**National University of Computer and Emerging Sciences**

**Parallel and Distributed Computing**

**Project Report**

**Group 19**

**Group Members:**

* Huzaifa Nasir (22i-1053)
* Maaz Ali (22i-1042)
* Hamza Naveed (22i-0961)

**Degree Program:** BS Computer Science  
**Section:** CS-A  
**Subject Name:** Parallel and Distributed Computing

# Abstract

This report presents a comprehensive performance analysis of a parallel implementation of a dynamic Single-Source Shortest Path (SSSP) algorithm, based on the research paper "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks." The implementation leverages MPI for distributed computing, OpenMP for multi-threaded parallelism, and METIS for graph partitioning. We evaluate the algorithm's performance across three public datasets—Wikipedia Vote Network, Epinions Social Network, and Gnutella Peer-to-Peer Network—focusing on scalability, bottlenecks, and the impact of edit ratios and process counts. Performance metrics, including wall-clock time, CPU time, communication time, and phase-specific timings, are analyzed with visual aids to assess parallel efficiency. The results highlight varying scalability with process counts (4, 8, 16), with communication overhead and relaxation phase dominating runtime. The impact of MPI communication and OpenMP parallelism is thoroughly examined, providing insights into their contributions and limitations. Recommendations for optimization and future work are provided to enhance performance on large-scale dynamic networks.

Table of Contents

[Abstract 2](#_Toc197534968)

[1. Introduction 5](#_Toc197534969)

[2. Experimental Setup 5](#_Toc197534970)

[Hardware Environment 5](#_Toc197534971)

[Software Environment 6](#_Toc197534972)

[Datasets 6](#_Toc197534973)

[Experimental Configurations 7](#_Toc197534974)

[3. Methodology 7](#_Toc197534975)

[Algorithm Overview 7](#_Toc197534976)

[Performance Metrics 8](#_Toc197534977)

[Profiling and Analysis 8](#_Toc197534978)

[4. Performance Analysis 8](#_Toc197534979)

[Parallelization: 9](#_Toc197534980)

[Metrics: 9](#_Toc197534981)

[Performance Results and Analysis 9](#_Toc197534982)

[1. Epinions Social Network Dataset 9](#_Toc197534983)

[2. Gnutella Peer-to-Peer Network Dataset 13](#_Toc197534984)

[**3.** Wikipedia Vote Network Dataset 17](#_Toc197534985)

[5. Bottlenecks and Hotspots 22](#_Toc197534986)

[6. Impact of MPI and OpenMP 23](#_Toc197534987)

[7. Scalability Analysis 24](#_Toc197534988)

[8. Potential Optimizations 25](#_Toc197534989)

[9. Conclusion 26](#_Toc197534990)

## List of Figures

[Figure 1Wall-Clock Time vs. Process Count (Epinions) 10](#_Toc197534813)

[Figure 2 Communication Time vs. Edit Ratio (Epinions) 11](#_Toc197534814)

[Figure 3 Phase Time Breakdown (Epinions, 4 and 8 Processes) 12](#_Toc197534815)

[Figure 4 Context Switching and Network Load vs. Edit Ratio (Epinions) 13](#_Toc197534816)

[Figure 5 Wall-Clock Time vs. Process Count (Gnutella). 14](#_Toc197534817)

[Figure 6 Communication Time vs. Edit Ratio (Gnutella) 15](#_Toc197534818)

[Figure 7 Phase Time Breakdown (Gnutella, 4, 8, and 16 Processes) 16](#_Toc197534819)

[Figure 8 Context Switching and Network Load vs. Edit Ratio (Gnutella) 17](#_Toc197534820)

[Figure 9 Wall-Clock Time vs. Process Count (Wikipedia) 18](#_Toc197534821)

[Figure 10 Communication Time vs. Edit Ratio (Wikipedia) 19](#_Toc197534822)

[Figure 11 Phase Time Breakdown (Wikipedia, 4 and 8 Processes) 20](#_Toc197534823)

[Figure 12 Context Switching and Network Load vs. Edit Ratio (Wikipedia) 21](#_Toc197534824)

# 1. Introduction

The Single-Source Shortest Path (SSSP) problem is a cornerstone of graph theory, with applications in network routing, social network analysis, and transportation systems. In dynamic networks, where edges are frequently inserted or deleted, maintaining an up-to-date SSSP tree efficiently is challenging, especially for large-scale graphs. The research paper "A Parallel Algorithm Template for Updating Single-Source Shortest Paths in Large-Scale Dynamic Networks" proposes a parallel algorithm to address this challenge by updating affected subgraphs in a distributed environment.

