Communication Primitives

1 Broadcast

1.1 Defintion

A piece of data from one processor is sent to all other processors.

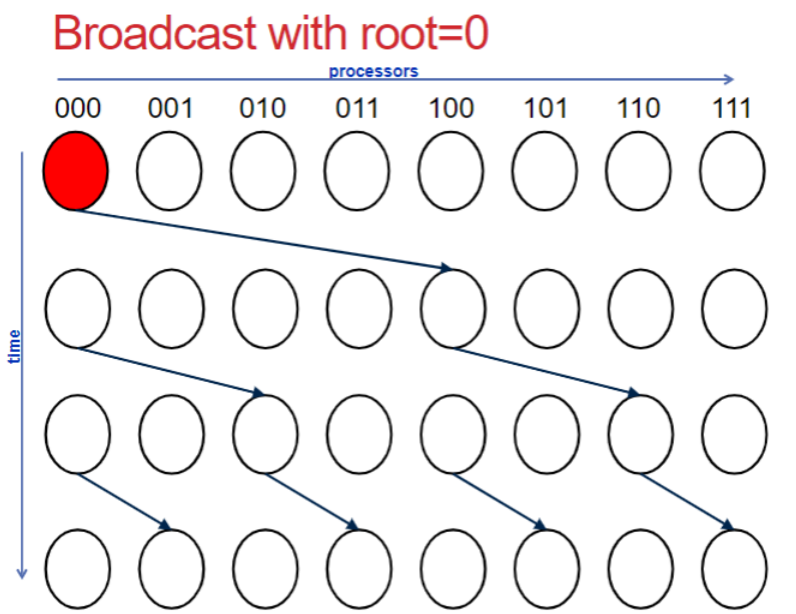

Figure 1: Broadcast

1.2 Arbitrary Cases

If p is not 2^d, we can find $p' = 2^d$ such that:

$$\frac{p'}{2} < p < p' \tag{1}$$

And then run the code as if we have p' processors and ignore the communications to/from non-existing processors.

1.3 Runtime

For message size m, we have $\log p$ steps to finally broadcast to all processors and at each level, **in each processor the message size is still** m. Therefore the runtime could be expressed as:

$$T_{Comm} = \theta(\tau \log p + \mu m \log p) \tag{2}$$

2 Reduce

2.1 Definition

Reduce operation aggregates data from all processors and combines them into a single processor.

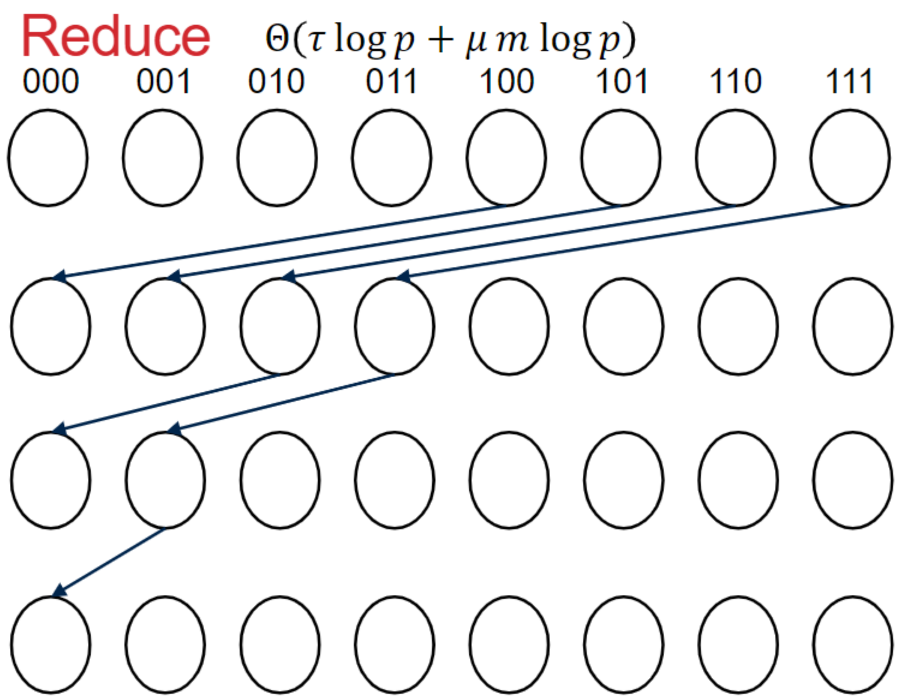

Figure 2: Reduce

2.2 Runtime

Notice in this operation, data is **aggregated**, so the combine process **will not increase the size**. Therefore, the runtime could be expressed as:

