Breadth-First Search over GPU Clusters using MPI and CUDA

Balaji Shashipreeth Racherla

1. Abstract

This project focuses on the implementation and performance analysis of a Breadth-First Search (BFS) algorithm designed to operate on large-scale graphs using a distributed GPU cluster. The core objective is to leverage the parallel processing capabilities of CUDA-enabled GPUs and the message-passing interface (MPI) to efficiently traverse extensive graph structures. The input graph data, sourced from SNAP (Stanford Network Analysis Platform), is first converted into the Compressed Sparse Row (CSR) format. This CSR representation is then distributed across multiple machines in the cluster.

The BFS algorithm is executed in a work-split manner, where each machine processes a portion of the graph's frontier in parallel. The primary output of the program is a level vector, which stores the shortest distance (in terms of number of edges) from a specified source vertex to all other reachable vertices in the graph. This report details the design, implementation specifics, compilation process, execution instructions, and a comparative analysis of the performance of the program

2. Rationale

The Breadth-First Search algorithm is a fundamental graph traversal technique with wide-ranging applications, including finding the shortest path in unweighted graphs, network analysis (e.g., social networks, web graphs), and solving puzzles. As real-world graphs, such as social networks (like LiveJournal) or road networks (like the California road network), continue to grow massively in size, with millions of nodes and tens of millions of edges, traditional single-machine BFS implementations become prohibitively slow or infeasible due to memory constraints.

This project addresses the challenge of performing BFS on such large graphs by distributing the computational load and data across a cluster of machines, each

equipped with a GPU. GPUs offer massive parallelism suitable for graph algorithms where operations on many vertices or edges can often be performed concurrently. MPI provides the communication backbone for coordinating the distributed processes and managing data exchange (like frontier information) between machines. By combining CUDA for intra-node parallelism and MPI for inter-node parallelism, this project aims to develop a scalable BFS solution capable of handling very large graphs and delivering significant speedups over traditional approaches. The ultimate goal is to produce a level vector indicating the shortest path distances, which is a common requirement in many graph analysis tasks.

3. Program Organization and Design

3.1. Overall Architecture

The BFS algorithm is implemented to run on a distributed GPU cluster. The overall workflow, as depicted in the presentation, can be summarized as follows:

1. Initialization (at ROOT node):
    - Load the graph data from an input file (e.g., data from SNAP) in the form of a csv file which contains the graph data in Compressed Sparse Row (CSR) format. The CSR format consists of srcPtrs (row pointers) and dst (column indices) arrays.
    - Initialize necessary data structures: level array (stores the BFS level/distance for each vertex, initialized to infinity/UINT32_MAX), prevFrontier array (stores vertices in the current BFS frontier), and currFrontier array (stores newly discovered vertices for the next frontier).
    - The starting vertex for BFS is marked with level 0.

2. Initial Local BFS (at ROOT node):
    - Perform a few iterations of BFS locally on the ROOT node using a CPU-based BFS (cpuBFS in bfs_mpi.cpp) until the currFrontier size reaches a significant threshold (e.g., 1000 vertices as mentioned in bfs_mpi.cpp). This is done to quickly expand the frontier before distributing the work.

3. Initial Scatter From Root:

    - The numCurrFrontier and currLevel is broadcasted to all nodes
    - If numCurrFrontier is greater than zero we broadcast level and currFrontier Vectors
    - Using numCurrFrontier we distribute the work among workers
    - Once we get our slice of work we do a cpuBFS
    - After cpuBFS we gather the variable number of frontiers in two steps
    - First we gather myNumCurrFrontier from all nodes which contain data about no of nodes discovered at each node
    - Next using this information we create recvCounts and displacements to use with MPI_Gatherv and gather all the nodes to everyone
    - We also sync the level vector using the gathered nodes in currFrontier vector

4. Distribution and Parallel GPU BFS:

    - Copy Level to GPU
    - Using the currFrontier gathered in previous level and copy required data to gpu i.e the slice of work
    - And perform BFS in GPU using gpuBFS

5. Synchronization and Aggregation:

    - After each GPU BFS step, the number of newly discovered frontier vertices (myNumCurrFrontier) from each process is gathered using MPI_Allgather (bfs_mpi.cpp).
    - The local myCurrFrontier arrays from all processes are then gathered into a global currFrontier array on all nodes using MPI_Allgatherv (bfs_mpi.cpp).
    - The global level array on each GPU is updated with the newly discovered levels using the updateLevel function which involves a CUDA kernel (updateLevelKernel in bfs_cuda.cu) to reflect changes from all processes.