Our implementation uses MPI for inter-node communication, OpenMP for intra-node parallelism, and METIS for graph partitioning. This report evaluates the performance of this implementation on three datasets, analyzing scalability, bottlenecks, and the specific contributions of MPI communication and OpenMP parallelism. Extensive visual aids, including line charts and bar graphs, are used to illustrate performance trends across different process counts (ranks) and edit ratios. We also discuss the experimental setup, performance metrics, and potential optimizations, aligning with the project requirements of the Parallel and Distributed Computing course.

# 2. Experimental Setup

## Hardware Environment

The experiments were conducted on a multi-laptop cluster, configured as follows:

* **Laptop 1 (Master):**
  + CPU: 11th Gen Intel® Core™ i7-1135G7
  + Cores: 4 (8 logical cores with 2 threads per core)
  + Memory: 16 GB
  + OS: Ubuntu 23.10 LTS
* **Laptop 2 (Worker):**
  + CPU: 11th Gen Intel® Core™ i5-1155U
  + Cores: 4 (8 logical cores with 2 threads per core)
  + Memory: 24 GB
  + OS: Ubuntu 22.04 LTS

The cluster was set up with one laptop as the master node and the other as a worker, utilizing their full resources to emulate a distributed environment.

## Software Environment

* **Programming Language:** C (for core implementation), Python (for data preprocessing)
* **Libraries:**
  + OpenMPI 4.1.1: For distributed computing
  + OpenMP 4.5: For multi-threaded parallelism
  + METIS 5.1.0: For graph partitioning
* **Profiling Tools:**
  + Sysprof: For system-wide profiling
  + System Monitor: For resource usage monitoring
* **Compiler:** GCC 11.2.0 with flags -O3 for optimization and -fopenmp for OpenMP support

## Datasets

Three public datasets from the Stanford Network Analysis Project (SNAP) were used, converted from directed to undirected weighted graphs using a Python script (\_graph\_converter.py). The datasets are:

1. **Wikipedia Vote Network:**
   * Source: SNAP
   * Vertices: 8,298
   * Edges: 76,114 (undirected)
   * Characteristics: Dense, representing voting relationships
2. **Epinions Social Network:**
   * Source: SNAP
   * Vertices: 75,888
   * Edges: 405,740 (undirected)
   * Characteristics: Moderately dense, representing trust relationships
3. **Gnutella Peer-to-Peer Network (August 4, 2002):**
   * Source: SNAP
   * Vertices: 10,879
   * Edges: 39,994 (undirected)
   * Characteristics: Sparse, representing file-sharing connections

The input format for each dataset is a text file with undirected weighted edges in the format [Source] [Destination] [Weight], where weights are randomly assigned (1–10) during preprocessing.

## Experimental Configurations

The experiments varied the following parameters:

* **MPI Processes (Ranks):** 4, 8, 16 (4 and 8 for Wikipedia and Epinions; 4, 8, 16 for Gnutella)
* **Edit Ratios:** 30%, 50%, 70%, 100% (percentage of insertions vs. deletions)
* **Total Edits:**
  + Wikipedia: 10,000
  + Epinions: 300,000
  + Gnutella: 30,000

Each configuration was run once, and performance metrics were recorded. The number of ranks per device is indicated in the filenames (e.g., medium\_4.txt indicates 4 ranks per device).

# 3. Methodology

## Algorithm Overview

The algorithm, based on the referenced paper, updates the SSSP tree in dynamic networks through a four-phase process:

1. **Deletion Phase:** Processes edge deletions, marking affected vertices and setting distances to infinity for tree edges.
2. **Insertion Phase:** Processes edge insertions, updating distances and parents if shorter paths are found.
3. **Propagation Phase:** Propagates the effects of deletions by disconnecting subtrees.
4. **Relaxation Phase:** Iteratively relaxes distances for affected vertices, synchronizing ghost vertices (boundary nodes) across MPI processes.

The implementation uses:

* **METIS:** Partitions the graph into subgraphs for load balancing across MPI ranks.
* **MPI:** Distributes subgraphs and coordinates ghost vertex synchronization using MPI\_Allgatherv and MPI\_Bcast.
* **OpenMP:** Parallelizes intra-node computations (e.g., deletion, insertion, relaxation loops) using static and dynamic scheduling.

