Aalto University
School of Electrical
Engineering

Master's Programme in Computer, Communication and Information Sciences

MPC in practice

MPC in practice

Tianxing Wu

Master's Thesis
2025

Aalto University
School of Electrical Engineering

Author Tianxing Wu

Title MPC in practice — MPC in practice

Degree programme Computer, Communication and Information Sciences

Major Algorithms

Supervisor Prof. Jukka Suomela

Advisor Hossein Vahidi

Date 10 July 2024 **Number of pages** 40 **Language** English

Abstract

This thesis investigates the performance of Congested Clique algorithms on modern supercomputers, specifically Puhti, a Finnish supercomputer hosted by CSC.
The study focuses on a representative algorithm, $MST\_\log\log n$, which is implemented using two different programming paradigms: OpenMPI (Message Passing Interface) for CPUs and CUDA (Compute Unified Device Architecture) for GPUs. The goal is to evaluate the compatibility of modern computing architectures with the Congested Clique model.
The evaluation examines several aspects, including how distributed computation and communication in the Congested Clique model are translated to supercomputer architectures, the overhead introduced by this translation, and the extent to which the theoretical time complexity of the algorithm holds in practice.
MPI is a standardized and portable message-passing library interface widely used for parallel computing, while CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs.

Keywords Distributed computing, Supercomputing, Congested clique, MST, Puhti, MPI, CUDA, MPC

Preface

I want to thank Professor Jukka Suomela and my advisor Hossein Vahidi for their guidance.

I also want to thank my friends for keeping me sane and alive.

Otaniemi, 10 July 2024

Tianxing Wu

Contents

Abstract 3

Preface 4

Contents 5

Symbols and abbreviations 7

1 Introduction 8
- 1.1 Methodology . . . 8
- 1.2 Objective . . . 9

2 Preliminaries 11
- 2.1 Congested clique (also known as CLIQUE) . . . 11
- 2.2 $MST\_\log\log n$ . . . 11
- 2.3 The supercomputer Puhti . . . 12
- 2.4 Parallel Computing Frameworks: MPI and CUDA . . . 13
  - 2.4.1 Message Passing Interface (MPI) . . . 13
  - 2.4.2 CUDA . . . 14

3 Congested Clique with MPI 15
- 3.1 Communication . . . 15
- 3.2 Simplifications and Optimizations in the Implementation . . . 17
  - 3.2.1 Simplification in the First Phase . . . 17
  - 3.2.2 Direct Communication to rank-0 . . . 18
  - 3.2.3 Union-Find for Cycle Detection . . . 18
- 3.3 Vertex Partitioning and Graph Generation . . . 18

4 Performance with MPI 20
- 4.1 Conclusion overview . . . 20
- 4.2 Performance . . . 20
- 4.3 Comparison with prim's algorithm . . . 23
- 4.4 Python vs C++ . . . 25

5 Congested clique with CUDA 27
- 5.1 Communication . . . 27
- 5.2 Memory read and write . . . 29
- 5.3 Complete and sparse graphs . . . 30
- 5.4 Best practices . . . 30
- 5.5 Profiling . . . 31

6 Performance with CUDA **32**
6.1 Conclusion overview 32
6.2 Performance 32
6.3 Complete vs sparse graph 34
6.4 Comparison with Prim's on single threaded CPU 35

7 Conclusion **37**

8 Future work **38**

References **39**

Symbols and abbreviations

Abbreviations

$MST\_\log\log n$	The algorithm described in [1]
MPI	Message Passing Interface
CUDA	Compute Unified Device Architecture
HPC	High-performance computing
AI	Artificial intelligence
GPU	Graphics processing unit
CPU	Central processing unit
MST	Minimum spanning tree
SLURM	Simple Linux Utility for Resource Management
SM	Streaming multiprocessor
SP	Streaming processor

1 Introduction

The Congested Clique model, as described in [2], is a framework for distributed computing. It consists of n computing nodes arranged in a clique network, where each node is directly connected to all other nodes. In this model, each node in the network corresponds to a node in an input graph G, and the nodes collaborate to solve a problem on G.

The efficiency of the Congested Clique model is measured by the number of communication rounds required for an algorithm to converge. In each round, nodes perform local computations and exchange $O(\log n)$ bits with their neighbors. The model emphasizes communication over local computation, as communication is considered the primary bottleneck in distributed computing. Local computation is assumed to be free, with each node having unlimited computational power and memory.

This study explores the Congested Clique model in the context of modern distributed computing hardware, specifically supercomputers. Supercomputers are high-performance machines designed to solve large-scale scientific and engineering problems. Unlike traditional computers, supercomputers utilize multiple processing units [13]. Their distributed nature makes them well-suited for executing Congested Clique algorithms.

To evaluate the compatibility of modern computing architectures with the Congested Clique model, we implement a representative algorithm, $MST\_\log\log n$, on a supercomputer. The implementation is carried out using two different programming paradigms: OpenMPI (Message Passing Interface) for CPUs and CUDA (Compute Unified Device Architecture) for GPUs.

1.1 Methodology

The algorithm $MST\_\log\log n$ from [1] is selected as the representative algorithm for this study. Its representativeness is discussed in detail in section 2.2. $MST\_\log\log n$ solves the Minimum Spanning Tree (MST) problem in $O(\log\log n)$ communication rounds in the Congested Clique model. The MST problem involves finding a spanning tree of a graph G with the smallest possible total weight. We demonstrate that $MST\_\log\log n$ is representative of algorithms designed for the Congested Clique model, and its properties generalize to the entire category of such algorithms.

To evaluate its performance, $MST\_\log\log n$ is implemented on Puhti, a Finnish supercomputer hosted by CSC [3]. The implementation is carried out using two distinct programming paradigms:

– Message Passing Interface (MPI) for CPUs.

– Compute Unified Device Architecture (CUDA) for NVIDIA GPUs.

The choice of both CPUs and GPUs reflects the diversity of modern computing hardware. These platforms follow different programming paradigms and have distinct hardware constraints. While implementing $MST\_\log\log n$, we adhere to best practices and optimize the implementation without altering the algorithm itself.

In the Congested Clique model, we assume that the graph to be solved is the same as the network graph, which is a complete graph. When implementing $MST\_\log\log n$, we separate the network graph and the problem graph G into two distinct graphs. Each node in the model is represented as a computing unit in our implementation, which can either be a physical CPU core/thread or a virtual computing resource assigned to the node. Nodes are not necessarily separate machines connected by a network; they can reside on a single computing core or thread. However, we ensure that each node can communicate with any other node in the network.

We do not allocate physically separate machines to the nodes in this work for two reasons: (1) it would be a waste of resources for a single node, and (2) the supercomputer does not have enough GPU/CPU nodes to allocate one per graph node. As we are dealing with large graphs, the graphs used in this study are complete graphs with n vertices, containing $\frac{n(n-1)}{2}$ edges. The focus is on large graphs, with n scaling up to 10^5 and the number of edges reaching up to 10^{11}.

1.2 Objective

The purpose of this work is to explore the connections between the Congested Clique model and modern distributed computing hardware, software, and tools. We present our approach to simulating Congested Clique algorithms on HPC hardware by making informed design decisions regarding the following aspects:

– How communication is handled across different programming paradigms.

– How workload is allocated to nodes to maximize hardware utilization.

– How communication is synchronized in a round-based manner.

The implementation of $MST\_\log\log n$ is detailed in section 3 for MPI and section 5 for CUDA.

We evaluate the performance of the simulation and analyze how hardware limitations affect the algorithm's performance in terms of memory, computation, and network constraints in section 4 and section 6. Specifically, we address the following questions:

– What assumptions of the Congested Clique model lead to performance bottlenecks.

– Under what settings do these bottlenecks appear.

– Under what conditions does the algorithm perform as expected by the Congested Clique model.

We compare the performance of $MST\_\log\log n$ with its sequential alternative in terms of:

– Capability to process large graphs with a high number of vertices and edges.

- Wall clock time when processing the same graphs.

Finally, we discuss the scalability of $MST\_\log\log n$:

- Strong and weak scaling: How adding more nodes speeds up computation and which hardware components become bottlenecks as scaling continues.
- Time composition during scaling: How the runtime is distributed between MPI communication and local computation. Does the time complexity align with theoretical expectations?

2 Preliminaries

2.1 Congested clique (also known as CLIQUE)

The Congested Clique model is a theoretical framework for distributed computing, where n nodes are fully connected, forming a complete graph. Each node is assigned a unique identifier that requires $O(\log n)$ bits for representation. The network operates in synchronized communication rounds, where all nodes begin and complete each round simultaneously. During a single round, each node performs local computations and communicates with its neighbors by sending at most $O(\log n)$ bits to each of them. Communication is achieved through unicast, allowing a node to send distinct messages to different neighbors over its links within the same round. This model emphasizes communication efficiency, as the nodes are assumed to have unlimited computational power and memory resources. Consequently, the primary metric for evaluating algorithm performance in the Congested Clique model is the number of communication rounds required, rather than the time spent on local computation. [2]

2.2 $MST\_\log\log n$

$MST\_\log\log n$ is an algorithm developed by Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg in 2005 [1]. It solves the Minimum Spanning Tree (MST) problem in $O(\log\log n)$ communication rounds within the Congested Clique model.

