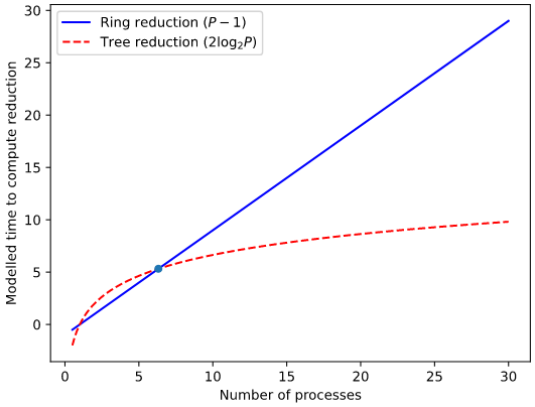
PHYS52015 Coursework:
Stencils and Collectives : MPI

1 Introduction

In this report, implementations of a different algorithm for a reduction in MPI collectives with Ring, Tree and MPI_Allreduce algorithm and comparing with various scaling and analyse the performance of these implementations are also studied and discussed.

2 Implmentations

In this coursework, there are three implementations which are Ring_all reduce, Tree_all reduce, and MPI_all reduce. From the conclusion, when the size is 6 or more, the results are derived in the order of mpi, tree, and ring, but when the size is 6 less, the order of ring and tree is changed. With following time formula based on processes prove the relation between ring and tree under the size of 6.



(Fig.2.1 modeled time-number of processes)

$T\text{ring}(B)=(P-1)(\alpha+\beta B)$, $T\text{tree}(B)=2log_2P(\alpha+\beta B)$

In the below, it will explain about algorithm based on **single program multiple data(SPMD)**

2.1 Ring all reduce

In this structure, send a message with rotate direction following algorithm 2.1.1 represent the source and destination of the ring structure.

Algorithm 2.1.1 Ring Allreduce

```
1: source = (rank + size - 1) % size; //To run around the loop using modular
2: dest = (rank + 1) % size; //For next rank
```

Unlikely python, in C modular acts like truncated division rather than remainder division, so it rotates in the circle and sends the message right next rank.

And in the buffer, message data should be duplicated in recvbuf which following algorithm 2.1.2 represent

Algorithm 2.1.2 Ring Allreduce

```
1: for (c = 0; c < count; c++) {
    recvbuf[c] = sendbuf[c]; //Duplicate buffur
```

With this process, the buffer could duplicate, and it will use for operand afterwards.

The following algorithm 2.1.3 buffer could receive and send the message program simultaneously.

Algorithm 2.1.3 Ring Allreduce

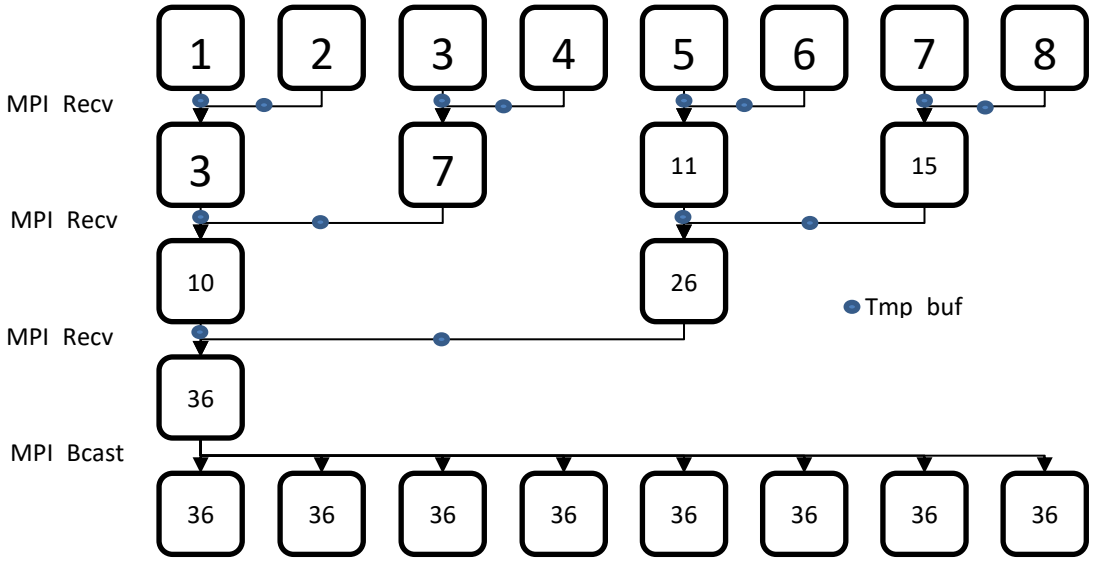
```
1: ierr = MPI_Sendrecv_replace(recvbuf, count, MPI_INT, dest, 0, source, 0, comm, MPI_STATUS_IGNORE);CHKERR(ierr);
//With MPI_Send,MPI_Recv could make memory error in big message data so with MPI_Sendrecv, exchange at the same time
```