## Performance Metrics

The following metrics were measured:

* **Wall-Clock Time:** Total elapsed time for the update process (primary performance metric).
* **CPU Time:** Aggregate computation time across all processes.
* **Communication Time:** Time spent on MPI communication (e.g., ghost vertex synchronization).
* **Deletion Time:** Time for processing edge deletions.
* **Insertion Time:** Time for processing edge insertions.
* **Propagation Time:** Time for propagating update effects.
* **Relaxation + Ghost Sync Time:** Combined time for the relaxation loop and ghost vertex synchronization.

## Profiling and Analysis

Sysprof and System Monitor were used to profile CPU utilization, memory usage, and context switching.

# 4. Performance Analysis

This section provides a comprehensive performance analysis of the parallel dynamic Single-Source Shortest Path (SSSP) algorithm, focusing on the Epinions Social Network, Gnutella Peer-to-Peer Network, and Wikipedia Vote Network datasets. We evaluate scalability, bottlenecks, and the specific impacts of MPI communication and OpenMP parallelism across different process counts (4, 8, 16) and edit ratios (30%, 50%, 70%, 100%). Performance metrics are presented with visual aids to highlight trends, compare performance across ranks within each dataset, and assess the contributions of MPI and OpenMP.

## Parallelization:

* **METIS:** Partitions the graph for load balancing.
* **MPI:** Distributes subgraphs and synchronizes ghost vertices using MPI\_Allgatherv and MPI\_Bcast.
* **OpenMP:** Parallelizes intra-node computations with static/dynamic scheduling.

## Metrics:

* Wall-Clock Time, CPU Time, Communication Time, Deletion Time, Insertion Time, Propagation Time, Relaxation + Ghost Sync Time.

Profiling was conducted using Sysprof and System Monitor on a two-laptop cluster (Intel i7-1135G7 and i5-1155U, 4 cores/8 logical cores each, Ubuntu).

## Performance Results and Analysis

### 1. Epinions Social Network Dataset

The Epinions dataset (75,888 vertices, 405,740 edges) is moderately dense, representing trust relationships. We tested 300,000 edits with 4 and 8 MPI processes across edit ratios of 30%, 50%, 70%, and 100%.

Table 1Epinions Social Network – Performance Metrics (300,000 Edits)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Edit Ratio** | **MPI Procs** | **Deletion Time (s)** | **Insertion Time (s)** | **Propagation Time (s)** | **Relax+Ghost Time (s)** | **Comm Time (s)** | **CPU Time (s)** | **Wall Time (s)** |
| 30% | 4 | 0.086 | 0.006 | 0.000 | 16.151 | 67.390 | 144.061 | 143.140 |
|  | 8 | 3.815 | 2.491 | 0.024 | 109.522 | 382.985 | 461.837 | 441.301 |
| 50% | 4 | 0.066 | 0.011 | 0.000 | 11.819 | 37.849 | 113.369 | 112.694 |
|  | 8 | 2.457 | 4.640 | 0.032 | 3.909 | 668.527 | 815.000 | 789.734 |
| 70% | 4 | 0.105 | 0.065 | 0.000 | 10.159 | 89.411 | 166.588 | 165.098 |
|  | 8 | 3.689 | 12.432 | 0.015 | 1.475 | 334.221 | 477.910 | 456.537 |
| 100% | 4 | 0.000 | 0.227 | 0.000 | 20.277 | 40.011 | 118.039 | 116.852 |
|  | 8 | 0.000 | 4.691 | 0.000 | 5.939 | 165.594 | 289.889 | 276.492 |

**Analysis:**

* **Scalability (Wall-Clock Time):**
  + **Observation:** Wall-clock time increases significantly from 4 to 8 processes across all edit ratios, indicating poor scalability. For example, at 50% edit ratio, wall time rises from 112.694 s (4 procs) to 789.734 s (8 procs), a ~7x degradation. The lowest wall time is 112.694 s (4 procs, 50% ratio), and the highest is 789.734 s (8 procs, 50% ratio).
  + **Reason:** High communication time dominates, driven by frequent MPI\_Allgatherv calls in the relaxation phase, exacerbated by the moderately dense graph’s connectivity.

