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
2024

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** 22 **Language** English

Abstract

The purpose of this thesis is to study how congested clique algorithms perform on modern supercomputers.
A representative algorithm $MST\_\log\log n$ for Congested Clique is studied. The algorithm is implemented on Puhti, a Finnish supercomputer hosted by CSC. The congested clique setting is simulated with the Message Passing Interface (MPI) and CUDA.
MPI is a message-passing library interface specification that uses a message-passing model for parallel computing.
result: TODO

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

Symbols and abbreviations

Abbreviations

$\boxed{MST\_\log\log n}$	The algorithm described in [1]

informative, but is there a visually nicer alternative?

maybe also implicitly define clique network (e.g. all pairs of nodes are connected via a comm link)

maybe add that local computation is free

1 Introduction

Congested Clique, described in [2] is a mathematical model for distributed computing. It features n inter-connected computers in a clique network, sending messages in a synchronized manner. In each round, each computer sends $O(\log n)$ to each other computer. The computation efficiency is measured by the communication rounds. In this work, the congested clique algorithm that solves the (Minimum spanning tree)MST in $O(\log\log n)$ rounds [1] is studied and implemented on the supercomputer Puhti [3]. We simulated the congested clique model with MPI and a CUDA GPU to run $MST\_\log\log n$.

bits

define MST

briefly describing these in a few words would be very helpful (it is ok to not define in abstract but here it is very helpful)

This is probably not the best way to sell your work. since you would want to claim that you study the connection to Clique model at least to some degree

The purpose of this work is to identify the connection between $MST\_\log\log n$ and modern distributed computing hardware and tools. The relations allow us to understand the bigger picture as $MST\_\log\log n$ is representative of the congested clique model

The representativeness of $MST\_\log\log n$ can be shown in the following ways:

- $MST\_\log\log n$ exploit the full connectivity of the graph. In each round, there is a step where each computer sends $O(\log n)$ messages to all the other computers.
- To comply with the $O(\log n)$ limit when sending to a unique leader, large data is spilled to other computers before sending.
- The algorithm needs several rounds before its convergence.

We measure how compatible $MST\_\log\log n$ is with the supercomputing infrastructure to find the connections. The congested clique is simulated with MPI and CUDA. When implementing the algorithm pseudo-code into runnable, the compatibility is measured by the effort needed for the conversion. The simulation consists of two parts, the local computation, and the message passing. When implementing $MST\_\log\log n$, the coding follows best practices and common sense.

Secondly, how the hardware is limiting the performance of the algorithm. Memory, CPU, GPU, network, which one is the bottleneck when the algorithm scales. How the algorithm performs compared to traditional sequential algorithms. By increasing the number of nodes, will it outperform the sequential algorithm with

- Speed comparing to sequential algorithms.
- Number of edges can be processed compared to sequential algorithms.

Scalability:

- Strong scaling and weak scaling (How having more processes speeds up the computation, which part of the hardware will become bottleneck when scaling continuously).
- Time composition for the run when scaling (MPI_time vs local). Does the time complexity hold?

btw if you need references for the theory side of things I can help. For example on importance of MST in general (sequential setting) on importance of it when data is big and hence its importance in distributed/parallel computing. These should give a good reason as to why we chose this problem.

I think there should also be a discussion on the fact that we have faster algorithms in the congested clique for MST but we chose not to use those. in particular we have O(1) algorithms for MSTs.

about the introduction: I think we should have these: 1. a bunch of definitions and explanation on what the problem is, what are the platforms, 2. what is our goal and what is its importance in a bigger picture, 3. what is your contribution, structured in a subsection and numbered. don't worry about talking too much and repeating a lot of stuff that you are saying in later chapters. If it is structed well, people can easily read things as much as they want, and once they are convinced they just move to the next item of the list (next contribution/ next chapters). having a smaller text but without structure means reader gets lost and has to parse at each time what they are reading (is it a contribution, is it an observation, a definition? a claim?) being very precise using structure is huge (other less effective ways: setting tone at the beginning of a paragraph, or using italic/ \emph{} are some other ways to make sure a message goes through)

2 Preliminaries

2.1 Congested clique (also known as clique)

what is a round? The round is synchronous. The communication bandwidth to each neighbor is $O(\log n)$.

2.2 The algorithm $MST\_\log\log n$

What is a phase and a step in $MST\_\log\log n$? In each phase, the algorithm consists of the following steps: (TODO)

1. Each vertex v computes the minimum weight edge $e(v, F')$ that connects v to a leader l.
2. The numbers start at 1 with each use of the `enumerate` environment.
3. Another entry in the list

The steps are repeated until the MST is found.

