ASYNCHRONOUS DISTRIBUTED GRAPH ANALYTICS ON MULTI-GPU CLUSTERS USING NCCL AND CUDASTF

SKYLER RUITER*

1. Introduction. HPC systems continue to push the limits of their performance for their traditional dense linear algebra workflows, aiming for high spots on the Top500 list, which uses the LINPACK benchmark. While these systems attain impressive performance on these benchmarks, they often only pull orders of magnitude slower results from the Graph500 benchmark[9]. This is for many reasons, but generally modern HPC systems are not fundamentally built to execute graph algorithms efficiently, as witnessed by Fugakagu topping the Graph500 list, which has a design that maximizes bandwidth and minimizes latency[8] allowing it to outperform traditional systems.

One of the primary kernels in the Graph500 benchmark is Breadth-First-Search (BFS), a fundamental graph traversal method which is the basis for many graph algorithms. This method has a memory access pattern that is highly data dependent and is very difficult to effectively optimize for a distributed environment. One of the primary ways to optimize this kernel is to maximize the overlap between computation and communication, leveraging asynchronous operations and exposing parallelism where possible.

In this project, I implemented three versions of distributed BFS. The first is CPU based and uses MPI[6] for inter-process communication, this is to serve as the baseline to compare against the GPU implementations and demonstrate the benefits of the CPU architecture. There are two GPU versions, one utilizing NVIDIA Collective Communications Library (NCCL)[1] and MPI, and another which expands on that version by adapting it to use the CUDA Sequential Task Flow (CUDASTF) programming model[2]. The basic pseudo-code for the implementations is provided in Figure 1.1.

I aim to analyze the performance of the CUDASTF implementation of distributed BFS in comparison to the non-CUDASTF and non-GPU versions in terms of communication and computation and see if CUDASTF is a viable optimization strategy to avoid the complex by-hand asynchronous optimizations and push the performance of distributed graph algorithms on the GPU.

2. Background and Related Work. BFS is an algorithm which systematically visits all vertices in the graph starting from a source vertex, creating a tree of all nodes that the algorithm visits. It's a core method for more complex graph problems, such as finding the diameter and community structures [3]. Due to the search being very data-dependent and strategies for effective optimization varying based on graph characteristics, there is much work involved in developing new BFS versions[3]. On its own, BFS has a number of challenges to its optimization, but distributed graph algorithms have a number of problems as a whole. As mentioned in this survey[5], distributed graph algorithms are challenged often by parallelism, load balancing, communication overhead, and bandwidth. Each of these has had lots of research into how to best work around them, but there is often no one-size fits-all solution to an optimized distributed graph algorithm.

One way in which programmers can gain a lot of performance for cheap is to have another team create very efficient code as a library to apply to a problem. There are a number of these libraries, but in this project I am focused on using MPI, NCCL, and CUDASTF, all libraries aimed at optimizing code for performance and ease of use.

*Indiana University, sruiter@iu.edu

```
0   partition graph
1   initialize frontier
2   while (global_neighbors > 0):
3       for (vertex in current frontier):
4           add local neighbors to next frontier
5           add non-local neighbors to seperate buffer
6       organize non-local neighbors for communication
7       pass vertices to other ranks
8       process recieved vertices into next_frontier
9       check for termination and swap frontiers
```

FIG. 1.1. *Distributed BFS Pseudo-code*

MPI and NCCL are collective communication libraries, allowing for message passing between processes either on the same node or across nodes, with their main difference being that MPI generally is used for passing memory between global CPU memory whereas NCCL is for passing messages between global GPU memory. There is such a thing as CUDA-aware MPI, however BigRed200 does not have this implemented, therefore we will be ignoring it for this project.

CUDASTF is a task framework released recently that aims to organize computation as asynchronous tasks with data-dependencies, and derive a task graph from these dependencies[2]. By writing code in this model, CUDASTF can use the task graph it builds from the data dependencies between tasks to maximize asynchronous work. The library also provides seamless scaling to multiple GPUs, but that feature is left for future work to implement in this project, as we focus on the gains from asynchronous task scheduling. An example of the task graph created for the CUDASTF implementation cut off after a few tasks is presented in Figure 2.1.