$$T_{Comm} = \theta(\tau \log p + \mu m \log p) \tag{3}$$

Reduce also has computation time. Although the addition operation will only take $O(1)$, but we have $\log p$ levels, so the final computation time is:

$$T_{comp} = \theta(\log p) \tag{4}$$

3 AllReduce

3.1 Defintion

AllReduce operation aggregates data from all processors and broadcast to all processors.

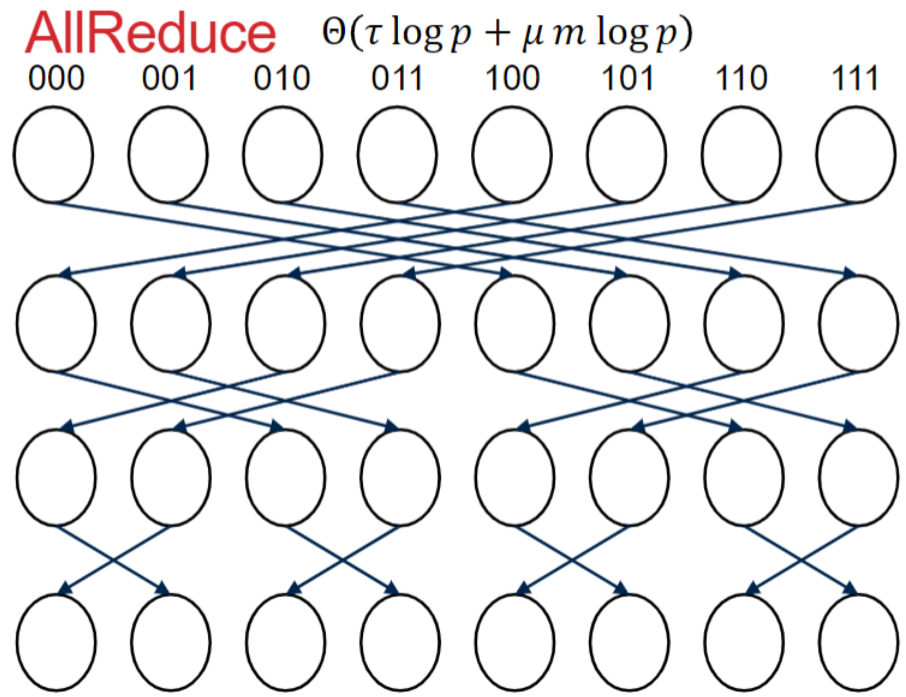

Figure 3: AllReduce

3.2 Runtime

Similar with Reduce, the runtime could be expressed as:

$$T_{Comm} = \theta(\tau \log p + \mu m \log p) \tag{5}$$

4 Scan

4.1 Definition

Scan, also known as the prefix sum operation, is a fundamental communication primitive in parallel computing, serving a unique role in both shared and distributed memory systems. The scan operation takes a sequence of data distributed across processes and computes partial aggregates of these data elements, **distributing the intermediate results back to each process**. Unlike reduce, which aggregates all data into a single result, scan provides each process with an intermediate aggregate that includes its own and all preceding data points in the sequence.