2.3 The super-computer Puhti

Used Slurm workload manager [6], to allocate computing nodes. Each node can have processes. The number of total threads is limited, can get 484 * 2 * 20 for the "M" cluster. Total memory available is 192 GiB. Xeon Gold 6230 is the CPU model (ref needed). The nodes are connected with a 100 Gbps HDR100 link, and the topology is a fat tree with a blocking factor of approximately 2:1. [5]

what is this? is it too common that it does not need a few words explaining it?

2.4 MPI

2.5 CUDA

What is device, what is host, what is a kernel function. How to allocate memory on the device. How to copy data between host and device. How to launch a kernel function (define thread count).

① this is true.

3 Congested clique to MPI

3.1 Congested Clique simulation

$MST\_\log\log n$ is simulated with OpenMPI and C++ on CPUs. Multiple CPU threads run cooperatively to simulate the congested clique model. Each thread has its dedicated memory space and computing power. Each thread simulates multiple machines in the congested clique model. ① The local computation of the congested clique is simulated by each thread's local computation while the communication is handled by OpenMPI.

abbr in ()

MPI (Message-passing interface) is a message-passing library interface specification that message-passing model for parallel computing [7]. Open MPI is an open-source Message-passing Interface implementation.

Wording needs care

One CPU thread can ~~hold~~ simulate one or multiple vertices. The thread number is limited. With each thread simulating more than one vertex, larger graphs can be simulated. To simulate more than one vertex, each thread holds all the edges adjacent to the vertices it is responsible for and run the algorithm for each vertex.

TODO how is the communication synchronized.

When running $MST\_\log\log n$, in the beginning, each rank receives input to run the algorithm. The input includes a list of vertices the rank is responsible for and all the adjacent edges.

what is a rank?

step 1 $\longrightarrow$ MPI_Alltoall (comm1)
step 2 $\longrightarrow$ MPI_Alltoall (comm2)
step 3 $\longrightarrow$ MPI_Gather (comm3)
step 4 $\longrightarrow$ Local computation on rank 0
step 5 $\longrightarrow$ MPI_Broadcast (comm4)
step 6 $\longrightarrow$ Local computation on each thread

Correctness by comparison is better as a secondary method. I think here our first argument should be: We are doing the same thing as MST_loglogn but translate in OpenMPI. We can discuss how to present this.

To ensure the correctness of the algorithm, the result of $MST\_\log\log n$ is compared to the result of the prim's algorithm on the same graph.

3.2 Simplification on the first phase

This is a bit vague. the first paragraph is always a great place to introduce the problem/complication or prepare readers' mind on what's about to follow in this section

We implement the algorithm with the best practices and common sense. The algorithm is modified to align with the implementation goal.

At the beginning of the first phase, there is no spanning tree or clusters formed yet. Each vertex itself is a cluster. In the first step, for each vertex v and edge $e \in E_v$, e is sent over the connection e. Because for every two clusters $F \neq F'$, there is only one unique edge connecting them. In step two, each cluster leader (the vertex itself) sorts all the edges it receives, which is all the edges the vertex is adjacent to check the minimum, sort and take only one as μ is 1.

Essentially, all the above steps find each vertex the minimum weight edge adjacent. Simplification is done to the algorithm during the first phase. In the simplified version, rank 0 gathers the minimum weighed edges adjacent to each vertex (comm0) and skips steps 1-3 during the first phase.

The Simplification has a significant impact on the performance of the algorithm. Because the time taken by MPI communication corresponds to the message size. During the first phase, each vertex is a cluster, and the communication needs are the biggest in the whole algorithm.

3.3 Union-find

When merging edges to the MST, each edge is checked if it is connecting two vertices that are already connected. Thus, a union-find is maintained. If two vertices have the same root, it means they are connected. When an edge is added to the MST, two sets containing the vertices are merged.

3.4 Random vertex partitioning

Each thread holds a list of vertices and all the edges adjacent to the vertices. The adjacency matrix represents a complete graph, and the weight on each edge is generated randomly. The adjacency matrix is partitioned and each thread holds a continuous part of the matrix.

3.5 Memory limit and input graph generation

Each node can have at most 40 threads and has a memory limit of 192 GiB. With n threads for a node, each thread can have at most $192/n$ GiB memory.

The graph is a complete graph represented by a square adj matrix. Each edge is assigned a random weigdht. There is a naive approach and a distributed approach to generate the graph.

Naive generation: generate the adj matrix on rank 0 and scatter it to all other ranks with MPI_Scatter. Each rank gets a matrix row. Memory eager ($O(n^2)$ memory needed), can't run large enough graph.