3. System Architecture and Implementation.

3.1. Experimental Setup. These experiments were run on Indiana University's BigRed200 supercomputer, specifically on its GPU nodes. These nodes each have 512GBs of RAM, one 64-core 2.0GHz AMD EPYC 7713 CPU, and four NVIDIA A100-SXM4 40GB GPUs. They run SLES version 15 and CUDA version 12.2 was used. I ran the experiments with 2 nodes and 2 GPUs per node. I was not able to find information on the inter-node interconnects, unfortunately. These implementations were tested on five different datasets. Four of these datasets are different sizes of Graph500 scale-free randomly generated graphs. Specifically, I used the scale 18, 19, 21, and 23 from Network Repository[7]. From the same repository I also used the web-uk-2002 dataset which is a large web graph and directed, as opposed to the undirected Graph500 datasets.

3.2. BFS Algorithm Overview. At its core the BFS algorithm is very simple, look at all neighbors of the current depth of the tree and add them as the next layer, and repeat until no nodes are left to add to the tree, and the top of the tree being the source vertex. Distributing this algorithm quickly devolves into a complex optimization problem however. First, there is the graph partitioning, with the best usually being the METIS partitioning[4]. Opting for simplicity I simply partitioned the graph using the modulus operator, with each rank owning each edge vertex that equals zero when modulused with the rank. I also store the outgoing edge from vertices owned by the rank in that same rank. The rest of the algorithm is outlined in 1.1, where I next add local neighbors to the next frontier, and non-

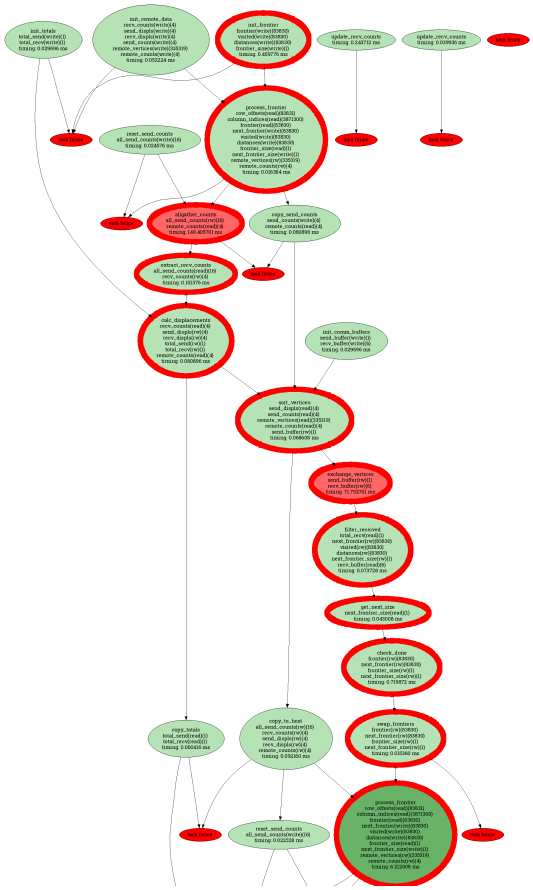

FIG. 2.1. *CUDASTF Visualized Task Graph (With relative timing coloring)*

local ones to a separate buffer. This buffer is then organized by each rank to be all-to-all communicated efficiently, where each rank then organizes it's received vertices into the next frontier, and then we repeat the loop if there are more layer to process.

For the CPU based implementation I am able to utilize standard library vectors and more powerful MPI collective operations allowing me to attain a simple implementation that even single threaded is somewhat optimized and not difficult to create. I did build and test an OpenMP multi-threaded version, but the large amount of atomic operations made the implementation slow and clunky, so I decided to stick with the single threaded version.

The first GPU implementation uses no MPI communication in the main loop, instead utilizing direct GPU-to-GPU memory operations utilizing NCCL. This step avoids having to move memory to the CPU first and then using MPI, which woud cause massive performance losses due to the extra communication time. The NCCL library however doesn't have a variable alltoall gather operation however, leaving the programmer to implement this. Utilizing almost exclusively device memory keeps the complexity down in the codebase, with the kernels also being small and simple generally.

