**CSE 6220 High Performance Computing**
**Programming Assignemnt 2**
**Submitted Wednesday, October 23 2024**

**Name: (Group 13)** Srinath Suresh Kumar, Kaitlyn Tung, Dhruv Raj Bangad

# 1 Custom Scatter

## 1.1 Working

The `Custom_Scatter` function implements a custom scatter operation that distributes data from a root processor to all other processors in an MPI communicator. The function begins by initializing the size and rank of the communicator, and then calculates the depth $d$ of the communication, determined by $\lceil \log_2(p) \rceil$, where $p$ is the number of processors. It checks whether $p$ is a power of two and initializes variables for managing communication patterns.

A temporary buffer, `tempbuf`, is allocated to store the data that will be scattered. If the current rank is the root, the function copies the data from `sendbuf` into `tempbuf`. The scattering process proceeds in a loop, where at each step, the rank checks its communication mask to determine if it should send or receive data from its neighboring ranks, calculated using XOR operations with a flip variable. The number of elements to send is adjusted based on the rank and whether the current process is involved in an "arbitrary" scatter due to a non-power-of-two number of processors.

As the loop iterates, data is sent using `MPI_Send` and received using `MPI_Recv` based on the communication pattern. After completing the scatter operation, the received data is copied from `tempbuf` to `recvbuf`, ensuring that each process has the appropriate subset of data. Finally, the temporary buffer is freed, completing the custom scatter operation.

## 1.2 Runtime Analysis

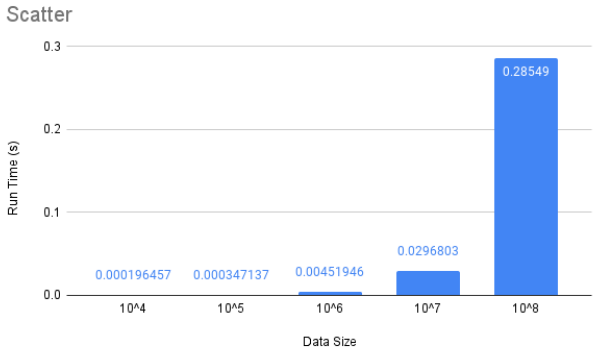
Given 'p' processors and each processor needs to receive m amounts of data. The root processor contains 'mp' amount of data to begin with. We use a permutation routing similar to Gather (but the amount of data transferred in each round reduces with iteration. $\log p$ rounds of communication happen in total.

$$T_{comm}(mp, n) = (\tau + \mu\frac{mp}{2}) + (\tau + \mu\frac{mp}{4}) + ... + (\tau + \mu m)$$
$$T_{comm}(mp, n) = \sum_{i=1}^{\log p}(\tau + \frac{\mu mp}{2^i})$$
$$T_{comm}(mp, n) = \theta(\tau \log p + \mu mp)$$

The custom scatter method does not perform any other computation than finding the processor rank to communicate which take constant time O(1) time in every round of communication there are total $\log p$ rounds. So it take $O(\log p)$ time. Since we use a temporary buffer to copy data from sendbuf of size mp and then again from tempbuf to recvbuf of size m. The copy operations take linear time complexity of O(mp) and O(m) respectively. Since $mp > m$, we take asymptotic time as O(mp).

$$T_{comp}(mp, n) = O(\log p) + O(mp)$$
$$T_{comp}(mp, n) = O(mp)$$

### 1.3 Results for $p = 16$



# 2 Custom AllGather

## 2.1 Working

The `Custom_Allgather` function implements an all-gather operation using a logarithmic number of steps, optimized for hypercube-like topologies. It first retrieves the rank and size of the communicator, computes the number of steps as $\lceil \log_2(p) \rceil$, and initializes the communication buffers.