Distributed generation: Each rank generates several rows of the adj matrix. And share the result with the destination peer with MPI_Alltoall. It saves the memory for graph storage and allows the running of larger graphs.

Some ephasis on "random input graph generation for the purpose of performance testing" would make the purpose of this section clearer

4 $MST\_\log\log n$ performance

it would be good if the title says we mean the performance with MPI

4.1 Time composition

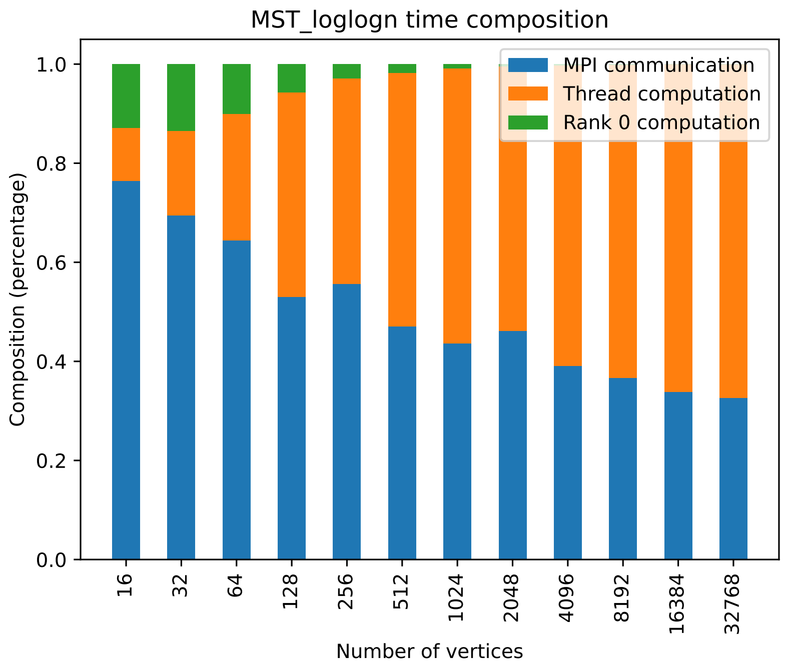


Figure 1: $MST\_\log\log n$ time composition with MPI on one node with 8 threads

Figure 1 shows the time composition of $MST\_\log\log n$ implemented with MPI with a different number of vertices. The hardware is fixed. There are 8 threads on a single node. Each thread has 16G memory for each.

some key words here can be in italic

MPI communication is the communication time spent on MPI functions. Rank 0 computation time is the time ~~step~~ spent on step 4, where rank 0 is the only active thread and other threads are idle waiting. The thread computation time is the time threads spend on local computation excluding step 6. MPI communication resembles the communication time in the congested clique model. Thread computation + rank 0 computation together resemble the local computation time in the congested clique model.

it would be nice to have a reference to where the steps are coming from. (i.e. the section)

In the congested clique model, each machine is in charge of only one vertex in the graph. Whereas with CPU threads and MPI, simulating only one vertex on each thread will not scale well with the hardware constraints. Typically, CPU threads are powerful enough to process large amounts of edges. But the number of threads is limited. To simulate larger graphs, having a thread responsible for multiple vertices is necessary.

here I think you can refer to the figure a few times when you analyze the time composition

As the number of vertices goes up on the fixed hardware, the thread computation time composition decreases. The MPI time and rank 0 computation time composition decreases. When the number of vertices is small, MPI computation is the dominant factor in time consumption, while thread and rank 0 computation combined constitute

another important thing is to really try to emphasize on the message. I think right now your message is in between explanation of the context and definitions. What you can do is that right when you are about to mention a result or observation change your tone and tell the reader that we are learning this new thing. For example saying "a key observation/point..." or "Figure x shows" or phrases of such are helpful here.

less than 25% percent of the time composition. When the number of vertices is large enough, the thread computation time becomes the dominant.

again I think your key message gets lost and would benefit if we try to highlight it a bit.
Something like expressing the key message of each figure in a structure like subsection or paragraph would probably help . what do you think?

so basically separating the part where you descibe the setting from observations/results/message would be good.