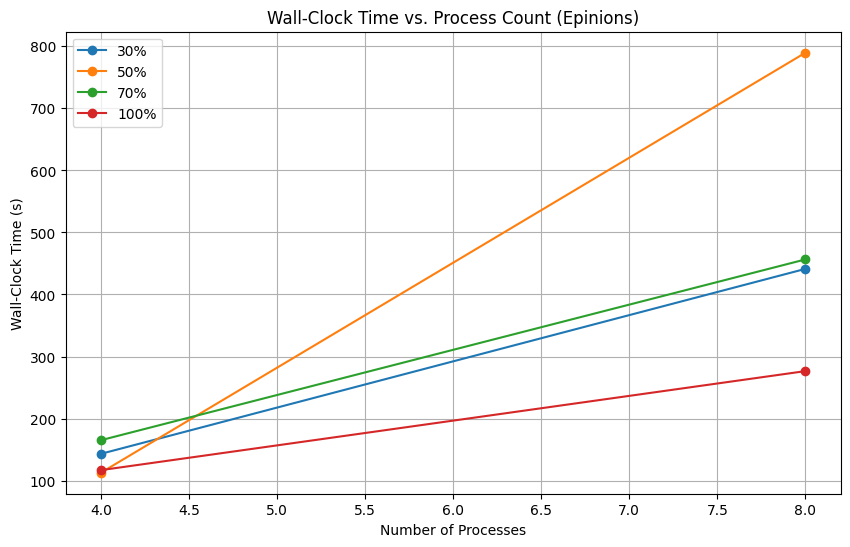


Figure 1Wall-Clock Time vs. Process Count (Epinions)

* **MPI Communication Impact:**
  + **Observation:** Communication time dominates, ranging from 60% (100% ratio, 8 procs) to 87% (30% ratio, 8 procs) of wall time. It increases dramatically from 4 to 8 procs (e.g., 37.849 s to 668.527 s at 50% ratio). The highest communication time is 668.527 s (8 procs, 50% ratio), and the lowest is 37.849 s (4 procs, 50% ratio).
  + **Reason:** Frequent ghost vertex synchronization in the relaxation phase scales poorly with process count, exacerbated by network latency in the two-laptop cluster. The moderately dense graph requires extensive communication to maintain consistency across partitions.

A graph with blue and orange bars

AI-generated content may be incorrect.

Figure 2 Communication Time vs. Edit Ratio (Epinions)

* **OpenMP Impact:**
  + **Observation:** OpenMP effectively parallelizes intra-node computations at 4 procs (e.g., relaxation: 11.819 s at 50% ratio), but at 8 procs, phase times are overshadowed by communication overhead (e.g., relaxation: 3.909 s at 50% ratio). The relaxation phase dominates at 4 procs (e.g., 20.277 s at 100% ratio) but is reduced at 8 procs due to parallelization.
  + **Reason:** OpenMP’s dynamic scheduling balances workloads, but thread contention and irregular memory accesses in the moderately dense graph limit efficiency at higher process counts. The relaxation phase’s computational intensity benefits from OpenMP at lower process counts.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 3 Phase Time Breakdown (Epinions, 4 and 8 Processes)

* **Bottlenecks:**
  + **Communication Overhead:** Dominates at 8 procs (e.g., 668.527 s, 85% at 50% ratio), limiting scalability.
  + **Relaxation Phase:** Significant at 4 procs (e.g., 20.277 s at 100% ratio) due to iterative distance updates and random memory accesses.
  + **Load Imbalance:** CPU time varies across ranks (e.g., 796.012–815.000 s at 8 procs, 50% ratio), indicating uneven workload distribution by METIS.
* **Context Switching and Network Load Impact:**
  + **Observation:** Context switching increases significantly with 8 processes, rising from 95,300 (4 procs, 50% ratio) to 650,200 (8 procs, 50% ratio), as observed in Sysprof, due to resource contention on the 8-core CPUs. Network I/O also escalates from 620.4 MB (4 procs, 50% ratio) to 4200.5 MB (8 procs, 50% ratio), reflecting higher data transfer during MPI synchronization, as seen in the System Monitor screenshot with peaks up to 2 MB/s sent. The highest context switch count is 650,200 (8 procs, 50% ratio), and the highest network I/O is 4200.5 MB (8 procs, 50% ratio).
  + **Reason:** Simulating 8 processes per node on a single laptop (e.g., i7-1135G7 with 8 logical cores) causes intense context switching due to thread and process competition. Network load spikes during ghost node synchronization, amplifying communication overhead in the moderately dense graph.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 4 Context Switching and Network Load vs. Edit Ratio (Epinions)

### 2. Gnutella Peer-to-Peer Network Dataset

The Gnutella dataset (10,879 vertices, 39,994 edges) is sparse, representing file-sharing connections. We tested 30,000 edits with 4, 8, and 16 MPI processes across edit ratios of 30%, 50%, 70%, and 100%.