The algorithm begins by partitioning the nodes of the graph G into disjoint clusters. Initially, each vertex v_i forms its own singleton cluster, i.e., $\mathcal{F}^0 = \{F_1^0, \ldots, F_n^0\}$, where $F_i^0 = \{v_i\}$ for $1 \leq i \leq n$. The clusters are iteratively merged in subsequent phases until only a single cluster remains, at which point the algorithm terminates.

In each phase $k \geq 0$, the size of the clusters grows quadratically. This ensures that the total number of phases is at most $O(\log\log n)$. Each phase consists of $O(1)$ communication rounds, during which the algorithm identifies and merges clusters using minimum-weight edges.

By combining the quadratic growth of clusters with the constant number of communication rounds per phase, $MST\_\log\log n$ achieves a total runtime of $O(\log\log n)$ communication rounds, making it highly efficient for solving the MST problem in the Congested Clique model.

In each phase, $MST\_\log\log n$ executes the following steps:

1. (a) Compute the minimum-weight edge $e(v, F')$ that connects v to any node in F' for all clusters $F' \neq F$.
   (b) Send $e(v, F')$ to $\ell(F')$ for all clusters $F' \neq F$.

2. If $v = \ell(F)$, then:
   (a) Using the messages received from Step 1, compute the lightest edge between F' and F for every other cluster F'.
   (b) Perform Procedure Cheap-Out locally. This procedure:

  * Selects a set $\mathcal{A}(F)$ containing the N cheapest edges that go out of F to $N = |F|$ distinct clusters.
  * Appoints for each such edge e a guardian node $g(e)$ in F, ensuring that each node in F is appointed as a guardian for at most one edge.

3. Let $e' \in \mathcal{A}(F)$ be the edge for which v was appointed as guardian, i.e., such that $g(e') = v$. Send e' to v_0, the node with the smallest ID in the graph.

4. If $v = v_0$, then:

   (a) Perform Procedure Const-Frags locally. This procedure computes E^k, the new set of edges to add.

   (b) For each edge $e \in E^k$, send a message to $g(e)$.

5. If v receives a message from v_0 indicating that $e \in E^k$, then v sends e to all nodes in the graph.

6. Each node adds all edges in E^k and updates T.

$MST\_\log\log n$ is not the only algorithm designed for the MST problem in the Congested Clique model. Ghaffari and Parter proposed an algorithm that solves the MST problem in $O(\log^* n)$ rounds [15], while Jurdzinski and Nowicki introduced a constant-round MST algorithm [14]. Additionally, Nowicki later developed a deterministic constant-round MST algorithm [24]. These advancements demonstrate the ongoing efforts to optimize MST algorithms in the Congested Clique model. However, we have chosen to use $MST\_\log\log n$ for the following reasons:

– $MST\_\log\log n$ is a representative Congested Clique algorithm. Its properties generalize to the entire category of such algorithms.

– $MST\_\log\log n$ is simple and easy to understand, saving time for studying and implementing the algorithm. This makes it suitable for implementation using different programming paradigms.

– Within the range of graph sizes considered (up to millions of edges), there is no significant round reduction compared to [15] and [14] (both require fewer than 4 rounds).

In the following sections, we will delve into the details of implementing $MST\_\log\log n$ using MPI and CUDA, highlighting the challenges and optimizations specific to each framework.

2.3 The supercomputer Puhti

Puhti is a supercomputer managed by CSC, designed for high-performance computing (HPC) and artificial intelligence (AI) applications. It employs the Slurm workload manager [6] to allocate computing resources across its nodes. Each node is equipped with two Intel Xeon Gold 6230 processors (Cascade Lake architecture), featuring 20 cores per processor running at 2.1 GHz. These processors support AVX-512 vector instructions and VNNI instructions optimized for AI inference workloads. With 20 cores per processor, a node can run up to 40 Slurm tasks, supported by a total of 192 GiB of memory [5].

Puhti's "M" cluster consists of 484 nodes, interconnected using Mellanox HDR InfiniBand. Each node is linked via a 100 Gbps HDR100 connection. The network topology follows a fat-tree structure with a blocking factor of approximately 2:1, meaning there are twice as many ports going down as going up [5] [16].

In addition to CPU nodes, Puhti includes an AI-dedicated partition with 80 GPU nodes, achieving a peak performance of 2.7 petaflops. Each GPU node contains two Intel Xeon Gold 6230 processors (20 cores each at 2.1 GHz) and four Nvidia Volta V100 GPUs, each with 32 GB of memory. These nodes are equipped with 384 GiB of main memory and 3.6 TB of local storage, making them well-suited for GPU-intensive workloads that scale across multiple nodes. The AI partition uses a dual-rail HDR100 network, providing an aggregate bandwidth of 200 Gbps with a non-blocking fat-tree topology, enabling efficient communication across nodes [5].

In the next sections, we will explore how the computational resources of Puhti are utilized to implement and evaluate $MST\_\log\log n$ using both MPI and CUDA. These implementations address the practical challenges of simulating the Congested Clique model on modern hardware.

2.4 Parallel Computing Frameworks: MPI and CUDA

Message Passing Interface (MPI) and CUDA (Compute Unified Device Architecture) are two widely used frameworks for parallel computing. While MPI is designed for distributed systems, CUDA is tailored for GPU-based parallelism. In this section, we discuss how these frameworks are utilized to implement $MST\_\log\log n$ on Puhti.

2.4.1 Message Passing Interface (MPI)

MPI is a standardized and portable message-passing library interface, widely used for parallel computing. It allows processes running on distributed systems to communicate with each other by sending and receiving messages. MPI supports both point-to-point communication (between individual pairs of processes) and collective communication (among multiple processes), enabling efficient data sharing and synchronization.

MPI is extensively used in high-performance computing (HPC) to coordinate computation across nodes in a cluster, leveraging both shared and distributed memory architectures [7]. In our implementation, each MPI rank simulates a node in the Congested Clique model, and communication between ranks is handled using MPI's collective communication functions. This approach ensures scalability and efficient utilization of Puhti's CPU resources.

2.4.2 CUDA

CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs. In CUDA, the CPU is referred to as the host, while the GPU is the device. CUDA programs typically consist of both host and device code.

A kernel function is a function that runs on the device and can be executed by many threads in parallel. Each thread performs computations independently, allowing for massive parallelism on the GPU.

Memory allocation on the device is performed using the `cudaMalloc` function, which takes a pointer and the size of memory to be allocated in bytes:

```
cudaMalloc((void **)&devicePointer, size);
```

Data can be copied between the host and device using `cudaMemcpy`, which specifies the source, destination, size, and direction (host-to-device or device-to-host). For example:

```
cudaMemcpy(devicePointer, hostPointer, size, cudaMemcpyHostToDevice);
```

To launch a kernel function, specify the number of threads per block and blocks per grid using the syntax `kernel«<gridDim, blockDim»>(args);` where `gridDim` and `blockDim` define the grid and block dimensions:

```
kernelFunction<<<numBlocks, threadsPerBlock>>>(arguments);
```

3 Congested Clique with MPI

We implement the $MST\_\log\log n$ algorithm using OpenMPI, an open-source implementation of the Message Passing Interface (MPI), on Puhti's Intel Xeon Gold 6230 CPU compute nodes. Each compute node has 40 cores and 192 GiB of memory, enabling parallel execution of Slurm tasks. In MPI terminology, a "rank" corresponds to a Slurm task, and each core provides one Slurm task. Each task is allocated an even portion of the memory on the compute node. The term "rank-0" refers to the Slurm task with index 0.

Since the graphs to be solved can have up to 10^5 vertices, which significantly exceeds the number of available cores, we utilize both multiple vertices per MPI rank and multiple compute nodes for communication. Assigning multiple vertices per rank means that each thread is responsible for a subset of vertices. Each thread handles the computation, memory, and communication for the vertices assigned to it and performs all operations required for those vertices.

The pseudo-code for the MPI-based implementation of $MST\_\log\log n$ is provided in algorithm 1.

3.1 Communication

Each Slurm task has its dedicated memory space and computing power, which is shared among all the vertices that the task is responsible for. Communication between tasks is handled by OpenMPI, an open-source implementation of the Message Passing Interface (MPI). Each task manages a batch of vertices with continuous indices, maintaining all edges adjacent to these vertices and their states. The task sends MPI messages on behalf of the vertices it represents.

The communication steps in each phase of $MST\_\log\log n$ can be categorized into three types:

- **All-to-All Communication (Steps 1 and 2):** All nodes send messages to all other nodes.
- **Gather Communication (Step 3):** Rank-0 gathers messages from all other nodes.
- **Broadcast Communication (Steps 5 and 6):** Rank-0 sends messages to all other nodes.

These communication patterns involve all nodes in the graph, making MPI collective communication functions well-suited for their implementation. Collective communication functions, such as `MPI_Alltoall`, `MPI_Gather`, and `MPI_Broadcast`, are designed to efficiently handle such operations across multiple ranks [7].