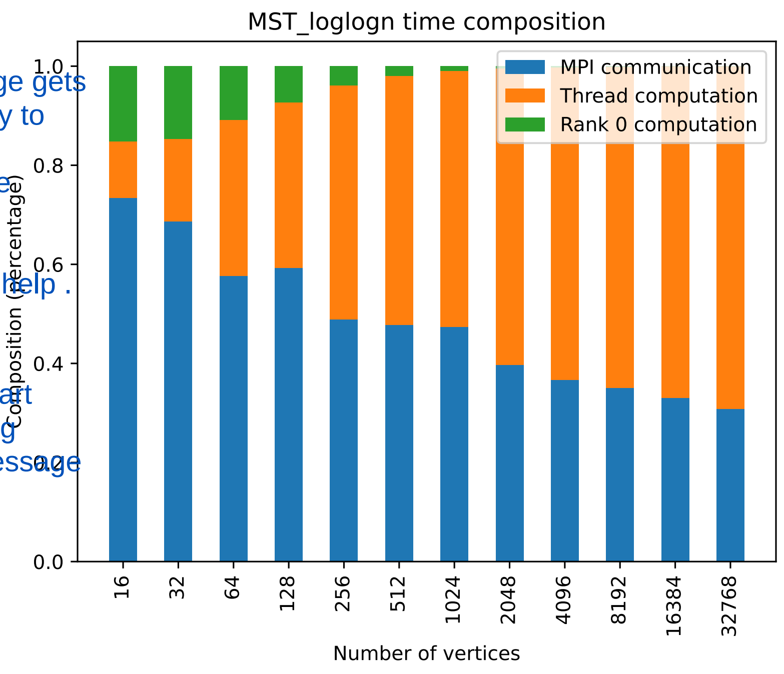


Figure 2: $MST\_\log\log n$ time composition with MPI on 8 nodes with one thread each

Figure 2 shows the same time composition of $MST\_\log\log n$ as in Figure 1. But instead of having 8 threads on a single node, there are 8 nodes with one thread each. A similar time composition pattern can be observed. Together with 1, it shows that the time composition of $MST\_\log\log n$ is not affected by how the threads are distributed.

Figure 3 is a demonstration of how the time composition changes with a fixed number of vertices but a varying number of threads. Horizontally, the number of threads increases as the number of vertices on each thread decreases. There are 16384 vertices in total. The time composition pattern is similar to 1. For a fixed number of vertices, as there are more vertices on each thread and fewer threads in total, the thread computation time becomes the dominant factor compared to MPI communication time and rank 0 computation time. Whereas when there are more threads and fewer vertices on each thread, the MPI communication time becomes the dominant factor.

This concludes that for larger graphs, the fewer vertices on each thread, the more time is spent on MPI communication and the rank 0 computation, which is more similar to the scenario of Congested Clique. On the opposite, the more vertices on each thread, the more time is spent on local computation, which is not similar to the scenario of Congested Clique.

good conclusion. Maybe we can better sell this by saying that in certain cases it behaves similarly and therefore you can translate your expectations from congested clique to here. What do you think?

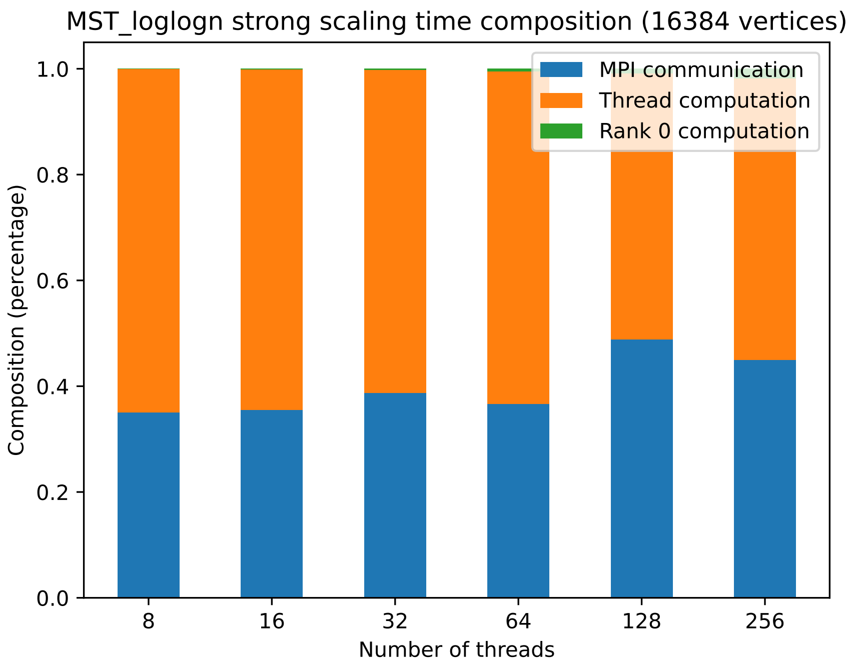
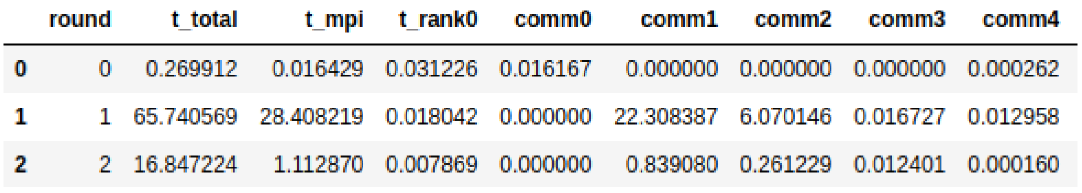


