MPI-Based Parallelization of Transformer Models

(BERT Optimization)

Assignment Report

Abishek Chakravarthy
Roll Number: CS22B2054
IIITDM

February 2025

Abstract

This report documents the parallelization of computationally intensive components of a Transformer encoder model, inspired by BERT, using the MPI (Message Passing Interface) programming model. The implementation distributes operations like matrix multiplication, layer normalization, softmax, GELU, and self-attention across multiple processes to reduce execution time. Benchmarking revealed a significant speedup of 17.68x when using 8 MPI processes compared to the serial implementation. This report covers design, methodology, timing results, and performance analysis of the MPI-based optimization.

0.1. Introduction

Transformer-based models like BERT have become foundational in NLP, delivering state-of-the-art results across diverse applications. However, these models are computationally expensive due to operations like self-attention, large matrix multiplications, and deep feedforward networks.

This project aims to parallelize these operations using MPI, a message-passing model well-suited for distributed memory systems. Unlike shared memory models such as CUDA, MPI allows processes running on different nodes or cores to communicate and collaboratively solve large problems by splitting both computation and memory.

The primary objective is to reduce inference time by distributing core components of a Transformer encoder:

- Matrix Multiplication
- Layer Normalization
- Positional Encoding
- GELU Activation
- Softmax
- Dropout
- Self-Attention
- Feedforward Network

0.2. Methodology

The Transformer encoder is implemented in two forms:

1. A **Serial Version** using standard C++ for baseline timing.
2. A **Parallel MPI Version** that leverages distributed computation.

0.2.1. Parallelization Strategy

- **Matrix Operations:** Matrix multiplication is parallelized by row-wise splitting among processes. Each process computes a subset of output rows.
- **Layer Normalization:** Each process computes normalization statistics (mean and variance) for a subset of rows and contributes to the full matrix output.
- **Dropout and GELU:** These point-wise operations are also computed in parallel per process block. A shared seed ensures deterministic behavior across processes.
- **Self-Attention:** Key attention mechanisms—projection, score computation, and final aggregation—are parallelized with collective MPI communication (e.g., `MPI_Allreduce`).
- **Feedforward Layers:** Two dense layers with GELU activation in between are parallelized like matrix multiplication.

0.2.2. Communication Techniques

- `MPI_Bcast`: Used for broadcasting input matrices and shared weights.
- `MPI_Gatherv`: Assembles partial results from all processes.
- `MPI_Allreduce`: Aggregates results such as max values (for softmax), attention scores, or final vectors.

0.2.3. Benchmarking

Each process contributes to computing a portion of the encoder output. Execution time is measured using `chrono` for serial runs and MPI `Barrier + chrono` for parallel runs.

0.3. Experimental Results

The experiments were conducted using an input of sequence length 1024 and embedding dimension 1024. The transformer encoder layer was benchmarked using 8 MPI processes.

Table 1: MPI vs Serial Transformer Encoder Performance

Processes	MPI Time (s)	Serial Time (s)	Speedup
8	24.890278	440.170098	17.68x

This result shows a significant reduction in execution time by distributing computation across processes.

0.4. Observation and Analysis

- **Effective Workload Distribution:** The row-wise partitioning of matrix operations ensures that each process handles a roughly equal portion of the workload, contributing to the high speedup.
- **Collective Communication Overhead:** While MPI allows efficient scaling, operations such as `MPI_Bcast` and `MPI_Allreduce` introduce synchronization overhead. These overheads become more significant with increasing layers or more frequent communication.
- **Deterministic Output:** By using fixed seeds for matrix initialization and random sampling, reproducibility is ensured across all MPI processes.
- **Scalability:** The current implementation is tested on 8 processes. Further testing on more processes would help analyze strong and weak scaling behaviors.
- **Component-Wise Speedup:** Matrix multiplication, layer norm, and GELU see near-linear speedups. However, operations with large reductions (like softmax) face bottlenecks due to global synchronization needs.
- **Bottleneck Identification:** The dominant bottlenecks arise from the softmax computation (due to max + sum reductions) and the self-attention projection steps where inter-process communication is more frequent.

0.5. Conclusion

This report demonstrated the successful parallelization of a Transformer encoder layer using MPI. With 8 processes, the parallel implementation achieved a **17.68x speedup**, reducing the execution time from 440 seconds to just under 25 seconds. This acceleration validates MPI's effectiveness for high-performance NLP workloads. Future work can include multi-layer support, pipeline parallelism, and hybrid MPI+OpenMP strategies for intra-node and inter-node parallelism.