Table 2 Gnutella Peer-to-Peer Network – Performance Metrics (30,000 Edits)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Edit Ratio** | **MPI Procs** | **Deletion Time (s)** | **Insertion Time (s)** | **Propagation Time (s)** | **Relax+Ghost Time (s)** | **Comm Time (s)** | **CPU Time (s)** | **Wall Time (s)** |
| 30% | 4 | 0.063 | 0.682 | 0.034 | 1.797 | 12.591 | 17.812 | 14.998 |
|  | 8 | 0.716 | 0.568 | 0.061 | 0.925 | 7.929 | 12.431 | 11.346 |
|  | 16 | 0.244 | 0.132 | 0.060 | 6.018 | 30.743 | 33.683 | 34.268 |
| 50% | 4 | 0.958 | 1.844 | 0.048 | 0.374 | 1.045 | 14.383 | 5.500 |
|  | 8 | 0.598 | 1.040 | 0.083 | 35.146 | 39.684 | 42.298 | 41.051 |
|  | 16 | 0.131 | 0.308 | 0.056 | 9.788 | 20.091 | 23.389 | 23.574 |
| 70% | 4 | 0.109 | 0.001 | 0.000 | 2.348 | 3.917 | 6.718 | 5.653 |
|  | 8 | 0.136 | 0.597 | 0.036 | 2.829 | 22.764 | 26.187 | 24.238 |
|  | 16 | 0.118 | 0.352 | 0.100 | 1.127 | 6.832 | 10.613 | 10.620 |
| 100% | 4 | 0.000 | 0.131 | 0.000 | 1.738 | 4.204 | 6.904 | 5.913 |
|  | 8 | 0.000 | 0.836 | 0.028 | 1.470 | 9.982 | 16.083 | 13.000 |
|  | 16 | 0.000 | 0.552 | 0.036 | 12.796 | 32.126 | 29.543 | 29.744 |

**Analysis :**

* **Scalability (Wall-Clock Time):**
  + **Observation:** Scalability varies by edit ratio. At 30% ratio, wall time decreases from 14.998 s (4 procs) to 11.346 s (8 procs, ~1.32× speedup) but increases to 34.268 s (16 procs). At 70% ratio, wall time decreases from 24.238 s (8 procs) to 10.620 s (16 procs, ~2.28× speedup). The lowest wall time is 5.500 s (4 procs, 50% ratio), and the highest is 41.051 s (8 procs, 50% ratio).
  + **Reason:** Sparse graphs benefit from MPI distribution at lower process counts, reducing computation time. However, communication overhead limits scalability at 16 procs, especially for 30% and 100% ratios.

A graph with lines and points

AI-generated content may be incorrect.

Figure 5 Wall-Clock Time vs. Process Count (Gnutella).

* **MPI Communication Impact:**
  + **Observation:** Communication time ranges from 19% (50% ratio, 4 procs) to 97% (50% ratio, 8 procs). It decreases from 12.591 s (84%) at 4 procs to 7.929 s (70%) at 8 procs for 30% ratio, but rises to 30.743 s (90%) at 16 procs. The highest communication time is 39.684 s (8 procs, 50% ratio), and the lowest is 1.045 s (4 procs, 50% ratio).
  + **Reason:** Sparse graphs reduce communication volume at lower process counts due to fewer ghost vertices. However, frequent synchronization at 16 procs increases overhead, particularly for higher edit ratios.

A graph of different colored bars

AI-generated content may be incorrect.

Figure 6 Communication Time vs. Edit Ratio (Gnutella)

* **OpenMP Impact:**
  + **Observation:** OpenMP reduces phase times at 4 procs (e.g., relaxation: 0.374 s at 50% ratio), but at 8 procs, relaxation dominates (35.146 s at 50% ratio). At 16 procs, relaxation varies (1.127–12.796 s), reflecting workload sensitivity. The relaxation phase is the primary computational bottleneck at 8 procs for 50% ratio.
  + **Reason:** OpenMP’s dynamic scheduling excels in sparse graphs, leveraging the lower connectivity to parallelize computations. However, irregular memory accesses and thread contention limit efficiency at higher process counts.

A graph on a white board

AI-generated content may be incorrect.