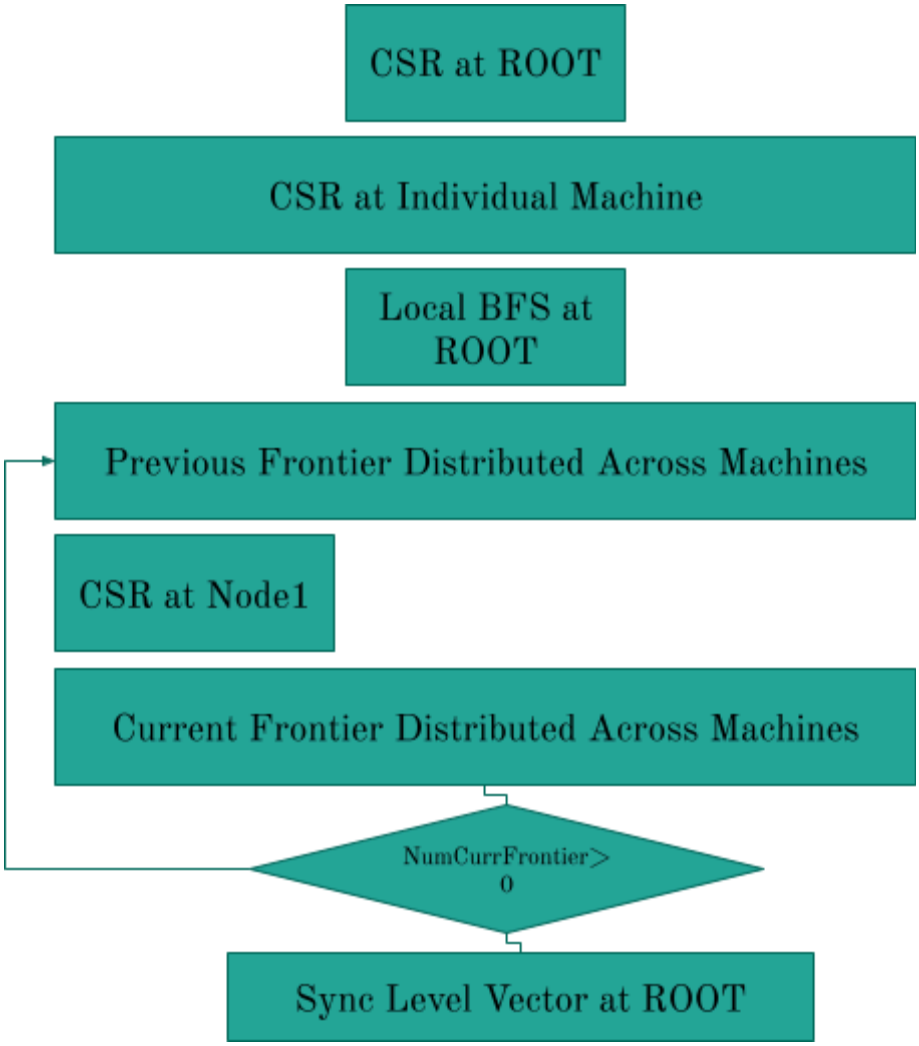
6. Iteration and Termination:

    - The global currFrontier becomes the prevFrontier for the next iteration.
    - The process repeats as long as the numCurrFrontier is greater than zero and exceeds a certain threshold (e.g., 1000). We continue this until the number of nodes discovered become small enough that we don't gain any performance gains as transfer to device memory becomes costlier than computation on GPU
    - If numCurrFrontier drops below the threshold but is still greater than zero, the remaining BFS steps are completed by the ROOT node using cpuBFS for efficiency with smaller frontiers

7. Output:

    - Once the BFS is complete (no new frontiers are generated), the final level array is copied from the GPU of the ROOT node (if applicable) back to its host memory. This vector represents the shortest distance from the source vertex to all other nodes.