Figure 3: $MST\_\log\log n$ strong scaling time composition for with 16384 vertices

we can maybe have this table in latex instead of a figure, no?



	round	t_total	t_mpi	t_rank0	comm0	comm1	comm2	comm3	comm4
0	0	0.269912	0.016429	0.031226	0.016167	0.000000	0.000000	0.000000	0.000262
1	1	65.740569	28.408219	0.018042	0.000000	22.308387	6.070146	0.016727	0.012958
2	2	16.847224	1.112870	0.007869	0.000000	0.839080	0.261229	0.012401	0.000160

Figure 4: $MST\_\log\log n$ detailed time composition with MPI with 8 threads and 16384 vertices

4.2 Round-based time composition

what is this?

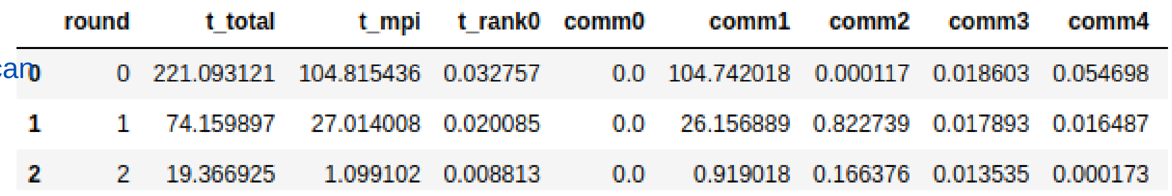
Figure 4 is a detailed time composition of $MST\_\log\log n$ with MPI with 8 threads on a single node and 16384 vertices. $mstalgo$ ran for 3 rounds. t_mpi is the time spent on MPI communication in a round. It is a sum of $comm0$, $comm1$, $comm2$, $comm3$ and, $comm4$. t_rank0 is the rank 0 computation time in each round. It corresponds to the step 4 of the algorithm. t_total is the total time spent on the round. The difference between t_total and $(t\_mpi + t\_rank0)$ is the time spent on parallel computation.

maybe you can use a style that's more code-oriented for these. like the type when you write a piece of code in slack/zulip that wraps it in a gray box. This can be helpful for whenever you are talking about scertain functions in MPI as well.

The first phase takes the least time as the result of the simplification. Without the simplification, the first phase takes the most time shown in 5. The Time decreases as the rounds go on. The number of clusters decreases as more and more vertices are added to the spanning tree along the process. This means, that the more rounds the algorithm runs, the fewer edges to process for the next round. The MPI communication time and the computation time show a correlation to the number of edges to process in the round. The unbalanced workload in each round causes an unbalanced running time.

Does this mean translating round complexity of an algorithm in congested clique to here is not as simple as multiplying it with some constant number? Then we can emphasize on this!

but if it turned out kind of ugly, you can maybe just use a different font for these, but defenitely not $$ math environment since it makes them look a bit awkward similarly, this can be a table.



	round	t_total	t_mpi	t_rank0	comm0	comm1	comm2	comm3	comm4
0	0	221.093121	104.815436	0.032757	0.0	104.742018	0.000117	0.018603	0.054698
1	1	74.159897	27.014008	0.020085	0.0	26.156889	0.822739	0.017893	0.016487
2	2	19.366925	1.099102	0.008813	0.0	0.919018	0.166376	0.013535	0.000173

Figure 5: $MST\_\log\log n$ with no simplification detailed time composition with MPI with 8 threads and 16384 vertices

The time spent on each round is unbalanced. The time composition of each round is different. In the first phase, the MPI communication time only takes less than 10% of the total time. In the second round, the MPI communication time takes more than 40%. In the third round, the MPI communication time takes less than 10%.