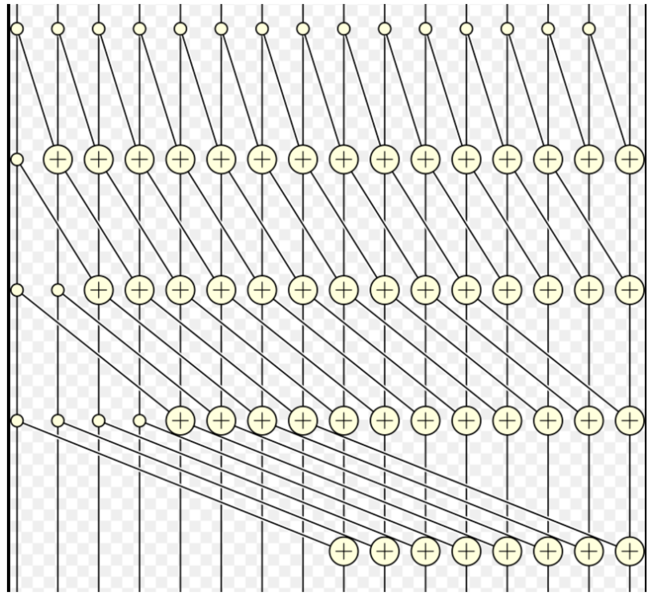

Figure 4: Scan

4.2 Runtime

In this case, we only care about the communication time. Same as Reduce, the sum operation will not increase the size of the data, so:

$$T_{Comm} = \theta(\tau \log p + \mu m \log p) \tag{6}$$

5 Gather

5.1 Definition

Collect data from all processors and **assemble** the data into a single processor.

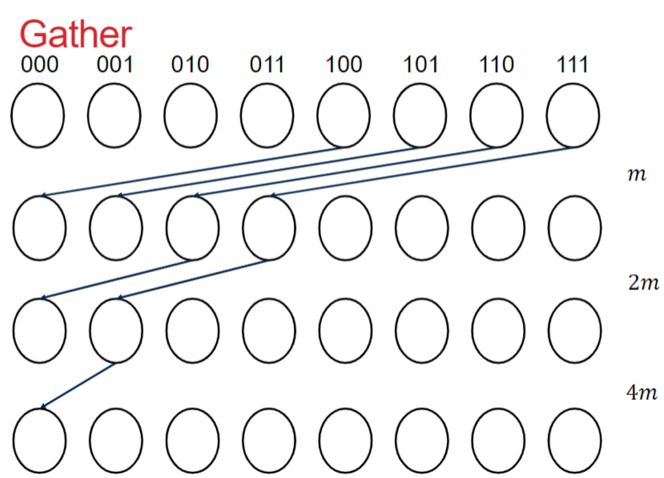

Figure 5: Gather

5.2 Runtime

Notice that here the data size in each processor increases at each level, so the previous runtime expression could not be used.

The data sending communication time is:

$$\theta(\sum_{i=0}^{\log(p)-1}(\tau + \mu m \cdot 2^i)) = \theta(\mu m \cdot (1 + 2 + ... + \frac{p}{2})) \approx \theta(\mu m p) \tag{7}$$

Therefore the total communication time is:

$$T_{Comm} = \theta(\tau \log p + \mu m p) \tag{8}$$

6 AllGather

6.1 Definition

Collect data from all processors, **assemble** the data and broadcast into all processors.

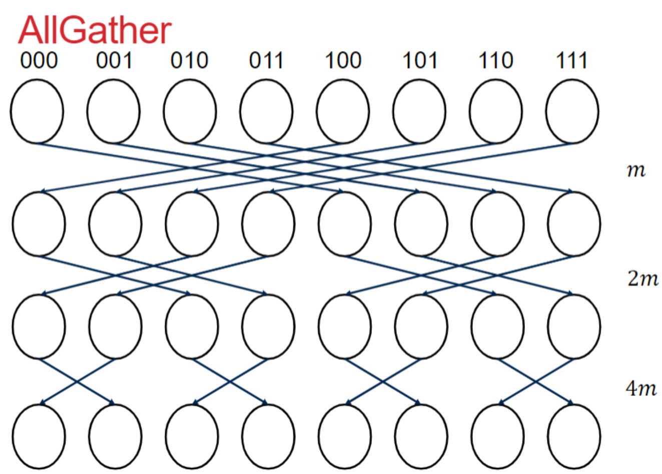

Figure 6: AllGather

6.2 Runtime

Same as Gather:

$$T_{Comm} = \theta(\tau \log p + \mu m p) \tag{9}$$

7 Scatter

7.1 Definition

Scatter is a key communication primitive in parallel computing, which performs the opposite operation of gather. In scatter, a single data source from one process, often the root process, is divided into segments and distributed among all processes in a communicator or group.