Algorithm 1 MINIMUM SPANNING TREE with MPI

Require: Number of vertices n, adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$.

On each rank: Rank index i, number of vertices per rank n_r, and adjacency sub-matrix $\mathbf{A}[i \times n_r : (i+1) \times n_r, :] \in \mathbb{R}^{n_r \times n}$.

```
 1: $vertex\_lo \leftarrow i \times n_r$
 2: $vertex\_hi \leftarrow (i+1) \times n_r$
 3: $T \leftarrow \emptyset$
 4: while there exists $v \notin T$ do
 5:     if First phase then                                         ▷ Step 1
 6:         for $v = vertex\_lo$ to $vertex\_hi$ do
 7:             Find the minimum edge $e(v, v')$ connected to $v$
 8:         end for
 9:         MPI_Gather all $e(v, v')$ to rank-0
10:     else                                                        ▷ Step 2
11:         for $v = vertex\_lo$ to $vertex\_hi$ do
12:             For each cluster $F' \neq F$, compute $e(v, F') \leftarrow \min_{v' \in F'}\{e(v, v')\}$
13:             Add $e(v, F')$ to the send buffer
14:         end for
15:         MPI_Alltoall all $e(v, F')$ to leaders $l(F')$             ▷ Step 3
16:         for $v = vertex\_lo$ to $vertex\_hi$ do
17:             if $v = l(F)$ then
18:                 For each cluster $F' \neq F$, compute $e(v, F') \leftarrow \min_{v' \in F'}\{e(v, v')\}$
19:                 Perform CHEAP-OUT and add the edge $e$ to the send buffer
20:             end if
21:         end for
22:         MPI_Gather all $e$ to rank-0
23:     end if
24:     if on rank-0 then                                           ▷ Step 4
25:         Perform CONSTRUCT-FRAGS and update $T$
26:         MPI_Broadcast the updated $T$ to all ranks
27:     end if                               ▷ Step 5: Updating Local State
28:     Each rank updates $T$ with the received data
29: end while
30: Return the spanning tree $T$
```

Synchronization between ranks is typically achieved using `MPI_Barrier`. However, in the implementation of $MST\_\log\log n$, all communication is performed using collective operations, which are inherently synchronous. These operations ensure that all participating ranks reach the collective call before proceeding, effectively acting as implicit synchronization points [7].

At the start of $MST\_\log\log n$, each rank receives its input, which includes a list of vertices it is responsible for and all adjacent edges. The MPI communication in each step is implemented as follows:

- **Step 1:** `MPI_Alltoall` is used to exchange minimum-weight edges between nodes.
- **Step 2:** `MPI_Alltoall` is used again to send minimum-weight edges to cluster leaders.
- **Step 3:** `MPI_Gather` is used to collect edges from cluster leaders to rank-0.
- **Step 4:** Local computation is performed on rank-0 to construct clusters.
- **Step 5:** `MPI_Broadcast` is used to distribute updated cluster information from rank-0 to all ranks.
- **Step 6:** Local computation is performed on each rank to update the spanning tree.

In Step 3, the original algorithm specifies that cluster leaders scatter messages to guardians, who then forward the messages to rank-0. However, in our implementation, this step is simplified by directly gathering edges on rank-0 using `MPI_Gather`. OpenMPI internally optimizes this operation using a tree-based gathering pattern [18] [17], which achieves the same goal as the original step. Since the allocation of nodes to ranks and the network topology can affect traffic balancing, it is more efficient to rely on MPI's built-in optimizations for this step.

3.2 Simplifications and Optimizations in the Implementation

The implementation of $MST\_\log\log n$ includes several simplifications and optimizations to improve performance and align with practical considerations. These modifications are detailed below.

3.2.1 Simplification in the First Phase

To streamline the algorithm and eliminate unnecessary computation and communication, we simplify the first phase of $MST\_\log\log n$. At the beginning of the first phase, no spanning tree or clusters have been formed yet, and each vertex is treated as its own cluster. In the original algorithm, the first three steps involve: 1. Each vertex v sending all its adjacent edges $e \in E_v$ to the corresponding vertices. 2. Each cluster leader processing the received edges to identify the minimum-weight edge connecting to other clusters. 3. The cluster leader scattering the selected edges to guardians, who then forward them to rank-0.

To simplify this process, we bypass steps 1–3 in the first phase. Instead, rank-0 directly gathers the minimum-weight edge adjacent to each vertex. This is achieved using `MPI_Gather`, which collects the minimum-weight edges from all ranks. By doing so, we achieve the same result as the original steps but with significantly reduced complexity and communication overhead.

This simplification is particularly effective because, during the first phase, each vertex is treated as an individual cluster, resulting in the highest communication requirements. Directly gathering the minimum-weight edges minimizes the communication overhead and streamlines the process.

3.2.2 Direct Communication to rank-0

In the original algorithm, during the third step, each cluster leader sends the minimum-weight edge to the guardians of the cluster, who then forward the edge to rank-0. This step is designed to avoid exceeding bandwidth limits in the Congested Clique model, where all-to-one communication can become a bottleneck.

However, in our MPI implementation, we simplify this step by having each cluster leader directly send the minimum-weight edge to rank-0. This approach leverages MPI's optimized collective communication functions, such as `MPI_Gather`, which internally use efficient tree-based patterns to handle such communication. As a result, this direct communication is more efficient in practice and avoids the additional complexity of involving guardians.

3.2.3 Union-Find for Cycle Detection

When adding edges to the Minimum Spanning Tree (MST), it is essential to ensure that no cycles are created. To efficiently perform this check, we use the union-find data structure. This structure allows us to determine whether two vertices belong to the same connected component by checking their roots. If two vertices have the same root, they are already connected, and adding the edge would create a cycle. When an edge is added to the MST, the union-find structure merges the sets containing the two vertices, ensuring efficient updates and queries.

By incorporating these simplifications and optimizations, the implementation achieves better performance while maintaining the correctness of the algorithm.

3.3 Vertex Partitioning and Graph Generation

As mentioned earlier, the number of vertices exceeds the available Slurm tasks. To address this, we partition the vertices among the ranks. Each rank is responsible for a subset of vertices, handling all the adjacent edges for those vertices.

The edges are represented as an adjacency matrix, implemented as a list of lists. Each element in the matrix corresponds to a randomly generated weight for the edge.

The matrix is partitioned such that each rank holds a contiguous subset of the matrix rows. Each rank is responsible for computation, communication, and storing the adjacent edges of the vertices it owns.

Graph generation is a memory-intensive process. Generating the entire graph on rank-0 leads to underutilization of memory resources allocated to other ranks, which limits the size of the graph that can be processed. A naive approach would involve generating the entire adjacency matrix on rank-0 and then using `MPI_Scatter` to distribute the matrix rows to the destination ranks. However, this method is highly memory-intensive for rank-0, requiring $O(n^2)$ memory, which becomes infeasible for large graphs.

To address the memory limitations, we employ distributed graph generation. This approach divides the task of graph generation among all ranks, ensuring that no single rank becomes a bottleneck. It not only reduces memory usage on rank-0 but also leverages the parallelism of the distributed system to accelerate graph generation.

The graph is generated as a triangular matrix with a zero diagonal, as the adjacency matrix is symmetric along the diagonal. Each rank is responsible for generating the portion of the triangular matrix corresponding to its assigned vertices. Specifically: - Each rank generates the weights for the edges in its portion of the triangular matrix. - The diagonal entries are set to zero to ensure no self-loops.

Once the triangular matrix is generated, the missing entries on the other side of the diagonal are filled using `MPI_Alltoall`. This ensures that all ranks have a complete view of the adjacency matrix for their assigned vertices. By distributing the graph generation process, we achieve scalability and efficient memory utilization, enabling the simulation of large graphs on distributed systems.

A single compute node can launch up to 40 tasks, with a total memory limit of 192 GiB. For a node running n tasks, each task is allocated $192/n$ GiB of memory. This memory constraint determines the maximum number of edges that can be stored locally. The distributed graph generation approach ensures balanced memory usage across all ranks, avoiding bottlenecks during graph creation.

4 Performance with MPI

4.1 Conclusion overview

When simulating the Congested Clique model with MPI, the parallelization from CPUs is limited. Partitioning the vertices among the tasks is inevitable. When there are only one or a few vertices assigned to each task, as predicted by the Congested Clique model, the communication time becomes the bottleneck. However, as we attempt to process larger and larger graphs with limited task resources, the communication time no longer dominates the total runtime. Instead, the computation time on each task becomes the bottleneck.

We compare the performance of $MST\_\log\log n$ with Prim's algorithm. For the same graph, $MST\_\log\log n$ requires more computation and memory than Prim's algorithm. However, by increasing the number of tasks, $MST\_\log\log n$ outperforms Prim's algorithm in terms of wall clock time. Additionally, $MST\_\log\log n$ is capable of processing larger graphs than Prim's algorithm. The bottleneck for processing larger graphs is the memory required for MPI messages.

