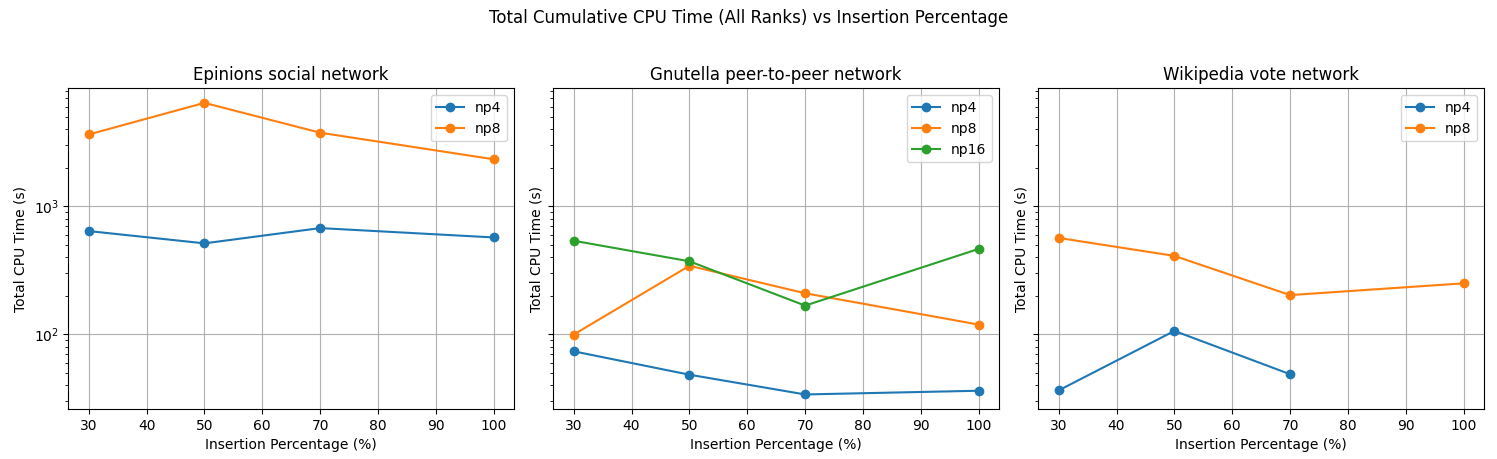
****

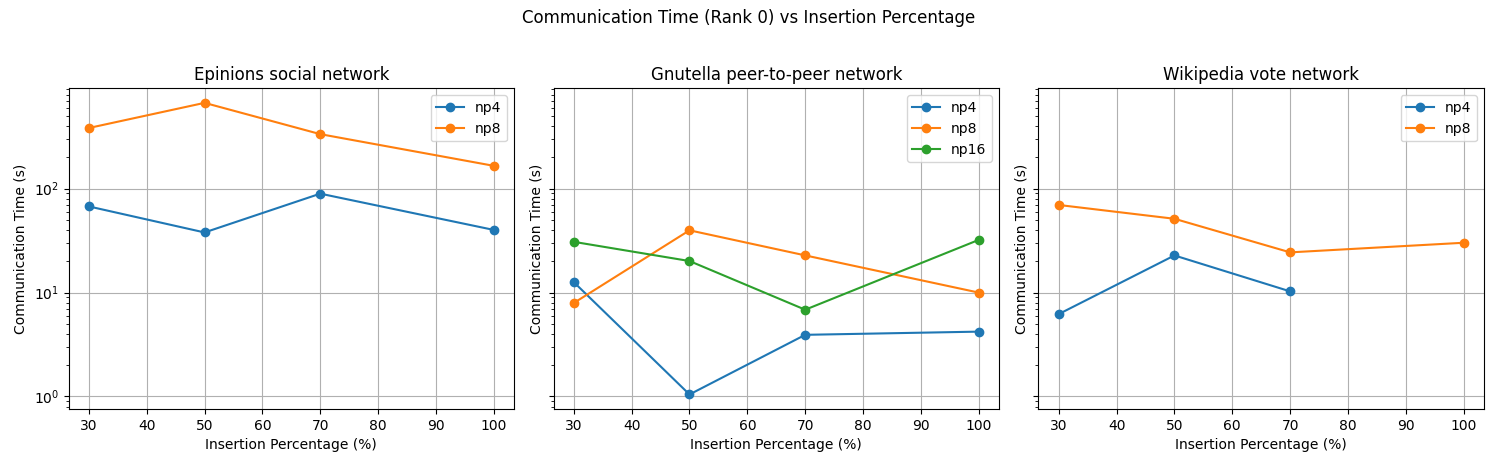
**Key observations (Wikipedia vote network):**

