Homework 3

CS 406

Parallel Programming

Spring 2020-2021

Instructor:

Kamer Kaya

Faculty of Engineering and Natural Sciences

Sabanci University

1) Problem Statement

The goal of this project is to implement all to all broadcast among processors with various message size and process numbers using the MPI framework. At the end of the implementation, each processor, should have the same all messages with the same message ordering that was broadcasted among the processors.

2) Implementation Details

Among different all to all broadcast methods, we choose hypercube implementation since it has the best asymptotic time complexity with (ts * logp + tw*m*(p-1)), where ts represents the cost of preparing the data to be sent, p represents the process size, m represents message size and tw represents the cost of transferring one word. The pseudocode implementation and the explanation of the hypercube is as follows:

All to All Broadcast with Hypercube:

```
dimension = log(p)
current_msg_size = msg_size
for  i  → msg_size
    rb[(rank * msg_size)] = rank
for  i  → dimension
    partner = rank ^ $2^i$
    for  i  → dimension
        result[i] = rb[(sb[0] * msg_size +i]
    if rank < partner do
        Send(result)
        Receive(msg)
    else do
        Receive(msg)
        Send(result)
    for  i  → current_msg_size
        find the minimum rank in the message and store it in sb[0]
        rb[(msg[0] * msg_size) + i]  =msg[i]
    current_msg_size *= 2
```

We first calculated the dimension of the hypercube by taking the logarithm of the processor size. We then initialized the receive buffer (rb) with the rank of the current processor with the corresponding index which is calculated by rank * msg_size. Afterwards, we started our main iteration that runs dimension many times. We calculated the rank of the neighbor by taking the xor of the current rank id with the power of 2^i in which i is the iteration index. Since MPI_Send and MPI_Receive are blocking operations, during the message sending process, we first let the processors send the message which has the lower rank compared to its neighbor rank. Likewise, the processes that has the higher rank will first receive the message, then it will send its own message to its particular neighbor. Hence, we avoided deadlock. After messages are being received by the neighbors, we updated the receive buffers of the neighbors and calculated the minimum rank of the send buffer which is further be stored in the first element of the send buffer. We then use that minimum rank value to calculate the starting index of the message that is contained the receive buffer and send that message to the corresponding neighbor. By doing that, we guarantee that every processor's receive buffer will have the same message ordering at the end of the communication process.

3) Run Times

Thread counts are denoted in the leftmost column. The rest of the columns represent the message sizes. Since, the execution times may differ in different runs, we take the average of 5 runs while reporting each timing. All of the units are seconds.

Thread Count	4	100	500	1000	5000	10000	50000
2	**0.000018**	**0.000020**	**0.000023**	**0.000028**	**0.000074**	**0.000113**	**0.000367**
4	**0.000022**	**0.000030**	**0.000053**	**0.000064**	**0.000172**	**0.000274**	**0.001145**
8	**0.000047**	**0.000055**	**0.000073**	**0.000127**	**0.000363**	**0.000624**	**0.003086**
16	**0.000100**	**0.000106**	**0.000149**	**0.000247**	**0.000803**	**0.001458**	**0.011300**
32	**0.000279**	**0.000304**	**0.000429**	**0.000655**	**0.002175**	**0.004573**	**0.029747**
64	**0.0266120**	**0.032044**	**0.035197**	**0.035408**	**0.077646**	**0.051667**	**0.17745**
128	**0.112404**	**0.161379**	**0.190827**	**0.193237**	**0.3146444**	**0.312893**	**0.664209**