The CUDASTF version of the code doesn't interface perfectly with NCCL, making the programmer move some values off the GPU to be used in the collective communication operation. Outside the few quirks of the library, the tasks force the programmer to understand

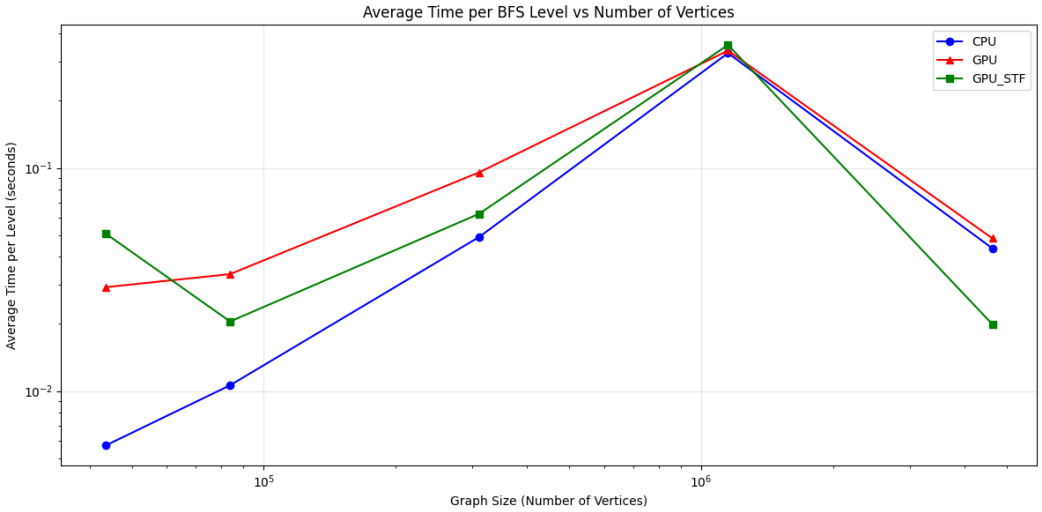

FIG. 4.1. *Plot showing average time per BFS level over the number of vertices in the traversed graph, with the CPU generally outperforming the GPU implementations, and the CUDASTF implementation being faster than the regular NCCL version.*

their algorithms data dependencies, and how their work being done asynchronously might have unintended consequences.

4. Results. Measuring each of the implementations, the first I looked at was the average time per BFS level. This will give a good idea of how performant the implementations are across the wide range of communication sizes expected to happen. We can see these values plotting in Figure 4.1. This plot shows us that the CPU implementation beats out both the GPU implementations for smaller graphs, with the margin decreasing slowly as the graph size increases, until we get to the largest graph and the STF version beats out the CPU version. The STF version almost always matches or exceeds the performance of the non-STF version, but only outperforms the CPU version at very large graph sizes.

To see in more detail where the gains are being achieved from we can look at the differences between the communication and computation times. To note however, these results should be taken with a degree of skepticism, as the STF version is difficult to gain detailed performance results on using a simple CPU timer as the asynchronous nature doesn't allow us to capture the meaningful overlapping work occurring. We can see this breakdown plotting in Figure 4.2. Here we can first see that all of the smaller graphs are clustered together, and are blown out into a subplot that somewhat mirrors the overall graph. Together with Figure 4.3, we can analyze the communication versus computation times of the implementations.

The CPU implementation has little communication time compared to its computation, as seen in Figure 4.3, and with Figure 4.2 we can see that the computation is relatively in line with the GPU implementations, this must be because the NCCL communication is slower in this instance than the MPI communication. For the GPU implementations we see that they experience a higher proportion of their runtime as communication. The main distinction here is that the GPU implementation with STF tends to scale its communication time better, with larger graphs undergoing less communication time than smaller ones in comparison to the non-STF version. As mentioned previously however, these values are misleading, as the communication times are not guaranteed to be measured accurately and