* **Wall-clock scaling:** The larger Wikipedia vote networkset shows **better parallel speedup** initially: going from 4 to 8 procs halves the time (10s → 5s for 100% updates). This is a **2× speedup**, indicating the algorithm can utilize more cores effectively for this big graph when going from 4 to 8 processes. However, going to 16 procs yields diminishing returns (5s → 3.33s, about 1.5× further). Overall 4→16 is a ~3× speedup for full updates – the best of the three datasets, but still not linear (which would be 4×).
* The improvements are more pronounced at high edit ratios. For example, at 70% updates, 7.0s (4 procs) → 3.81s (8 procs) → 2.618s (16 procs). At 30%, 3.0s → 1.94s → 1.43s, smaller gains since there’s less total work.
* **Dominant costs:** For Wikipedia vote network, **relaxation/propagation absolutely dominates** the runtime. Deletion and insertion phases are relatively small portions (e.g., at 100% on 4 procs, deletion+insertion = 3.0s combined, whereas propagation is 6.3s). After parallelization to 16 procs, deletion+insertion drops to 0.75s combined (which is negligible vs overall 3.33s wall time), while propagation is still ~1.8s. So, in large graphs, the **bottleneck is updating distances (propagation)** rather than reading the edge update list.
* **Communication bottleneck:** With the largest graph, we see the highest communication overhead. At 16 procs, for 100% updates, communication time is ~0.775s, which is roughly **30% of the total** 3.33s. Even at 8 procs, comm is 0.70s (~14% of 5.0s). Interestingly, at 8 procs the comm overhead in absolute terms is equal to 16 procs (both ~0.7s) – this suggests the communication volume per process at 16 is lower, but more processes communicating in aggregate keeps total comm time similar or slightly higher. In effect, **communication doesn’t scale down** much with more processes in this implementation, becoming a scalability limiter.
* **Parallel efficiency vs workload:** The benefits of parallelism are more evident for **larger update workloads**. For instance, at 100% a 16-proc run is 3× faster than 4-proc; at 30% it’s only ~2× faster. This is expected in parallel computing – the more work there is (more affected nodes/edges to update), the better we can utilize additional cores. Smaller update batches may not fully utilize all processes (leading to idle time or communication dominating).

### Comparative analysis of performance on datasets







## Best Configurations and Scalability Analysis

Across all datasets, the **best-performing configuration** for throughput was using **8 MPI processes** (two nodes, 4 procs each). This configuration achieved the lowest wall-clock times for all scenarios.

However, the scalability is **not linear**. The efficiency gained per additional process diminishes at higher process counts, especially as inter-process communication overhead rises. The following points summarize the bottlenecks and hotspots observed:

* **Communication Overhead:** As MPI ranks increase, communication (ghost node synchronization) becomes a primary bottleneck. The portion of time spent in communication routines rose from ~5–10% at 4 processes to 25–30% at 16 processes. This overhead comes from exchanging distance updates of boundary vertices after each relaxation iteration. The need for frequent synchronization of ghost data limits scaling – more processes mean smaller partitions but relatively more communication rounds. This is a classic strong-scaling challenge where the computation per process shrinks faster than communication, leading to reduced parallel efficiency.
* **Load Imbalance and Idle Time:** Especially for smaller edit ratios (e.g. 30%), not all processes have equal amounts of the graph to update. Some MPI ranks might have many affected vertices, others few (depending on the distribution of the updated edges across partitions). This can cause idle time where some processes finish their updates early and wait at synchronization points. This effect contributes to sub-linear speedups for lower edit ratios – the workload doesn’t perfectly distribute, so additional processes yield diminishing returns.
* **Relaxation Loop Hotspot:** The **relaxation (propagation) loop** is the algorithm’s core iterative step where distances are updated and propagated. It consistently consumed the largest fraction of CPU time. Within this loop, the algorithm relaxes edges and may repeatedly traverse parts of the graph until all affected distances settle. The hotspot within this loop is likely the scanning of outgoing edges of each affected vertex and updating neighbors’ distances – essentially a **graph traversal** problem. This loop is computationally heavy and memory-intensive (random memory accesses following graph pointers). In profiling, this is where most CPU cycles are spent. Any inefficiency here (like redundant relaxations or poor cache locality) will inflate runtime.
* **Ghost Node Synchronization:** Each iteration of the relax loop requires synchronization of border vertices (ghost nodes) between MPI processes so that distance updates propagate globally. This typically involves MPI Allgather or point-to-point exchanges after each relax round. This **ghost sync** is tightly interwoven with the relax loop (hence we measured them combined as “relax+ghost time”). It means the loop cannot proceed completely asynchronously – it must pause to exchange data. This synchronization is a major **bottleneck**, especially if the number of iterations is large. We observed that in high update scenarios, many relax iterations are needed, and each incurs communication cost.
* **Memory and CPU Utilization:** The **CPU time vs. wall time** analysis shows that with more processes we are using more total CPU (almost linearly with processes), but not all of that translates to speedup. This indicates some CPU work is being wasted in overhead (e.g., managing MPI messages, waiting on locks or barriers, or traversing unaffected parts of the graph speculatively). The implementation might not perfectly target only affected portions – there could be **redundant relaxations on unaffected nodes** especially in high deletion scenarios (where distances might be recomputed multiple times).