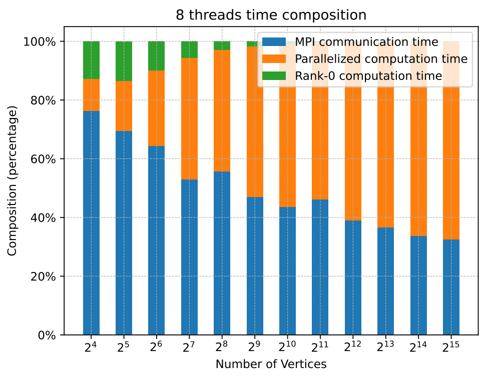
The number of rounds is as predicted: only $O(\log\log n)$. From a runtime perspective, Python is significantly slower than C++. The slowdown comes from both MPI communication time and computation time. The slowdown in MPI is primarily due to the large memory footprint of Python's number objects. The computation slowdown is expected, as Python is an interpreted language. With Python, the algorithm becomes more memory-bound, requiring significantly more memory to construct MPI messages compared to the C++ implementation.

4.2 Performance

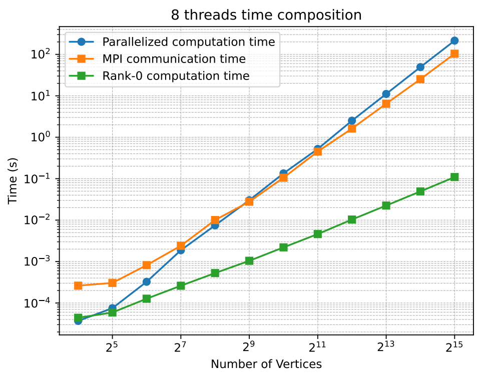
We executed the $MST\_\log\log n$ implementation (algorithm 1) using OpenMPI on Puhti and benchmarked it with complete graphs. The primary objectives of this evaluation were as follows:

- To determine the best practices for partitioning vertices when the number of tasks is limited.
- To analyze how the time composition changes under different partitioning strategies and whether these changes align with the predictions of the Congested Clique model.
- To evaluate the algorithm's capacity—specifically, how many more edges it can process compared to a single-node implementation of Prim's algorithm.
- To assess whether the algorithm can achieve better runtime performance than Prim's algorithm when processing the same graph.

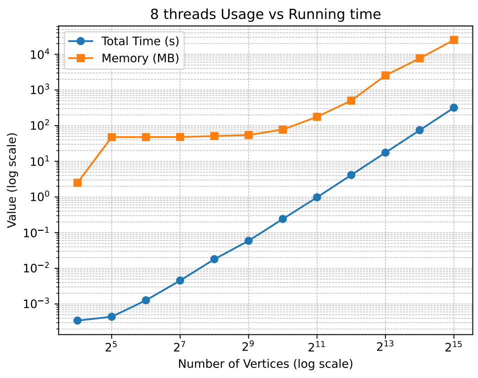
As the number of vertices per rank increases, the number of rounds grows as predicted by $O(\log\log(n))$. However, the runtime does not benefit significantly from the slower round growth, as shown in fig. 1c and fig. 1d.



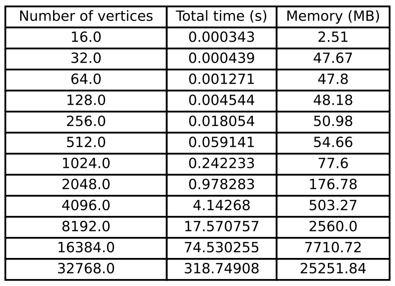
(a) 8 threads time composition



(b) 8 threads time



(c) 8 threads time and memory usage



Number of vertices	Total time (s)	Memory (MB)
16.0	0.000343	2.51
32.0	0.000439	47.67
64.0	0.001271	47.8
128.0	0.004544	48.18
256.0	0.018054	50.98
512.0	0.059141	54.66
1024.0	0.242233	77.6
2048.0	0.978283	176.78
4096.0	4.14268	503.27
8192.0	17.570757	2560.0
16384.0	74.530255	7710.72
32768.0	318.74908	25251.84

(d) 8 threads time and memory usage table

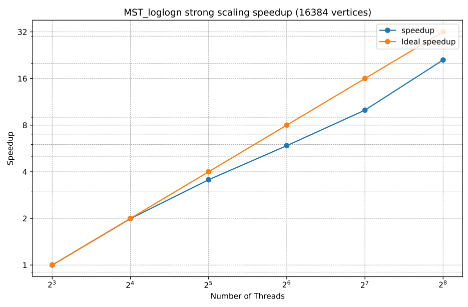
Figure 1: $MST\_\log\log n$ with 8 threads

When there are only one or a few vertices assigned to each Slurm task, as predicted by the Congested Clique model, the communication time constitutes the majority of the total runtime. Additionally, the rank-0 computation time accounts for a significant fraction of the total runtime, meaning that at least 10% of the time, all other tasks remain idle, waiting for rank-0 to complete its computation. However, as we process larger and larger graphs, the communication time is no longer the dominant factor. Instead, the parallelized computation time starts to dominate, while the rank-0 computation time diminishes, as shown in fig. 1a.

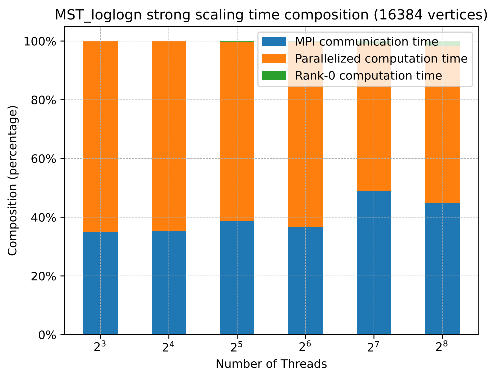
The parallelized computation time includes all operations other than communication and rank-0 computation. Calculating the minimum-weight edges is a memory-bound task and constitutes a significant portion of the parallelized computation time. The memory required for the graph grows quadratically with the number of vertices, which means the time spent preparing MPI messages also grows accordingly. Further optimization of this algorithm would require more memory-efficient data structures.

In the 2^{15}-vertex example, the expected memory usage is $2^{15} \times 2^{15} \times 4 = 2^{31}$ bytes = 4096 MB (assuming double precision). However, the actual memory usage is 25251 MB, indicating that the message buffers consume the majority of the memory.

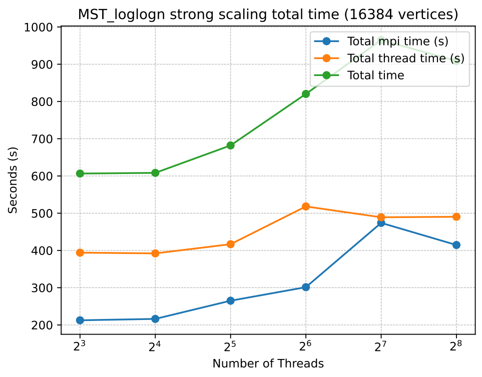
We measure the strong scaling performance of $MST\_\log\log n$ on Puhti with 16384 vertices and up to 512 Slurm tasks (as provided by Puhti).



(a) 16384 vertices strong scaling speedup



(b) 16384 vertices strong scaling time composition



(c) 16384 vertices strong scaling total time

The speedup shown in Figure fig. 2a demonstrates good scalability. It closely follows the ideal speedup line. With 512 tasks, the algorithm achieves a speedup of over 32 compared to the optimal speedup of 64. The parallelized computation time decreases as the number of tasks increases, while the communication and rank-0 computation times grow.

Key observation: Compute and communication trade-off. For the same-sized graph, more tasks result in less parallelized computation time (fraction) per task (as it is averaged out) but more communication time (fraction). Vice versa.

Figure fig. 2c demonstrates that MPI communication time is the major contributor to the slowdown in speedup. When vertices are partitioned, multiple vertices (nodes in Congested Clique terminology) are assigned to the same task. Within each group of vertices, no MPI communication is required. By increasing the number of tasks and further partitioning the vertices, the size of these local groups decreases, resulting in a higher total communication overhead. This explains the poor scalability of MPI communication time in the graphs.

On the other hand, the parallelized computation time shows excellent scalability. The total parallelized computation time remains relatively flat in Figure fig. 2c.

If more tasks were available, there would likely be a turning point in speedup where the communication overhead outweighs the benefits of additional tasks. However, due to the limited number of tasks, we cannot reach that point. Nonetheless, $MST\_\log\log n$ demonstrates good scalability with the available number of tasks and excellent compatibility with vertex partitioning.