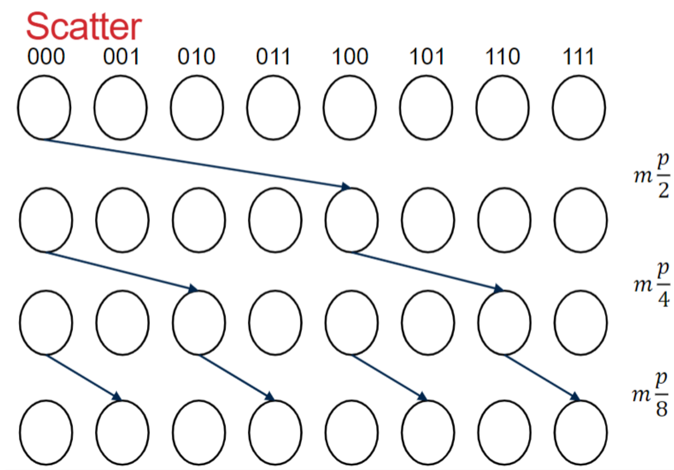

Figure 7: Scatter

7.2 Runtime

Assume the root processor has data size as pm, assume $p = 2^d$. Then at each level, the data size in each processor is divided by 2. The data sending runtime will be:

$$\theta(\mu mp \cdot (\frac{1}{2} + \frac{1}{4} + ... + \frac{1}{p})) \approx \theta(\mu mp) \tag{10}$$

So the total communication time is:

$$T_{Comm} = \theta(\tau \log p + \mu mp) \tag{11}$$

8 All to All

8.1 Definition

All to All communication in parallel computing refers to a communication pattern where every process (or node) sends data to, and receives data from, **all other processes** in the computing system.

8.2 Arbitrary Permutations

The arbitrary permutation could be implemented in this way:

Algorithm (for P_i)
for j=0 to (p-1) do
  P_i sends $m_{i,j}$ to P_j

Figure 8: Arbitrary Permutation Implementation 1

But this is not efficient algorithm because every time same processor is working, which is actually a serial communication. The runtime for this implementation is $O(\tau p^2 + \mu m p^2)$. Another implementation is:

Algorithm (for P_i)
for j=1 to (p-1) do
  P_i sends $m_{i,(i+j) \bmod p}$ to $P_{(i+j) \bmod p}$

Figure 9: Arbitrary Permutation Implementation 2

Now all the processors could work at the same time. The runtime for this implementation is $O(\tau p + \mu m p)$.

8.3 Hypercubic Permutations

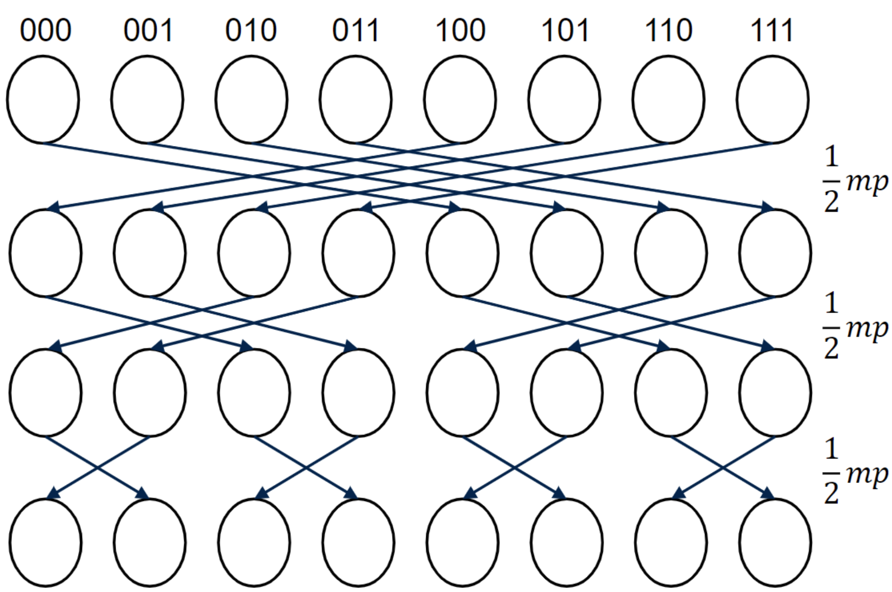

Figure 10: Hypercubic Permutation Implementation

At each step, the processors at one side send the messages for the other side to the other side. More details:

- P_0: m_{00} m_{01} m_{02} m_{03} m_{04} m_{05} m_{06} m_{07}
- P_0: m_{00} m_{40} m_{01} m_{41} m_{02} m_{42} m_{03} m_{43}
- P_0: m_{00} m_{20} m_{40} m_{60} m_{01} m_{21} m_{41} m_{61}
- P_0: m_{00} m_{10} m_{20} m_{30} m_{40} m_{50} m_{60} m_{70}

Figure 11: Hypercubic Permutation Step by Step

For this permutation, the runtime will be $O(\tau \log p + \mu m p \log p)$.

9 Many to Many

9.1 Definition

In a "Many to Many" communication scenario, a selected subset of processors (not necessarily all) sends data to and receives data from another selected subset of processors.

Some notations for this operation.

1. m_{ij}: message from P_i to P_j
2. $|m_{ij}|$: size of the message
3. $max(i) \sum_j |m_{ij}| \leq S_i$: this represents the sending message limit
4. $max(j) \sum_i |m_{ij}| \leq R_j$: this represents the receiving message limit

$$\begin{array}{rc l}
\begin{array}{r} 0: \\ 1: \\ 2: \\ 3: \end{array} &
\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} &
\begin{array}{l} S_0 \\ S_1 \\ S_2 \\ S_3 \end{array} \\
& \begin{array}{cccc} R_0 & R_1 & R_2 & R_3 \end{array} &
\end{array}$$

Figure 12: Many to Many Notations

Assume we have 4 processors, and in processor P_2 we have messages want to send to P_0, P_1 and P_3, with different size:

Figure 13: Initial Condition

Now, we divide each message into 4 pieces for each processor:

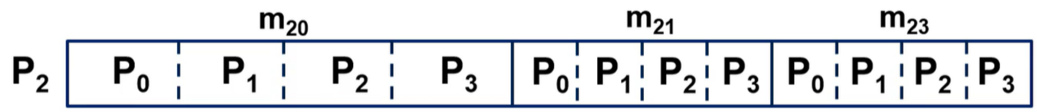

Figure 14: Division

Now we can use All to All operation to make sure the message size in each processor are the same, with max message size $\leq \frac{S}{p}$:

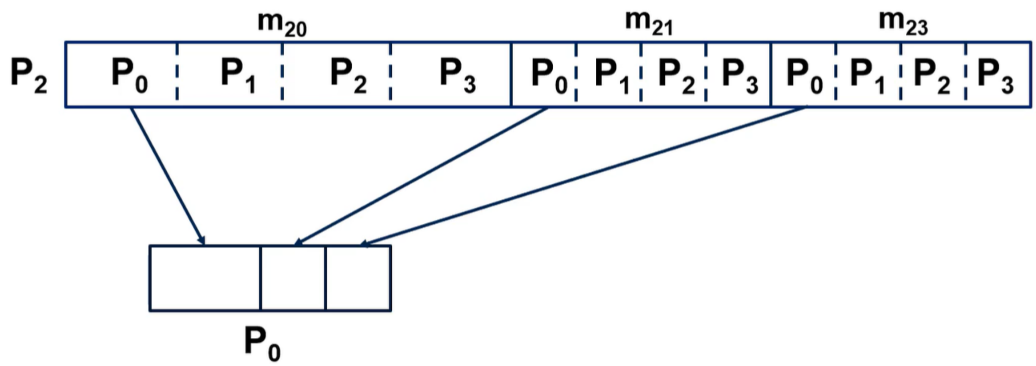

Figure 15: Combination

Then, **route the message fragments to actual destinations and assemble, using All to All**. The max message size now is $\leq \frac{R}{p}$.

Recall the runtime for All to All is $O(\tau p + \mu mp)$. Notice that we need to set a boundary label to let the processor know the size of the box, this will take $O(p)$. So for stage 1, the runtime is $O(\tau p + \mu(\frac{S}{p} + p)p)$. For stage 2, the runtime is $O(\tau p + \mu(\frac{R}{p} + p)p)$. The total runtime is $O(\tau p + \mu(R + S + p^2))$.