Each process begins by placing its local data into the appropriate position in the `recvbuf` using `memcpy`. The core of the function is the loop that performs $\log_2(p)$ steps, where $p$ is the number of processes. In each step, the process communicates with a neighbor determined by XOR-ing its rank with a flip (i.e., neighbor = rank $\oplus$ flip), and exchanges a progressively larger chunk of data. The amount of data exchanged doubles in each step, allowing each process to gather data from more peers. The `MPI_Sendrecv` call facilitates simultaneous sending and receiving of data between the process and its neighbor. After each step, the flip and step size are updated for the next iteration, until all processes have gathered data from every other process.

This logarithmic communication scheme reduces the time complexity compared to a naive approach, making it more efficient for large-scale systems.

## 2.2 Runtime Analysis

`Communication Time` In the AllGather primitive we send 'm' amounts of data in first round, '2m' amounts of data in the second round, and so one and finally '$\frac{mp}{2}$' amount of data in the final round. There are $\log p$ number of rounds in total.

$$
\begin{aligned}
T_{comm}(mp, n) &= (\tau + \mu m) + (\tau + \mu 2m) + ... + (\tau + \mu \frac{mp}{2}) \\
T_{comm}(mp, n) &= \sum_{i=0}^{\log p - 1} (\tau + \mu m 2^i) \\
T_{comm}(mp, n) &= \theta(\tau \log p + \mu m(1 + 2 + .. + \frac{p}{2})) \\
T_{comm}(mp, n) &= \theta(\tau \log p + \mu mp)
\end{aligned}
$$

In case of number of processors not being an power of 2 we need to distribute the data to processor which did not participate in certain rounds of hypercubic permutation. For this we use the k-to-All approach where k processors have complete data and p - k do not have complete data. The minimum value of k will be 2 and maximum value of k will be p - 1. Here k processors begin by sending to maximum of k other processors in the first round. After first round 2k processor have complete data and now these 2k processors will send remaining data to maximum of 2k processors in parallel. In this method it will take a maximum of $log(\frac{p}{k})$ rounds for all processors to have data. In each round maximum of O(mp) amount of data is shared. Each round communication is $O(\tau + \mu mp)$. Communication time for sending data from k processors to p-k processors in parallel $O(\tau \log \frac{p}{k} + \mu mp \log \frac{p}{k})$. So total communication is

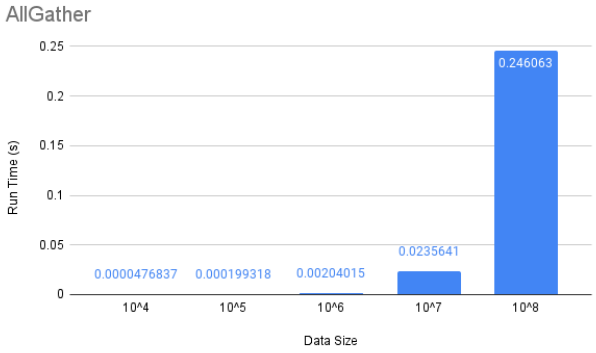
$$
T_{comm}(mp, n) = \theta(\tau \log p + \mu mp) + O(\tau \log \frac{p}{k} + \mu mp \log \frac{p}{k})
$$

`Computation Time` The custom AllGather method does not perform any other computation than finding the processor rank to communicate which take constant time O(1) time in every

round of communication there are total $\log p$ rounds. So it take $O(\log p)$ time. The computation time is $O(m)$ for copying m data from sendbuf to recvbuf in the beginning only one. Calculating the processor rank, transfer size to send and receive data is constant time from every round. In the subroutine to send data from from k processors with data to p-k processors without data we incur $O(p)$ per round computation cost to calculate the list of processors with and without data after each round. T his takes total of $plg\frac{p}{k}$ computation. Total computation

$$T_{comp}(mp, n) = O(m) + O(\log p) + O(p \log \frac{p}{k})$$

## 2.3 Results for $p = 16$



# 3 Custom AllReduce

## 3.1 Working

The `Custom_Allreduce` function implements a custom all-reduce operation for summing values across all processors in an MPI communicator. First, the function initializes variables for the communicator's size and rank, and calculates the depth of the communication, which is determined by the number of steps in a hypercube-like communication pattern (i.e., $\log_2(p)$, where $p$ is the number of processors). If the number of processors is not a power of two, the function rounds up to the next power of two for communication depth.

