WITS UNIVERSITY
School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040A & COMS7045A Course Project

Hand-out date: May 2, 2025
Due date: June 1, 2025

General Information

1. You are expected to work in a group of 2 persons (or individually) for this project.
2. Each group is required to work on both problems, where one is to use MPI, and the other to use CUDA.
3. The descriptions of the problems are only outlines. For your report, the problem must be set up more specifically.
4. Start early and plan your time effectively. Meet the deadlines to avoid late submission penalties (20% to 40%).

Deliverables and Evaluation

1. Report — must be written in a double-column, 10pt, single line spacing using latex; the page number should not exceed 8 pages in total.
   - General problem introduction
   - Methodology (including solution, validation, and evaluation methods) and pseudo codes
   - Experimental setup (includes data description and performance evaluation approaches among others.)
   - Evaluation results and discussions.
   - Show evidence in your report that you have ran your MPI implementation on the MSL cluster using more than one node.
   - In your report, proper citations and references must be given where necessary.
2. Source codes with `Makefile`, train/test data where applicable, run scripts (`run.sh`), and `readme` files.
3. Total marks: 100.

Problem 1 Federated Clustering using MPI [60]

The goal of this project is to implement a parallel machine learning algorithm within a simplified federated learning (FL) framework. The core idea of this framework is to enable model training across distributed datasets without the need to centralize the data. In many real-world scenarios, data is generated and stored across different locations (e.g., on user devices), yet we aim to train a machine learning model that benefits from all the data collectively. Federated learning addresses this challenge by allowing each data holder (or "worker") to train a local model on its own data and then communicate only the model parameters—not the raw data—to a central server. In our implementation, we assume a small number of worker machines and a central server. The workers could represent edge devices like smartphones, which often exist in large numbers, or more traditional computers, typically fewer in number. Each worker has access to its own local dataset. During training, each worker independently trains a local model and sends the resulting parameters to the server. The server aggregates these local models to update a global model, which is then sent back to all workers. This process repeats for a fixed number of iterations or until the global model converges.

Data heterogeneity is a significant challenge in federated learning, as the data available at each worker (or client) often originates from different distributions. This phenomenon is commonly referred to as non-i.i.d. (non-independent and identically distributed) data. Addressing data heterogeneity is crucial in FL to ensure that the global model learned at the central server generalizes well across all clients.

For the machine learning model, you may choose a simple algorithm such as k-means clustering, or opt for a more complex model like a convolutional neural network (CNN).

In terms of implementation, meet the following requirements.

(a) The FL framework must be able to handle an arbitrary number, m, of workers plus the server, where $m \geq 2$. Although it is more straightforward to implement such a framework using one of the Python machine learning libraries, you are required to implement the FL framework using C/C++ as our main focus is to use MPI.

(b) The FL framework should support an arbitrary number m of worker nodes plus a central server, where $m \geq 2$. While it may be more convenient to implement such a framework using Python machine learning libraries, you are required to implement it in C/C++, as the primary focus of this project is on utilizing MPI for distributed computing.

(c) Regarding the training and testing datasets, you are required to use at least one widely adopted benchmark dataset, such as CIFAR-10 or MNIST. To simulate real-world data heterogeneity, consider the data distribution approach proposed in [1], where each client is assigned a distinct transformation of the dataset. For example, one client may receive images rotated by 90 degrees, while another receives images ro-

tated by 180 degrees. This setup introduces realistic variations in local data distributions across clients. Follow the standard train/test split for your training and testing.

(d) For aggregating the local models at the central server, apply a simple averaging strategy – commonly referred to as federated averaging [2] – to compute the global model.

(e) Evaluate and compare the performance of your federated model against a baseline model trained in a centralized manner using the full dataset.

Problem 2 CUDA-accelerated Ray Tracing [40]

Ray tracing is a long-standing computer graphics technique used to generate realistic images by simulating the way rays of light interact with objects in a scene. It traces the path of light as it travels through pixels in an image plane, calculating color, shadows, reflections, and refractions to create highly detailed and accurate visual effects. Based on the physical phenomenon of light transportation, ray tracing is effectively simulating how light would interact with the world.

Unsurprisingly, such a technique is computationally intensive, as it requires simulating numerous light rays interacting with complex scenes, involving many recursive calculations for different lighting phenomenon across millions of pixels.

Fortunately, ray tracing is highly parallelisable, as each ray can be processed independently – allowing thousands or even millions of rays to be computed simultaneously across multiple threads, significantly accelerating rendering performance.

Following the simple ray tracing example in `Lec8` lecture slides, we would like to develop a more realistic raytracer using CUDA. As a starting point, a raytracer has been provided (`https://github.com/kaustubh0201/Ray-Tracer/blob/main/`). The code is written in C/C++ with OpenMP parallelization. Our goal is to adapt the code into a CUDA-accelerated raytracer. More specifically, we would like to achieve the following.

(a) Develop a basic CUDA-accelerated raytracer, called **cuRaytracer-base**, using global memory. [20]

(b) Explore the possibility of further optimization of cuRaytracer-base by utilizing CUDA memory hierarchy including shared memory, constant memory, texture memory, or any combination of the above-mentioned memory. [16]

(c) Any further optimization of raytracer in terms of enhancing its realistic effects. [4]

The code base with sample texture images[1] are provided in `craytracer` folder. (Type `make` to compile and run using `./raytracer <output image name>`. The image size and texture image sizes need to match for the code to run at the moment.) Further, `raytracing.md` in `craytracer/project_desc` provides brief explanations of some snippets in the base code [2].

[1] Sample texture images are from `https://www.pauldebevec.com/Probes/`

[2] Thanks to one of our graduates, Ireton Liu, for preparing this document

References

[1] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. An efficient framework for clustered federated learning. *Advances in neural information processing systems*, 33:19586–19597, 2020.

[2] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.