Send and receive a message in the circle of rank. Operator should execute after communicating to next rank. If the number of processes was increased, the calculation time would increase linearly, as shown in Fig.2.1

2.2 Tree all reduce

This structure sends a message to the previous rank from the next rank. The message will save a temporary buffer and

prevent deadlock. following Fig2.2.1 will represent the structure.



(Fig2.2.1 Tree structure)

Firstly, **send buffer** will **copy** to receive buffer and temporary buffer. which could use in MPI_Recv and operator. Then, refer to the following algorithm 2.2.1.

Algorithm 2.2.1 Tree Allreduce

```
1: // copy elements from sendbuf to recvbuf
2: for (c = 0; c < count; c++) {
3:    tmpbuf[c] = sendbuf[c];
4:    recvbuf[c] = sendbuf[c];}
```

According to algorithm 2.2.2, The tree structure would iterate until the size, and if the rank is even number, MPI_recv will perform the **reduction**. After an odd number of ranks would get the send buffer.

Algorithm 2.2.2 Tree Allreduce

```
1: // Process the Reduce operation from all processes to the process 0 (the root)
2:  for (i = 2; i <= size; i *= 2) {
3:  if (rank % i == 0) {
4:    //reciever = rank;
5:     sender = rank + ( i / 2); // devide by 2
6:     terr = MPI_Recv(recvbuf, count, MPI_INT, sender, 2022, comm, MPI_STATUS_IGNORE);CHKERR(terr);
```

--- Operator ---

```
1:   else {
2:     if ( rank % (i / 2) == 0) {
3:      receiver = rank - (i / 2);
4:      //sender = rank;
5:      terr = MPI_Send(sendbuf, count, MPI_INT, receiver, 2022, comm);CHKERR(terr);
```

Middle of the algorithm 2.2.2, tree allreduce will **operate** MPI_SUM, MPI_PROD, MPI_MIN and MPI_MAX operand. With algorithm 2.2.3 it will explain sum, production.

Algorithm 2.2.3 Tree Allreduce

```
1: if (op == MPI_SUM) { // In the production using MPI_PROD
2:      for (c = 0; c < count; c++) {
3:        recvbuf[c] += tmpbuf[c];} // In the production using *=
4:      for (c = 0; c < count; c++) {
5:        tmpbuf[c] = recvbuf[c];
```

Using **conditional handle** (select MPI_SUM, MPI_PROD), the operator. From temporary buffer operate to receive buffer and receive buffer store to temporary buffer again. Moreover, operators (MPI_MAX, MPI_MIN) will explain in algorithm 2.2.4.

Algorithm 2.2.4 Tree Allreduce

```
1: #define min(a,b) (((a) <= (b)) ? (a) : (b)) // Define the min, max definition
2: #define max(a,b) (((a) >= (b)) ? (a) : (b))
```

--- Process ---

```
3:  else if (op == MPI_MAX) {
4:      for (c = 0; c < count; c++) {
5:        recvbuf[c] = max(recvbuf[c], tmp_max[c]);} // in the MPI_MIN use min
```

```
6:      for (c = 0; c < count; c++) {
7:        tmp_max[c] = recvbuf[c]; //in the MPI_MIN use tmp_min[c]
```

Compare recvbur and tmp_max with definition, and result will be store recvbuf. Which is also stored in tmp_max.