The process begins by copying the data from the `sendbuf` into the `recvbuf`. Then, the function enters a loop, where at each step, the current process communicates with a "neighbor" process determined by XOR-ing the rank with a bitmask. The ranks exchange their data using `MPI_Send` and `MPI_Recv`, and the received values are accumulated (summed) into the `recvbuf`.

After completing the log-based communication, the function employs the `ktoAll` function, which acts as a final distribution phase. This step involves partitioning the ranks into two groups (sends and recvs), and redistributing the reduced data across all ranks using a custom broadcasting procedure. The communication in `ktoAll` ensures that each process receives the final result, completing the all-reduce operation.

## 3.2 Runtime Analysis

`Communication time:`Given 'p' processors with 'm' amounts of data each we use an hypercubic permutation to send 'm' amounts of data in each round with $\log p$ such rounds. Communication time per round is $\tau + \mu m$.

$$T_{comm}(mp, n) = (\tau + \mu m) * \log p$$
$$T_{comm}(mp, n) = \theta(\tau \log p + \mu m \log p)$$

In case of number of processors not being an power of 2 we need to distribute the data to processor which did not participate in certain rounds of hypercubic permutation. For this we use the k-to-All approach where k processors have complete data and p - k do not have complete data. The minimum value of k will be 2 and maximum value of k will be p - 1. Here k processors begin by sending to maximum of k other processors in the first round. After first round 2k processor have complete data and now these 2k processors will send remaining data to maximum of 2k processors in parallel. In this method it will take a maximum of $log(\frac{p}{k})$ rounds for all processors to have data. In each round maximum of 'm' amount of data is shared. Each round communication is $O(\tau + \mu m)$. Communication time for sending data from k processors to p-k processors in parallel $O(\tau \log \frac{p}{k} + \mu m \log \frac{p}{k})$. So total communication is

$$T_{comm}(mp, n) = \theta(\tau \log p + \mu m \log p) + O(\tau \log \frac{p}{k} + \mu m \log \frac{p}{k})$$

Since $\log \frac{p}{k}$ is lesser term than $\log p$

$$T_{comm}(mp, n) = O(\tau \log p + \mu m \log p)$$

`Computation time:`The custom AllReduce method does not perform any other computation than finding the processor rank to communicate which take constant time $O(1)$ time in every round of communication there are total $\log p$ rounds. So it take $O(\log p)$ time. In AllReduce computing

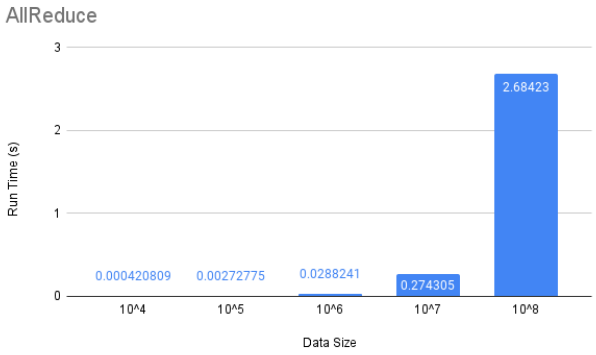
the send and receiving processor is constant time but the operation, here sum, take linear time over the data size which is 'm' per round. So there are $\log p$ rounds and total computation is $O(mlgp)$. (We copy the sendbuf into recvbuf one time initially, which is also $O(m)$ and can be ignored as $O(mlgp)$ is greater than $O(m)$ time). In the subroutine to send data from from k processors with data to p-k processors without data we incur $O(p)$ per round computation cost to calculate the list of processors with and without data after each round. T his takes total of $plg\frac{p}{k}$ computation. Total computation

$$T_{comp}(mp, n) = O(\log p) + O(mlgp) + O(plg\frac{p}{k})$$

Assuming data is larger than processor count, $m > p$, and known that $\log p > \log \frac{p}{k}$, mlgp is the larger term.

