Coding Assignment: Distributed Parallel Reductions

1. Submission Guidelines

- Implement solutions in C-like language and share .c or .pdf file (**mandatory**, handwritten submissions will be ignored).
- Make sure to comment the code adequately to make it easy to read. Included details of the algorithm selected and any trade-offs made.
- Use clean code organization and intuitive names for variables and functions.
- Solution Template:
  - **Solution Explanation**: <short paragraph>, explain algorithm in steps 1, 2, 3, etc.
  - **Corner case handling:** <short note>
  - **Complexity:** <short note>
  - **Code:** <code with comments>

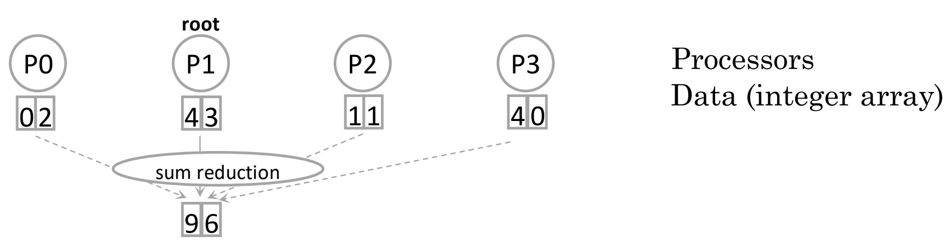
2. Programming Environment

This assignment assumes hypothetical distributed-memory systems composed of **P** processors interconnected with a network fabric. Each processor has private memory and communication between processors is handled through basic send-receive message passing primitives. For the purpose of this assignment, we provide the following primitives:

int get_my_id()	returns the caller's processor ID
int get_proc_count()	returns the number of processors in the system
void send(int trg, void* buf, int sz)	send to target processor ID **trg** the data in buffer **buf** of size **sz**. This call is synchronous, i.e., the caller is blocked until data is sent out.
void recv(int src, void* buf, int sz)	receive data from source processor ID **src** and store it in buffer **buf** of size **sz**. This call is synchronous, i.e., the caller is blocked until data is received.

3. Basic Reduction

A reduction performs an operation **X** (e.g., sum) on data distributed among **P** processors. The result is stored at one of the processors called the "root". The example below shows how a sum reduction is performed on an array of integers, with processor 1 being the root.



Let us assume that we only do sum reductions. The signature of a reduction routine is as follows:

void reduce(int root, void* buf, int sz)	each processor contributes its local buffer **buf** of size **sz.** Sum the data from all processor buffers and store the result at processor **root**

For reference, we provide a basic implementation of this routine below assuming a C-like programming style.

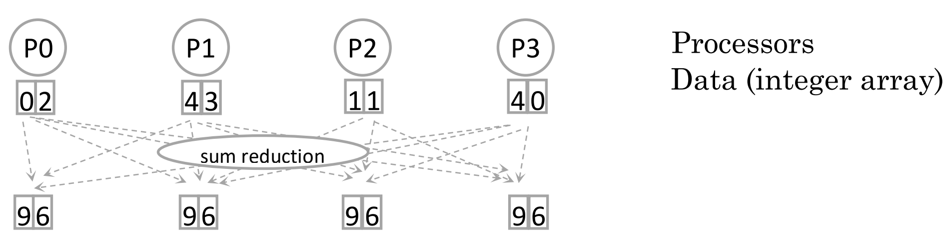
```
void reduce(int root, void* buf, int sz) {
   int self = get_my_id();
   int proc_count = get_proc_count();
   int *recv_buf = (int*)malloc(sz * sizeof(int));
   if (self == root) {
      for (int p = 0; p < proc_count; ++p) {
         if (p != self) {
            recv(p, recv_buf, sz);
            for (int i = 0; i < sz; ++i) {
               buf[i] += recv_buf[i];
            }
         }
      }
   } else {
      send(root, buf, sz);
   }
}
```

Question #1

1. What is the complexity in terms of steps and number of messages exchanged in the above **reduce** implementation? (use big-O notation) **(0.5 pts)**
2. Provide an implementation that executes in O(Log(P)) steps and exchanges O(P) messages. **(1.5 pts)**

4. Allreduce Operation

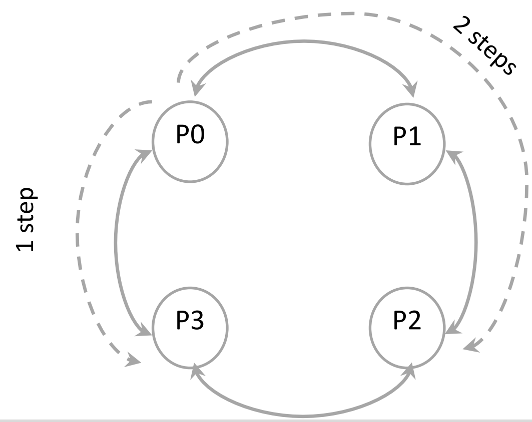
An **allreduce** operation is similar to a **reduce** operation but without a root processor; instead, all processors involved in the reduction store the result locally. Below is the same example as above where an **allreduce** operation is performed instead of **reduce.**



Question #2
Provide an implementation of the **allreduce** that executes in O(Log(P)) steps and exchanges O(P) messages. **(1 pt)**

5. Allreduce on a Ring Topology

Suppose our multiprocessor system features a ring topology; that is, each processor is connected to exactly two neighbor processors with a bidirectional link in a ring pattern. We assume that the link bandwidth between two neighbor processors is unlimited. Communication between neighbor processors is performed with a pair of point-to-point send-receive operations and takes one **sequential step**. Consequently, communication between non-neighbor processors takes multiple sequential steps, and communication between independent pairs of processors occurs in parallel and thus only takes one sequential step. Assume that process ids start from **0** and increase monotonically to **P-1**, where **P** is the total number of processors, in clockwise direction of the ring.



Question #3
Provide an implementation of the **allreduce** routine on this ring topology that minimizes the number of sequential steps. **(1.5 pts)**

6. Softmax on a Ring Topology

The Softmax function takes a vector of real numbers and normalizes it into a probability distribution. Specifically, for an input vector A of size N, the Softmax routine outputs vector B of size N where:

$$B_i = \frac{e^{A_i}}{\sum_{j=0}^{N-1} e^{A_j}}, for\ i = 0, \ldots, N-1$$

The programming environment provides the following routine for computing the exponential of a floating-point number.

float exp(float x)	return the exponential of **x**

We want to implement the Softmax function with the following signature:

void softmax(float *A, int sz, float *B)	apply the Softmax function on input vector **A** and store the result in output vector **B. A** and **B** are distributed among all processes and each processor allocates and contributes its local vectors of size **sz**.

Question #4
Provide an implementation of the **softmax** routine on the previous ring topology. **(0.5 pts)**