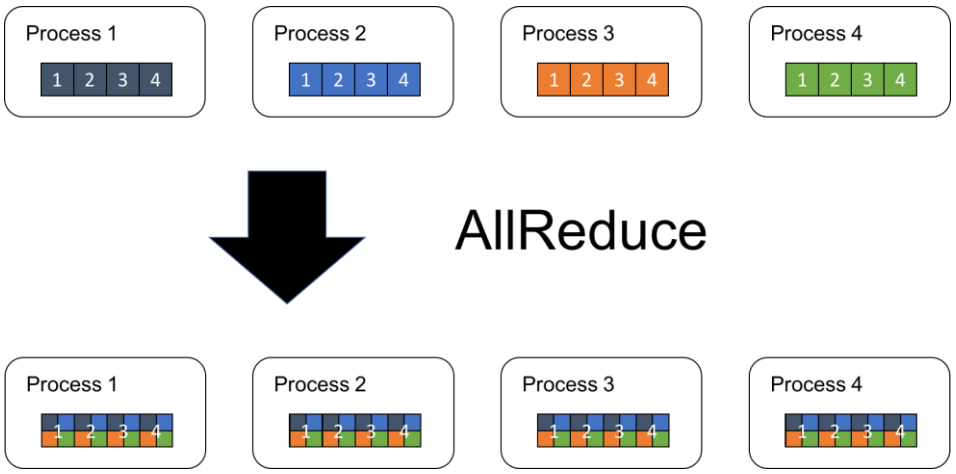
2.3 MPI_Allreduce

MPI_Allreduce is one of MPI internal function of communication between ranks. AllReduce is an operation that reduces the target arrays in all processes to a single array and returns the resultant array to all processes. Let P the total number of processes. Each process has an array of length N called A_P. i-th element of the array of process p (1≤p≤P) is $A_{P,i}$.

The resulting array B is to be:

$$B_i = A_{1,i}\ Op\ A_{2,i}\ OP\ \ldots\ldots Op\ A_{P,i}$$

Op is a binary operator. SUM, MAX, and MIN are frequently used. In distributed deep learning, the SUM operation is used to compute the mean of gradients. Assume that the reduction operation is SUM. Fig.2.3.1 illustrates how the AllReduce operation works by using an example of P=4 and N=4.



(Fig.2.3.1 AllReduce Operation)

There are several algorithms to implement the operation. For example, a straightforward one is to select one process as a master, gather all arrays into the master, perform reduction operations locally in the master, and then distribute the resulting array to the rest of the processes. Although this algorithm is simple and easy to implement, it is not scalable. The master process is a performance bottleneck because its communication and reduction costs increase in proportion to the number of total processes.

3 Scaling

3.1 Methodology

The three main algorithms for assessing the performance of an MPI program are the **Ring, Tree and AllReduce** algorithms. In the three models, algorithms' performance is assessed in terms of the **Runtime** - how much time is consumed to execute the process. Defined as $T\text{ring}(B)=(P-1)(\alpha+\beta B)$ and $T\text{tree}(B)= 2log_2P\ (\alpha+\beta B)$ in previous algorithm section. With the main three models, scaling is programmed to run two other variables. First, to find out the effect of the number of processes and message size. In the case of message size, because lack of memory(issued by too big number) makes maximum size up to 2048 and the number of processes up to 128.

To assess the result, it was performed by the batch script for compiling sequential codes. strong scaling performance, the run time of the 3 algorithms was measured on the Hamilton par.7 partition for message sizes n of 2 to 2048, for between 1 to 128 processes.

3.2 Results with Conclusion

The run times recorded for comparing three algorithms and graphs are shown in Tables.3.2.1 and Fig.3.2.1 and the run times recorded for comparing the number of processes and its graph shown in Table 3.2.2 and Fig.3.2.2. For example, with Table.3.2.1it is clear that **tree allreduce** runtime is almost **10%** of run time in ring allreduce with 2048 size. Likewise, Table 3.2.2 shows the128 processes is 4 times slower than 64 processes.