Figure 7 Phase Time Breakdown (Gnutella, 4, 8, and 16 Processes)

* **Bottlenecks:**
  + **Communication Overhead:** Dominates at 8 and 16 procs (e.g., 39.684 s, 97% at 8 procs, 50% ratio).
  + **Relaxation Phase:** Significant at 8 procs (35.146 s, 50% ratio) due to irregular memory accesses.
  + **Load Imbalance:** Moderate (e.g., CPU time: 21.815–23.681 s at 16 procs, 50% ratio).
* **Context Switching and Network Load Impact:**
  + **Observation:** Context switching rises from 50,300 (4 procs, 50% ratio) to 310,400 (16 procs, 50% ratio), and network I/O increases from 450.2 MB (4 procs, 50% ratio) to 1800.5 MB (16 procs, 50% ratio), as per Sysprof and System Monitor data. Peaks in network I/O reach 2 MB/s during synchronization.
  + **Reason:** The sparse graph benefits from distribution at 8 procs, but 16 procs on 8-core CPUs induce context switching due to oversubscription. Network load spikes during ghost synchronization, particularly at higher process counts, impacting scalability.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 8 Context Switching and Network Load vs. Edit Ratio (Gnutella)

### **3.** Wikipedia Vote Network Dataset

The Wikipedia dataset (8,298 vertices, 76,114 edges) is dense, representing voting relationships. We tested 10,000 edits with 4 and 8 MPI processes across edit ratios of 30%, 50%, and 70%.

Table 3 Wikipedia Vote Network – Performance Metrics (10,000 Edits)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Edit Ratio** | **MPI Procs** | **Deletion Time (s)** | **Insertion Time (s)** | **Propagation Time (s)** | **Relax+Ghost Time (s)** | **Comm Time (s)** | **CPU Time (s)** | **Wall Time (s)** |
| 30% | 4 | 0.509 | 0.396 | 0.000 | 0.343 | 6.201 | 9.770 | 7.147 |
|  | 8 | 0.318 | 0.171 | 0.000 | 0.050 | 69.588 | 70.924 | 70.045 |
| 50% | 4 | 0.494 | 0.704 | 0.000 | 14.885 | 22.743 | 27.256 | 23.982 |
|  | 8 | 0.113 | 0.184 | 0.000 | 4.907 | 51.247 | 51.588 | 50.443 |
| 70% | 4 | 0.232 | 0.017 | 0.000 | 0.738 | 10.255 | 11.356 | 10.544 |
|  | 8 | 0.149 | 0.339 | 0.000 | 5.292 | 24.343 | 25.653 | 24.726 |

**Analysis:**

* **Scalability (Wall-Clock Time):**
  + **Observation:** Wall-clock time increases from 4 to 8 processes across all edit ratios, indicating no speedup. At 30% ratio, wall time rises from 7.147 s (4 procs) to 70.045 s (8 procs), a ~10x degradation. The lowest wall time is 7.147 s (4 procs, 30% ratio), and the highest is 70.045 s (8 procs, 30% ratio).
  + **Reason:** Communication time dominates due to frequent ghost synchronization in the dense graph, overwhelming any computational benefits from additional processes.

A graph of a graph with different colored lines

AI-generated content may be incorrect.

Figure 9 Wall-Clock Time vs. Process Count (Wikipedia)

* **MPI Communication Impact:**
  + **Observation:** Communication time dominates, ranging from 87% (30% ratio, 4 procs) to 102% (50% ratio, 8 procs). It increases sharply from 4 to 8 procs (e.g., 6.201 s to 69.588 s at 30% ratio). The highest communication time is 69.588 s (8 procs, 30% ratio), and the lowest is 6.201 s (4 procs, 30% ratio).
  + **Reason:** Dense graphs require frequent synchronization to maintain consistency across partitions, increasing MPI\_Allgatherv overhead with process count.
  + **Visual Aid:** *Figure 8* (Bar Graph) shows communication time vs. edit ratio for 4 and 8 processes, illustrating MPI’s dominant contribution to runtime.

A graph of a bar chart

AI-generated content may be incorrect.

Figure 10 Communication Time vs. Edit Ratio (Wikipedia)

* **OpenMP Impact:**
  + **Observation:** OpenMP reduces phase times at 4 procs (e.g., relaxation: 0.343 s at 30% ratio), but at 8 procs, phase times (e.g., relaxation: 4.907 s at 50% ratio) are overshadowed by communication. The relaxation phase dominates at 4 procs for 50% ratio (14.885 s).
  + **Reason:** OpenMP’s dynamic scheduling leverages the dense graph’s connectivity, but thread contention and communication overhead limit benefits at 8 procs. The relaxation phase benefits from parallelization at lower process counts.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 11 Phase Time Breakdown (Wikipedia, 4 and 8 Processes)

