Report for Lab 5

Task 1 – MPI version

- We followed the given instructions to build and run the code.
- All the work is done in module `nnetwork.cxx`
- We distributed the weight vectors `W1`, `W2` and `W3` to allow the processes to work on a random portion of the samples on their own.
- Each call to `dot` function will carry out a matrix multiplication and the result is stored in output vectors, therefore, we split and distribute the generation of each output vector into different MPI processes based on the the rows. Each MPI will process will get `total_row_num/process_num` rows to deal with.
- Every process computes the change of the weights, i.e, the variables `dWx`.
- Root process is recollecting and displaying(every 100 iterations):

```
MPI_Gather(dW3.data(), ...);
MPI_Gather(dW2.data(), ...);
MPI_Gather(dW1.data(), ...);
 if (mpirank == 0) {
    W2 = W2 - lr * dW2;
    W3 = W3 - lr * dW3;
    W1 = W1 - lr * dW1;
    W2 = W2 - lr * dW2;
}
    ...
if ((mpirank == 0) && (i+1) % 100 == 0){
    ...
};
```

- We used the given commands to figure out the number of total processes as well as the rank of the executing process.
- We are mainly using `MPI` collectives in the implementation:

  For the data needed only by the root process, we used `MPI_Gather` to gather all the data partitions from all processes into root process.

  For the data needed by all the processes, we used `MPI_Allgather` to make all processes get the same copy of the data.

  For the data needed reduced at root process but needed by all the processes, we used `MPI_Bcast` to distribute the data to each processes.
- We used the following strategy.
  - Broadcast the randomly initialized weights to all processes.

```
// Random initialization of the weights in root process
vector<float> W1;
vector<float> W2;
vector<float> W3;

if (process_id == 0) {
    W1 = random_vector(784 * 128);
    W2 = random_vector(128 * 64);
    W3 = random_vector(64 * 10);
} else {
    W1.reserve(784 * 128);
    W2.reserve(128 * 64);
    W3.reserve(64 * 10);
}

// Broadcast initial weights
MPI_Bcast((void *)W1.data(), 784 * 128, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast((void *)W2.data(), 128 * 64, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast((void *)W3.data(), 64 * 10, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

– When training the model, we use `MPI_Allreduce` in order to sum over all weight changes and the original weights. After the reduction, the new weights values `W1`, `W2` and `W3` are present in all processes.

```
// Training loop
for (unsigned i = 0; i < 1000; ++i) {

    ... Calculate weight changes dW1, dW2, dW3 ...

    // Allreduce the weight deltas to all processes
    const auto dW1_aggr = process_id == 0 ? W1 - lr * dW1 : -lr * dW1;
    const auto dW2_aggr = process_id == 0 ? W2 - lr * dW2 : -lr * dW2;
    const auto dW3_aggr = process_id == 0 ? W3 - lr * dW3 : -lr * dW3;

    MPI_Allreduce((const void *)dW1_aggr.data(), (void *)W1.data(),
                  784 * 128, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce((const void *)dW2_aggr.data(), (void *)W2.data(),
                  128 * 64, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Allreduce((const void *)dW3_aggr.data(), (void *)W3.data(),
                  64 * 10, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);

    if ((process_id == 0) && (i + 1) % 100 == 0) {
        ... Logging ...
    }
}
```

- The table below shows the speedup for each number of processes:

	1 process	2 processes	4 processes
Total time	66.616 s	38.100 s	22.751 s
Speedup	1.0000	1.7485	2.9280

Table 1: Speedup for different number of processes

Task 2 – Parallel GEMM per process

We made an hybrid implementation of `MPI` and `OpenMP`. For `OpenMP`, we parallize the most outter loop of the `GEMM`. We set `OMP_NUM_THREADS`=2 and start 2 MPI processes by `mpirun -bind-to none -np 2 ./nnetwork_mpi`, so that we still fully utilize the available CPU resource without over utilization.

The result shows that the setup with 2 MPI processes + 2 threads per process (total: 21.616 s) slightly outperforms the setup with 4 MPI processes + 1 thread per process (total: 22.751 s). This is simply because whenever shared memory space is possible, it should be better than alternative distributed memory space due to smaller overhead.