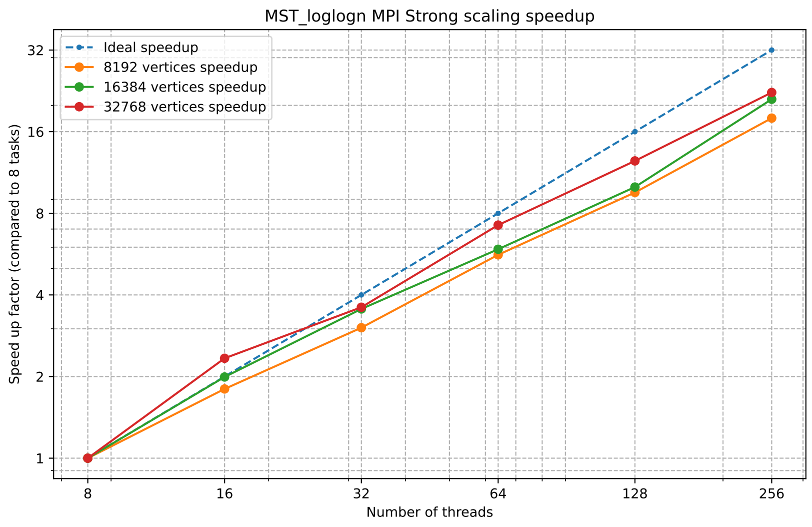


Figure 3: Strong scaling performance of $MST\_\log\log n$

Figure fig. 3 illustrates the strong scaling performance of $MST\_\log\log n$. The plot includes three lines corresponding to graphs with 2^{13}, 2^{14}, and 2^{15} vertices. The number of tasks increases horizontally from 8 to 256.

All three strong scaling plots closely follow the ideal speedup line, indicating that the parallelized computation time dominates the algorithm's execution time [9]. Among the three cases, the algorithm scales best with 2^{15} vertices, as its speedup curve is closest to the ideal speedup line. Conversely, the algorithm scales worst with 2^{13} vertices, as its speedup curve deviates the most from the ideal speedup line.

From Figure fig. 3, it is evident that larger graphs achieve better speedup with the same number of Slurm tasks. This observation aligns with the time composition shown in Figure fig. 1a. For a fixed number of tasks, increasing the number of vertices per task results in more time spent on parallelized computation and less time on MPI communication and rank-0 computation.

4.3 Comparison with prim's algorithm

Prim's algorithm is used as the sequential algorithm to compare the performance of $MST\_\log\log n$. It is implemented in C++ and runs on a single (Slurm) task on Puhti, with 16 GiB of memory.

With 16 GiB of memory, Prim's algorithm can process complete graphs with up to 2^{15} vertices (but not 2^{16}), which corresponds to $\frac{2^{15}\times(2^{15}-1)}{2} = 5.33 \times 10^8$ edges, completing in 88 seconds. The bottleneck for Prim's algorithm is the inability to fit larger graphs into the limited memory of a single node.

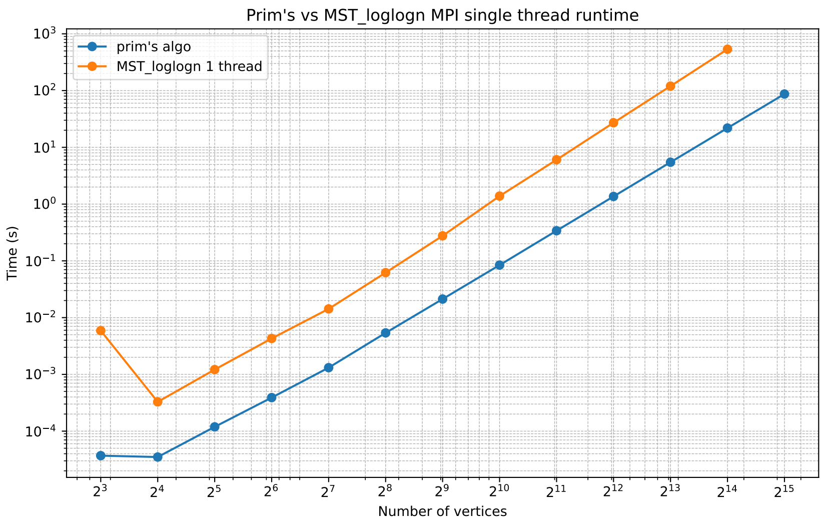


Figure 4: Prim's algorithm compared to $MST\_\log\log n$ performance

In contrast, due to the distributed nature of $MST\_\log\log n$, larger graphs can be processed as the number of tasks increases. However, when working on graphs of the same size, $MST\_\log\log n$ requires more memory than Prim's algorithm due to the additional memory needed for MPI buffers.

As shown in fig. 4, with more than 16 tasks, $MST\_\log\log n$ is capable of processing complete graphs with up to 2^{16} vertices (but not 2^{17}). Furthermore, $MST\_\log\log n$ outperforms Prim's algorithm in terms of speed when there are more than 64 tasks.

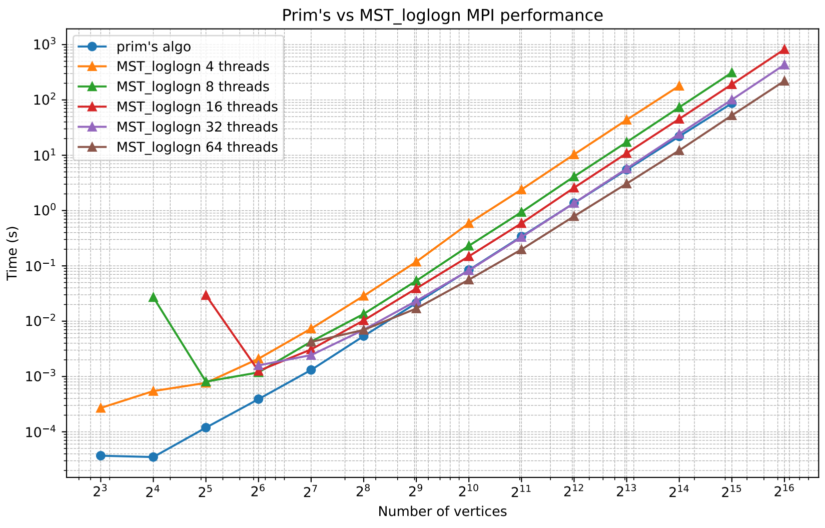


Figure 5: Prim's algorithm compared to $MST\_\log\log n$ performance with single thread

When both the prim's algorithm and $MST\_\log\log n$ are run on a single thread, the prim's algorithm outperforms $MST\_\log\log n$. The prim's algorithm is more memory efficient and has less computation time. Within the range of the graph size of our work, we see only a minor difference in the time complexity of the two algorithms.

4.4 Python vs C++

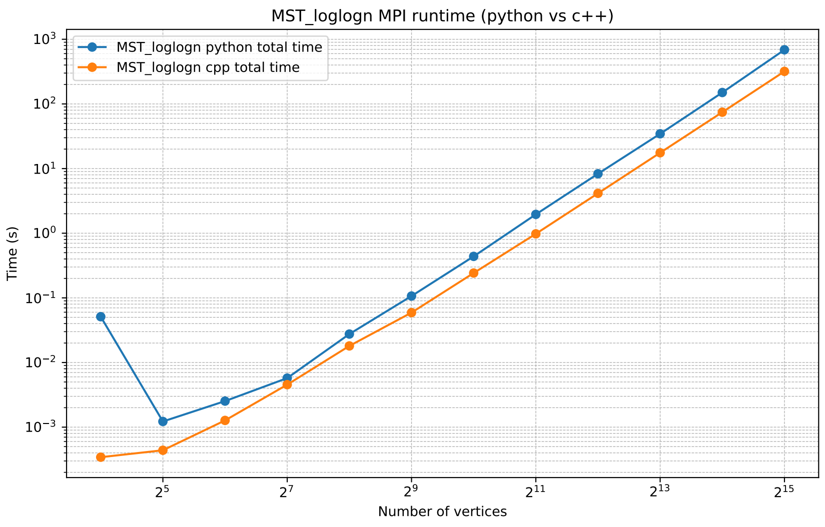


Figure 6: $MST\_\log\log n$ Python vs C++ time total time

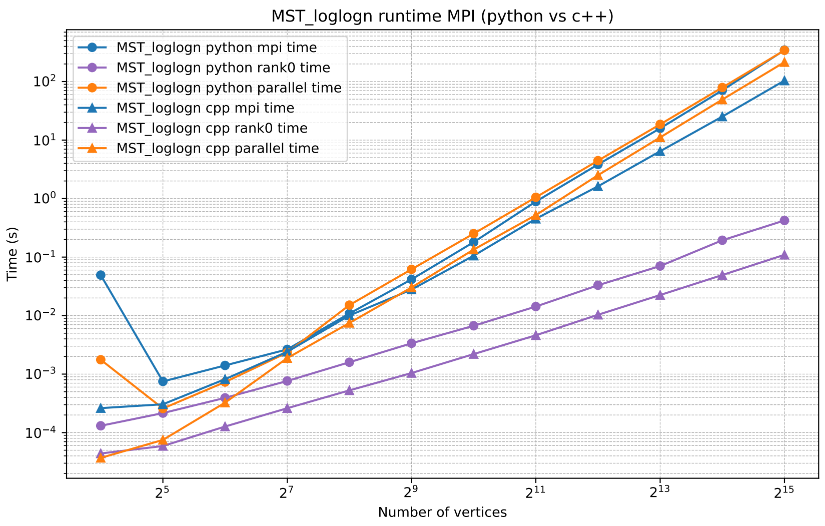


Figure 7: $MST\_\log\log n$ Python vs C++ time composition

The runtime difference between the C++ and Python implementations is significant. The slowdown in the Python implementation is primarily due to the memory usage of Python objects, which increases the time required for MPI communication and data processing during computation.