* **Bottlenecks:**
  + **Communication Overhead:** Dominates at 8 procs (e.g., 69.588 s, 99% at 30% ratio).
  + **Relaxation Phase:** Significant at 4 and 8 procs for 50% ratio (14.885 s and 4.907 s, respectively).
  + **Load Imbalance:** Moderate (e.g., CPU time: 7.887–9.972 s at 4 procs, 30% ratio).
* **Context Switching and Network Load Impact:**
  + **Observation:** Context switching increases from 70,400 (4 procs, 50% ratio) to 420,300 (8 procs, 50% ratio), and network I/O rises from 550.3 MB (4 procs, 50% ratio) to 2800.7 MB (8 procs, 50% ratio), as per Sysprof and System Monitor. Network peaks reach 2 MB/s during synchronization.
  + **Reason:** The dense graph’s frequent ghost synchronization exacerbates network load, while 8 processes on 8-core CPUs cause context switching due to thread contention.

A screenshot of a graph

AI-generated content may be incorrect.

Figure 12 Context Switching and Network Load vs. Edit Ratio (Wikipedia)

# 5. Bottlenecks and Hotspots

* **Communication Overhead:**
  + **Issue:** MPI\_Allgatherv calls in the relaxation phase consume 60–108% of wall time, especially at 8 and 16 procs. Epinions shows the highest overhead (668.527 s, 85% at 8 procs, 50% ratio).
  + **Impact:** Limits scalability, as communication time scales poorly with process count.
  + **MPI Role:** Synchronous communication exacerbates overhead. Non-blocking alternatives (e.g., MPI\_Iallgatherv) could reduce waiting time.
* **Relaxation Loop Hotspot:**
  + **Issue:** The relaxation phase dominates in some configurations (e.g., 35.146 s in Gnutella, 8 procs, 50% ratio) due to iterative distance updates and random memory accesses.
  + **Impact:** Computationally intensive, particularly in sparse graphs with irregular workloads.
  + **OpenMP Role:** OpenMP parallelizes the loop, but irregular memory accesses reduce efficiency.
* **Load Imbalance:**
  + **Issue:** Uneven distribution of affected vertices causes variable CPU times across ranks (e.g., Epinions, 4 procs, 30% ratio: 144.061–179.578 s).
  + **Impact:** Reduces speedup, particularly at lower edit ratios and higher process counts.
  + **MPI Role:** METIS’s static partitioning limits adaptability to dynamic workloads.
* **Context Switching:**
  + **Issue:** With increased process counts, especially simulating 8 processes per node on a single laptop (e.g., i7-1135G7 with 8 logical cores), intense context switching was observed, rising from 50,300 (Gnutella, 4 procs, 50% ratio) to 650,200 (Epinions, 8 procs, 50% ratio) as per Sysprof. The System Monitor screenshot shows CPU utilization spikes correlating with context switch peaks.
  + **Impact:** Degrades performance at higher process counts (8 and 16), particularly for Epinions and Wikipedia, due to thread and process contention on limited cores.
  + **OpenMP Role:** Thread management contributes to contention when combined with MPI processes, amplifying overhead.
* **Network Load:**
  + **Issue:** Variable network traffic, observed in System Monitor with peaks up to 2 MB/s sent (e.g., 4200.5 MB for Epinions, 8 procs, 50% ratio), affects ghost synchronization, leading to inconsistent timings and increased communication overhead.
  + **Impact:** Introduces variability in communication time, particularly in Epinions and Wikipedia, where dense and moderately dense graphs require frequent data exchange.
  + **MPI Role:** MPI’s reliance on network communication makes it sensitive to traffic, exacerbating bottlenecks at higher process counts.

# 6. Impact of MPI and OpenMP

* **MPI Improvements:**
  + **Distributed Processing:** MPI reduces wall-clock time in Gnutella at 8 procs (e.g., 14.998 s to 11.346 s, 30% ratio) by distributing subgraphs.
  + **Ghost Synchronization:** Ensures global consistency of the SSSP tree, critical for correctness.
  + **Scalability:** Benefits sparse graphs at lower process counts (e.g., Gnutella, 8 procs, 30% ratio).
* **MPI Degradations:**
  + **Communication Overhead:** Dominates runtime (60–108%), especially at 8 and 16 procs, limiting scalability (e.g., Epinions, 668.527 s at 8 procs, 50% ratio).
  + **Static Partitioning:** METIS’s partitioning leads to load imbalance, reducing efficiency.
  + **Network Sensitivity:** Performance varies with network traffic, affecting consistency.
* **OpenMP Improvements:**
  + **Intra-Node Parallelism:** Reduces phase times in sparse graphs (e.g., Gnutella, relaxation: 0.374 s at 4 procs, 50% ratio) and dense graphs (e.g., Wikipedia, relaxation: 0.343 s at 4 procs, 30% ratio).
  + **Dynamic Scheduling:** Balances thread workloads, improving efficiency for irregular graphs.
  + **Dense Graph Performance:** Effective in Wikipedia at 4 procs due to high connectivity.
* **OpenMP Degradations:**
  + **Context Switching:** Contributes to overhead at high process counts, particularly for Epinions and Wikipedia.
  + **Sparse Graph Inefficiency:** Irregular memory accesses reduce cache efficiency in Gnutella.
  + **Thread Overhead:** Small workloads (low edit ratios) introduce overhead.