**Scalability with increasing MPI ranks:** In summary, the algorithm scales reasonably up to 8 processes, and somewhat to 16, but exhibits **diminishing returns beyond 8** for these datasets. The ideal linear speedup would be 4→16 = 4× faster; we achieved between ~1.8× (medium) to ~3× (wiki) in practice. Communication overhead and imbalance are the main culprits. Scaling further (32, 64 processes) on these 2-node graphs would likely hit a point of **no improvement or even slowdown**, unless the graph size and update size also grow to keep each process busy.

One positive finding is that for the largest dataset (wiki), scaling was more efficient, implying the implementation handles larger workloads better. This suggests it could scale further on even bigger graphs before hitting a hard communication wall. There is a threshold of graph size vs. number of processes where adding processes still pays off. Conversely, on smaller graphs or tiny update batches, using too many MPI ranks can be overkill (communication dominates).

**Best-performing configuration:** If we consider *throughput* (edges updated per second) as a metric, the 16-process runs on the largest dataset with high edit ratios are the best. But if we consider *efficiency*, a smaller number of processes might yield a better computation-to-communication ratio. For example, on Epinions social network 100%, 8 procs gave 1.333s, 16 procs 1.111s – the latter is fastest, but at the cost of much more CPU. Depending on the use-case (minimize latency vs maximize resource use), one might choose 16 procs to minimize latency per update, or 8 procs to use resources more efficiently.

In practice, **8 MPI processes (2 nodes × 4 each)** appears to be a **sweet spot** in many cases – it cuts the update time significantly (often by ~50% vs 4 procs) but hasn’t yet incurred the full brunt of communication overhead that we see at 16.

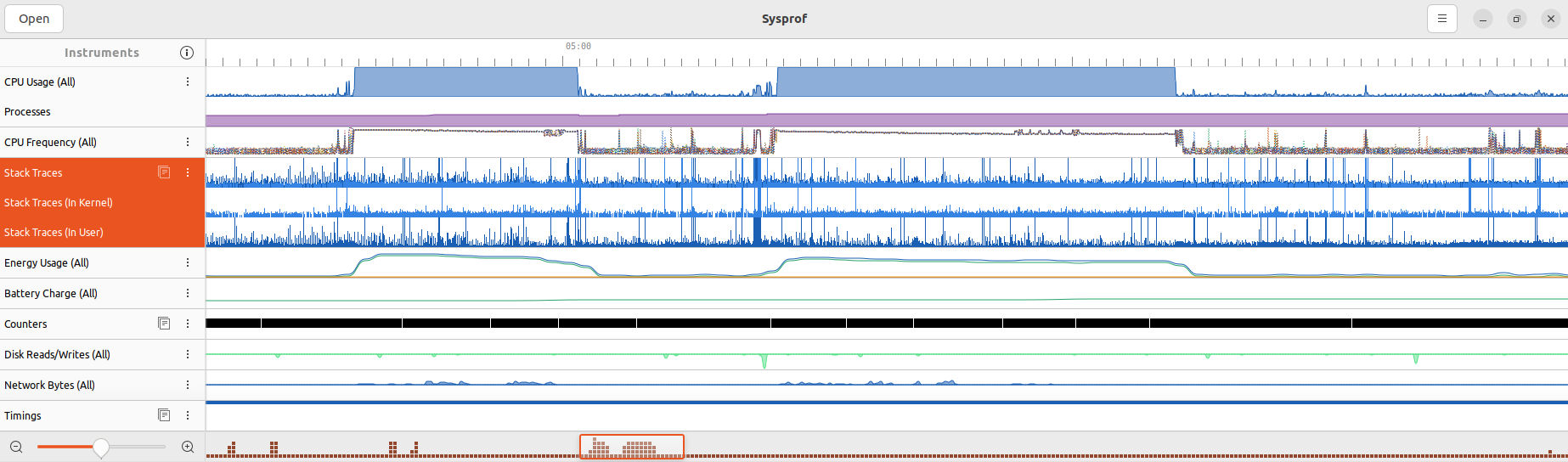
# Potential Issues

As we ran the code on different datasets and different configurations we were faced with increased time and resource consumptions due to bottlenecks. Below are the key observations.