4.3 Scalability

have we explained what we mean by scalability? or is this the first proper place? Maybe we can re-iterate why we care about this

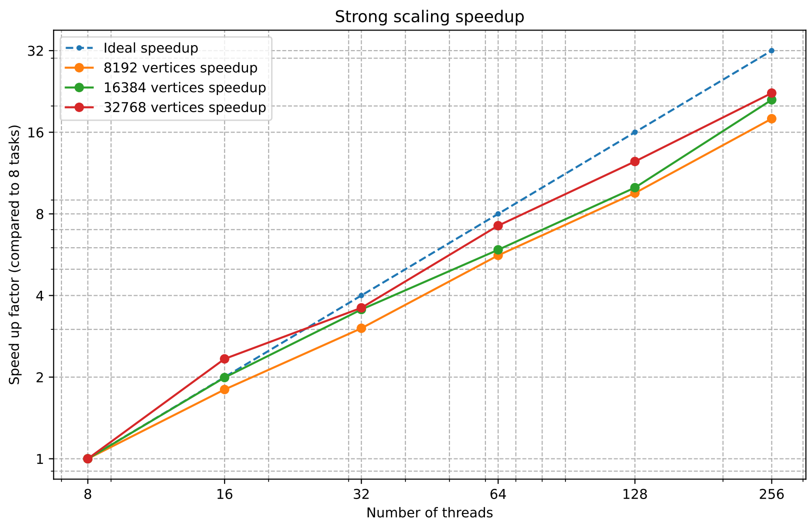


Figure 6: Strong scaling performance of $MST\_\log\log n$

Figure 6 shows the strong scaling performance of $MST\_\log\log n$. There are three different lines in the plot with 8192, 16384, and 32768 vertices. The number of threads increases from 8 to 258 Horizontally.

we should define this

All three strong scaling plots are very close to the ideal speedup line. The proportion of execution time spent on the part that can be parallelized is dominant [9]. The algorithm scales the best in the strong scaling with 32768 vertices compared to the rest lines, as it is the closest line to the ideal speedup line. The algorithm scales the worst in the strong scaling with 8192 vertices compared to the rest lines, as it is the furthest line to the ideal speedup line.

From figure 6, the larger the graph, the better speedup the algorithm can achieve with the same amount of threads. This result corresponds to the time composition in figure 1. For a fixed amount of threads, more vertices on each thread will lead to more time spent on thread computation and less time on MPI communication and rank 0 computation.