The following figure, adapted from the presentation, illustrates the high-level data flow:



3.2. Core Modules and Functions

3.2.1. MPI Communication Module (bfs_mpi.cpp)

This module handles the overall orchestration of the distributed BFS algorithm, including data distribution, work assignment, and synchronization among MPI processes.

- main function:
    - Initializes MPI environment (MPI_Init).
    - Loads graph data (CSR format) on the ROOT process using loadCSR.
    -
    - Broadcasts graph metadata (lengths of arrays) and the CSR data (srcPtrs, dst) to all processes (MPI_Bcast).
    - Calls copyCSRToGPU to allocate and transfer graph data to each node's GPU memory.
    - Calls frontierParallel to execute the main BFS logic.
    - Prints timing results and finalizes MPI (MPI_Finalize).
- frontierParallel function:
    - Implements the iterative BFS process described in the architecture.
    - Manages the initial local BFS on the ROOT node if my_rank == 0.
    - Handles the distribution of the frontier (MPI_Bcast, MPI_Scatterv implicitly through work division logic).
    - Calls gpuBFS (for GPU execution part) or cpuBFS (for CPU execution part) for its portion of the work.
    - Uses MPI_Allgather to collect the number of newly discovered frontiers from each process.
    - Uses MPI_Allgatherv to gather all local current frontiers into a global current frontier on all nodes.
    - Calls updateLevel to synchronize the level array on the GPU after gathering frontiers.
    - Manages the loop termination conditions (e.g., numCurrFrontier > 0 and numCurrFrontier > 1000 for GPU phase).
    - If the frontier becomes small, it consolidates the remaining work on the ROOT node for CPU processing.
- cpuBFS function:
    - A standard CPU-based BFS implementation. It iterates through the vertices in prevFrontier. For each vertex, it explores its neighbors using the srcPtrs and dst arrays. If a neighbor is unvisited (level[neighbor] ==

UINT32_MAX), its level is updated to currLevel, and it's added to currFrontier. The count numCurrFrontier is updated.

- loadCSR function:
    - Reads a graph from a CSV file where the first line contains comma-separated row pointers (srcPtrs) and the second line contains comma-separated column indices (dst).

3.2.2. CUDA Kernel Module (bfs_cuda.cu)

This module contains the CUDA C/C++ code for GPU-accelerated BFS operations.

- gpuBFS (extern "C" wrapper function):
    - This C-style function is called from bfs_mpi.cpp.
    - It handles memory transfers between host and device for the relevant portion of prevFrontier and numCurrFrontier.
    - Launches the CUDA BFS kernel (bfs_kernel_shared_memory or bfs_kernel_global_memory).
    - Copies results (numCurrFrontier, currFrontier) back from device to host.
- bfs_kernel_shared_memory / bfs_kernel_global_memory (CUDA Kernels):
    - These are the core GPU BFS kernels. Each thread processes one or more vertices from its assigned portion of the prevFrontier.
    - It iterates through the neighbors of its assigned vertex.
    - atomicCAS (Compare-And-Swap) is used on the global d_level array to ensure that a neighbor's level is updated only once and by only one thread, setting it to currLevel if it was UINT32_MAX.
    - If a thread successfully updates a neighbor's level, it adds the neighbor to the d_currFrontier array using atomicAdd to update d_numCurrFrontier and get a unique index for storing the new frontier vertex.
    - The bfs_kernel_shared_memory version attempts to use shared memory (currFrontier_s) for temporarily storing newly found frontier vertices within a block to reduce global memory atomics, committing to global memory in a coalesced manner. This is likely beneficial if many threads in a block find frontiers.
- copyCSRToGPU function:
    - Allocates memory on the GPU for d_srcPtrs, d_dst, d_level, d_prevFrontier, d_currFrontier, and d_numCurrFrontier.
    - Copies srcPtrs and dst arrays from host to device.
- copyLevelToGPU function:
    - Copies the host level array to the d_level array on the GPU.