Size of message	Test Case Runtime		
Algorithm	Ring	Tree	AllReduce
Number of Processes	128	128	128
1	5.59E-04	8.47E-05	4.45E-05
2	5.58E-04	6.66E-05	3.85E-05
4	5.39E-04	6.89E-05	4.04E-05
8	5.69E-04	7.23E-05	5.12E-05

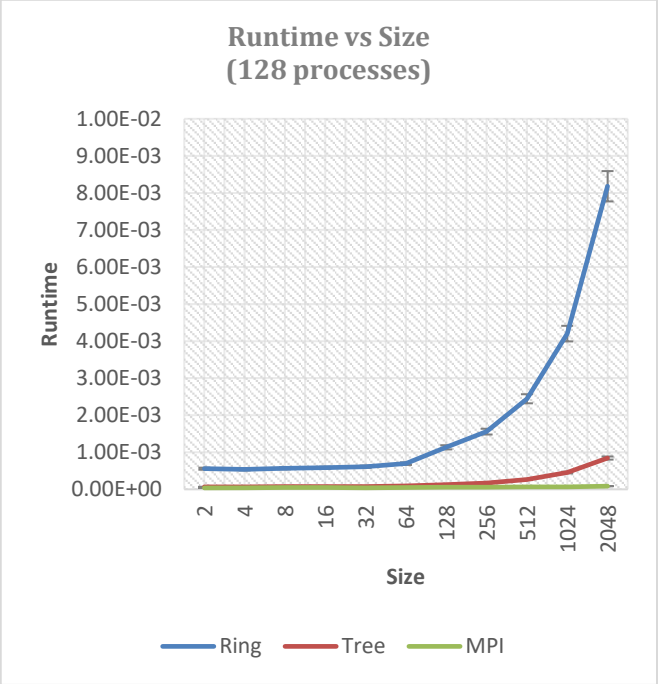
Size of Message	Test Case Runtime						
Algorithm	Ring						
Number of Processes	2	4	8	16	32	64	128
1	7.13E-07	1.66E-06	3.64E-06	7.38E-06	3.59E-05	1.39E-04	5.59E-04
2	6.20E-07	1.61E-06	3.63E-06	7.41E-06	3.42E-05	1.40E-04	5.58E-04
4	7.53E-07	1.66E-06	3.52E-06	7.58E-06	3.49E-05	1.45E-04	5.39E-04
8	7.94E-07	1.69E-06	3.83E-06	8.11E-06	3.75E-05	1.48E-04	5.69E-04

16	5.82E-04	7.27E-05	4.93E-05
32	6.10E-04	7.82E-05	4.36E-05
64	6.93E-04	9.05E-05	5.24E-05
128	1.13E-03	1.26E-04	5.58E-05
256	1.56E-03	1.66E-04	5.40E-05
512	2.44E-03	2.62E-04	6.59E-05
1024	4.20E-03	4.54E-04	6.71E-05
2048	8.18E-03	8.44E-04	8.61E-05

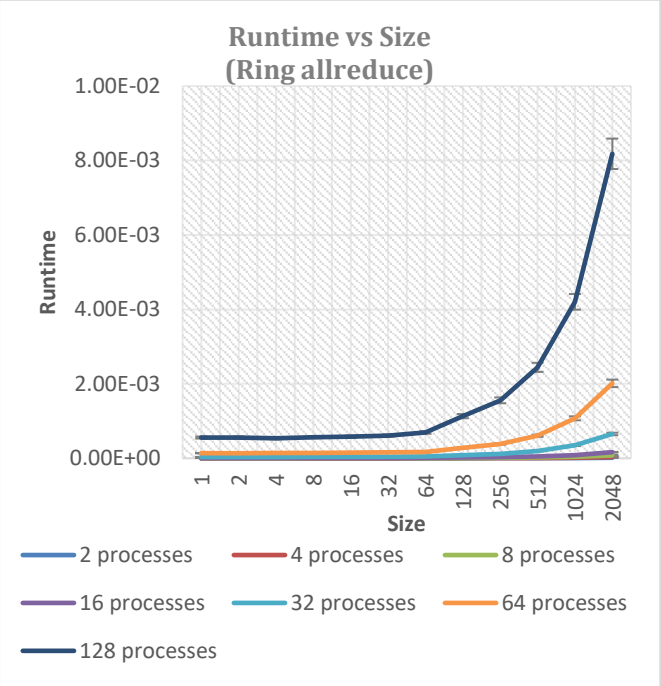
(Table.3.2.1 Scaling by size in three algorithm)