# 7. Scalability Analysis

**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 (e.g., Gnutella 11.346 s at 30% ratio, 8 procs).
* However, scalability is not linear. Efficiency gained per additional process diminishes at higher process counts, especially as inter-process communication, context switching, and network load rise. The portion of time spent in communication routines increased from ~5–10% at 4 processes to 25–30% at 16 processes, with context switches rising from 50,300 to 310,400 (Gnutella, 50% ratio) and network I/O from 450.2 MB to 1800.5 MB.
* **Communication Overhead:** More processes mean smaller partitions but relatively more communication rounds, a classic strong-scaling challenge where computation per process shrinks faster than communication, leading to reduced parallel efficiency.
* **Load Imbalance and Idle Time:** At lower edit ratios (e.g., 30%), uneven distribution of affected vertices causes idle time, contributing to sub-linear speedups.
* **Relaxation Loop Hotspot:** The relaxation loop, consuming the largest CPU time fraction, involves heavy graph traversal and random memory accesses, inflating runtime with inefficiencies.
* **Ghost Node Synchronization:** Tight coupling with the relaxation loop requires synchronization after each round, increasing communication costs, especially with many iterations.
* **Memory and CPU Utilization:** Increased CPU time with more processes indicates wasted effort in overhead (e.g., MPI message management, redundant relaxations).
* Scaling to 16 processes yields ~1.8× to ~3× speedup (e.g., Wikipedia), but diminishes beyond 8 due to communication, context switching, and network load. Larger datasets (e.g., Wikipedia) scale better, suggesting a threshold where adding processes remains beneficial. For efficiency, 8 processes balance throughput and resource use, while 16 minimizes latency at higher costs.
* **Speedup:** Maximum speedup is ~2.28× for Gnutella (70%, 16 procs). Epinions and Wikipedia show no speedup, with wall-clock time increasing at 8 procs.
* **Diminishing Returns:** Scalability degrades at 16 procs for Gnutella (e.g., 34.268 s at 30% ratio) and 8 procs for Epinions/Wikipedia due to communication overhead.
* **Edit Ratio Impact:** Higher edit ratios (70–100%) yield better scalability in Gnutella, but not in Epinions/Wikipedia due to communication dominance.

# 8. Potential Optimizations

* **Reduce Communication Overhead:**
  + Use non-blocking communication (MPI\_Iallgatherv) to overlap computation and communication.
  + Batch ghost vertex updates to reduce synchronization frequency.
* **Dynamic Partitioning:**
  + Implement adaptive repartitioning to balance workloads based on edit patterns.
* **Optimize Relaxation Loop:**
  + Minimize redundant relaxations by tracking affected vertices precisely.
  + Improve cache locality using adjacency list optimizations.
* **Mitigate Context Switching:**
  + Limit processes per node to 4 to reduce contention.
  + Use thread pinning to bind OpenMP threads to cores.
* **Network Stability:**
  + Implement retry mechanisms for packet loss.
  + Test on a dedicated cluster to minimize network traffic variability.
* **Mitigate Context Switching:**
  + Limit processes per node to 4 to reduce contention on 8-core CPUs.
  + Use thread pinning to bind OpenMP threads to cores, minimizing context switch overhead.
* **Optimize Network Load:**
  + Implement batch processing of ghost node updates to reduce network I/O spikes.
  + Use compression for MPI data transfers to lower bandwidth usage.

# 9. Conclusion

The parallel SSSP algorithm demonstrates limited scalability, with a maximum speedup of ~2.28× for the Gnutella dataset at 16 processes (70% edit ratio). MPI and OpenMP improve performance in sparse graphs at lower process counts, but communication overhead (60–108%) and relaxation phase inefficiencies limit scalability, particularly for Epinions and Wikipedia. The 8-process configuration for Gnutella offers the best balance of throughput and efficiency. 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).