- copyLevelToHost function:
    - Copies the d_level array from GPU to the host level array.
- updateLevelKernel (CUDA Kernel):
    - This kernel is launched after frontiers from all MPI processes are gathered.
    - It takes the global d_currFrontier (populated with all newly discovered vertices) and updates the d_level array on the GPU for these vertices to the currLevel. This step ensures that the d_level array on each GPU is consistent before the next BFS iteration.
- freeDeviceMemory function:
    - Frees all allocated GPU memory.

3.2.3. CPU BFS Implementation (bfs_mpi.cpp)

The cpuBFS function within bfs_mpi.cpp provides a serial BFS logic used by the ROOT node for the initial BFS phase and potentially for the final cleanup phase if the frontier size becomes very small. Its logic is standard: iterate the current frontier, visit unvisited neighbors, update their levels, and add them to the next frontier.

3.2.4. Data Loading and Preprocessing (bfs_mpi.cpp)

The loadCSR function in bfs_mpi.cpp is responsible for reading the graph data. It expects a CSV file where the first row contains the CSR row pointers (offsets into the destination array) and the second row contains the CSR column indices (the actual destination vertices of edges). This data is read into host memory before being distributed and copied to GPUs.

4. Compilation and Execution

4.1. Compilation

The project is compiled using the provided makefile. The key compilation steps and components involved are:

- CUDA Toolkit Path: The CUDA_PATH variable points to the location of the CUDA Toolkit, defaulting to /usr/local/cuda.
- Host Compiler: The HOST_COMPILER is typically g++ or clang++. The makefile allows specifying a custom host compiler via the HOST_COMPILER make variable.
- NVCC: The CUDA code (.cu files) is compiled using nvcc.

- Build Process:
    1. bfs_cuda.cu is compiled into bfs_cuda.o using nvcc with ALL_CCFLAGS and GENCODE_FLAGS.
    2. bfs_mpi.cpp is compiled into bfs_mpi.o using mpicxx with INCLUDES and MPI_CCFLAGS.
    3. The object files bfs_mpi.o and bfs_cuda.o are linked together using mpicxx with MPI_LDFLAGS and CUDA runtime libraries (-lcudart) to create the final executable bfs.
    4. The executable is then copied to a path like ./bin/somePath/bfs.
    5.

4.2. Execution

The program is executed using mpirun (or a similar MPI job runner like mpiexec).

Command Syntax:

Bash

mpirun -np <number_of_processes> --output-filename ../logs/out_<nprocs>_procs -hostfile myHosts ./bin/pathToBinary/bfs <csr_file_path> <start_vertex_id>

- -np <number_of_processes>: Specifies the number of MPI processes to launch.
- --output-filename ../logs/out_<nprocs>_procs: (Specific to some mpirun versions like OpenMPI) Redirects stdout/stderr from processes to specified files.
- -hostfile myHosts: Specifies a hostfile listing the machines to run on. This is necessary for multi-machine execution. For local execution (all processes on one machine), this might be optional or configured differently depending on the MPI implementation.
- ./bin/<arch>/<os>/<build_type>/bfs: Path to the compiled executable.
- <csr_file_path>: Path to the input graph file in the specified CSV-based CSR format. The makefile defaults this to ../graphData/csr_LiveJournal.csv.
- <start_vertex_id>: The 0-indexed ID of the vertex from which to start the BFS. The makefile defaults this to 0.

5. Results and Analysis

5.1. Experimental Setup

The experiments were conducted using graph data from SNAP, specifically the "LiveJournal" social network and the "roadNet-CA" California road network. The MPI+CUDA configurations. The MPI+CUDA configuration was tested both locally (multiple processes on one machine) and on a network, though results for networked execution are noted to be significantly slower if not optimized.