$$T_{comp}(mp, n) = O(mlgp)$$

### 3.3 Results for $p = 16$



# 4 Custom All-to-all Arbitrary

## 4.1 Working

The `Custom_Alltoall_Arbitrary` function implements an all-to-all communication where each process exchanges data with every other process in a fully connected topology. The function first retrieves the rank and size of the communicator, and then iterates over all processes in the communicator. First the particular portion of data of size 'm' belonging to its own rank is copied from sendbuf to recvbuf. After this is over, every processor loops for p-1 times to communicate with p-1 processor. Let the iteration variable be 'i'. In every round processor of rank 'r' sends to processor ((r + i) mod size) the portion to data that should belong to it. It also receives data from the processor ((r - i + size) mod size). So every processor will send data of size 'm' from sendbuf with an offset (receiver_rank * m). Similarly while receiving 'm' amounts of data it writes to recvbuf with an offset of (sender_rank * m).

## 4.2 Runtime Analysis

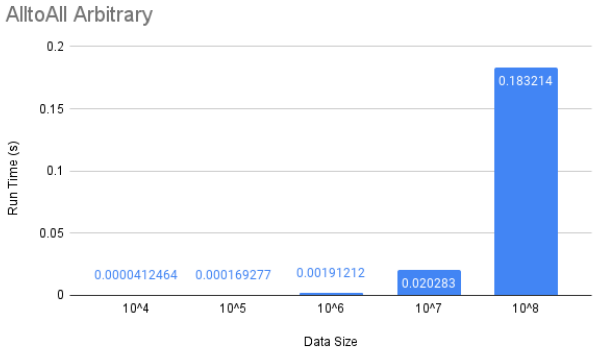
`Communication Time:` When looking at the communication time, each processor will incur $\tau$ latency as it is being looped through and since each processor needs $\mu$ bandwidth, our communication time will be:

$$T_{comm}(mp, n) = O(\tau \cdot p + \mu \cdot m \cdot p)$$

`Computation Time:` The computation cost of our function is from copying the data from `sendbuf` to `recvbuf` for the given portion of data in the beginning, which comes out to be O(m). Each processor loops for O(p) time and each loop takes constant time to calculate send and recv rank.

$$T_{comp}(mp, n) = O(m) + O(p)$$

## 4.3 Results for $p = 16$



# 5 Custom All-to-all Hypercube

## 5.1 Working

The `Custom_Alltoall_Hypercube` function performs an all-to-all communication in a hypercube topology, where each process exchanges data with others in a systematic way. The number of processors, $p$, determines the depth of the hypercube, and communication occurs over $\log_2(p)$ steps. In each step, every process exchanges data with a neighboring process determined by flipping specific bits of its rank (via XOR with $1 << j$). The data to be exchanged is partitioned, and processes send and receive corresponding portions of the data. After each communication, the data is merged back into the receiving buffer using `memcpy`. The function efficiently manages data transfers and distribution, ensuring that after all steps, each process has a complete set of data from all other processes. The communication and copying steps depend on the number of processors and the size of the data being exchanged, following the hypercube's logarithmic structure.

## 5.2 Runtime Analysis

`Communication Time:` In our code, at each step, the amount of data that is being exchanged by each neighbor is decreasing by half. The latency for initiating these messages will be $O(\log p\tau)$ since there are $\log p$ communication steps. The bandwidth will be time it takes to send/receive a unit of data, adding a factor of $O(\mu \cdot m \cdot p \cdot \log p)$ to our communication time. Hence we get,

$$T_{comm}(mp, n) = O(\tau \cdot \log p + \mu \cdot m \cdot p \cdot \log p)$$

`Computation Time:` Our computation time is solely due to the overhead of `memcpy`. The `memcpy` takes $O(m)$ for copying data of size $m$. This operation is repeated for O(p) processors, for $O(\log p)$ rounds of communication. Making our computation time:

$$T_{comp}(mp, n) = O(m \cdot p \cdot \log p)$$

## 5.3 Results for $p = 16$