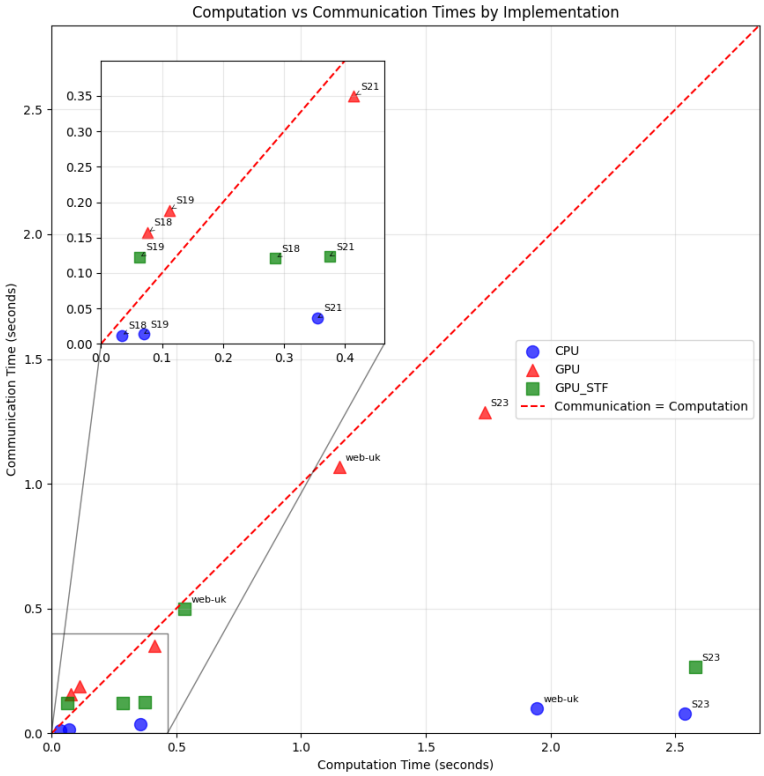

FIG. 4.2. *Plot showing the communication and computation times together with the diagonal being where communication equals computation time. Shows how the three algorithms scale as the dataset changes.*

the finer grain measurements don't consider overlapping communication and computation.

5. Discussion. The results demonstrate the nuanced trade-offs BFS has for being run on the CPU versus the GPU. With the CPU we are able to leverage lower latency instructions, better cache, and better data-dependent execution. The GPU has impressive memory bandwidth and parallelism however, meaning if we can leverage it even sub optimally it could increase performance. What we found however is that the question can depend on the graph size and communication structure, as we saw the STF model being most performant at the largest data point and if we could lower communication costs on the GPU further we could continue to optimize the GPU versions. For other graph algorithms we would need to analyze how much communication is needed and how well the computation can be parallelized by the GPU, and since BFS is difficult to parallelize and has very irregular communication the GPU optimizations are limited.

The CUDASTF version of the code is promising as a method to leverage an asynchronous tasking model for distributed graph algorithms. More research is needed for how to compare the communication and computation times of asynchronous distributed algorithms versus not on NVIDIA GPUs as the already unreliable timers are even less meaningful when working with asynchronous tasks. From what we can see however, it seems that CUDASTF helps to mitigate the cost of data movement and latency for difficult to optimize graph algorithms by processing work asynchronously when possible, while keeping the programming itself as simple as possible.