5.2. Performance Metrics

The primary performance metric collected is the execution time in milliseconds (ms) for the BFS traversal. Additionally, the maximum level discovered during the BFS is recorded, which indicates the diameter of the graph component reachable from the source vertex.

The provided metrics are:

Graph	No of Graph Nodes	No of Graph Edges	Serial Execution Time (ms)	CUDA Execution Time (ms)	MPI+CUDA Execution Time (ms) (2 procs locally)	Max Level Discovered
LiveJournal (A social Network)	4847571	68993773	917	257	647 (>5000 on network)	14
roadNet_CA (a network of roads in state of California)	1971281	5533214	107	616	1715	545

6. Conclusions

6.1. Achievements

The project successfully implemented a distributed Breadth-First Search algorithm leveraging both MPI for inter-node parallelism and CUDA for intra-node GPU acceleration. The primary achievement was the development of a framework capable of loading graph data in CSR format, distributing the BFS workload, and computing the level (shortest distance) of each vertex from a given source. The implementation incorporates mechanisms for GPU memory management, including persisting the level array in global GPU memory to reduce data transfers, and CUDA kernels for parallel frontier expansion.

The system was tested with large graph datasets (LiveJournal and roadNet-CA), demonstrating its capability to handle graphs with millions of nodes and tens of millions of edges. Performance metrics were collected, showing that for certain graph types (like social networks), the CUDA-accelerated version can significantly outperform a serial CPU implementation. The pseudocode and architecture illustrate a clear strategy for parallelizing the BFS algorithm across a cluster.

6.2. Limitations and Discrepancies

- Performance on Sparse/Deep Graphs: The results indicate that for graphs like roadNet-CA, which are sparse and lead to deep BFS traversals, both the CUDA-only and MPI+CUDA (with 2 local procs) versions performed worse than the serial CPU version. This suggests that the overhead associated with GPU kernel launches, data transfers, and MPI communication can outweigh the benefits of parallelism for such graph structures, especially when the frontier size per iteration is small.
- MPI+CUDA Scalability with Few Local Processes: The MPI+CUDA version with 2 local processes showed slower times compared to the single-GPU CUDA version for the LiveJournal graph. This is likely due to MPI communication and synchronization overheads becoming dominant when the degree of parallelism is low and processes are co-located. The true benefits of MPI+CUDA are expected with a larger number of distributed nodes
- Initial Local BFS Threshold: The strategy of performing initial BFS levels on the ROOT node until the frontier reaches 1000 nodes before distributing work (bfs_mpi.cpp) is a heuristic. Its optimality might vary depending on the graph structure and cluster configuration.

Overall, while a functional MPI+CUDA BFS framework was achieved, its performance benefits are highly dependent on the graph characteristics and the scale of the distributed environment. The project successfully explored the complexities of combining MPI and CUDA for graph traversal.

7. Future Work

The current project lays a solid foundation for BFS on GPU clusters. Several avenues exist for further exploration and enhancement, building upon the suggestions for improvement and the ideas presented:

1. Asynchronous Communication and Computation Overlap:
    - Utilize non-blocking MPI operations (MPI_Isend, MPI_Irecv, MPI_Ibcast, MPI_Iallgather, MPI_Iallgatherv) to overlap communication phases (e.g., gathering current frontiers, broadcasting next frontier) with computation (GPU kernel execution for the next level or other graph processing tasks).
    - This would require careful management of MPI request handles and synchronization points (MPI_Wait, MPI_Test) to ensure data consistency while maximizing resource utilization.
2. CUDA-Aware MPI:
    - Integrate CUDA-aware MPI capabilities if the MPI implementation and hardware support it. This allows MPI functions to directly operate on GPU device memory buffers (d_level, d_currFrontier, etc.), eliminating explicit host-to-device and device-to-host cudaMemcpy calls before and after MPI communication. This can significantly reduce data transfer overheads, especially for frequent synchronizations of the level array or frontier data.
    - Functions like MPI_Allgatherv could directly gather data from GPU buffers of all ranks into a GPU buffer on each rank.