When sending messages with MPI, the data of an edge is encapsulated in an `Edge` object. This object consists of two integers and a double: the integers represent the origin and destination of the edge, while the double represents the weight of the edge. In Python 3.8.0, an integer object occupies 28 bytes, as indicated by `sys.getsizeof(int())`. Python integers are instances of the `PyLongObject` class [19], which includes auxiliary fields such as `ob_refcnt`, `ob_size`, and `ob_digit`, leading to their large size. Similarly, a double object in Python occupies 24 bytes. Consequently, the Python `Edge` object, which combines these components, occupies 80 bytes.

In contrast, the C++ implementation of the `Edge` object is significantly more memory-efficient. The C++ `Edge` is implemented as a `struct` containing two `int32` integers and a `float64` double, resulting in a total size of 16 bytes (128 bits). This makes the Python `Edge` object five times larger than its C++ counterpart.

The larger size of the Python `Edge` object leads to significantly larger MPI messages, which increases the time spent on MPI communication, as shown in Figure fig. 7. Additionally, Python incurs more computation overhead compared to C++, further contributing to the slowdown. This is evident in the increased parallelized computation time in Figure fig. 7.

As a result, the total runtime of the Python implementation is substantially higher than that of the C++ implementation, as demonstrated in Figure fig. 6, despite both implementations using the same algorithm and communication framework.

5 Congested clique with CUDA

GPUs are inherently parallel computing devices and more suitable for parallel computing than CPUs. A Nvidia V100 [5] has 80 streaming multiprocessors. Each is capable of handing 2048 threads at a time, or 64 warps (with 32 threads each warp) [10]. A warp is a group of threads that are executed on a Single Instruction Multiple Thread (SIMT) architecture. The threads in a warp execute the same instruction at the same time but on different data [21]. Multiple warps can be executed at the same time on a streaming multiprocessor grouped into blocks. A streaming multiprocessor can handle one block at a time or multiple blocks concurrently. This architecture allows Nvidia GPUs to issue a large number of threads and execute them in parallel.

Compared to the CPU, the GPU is a device with more lightweight threads and more parallelism. The number of vertices in a Congested clique is comparable to the number of threads that can be issued on a GPU, which allows the GPU to simulate a node in the Congested Clique model with a thread.

algorithm 2 describes the pseudo code of the $MST\_\log\log n$ implemented with CUDA, "a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU" ([11]).

5.1 Communication

With CUDA, the computation and communication are handled by kernel functions. Kernels functions are functions that are launched to run on the GPU. The threads lives in the context of kernel function. Each thread simulates one machine in the Congested Clique model, which is a vertex in the clique graph. Each CUDA thread has its own memory space corresponds to the machine memory in the Congested Clique model.

Threads can write and read from the memory space to achieve communication. When thread A sends a message to thread B in the Congested Clique Model, thread A writes the message to the memory space and thread B reads the message from the memory space.

Each kernel function handles only one communication round. This ensures all the communication rounds are synchronized. We don't need barrier with in the kernel. The kernel function exits when all the threads finish their communication and computation. The result of the kernel function resides in the memory space for communication.

The communication is handled in the following way:

– step 1: `min_to_cluster_kernel`

– step 2: `min_from_cluster_kernel` (comm2)

– step 3: Buffers are copied to the host

– step 4: CPU computation

– step 5: The updated cluster info is copied backed in to the Device memory

Algorithm 2 MINIMUM SPANNING TREE with CUDA

Require: Number of vertices n, adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$.
On each thread: Block index j, thread index i, block dimension x, and adjacency sub-matrix $\mathbf{A}[j \times x + i, :] \in \mathbb{R}^{1 \times n}$.

```
 1: $v \leftarrow j \times x + i$
 2: $T \leftarrow \emptyset$
 3: **while** there exists $v \notin T$ **do**
 4:     **if** First phase **then**
 5:         speed_up_kernel()
 6:     **else**                                              ▷ Step 1
 7:         min_to_cluster_kernel()                           ▷ Step 2
 8:         min_from_cluster_kernel()                         ▷ Step 4
 9:     **end if**
10:     Perform CONSTRUCT-FRAGS and update $T$               ▷ Steps 5 and 6
11: **end while**
12: **Return** the spanning tree $T$
13:
14: **function** SPEED_UP_KERNEL()
15:     Find the minimum edge $e(v, v')$ connected to $v$
16:     Write the edge to $v_0$
17: **end function**
18:
19: **function** MIN_TO_CLUSTER_KERNEL()
20:     For each cluster $F' \neq F$, compute $e(v, F') \leftarrow \min_{v' \in F'} \{e(v, v')\}$.
21:     Write all $e(v, F')$ to leaders $l(F')$
22: **end function**
23:
24: **function** MIN_FROM_CLUSTER_KERNEL()
25:     For each cluster $F' \neq F$, compute $e(v, F') \leftarrow \min_{v' \in F'} \{e(v, v')\}$
26:     Perform CHEAP-OUT and write the edge $e$ $v_0$
27: **end function**
```

– step 6: Same as above

There are two kernels in algorithm algorithm 2, namely `min_to_cluster_kernel` and `min_from_cluster_kernel`, corresponding to the first and the second steps in $MST\_\log\log n$. Between the two kernel calls, the buffers are not copied back to the host but are kept in the device memory.

When there is only on machine active in step 3, instead of launching a kernel function with a single thread, that part of the computation is done on the CPU. CPU is far more efficient in handling single-threaded computation than the GPU. Thus, we simply write the edge to the memory space of the machine 0 to simulate the gathering process at step 3. Similarly, step 5 and 6 are done by copying the updated cluster information back to the device memory.

5.2 Memory read and write

Threads communicate with each other by reading and writing into the vram of V100. When thread A sends a message to thread B, thread A. For the correctness of the communication, we eliminate the data race scenarios where multiple threads write to the same memory space e.g. when thread A and thread C all send a message to thread B.

Two pointers are passed as input arguments into the first kernel `min_to_cluster_kernel`: `min_edges_bufGPU`, which has all the graph edges with shape $n \times n$ and `to_cluster_bufGPU`, which is the output of the kernel with shape also $n \times n$, where n is the number of vertices in the complete graph. When thread A sends a message to the leader of cluster F, it writes the edge to `to_cluster_bufGPU` $[A, leader(F)]$. There are at least n clusters in the graph, so $leader(F)$ never overflows the buffer. One thread send at most one edge to another cluster leader, so each entry in the buffer is a unique space analogous to a link in the Congested Clique model.

With the second kernel `min_from_cluster_kernel`, two pointers `to_cluster_bufGPU` and `from_cluster_bufGPU` are passed as input arguments. Among them, `from_cluster_bufGPU` is the output of the kernel. It is another n by n buffer. Because at the end of this stage, one cluster will not send more edges than the number of member of the cluster to vertex 0. There will be at most $n \times n$ edges sent to vertex 0.

The space of `from_cluster_buf` is arranged in the following way. Each cluster will gets a `start_idx` and an `end_idx` to indicate the space allocated to the cluster in the array. `start_idx` − `end_idx` is the number of vertices in the cluster. The `end_idx` of cluster F is the same as the `start_idx` of cluster $F + 1$, making sure the space is not overlapped. This makes up the space for the edges sent to vertex 0 with the size of maximum $n \times n$.

Finally, rank 0 or the CPU reads the buffer `from_cluster_buf` and filters out all the edges not `null` and continues to perform `Construct-Frags`.

5.3 Complete and sparse graphs

The Nvidia V100 GPU has the capability to issue a huge amount of threads but with a limited VRAM of $16G$ / $32G$. The total memory can be used is much lower than the previous MPI simulation of $8G$ / $16G$ per thread. Compared to the MPI implementation, V100 can only store much less edges in the memory, even it has the capability to launch a large quantity of threads for vertices. A V100 can issue 2048 threads on each streaming multiprocessor, which is $80 \times 2048 = 163840$ threads in total. We walk around this limitation by introducing sparse graphs. Sparse graphs saturate the GPU's thread capability better by having less edges on each vertex. In this way, the computation is better balanced among the threads.

In the Congested Clique, the network graph is the equivalent as the graph to be solved, which is a complete graph / network. Each vertex / thread, has a list of weight of adjacent edges in memory. In the sparse graph setting, each vertex has a varying number of adjacent edges. The edges of the graph are represented as a adjacent list when generated. The network remains fully connected in the sparse graph setting, meaning any thread and send message to any other threads.

$MST\_\log\log n$ works on the sparse graph as well. In memory, a complete graph is represented by an adjacency matrix, where each column represents the adjacent edges of a vertex. Whereas the sparse graph is represented by an adjacency list due to the varying rank of each vertex. Each vertex has a list of adjacent edges.