16	8.38E-07	1.91E-06	4.23E-06	8.78E-06	3.98E-05	1.51E-04	5.82E-04
32	1.00E-06	2.17E-06	4.89E-06	9.91E-06	4.48E-05	1.65E-04	6.10E-04
64	1.32E-06	3.12E-06	6.30E-06	1.28E-05	5.30E-05	1.70E-04	6.93E-04
128	1.87E-06	4.10E-06	9.82E-06	2.03E-05	8.45E-05	2.79E-04	1.13E-03
256	2.73E-06	6.12E-06	1.41E-05	3.01E-05	1.22E-04	3.88E-04	1.56E-03
512	4.38E-06	1.05E-05	2.30E-05	4.80E-05	1.96E-04	6.07E-04	2.44E-03
1024	7.52E-06	1.83E-05	4.05E-05	8.64E-05	3.48E-04	1.07E-03	4.20E-03
2048	1.49E-05	3.54E-05	7.89E-05	1.65E-04	6.55E-04	2.01E-03	8.18E-03

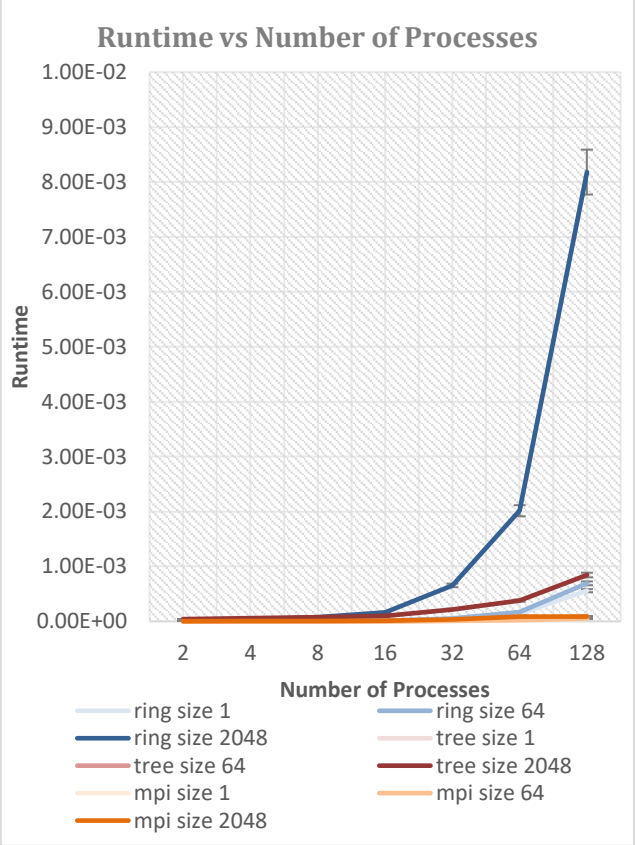
(Table.3.2.2 Scaling by size in the number of processes)



(Fig.3.2.1 Runtime vs Size)



(Fi.g3.2.2 Runtime vs Size)



(Fig3.2.3 Runtime vs Num. Processes)

Fig3.2.1 finds that the runtime is **smaller in Allreduce** MPI internal function and following **Tree algorithm**, at **last ring** was slowest run time. This means the structure of sending a message on the Tree algorithm is simpler than the Ring algorithm. The graphs increase exponentially (e^x) with message size. In this course work, size also sours exponentially. Therefore this could guess that the **increasing size** will influence to **run time linearly**. In Fig.3.2.2, it can clearly show the **bigger processes** make **longer run time**. Fig.3.2.3 can derive the tree allreduce time function of processes with T=b*$log_2(P)$ and it has the **same tendency** as lecture data.

The MPI_Allreduce and Tree_allreduce do **not** have the **same behaviour** in many algorithmic scaling refer to graphs and tables. The biggest difference between the two ways is the way to communicate. First of all, **MPI_Allreduce** use a **master process** to gather **all array** and execute operator for calculating and spreading to other processes. However, **Tree allreduce** gather array $log_2(p)$ times and calculate **each time**. After that, it **broadcast** to processes again. Therefore, if the **processes are increased**, the runtime **gap** between MPI_Allreduce and Tree Allredude will be much **bigger**.