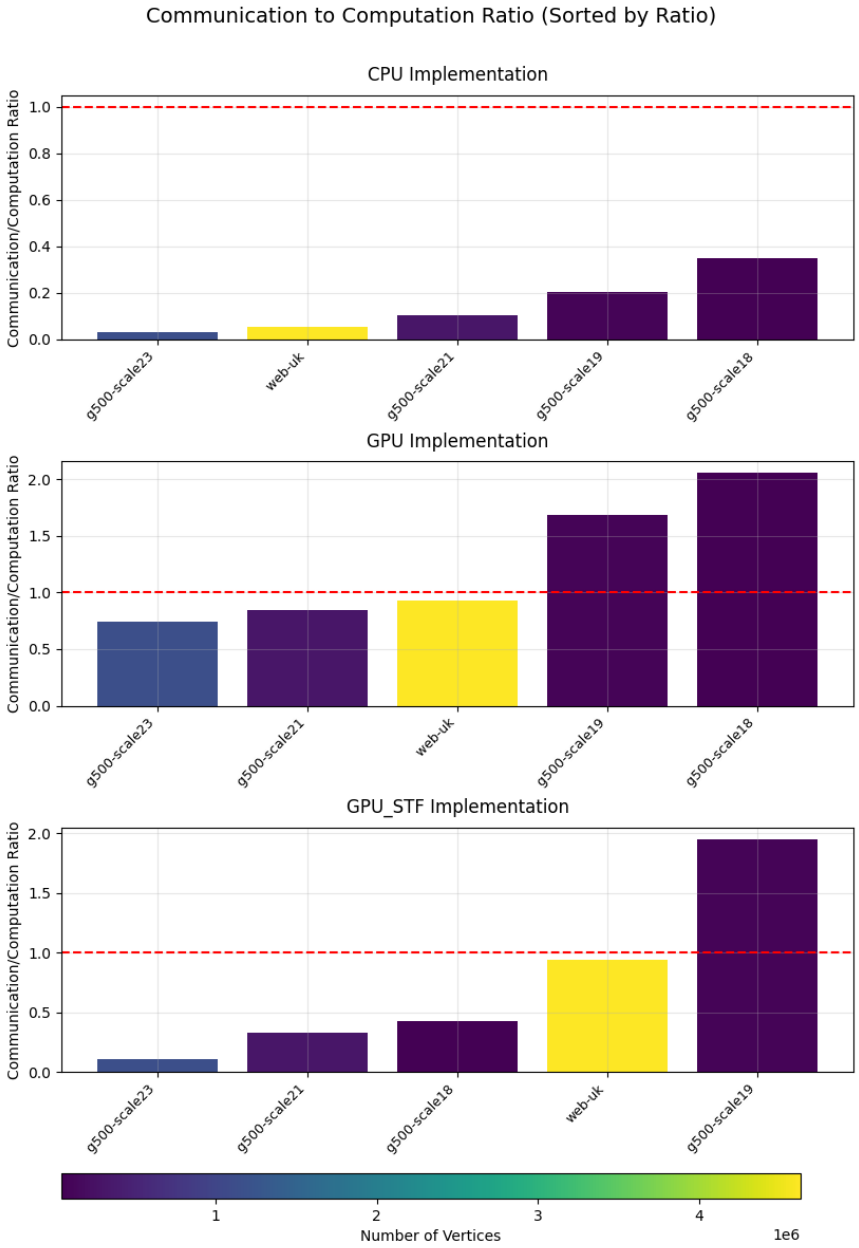

FIG. 4.3. *Plot showing the communication computation time ratio, with each bar colored in to reflect the number of vertices in the dataset. Shows the CPU implementation has far less communication time than the GPU implementations.*

6. Limitations and Future Work. There were many difficulties in building these implementations. The primary one being that to my knowledge, this is the only existing codebase to use specifically CUDASTF, NCCL, and MPI in the same codebase, especially for a graph algorithm. The lack of existing documentation and examples means there is likely room left for optimizing the particulars of the tasking model to fully leverage its usefulness.

Due in part to this bottleneck, and the limited time and scope of the project, only BFS was implemented when originally PageRank was also to be included and compared. I was also limited in the system architecture available to me, as I only had access to one cluster with multiple nodes and GPU accelerators and only one type of GPU. The limitations of the system I had access to meant I also had difficulty managing to build a suite of testing datasets that reached into the dozens of GB's.

With further time and resources, I could build out a further optimized version of all three implementations, fully leveraging the strengths of the CUDASTF library and maximizing overlapping efforts. One direction I thought would be to allow for inter-node communication using NCCL and intra-node processing using CUDASTF, but I was not able to achieve this level of granularity.

REFERENCES

[1] *Nvidia collective communications library (nccl).* `https://github.com/NVIDIA/nccl`.

[2] C. Augonnet, A. Alexandrescu, A. Sidelnik, and M. Garland, *Cudastf: Bridging the gap between cuda and task parallelism*, in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '24, IEEE Press, 2024.

[3] M. Bisson, M. Bernaschi, and E. Mastrostefano, *Parallel distributed breadth first search on the kepler architecture*, IEEE Transactions on Parallel and Distributed Systems, 27 (2016), pp. 2091–2102.

[4] G. Karypis and V. Kumar, *Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices*, Technical Report 97-061, University of Minnesota, Department of Computer Science and Engineering, 1997. Supported by IST/BMDO through Army Research Office contract DAAH04-93-G-0080 and by Army High Performance Computing Research Center under cooperative agreement DAAH04-95-2-0003.

[5] L. Meng, Y. Shao, L. Yuan, L. Lai, P. Cheng, X. Li, W. Yu, W. Zhang, X. Lin, and J. Zhou, *A survey of distributed graph algorithms on massive graphs*, ACM Comput. Surv., 57 (2024).

[6] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[7] R. A. Rossi and N. K. Ahmed, *The network data repository with interactive graph analytics and visualization*, in AAAI, 2015.

[8] M. Sato, Y. Kodama, M. Tsuji, and T. Odajima, *Co-design and system for the supercomputer "fugaku"*, IEEE Micro, 42 (2022), pp. 26–34.

[9] K. Ueno and T. Suzumura, *Highly scalable graph search for the graph500 benchmark*, in Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC '12, New York, NY, USA, 2012, Association for Computing Machinery, p. 149–160.