$MST\_\log\log n$ is designed to work on a complete graph but it does work for connected spares graphs as well.
Proof: A sparse graph G is analogous to a complete graph G' with weight of edges which are disconnected in the sparse graph set to infinity. $MST\_\log\log n$ is capable of finding the MST in G'. The edges with infinite weight will not be picked in step 4 for G', meaning they can be removed from the candidate list of step 4. Same for the previous steps. As the network remains fully connected, the infinite weight edges will not affect the communication. Thus, the edges with infinite weight can be ignored in the computation to form a sparse graph.

To generate a connected sparse graph, the following steps are taken.

- Generate a Prüfer sequence with all the vertices. A Prüfer sequence is an unique sequence of vertices that encodes a tree [22]. The tree serves as the skeleton of the sparse graph making sure the graph is connected.
- Convert the Prüfer sequence into a tree.
- Add edges to the tree randomly until the expected rank is met.

5.4 Best practices

Similar practices used in the MPI implementation are used on the CUDA implementation to ensure a relatively fair comparison.

- The constructed MST is maintained as a union-find data structure.
- Same edge objects and weight datatype are used in the CUDA implementation as in the MPI implementation.
- The simplification on the first phase is applied.

5.5 Profiling

To accurately measure the performance of the CUDA implementation, we need to profile the program. The profiling is done with nvprof, a command-line profiler provided by Nvidia. The profiling result shows the time spent on the memory copying and the kernel functions. The time spent on each kernel function is the time spent on the communication and local computation in each round.

6 Performance with CUDA

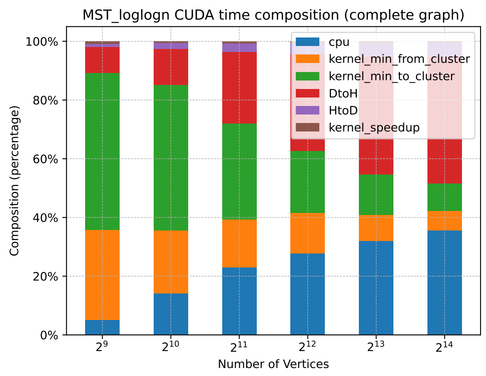
6.1 Conclusion overview

- CUDA implementation of $MST\_\log\log n$ demonstrates significant speedup compared to single-threaded Prim's algorithm, leveraging the GPU's parallelism.
- The communication between threads is efficiently handled using memory read and write operations, analogous to message passing in the Congested Clique model.
- The time spent on kernel execution scales well with the number of vertices, but memory operations and CPU computation become bottlenecks as graph size increases.
- Sparse graphs show potential for better workload balancing on GPUs, but the overhead of adjacency list operations can negate performance gains.
- Smaller block sizes improve workload distribution among Streaming Multiprocessors (SMs), reducing kernel execution time.
- CUDA implementation is limited by GPU memory capacity, making it challenging to handle large complete graphs without sparsity.

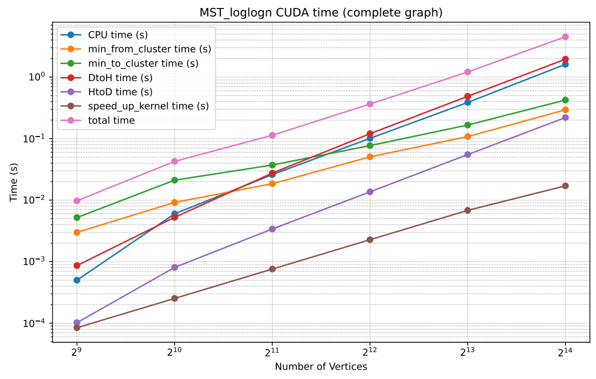
6.2 Performance

As shown in the figure, DtH refers to the initial data transfer from the host to the device. This operation scales proportionally with the graph size, as the entire graph data must be copied to the GPU memory. Conversely, HtD represents the final data transfer from the device back to the host, which scales at a slower rate due to the reduced amount of data being transferred.
Within the kernels, each thread simulates a machine in the congested clique model. At the beginning of each round, threads are responsible for receiving messages, which corresponds to reading from the buffer. Subsequently, threads process the received data and send their own messages, which involves writing to a new buffer.
The workload on the `to_cluster_kernel` is notably larger than that on the `from_cluster_kernel`. This is because, at the start of the algorithm, there are significantly more edges to process, resulting in a higher computational demand for the `to_cluster_kernel`.

We run the CUDA implementation algorithm 2 on Puhti with a Nvidia V100 GPU. There are massive amount of thread resources on one V100. The workload of complete graphs are not able to saturate the thread resources before the vram is full. fig. 8 shows the time composition and total time of the CUDA implementation with complete graphs. The number of vertices scales from 2^9 to 2^{14}. Considering the



(a) $MST\_\log\log n$ CUDA time composition with complete graph



(b) $MST\_\log\log n$ CUDA time with complete graph

Figure 8: $MST\_\log\log n$ CUDA time composition with complete graph. Each block has 1024 threads, each thread simulates a vertex.

communication buffers used in the program, 2^{14} is the largest graph that can be fit into the vram of a V100. The number of threads in each block is 1024.

Because the maximum amount of threads can be issued is around 2^{26} on V100, Every vertices can be mapped to a thread during the scaling. Compared to the $|E|$ of the graph, which scales quadratically with the number of vertices, the workload among the threads scale linearly within the kernel. However, having the workload scales linearly in the kernels doesn't mean it scales linearly throughout.

fig. 8b shows the time of each kernel, memory operation, CPU and total time of the program as the number of vertices increases. The time of `min_to_cluster_kernel` and `min_from_cluster_kernel` increases noticeably slower compared to other time categories. On contrast, the time of memory operations and CPU computation increases noticeably faster. The workload for them can not be distributed to more work units like with the kernels. Thus they scale corresponds to $|E|$.

From fig. 8a, we see the composition changes during the scaling. As explained earlier, The CPU time and memory copying time remain insignificant when the number of vertices is small, but as the number of vertices increases, these times become more prominent. Eventually, the kernels time is insignificant, and the nonparallelizable part of the program contributes over 80% of the total time, when graph size reaches 2^{14} vertices.

However, the CUDA time can be further reduced by having smaller block sizes, when the SMs are not well saturated. fig. 9 shows the time of the CUDA implementation with complete graphs with 8192 vertices and varying block sizes. The number of vertices is fixed to 8192. The time of the program decreases as the block size decreases. The time of the program is the lowest when the block size is 256. The time of the program increases when the block size is further decreased.

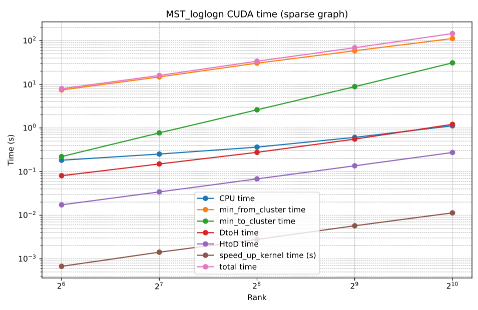
Going from block size 1024 to 128, the CUDA time is reduced almost by half. More blocks are launched with smaller block size, which means more SMs are utilized. The reduction in time is from the less warps on each Streaming Process (SP), going

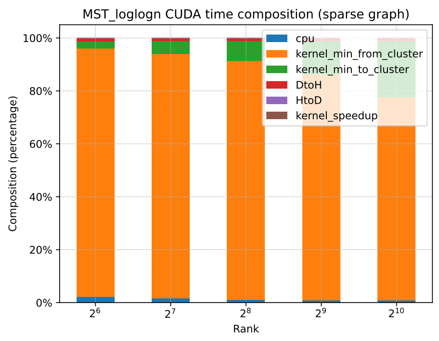
Number of vertices per block	Number of blocks	Total time (s)	CUDA time (s)	Memory ops time (s)
1024.0	8.0	1.2169	0.2992	0.5307
512.0	16.0	1.1281	0.2152	0.5289
256.0	32.0	1.1088	0.1766	0.5361
128.0	64.0	1.0741	0.1523	0.5318

Figure 9: $MST\_\log\log n$ CUDA time with complete graph with varying block size

from 32 to 4 warps with 32 threads in each warp. When there are SMs idling, this is a good way balancing the workload among the SMs.

6.3 Complete vs sparse graph





(a) $MST\_\log\log n$ CUDA time composition with sparse graph

(b) $MST\_\log\log n$ CUDA time with complete graph

Figure 10: $MST\_\log\log n$ CUDA time composition with sparse graph. There is a block on each SM and each block has 1024 threads/vertices.

We modified algorithm 2 to work with sparse graphs. The algorithm itself works out of the box with sparse graph. But the buffer allocation and accessing need to be modified to comply with the sparsity. Each SM possess 1024 vertices with 1024 threads, resulting in $80 \times 1024 = 81920$ vertices in total in the sparse graph. fig. 10a shows the time of the CUDA implementation with sparse graphs. The x-axis is the average rank of the sparse graph, which goes from 64 to 1024.