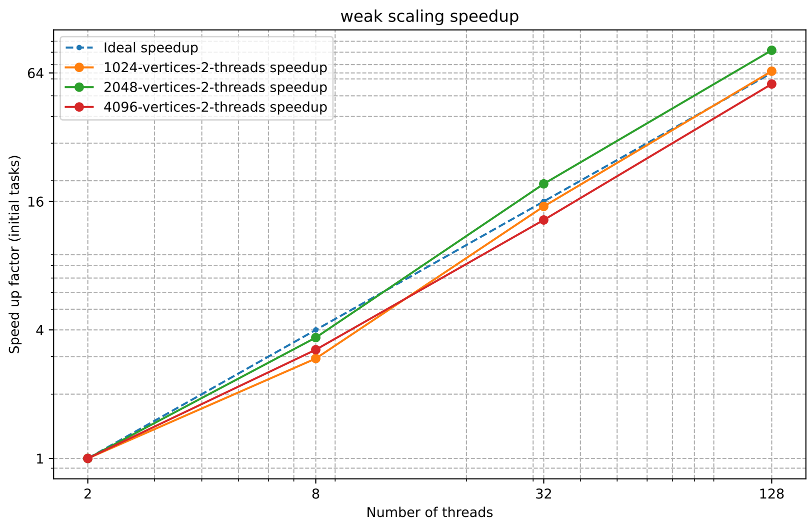


Figure 7: Weak scaling performance of $MST\_\log\log n$

4.4 Comparison with prim's algorithm

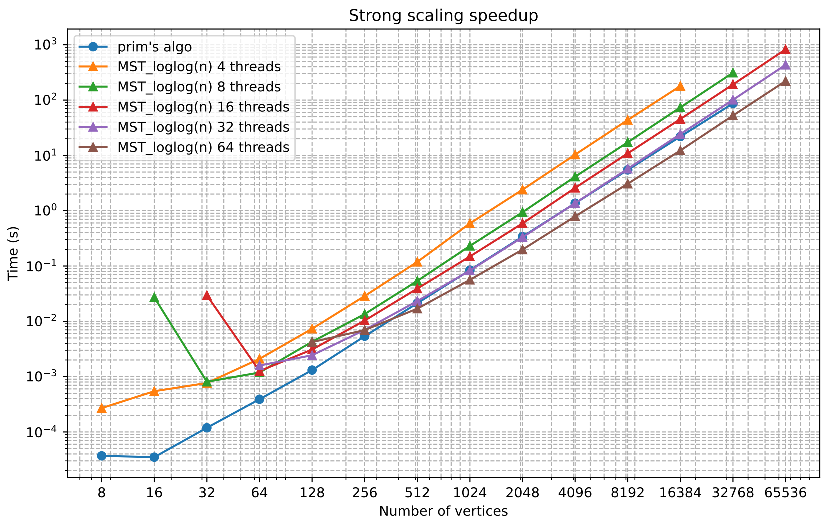


Figure 8: Prim's algorithm compared to $MST\_\log\log n$ performance

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

With 16G memory, prim's algorithm can process 32678-vertex complete graph (but not 65356), which has $32678 \times 32677/2(5.33 \times 10^8)$ edges, using 88 seconds ().

The bottleneck for prim's algorithm to process more edges is to fit the graph to the memory.

Because of the distributed nature of $MST\_\log\log n$, larger graphs can be fit into the memory as the number of threads increases. When working on graphs with the same size, the program needs more memory for MPI's buffer compared to the prim's algorithms.

if you use \Cref{} it will automatically say Figure 8. you will need to add it's package tho, i think it is cleveref

As shown in 8, with more than 16 threads, $MST\_\log\log n$ can process 65536-vertex complete graph (but not 131072). And $MST\_\log\log n$ outperforms the prim's algorithm by speed when there are more than 64 threads.

From 8 as the number of threads increases, the time taken by $MST\_\log\log n$ decreases. When

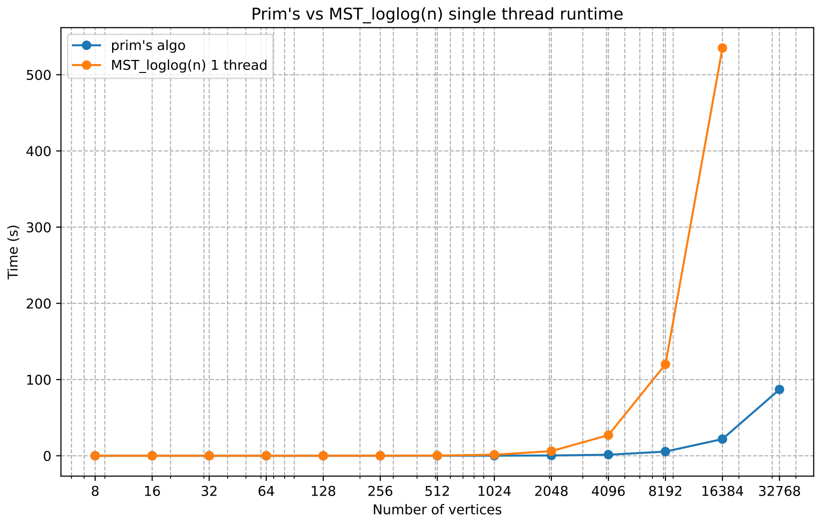


Figure 9: Prim's algorithm compared to $MST\_\log\log n$ performance with single thread

4.5 Python vs C++

5 Congested clique to CUDA

5.1 Congested Clique simulation

part of this can be used in the introduction too. It's ok if it's repeated.

$MST\_\log\log n$ is simulated with CUDA on Nvidia GPUs. Specifically, the Nvidia Tesla V100 GPU according to [5]. The V100 GPU has the capability for issuing 163,840 threads. Compared to CPU threads, GPU threads are more lightweight and can be issued in a larger number. This features allows each CUDA thread to simulate one vertex in the graph.

and it has significantly less computational power in comparison to a cpu thread

maybe we can also mention blocks and warps in a GPU

CUDA is "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]).

With CUDA, each thread simulates one machine in the Congested Clique model, which is a vertex in the clique graph. Kernels function are launched to run on the GPU. Each kernel function corresponds to a round in the Congested Clique. A round in the Congested Clique consists of communication and local computation in each machine, so as in the kernel function, each thread can do own location computation and a synchronized communication round. The kernel function an extra pointer that points to the GPU memory allocated for communication.

Threads can write and read from the memory space to achieve communication. Each thread has its chunk of memory space in the communication memory space. 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 as a simulation.

transition from definition of GPU and CUDA to how you use them to simulate things is not clear. maybe we can bundle things together and start the paragraph in a way that sets the tone for simulation process

Each kernel function is only responsible for one communication round. 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.

When there is only on machine active in the Congested Clique Model, 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.