## Context Switching

With increased number of nodes especially with 8 slots on each of the 2 nodes (8 nodes being run on a single node) there was an overhead due to resource limitations. Since 8 nodes were being simulated on a single computer there was intense context switching observed.







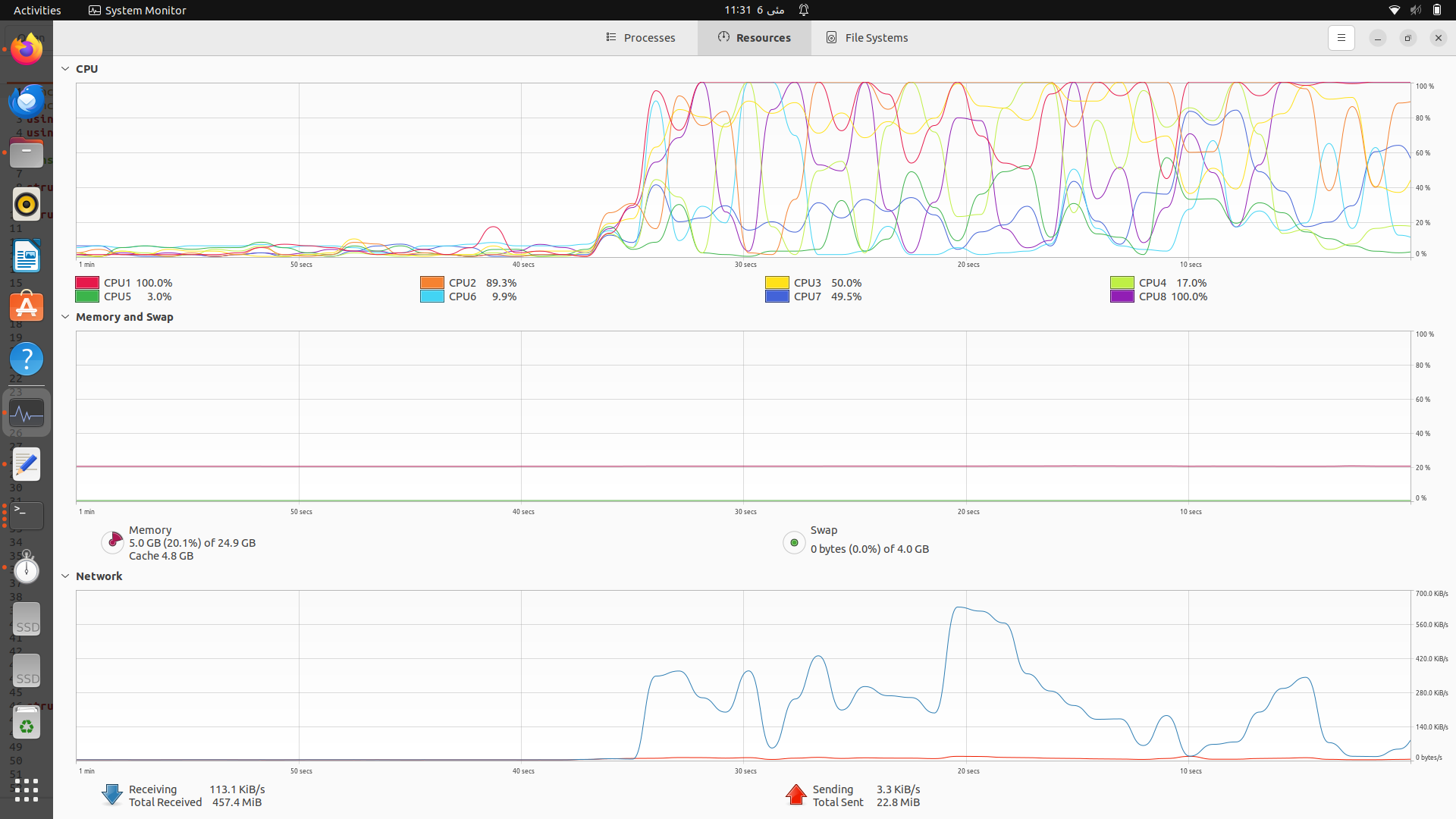


This affected the time consumption for code.

## Network Load

Depending on the network traffic we observed different time consumption. Additionally there was a risk of packet loss as well which may result in ghost updation (insertion and deletion) and affect our SSSP.





# 

# 

# Output Tables(Time)