However, the time of the CUDA implementation with sparse graphs is even larger than the time of memory operations and CPU computation. Whereas in the complete graphs, the time of the CUDA implementation is smaller than the time of memory operations and CPU computation. This is the exact opposite of the expectation. Most of the slow down is caused by the data structure overhead from the sparse graph. Because of the sparsity of the adjacent list. When you access the adjacent list, you have to loop through all the edges to find the edge to vertex v. This is a $O(n)$ operation. The time of the kernel is $O(n^2)$ in the worst case. And there are more clusters at the beginning which means more workload in the kernel. Thus in fig. 10, the kernels are taking the most time, and fig. 10 the sparse graph is slower than the complete graph with the same amount of edges.

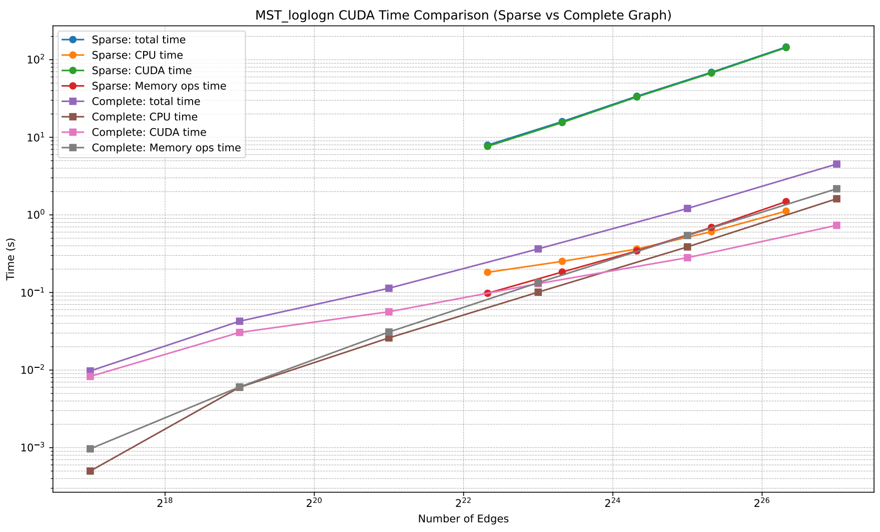


Figure 11: $MST\_\log\log n$ CUDA time comparison with complete and sparse graph

In the kernel `min_from_cluster_kernel`, there a nested loop

6.4 Comparison with Prim's on single threaded CPU

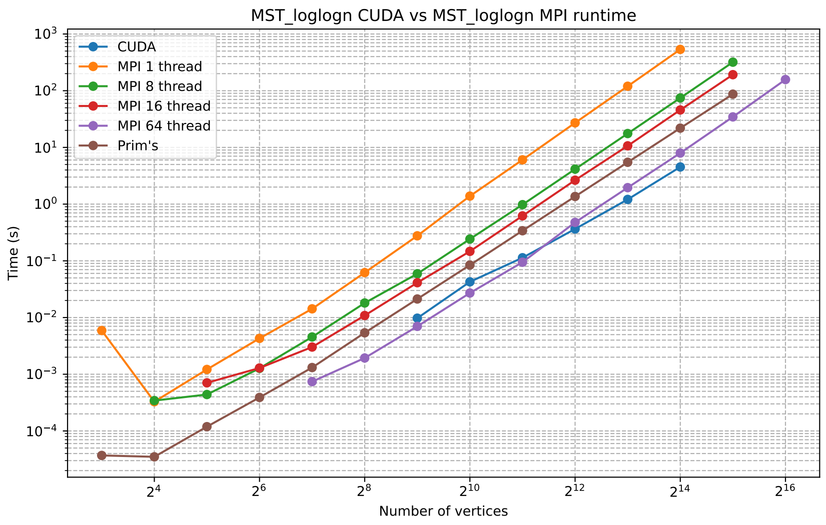


Figure 12: $MST\_\log\log n$ CUDA time comparison with MPI and Prim's algorithm

Compared with the vanilla Prim's algorithm, the CUDA implementation is significantly faster, with a slower grow in time as the graph size grows, according to fig. 12. While the single-threaded MPI implementation is the slowest, it beats the prim's algorithm when the number of threads is larger than 64. With 64 threads, the performance is comparable to the CUDA implementation but with steeper slope.

7 Conclusion

When simulating the Congested Clique model with MPI on CPU threads, limited thread resources necessitate vertex partitioning. With few vertices per rank, communication time dominates, but as graph size increases, computation time becomes the bottleneck. Comparing $MST_\log\log n$with Prim's algorithm, $MST_\log\log n$requires more compute and memory but outperforms Prim's in wall clock time with more work nodes and handles larger graphs. The bottleneck for larger graphs is MPI message memory. The number of rounds is $O(\log\log n)$, as predicted. Python is significantly slower than C++ due to larger memory usage for MPI messages and slower computation, making it more memory-bound.

The high parallelism of the GPU makes it good at parallelize the vertex workload very well. The assignment of the vertex to the thread is very natural. However, when comes to clique graphs, how to fit graphs into vram is a challenge. The high parallelism can be underutilized if the graph is too large to fit into the vram. Sparser graphs show better affinity to the memory-compute trade-off of the GPU.

However, Congested Clique algorithms are often not optimized to run on GPUs. The computation paradigm shows characteristics of CPU computation. Branching in the algorithms brings divergence among the threads on the same SIMT unit, potentially causing the performance to degenerate. Moreover, it is not intuitive to implement the communication between threads. Is it often required that one thread is globally accessible by another even if they are not in the same block. With this constraint, is it not intuitive how to utilize the shared memory of the block. We have to read and write to the vram, which is designed to be slower than the shared memory. We are also missing some matrix operations that are common in the GPU programming, such as MFMA, MMA operations, and their underlying hardware support such as tensor cores.

TODO insert more CUDA conclusion here.

8 Future work

How different datatypes affect the performance. We tried double precision, but how about single precision? Hardware typically have different support on different datatypes. E.g. on GPU, single / half precision has better support than double precision.

Can we optimize the algorithm to comply with the GPU architecture better? E.g. how to utilize the shared memory better? How to reduce the divergence among the threads? How to utilize the tensor cores?

References

[1] Zvi Lotker, Boaz Patt-Shamir, Elan Pavlov, and David Peleg, "MST Construction in $O(\log\log n)$ Communication Rounds," *SIAM Journal on Computing*, vol. 35, no. 1, pp. 120–131, 2005. https://doi.org/10.1137/S0097539704441848

[2] M. Ghaffari, F. Kuhn, and C. Lenzen, "On the power of the congested clique model," *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*, pp. 367–376, 2013. https://www.researchgate.net/publication/266659337_On_the_power_of_the_congested_clique_model

[3] "Puhti supercomputer," CSC - IT Center for Science. https://research.csc.fi/-/puhti

[4] "Puhti Service Description," CSC - IT Center for Science, Jun. 2021. https://research.csc.fi/documents/48467/640068/Puhti-servicedescription-0621.pdf/e6fd32ae-92ea-2d42-0b6c-9f73c1a4c418?t=1628595451763

[5] "Puhti Computing Systems," CSC - IT Center for Science. https://docs.csc.fi/computing/systems-puhti/

[6] "Slurm Overview," SchedMD. https://slurm.schedmd.com/overview.html

[7] "MPI Forum." https://www.mpi-forum.org/

[8] "Open MPI." https://www.open-mpi.org/

[9] "HPC Scaling," HPC Wiki. ://hpc-wiki.info/hpc/Scaling

[10] "NVIDIA Volta Architecture," NVIDIA Corporation, White Paper, 2017. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[11] "CUDA C++ Programming Guide," NVIDIA Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[12] H. Prüfer, "Neuer Beweis eines Satzes über Permutationen," *Archiv der Mathematischen Physik*, vol. 27, pp. 742–744, 1918.

[13] "Supercomputing," IBM. https://www.ibm.com/topics/supercomputing

[14] K. Nowicki, "A deterministic algorithm for the MST problem in constant rounds of congested clique," *arXiv.org*, Dec. 9, 2019. https://arxiv.org/abs/1912.04239

[15] M. Ghaffari and M. Parter, "MST in Log-Star rounds of congested clique," *ACM*, 2016. https://doi.org/10.1145/2933057.2933103

[16] "Fat-Tree Design," Cluster Design. https://clusterdesign.org/fat-trees/

[17] "Open MPI gather operation," Open MPI. https://github.com/open-mpi/ompi/blob/main/ompi/mpi/c/gather.c

[18] "MPI tutorial: collective communication," Open MPI. https://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/

[19] "CPython API documentation: Integer Objects," The Python Software Foundation. https://docs.python.org/3/c-api/long.html

[20] "CPython API documentation: Floating-Point Objects," The Python Software Foundation. https://docs.python.org/3/c-api/float.html

[21] "CUDA C programming SMIT architecture" NVIDIA Corporation, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#simt-architecture

[22] Heinz Prüfer, "Neuer Beweis eines Satzes über Permutationen," *Archiv der Mathematik und Physik*, vol. 27, pp. 742–744, 1918.

[23] "Profiler User's Guide" NVIDIA Corporation, 2025. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/

[24] Tomasz Jurdzinski and Krzysztof Nowicki, "MST in O(1) Rounds of the Congested Clique," *arXiv.org*, Jul. 26, 2017. https://arxiv.org/abs/1707.08484