The input graph of $MST\_\log\log n$ is generated on the CPU and copied to the GPU memory. Each thread learns its adjacent edges by reading the graph from the GPU memory.

step 1 $\longrightarrow$ min_to_cluster_kernel

these can be styled differently

step 2 $\longrightarrow$ min_from_cluster_kernel the min edges are not sent to the guardians
step 3 $\longrightarrow$ The memory is copied back to host memory and edges are collected
step 4 $\longrightarrow$ Local computation on the host
step 5 $\longrightarrow$ The updated cluster info is copied backed in to the Device memory. The threads will read about it in the next phase
step 6 $\longrightarrow$ Same as above

Maybe having a pseudocode/algorithm environment here is better. If not, then maybe using something like \begin{itemize} or enumerate give a better look

this is a very important part, tidying things up, having a structure that shows where the simulation techniques start and having an algorithm together with extensive epxplanation goes a long way.

you can also later refer to this as e.g. Algorithm 1. or Table 1. if you put it in a box or something.

5.2 Complete and sparse graphs

The Nvidia V100 GPU has the capability to issue a huge amount of threads but has limited to 16G / 32G. That means, with complete graphs, the number of vertices that can be simulated is limited as the $|E|$ grows quadratically corresponds to the $|V|$. As $|V|$ is limited and each thread simulates one vertex, the number of threads that can be issued is limited. This leads a waste of the GPU's capability.

limited to what?

Thus, Introducing sparse graphs address the under-utilization of the GPU's capability. For the same $|E|$, more threads can be issued to balance out the work load.

I think this section can be structured in 3 subsection/paragraphs: 1. limitations from hardware, hence the need for sparse graphs, 2. Algo works on sparse graphs, 3. actually generating random sparse graphs for input.

The 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, where 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 spares graphs as well. In the Congested Clique, the network graph is the equivalent as the graph to be solved, which is a complete graph / network. To solve the sparse graph, the network graph stays complete and the sparse graph is a subgraph of the network graph.

TODO proof that the algo works

A MST can always be found in a complete graph, but not in a sparse graph. To make a fair comparison for $MST\_\log\log n$ on complete and sparse graphs, the sparse graph is generated in a way that a MST can be found. The adjacency list of the sparse graph is generated in two steps.

First, a Prüfer sequence is generated with all the vertices and converted into a tree. A Prüfer sequence is an unique sequence of vertices that encodes a tree. The tree serves as the skeleton of the sparse graph making sure the graph is connected. The second step is to add edges tree randomly. When create the sparse graph, the expected rank and the number of vertices are given. So when the number of edges meets the rank requirement, the second step stops.

what is a prufer sequence

5.3 Best practices

$MST\_\log\log n$ on CUDA follows the best practices and common sense as it with MPI. All the tree information is maintained on the host and shared with the device when the kernel functions are launched. The tree is kept as a union-find data structure.

maybe have a command for it so the style remain consistent throughout the thesis?

(?) The simplification is not applied to the CUDA implementation, because the bandwidth is not a bottleneck anymore.

why? a good chance to remind the reader how the communication works

6 CUDA performance ✓ reasonable

6.1 Time composition

6.2 Comparison with Prim's on single threaded CPU

6.3 Comparison with MPI (time composition)

6.4 Scalability

6.5 Complete vs sparse graph

7 Conclusion

7.1 implementation compatibility

The MPI implementation is more natural than the CUDA implementation.

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*, 35(1):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)*, 2013, pp. 367-376. https://www.researchgate.net/publication/266659337_On_the_power_of_the_congested_clique_model

[3] "Puhti supercomputer" CSC - IT Center for Science, Accessed: Jul. 16, 2024. [Online]. Available: https://research.csc.fi/-/puhti

[4] "Puhti Service Description" CSC - IT Center for Science, Jun. 2021. [Online]. Available: 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, Accessed: Jul. 16, 2024. [Online]. Available: https://docs.csc.fi/computing/systems-puhti/

[6] "Slurm Overview" SchedMD, Accessed: Jul. 16, 2024. [Online]. Available: https://slurm.schedmd.com/overview.html

[7] "MPI Forum" Accessed: Jul. 16, 2024. [Online]. Available: https://www.mpi-forum.org/

[8] "Open MPI" Accessed: Jul. 16, 2024. [Online]. Available: https://www.open-mpi.org/

[9] "HPC Scaling," HPC Wiki, Accessed: Jul. 16, 2024. [Online]. Available: https://hpc-wiki.info/hpc/Scaling

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

[11] "CUDA C++ Programming Guide" NVIDIA Corporation, Accessed: Jul. 16, 2024. [Online]. Available: 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.