PHYS52015 Coursework
Part 2: MPI

CIS Username: bgtt65

1 Introduction

Implementations of two parallel all-reduce algorithms for the ring collective communication and the tree collective communication and a method for native all-reduce operation in MPI are described in this report. With the experimental data from a series of benchmarking, the performance of these implementations is also compared and discussed.

2 All-Reduce Algorithms

All-reduce is a collective-combine operation utilised in synchronous parameter updates in parallel algorithms. The following algorithms are restricted to reductions over processes whose size is a power of two (e.g, 1, 2, 4, …, 2^n) and parallel implementations are achieved using MPI. For 2^n processes, each process can be labelled as *rank* where $0 \leq rank < 2^n$. Therefore, $P(rank)$ describes a process whose label is *rank*. And the message array in $P(rank)$ can be defined as M_{rank}. The purpose of the all-reduce algorithm is to access the reduced results across all processes rather than the root process, and the reduced results are calculated using the method of calculating option *Op* (i.e., sum, multiplication, maximum and minimum).

2.1 Ring All-Reduce

Ring all-reduce algorithm for 2^n processes is given in Alg. 1. In the beginning, each process $P(rank)$ sends its message array M_{rank} to the next process $P(next\_rank)$ and received a message array from the previous process $P(previous\_rank)$. Then, based on *Op,* calculate a reduced result $M_{reduction}$ that will be sent in the next round. Repeat the same procedure until all the processes get the final reduced result. In this case, `MPI_Sendrecv_replace` can be used to rotate messages.

Algorithm 1 Ring all-reduce for 2^n processes

1: **Input:** Message array M_{rank} in current rank of process $P(rank)$ and calculating option *Op*
2: **for** $i = 0, 1, \ldots, 2^n - 1$ **do** ▷ From the first process to the last process
3: Send $M_{reduction}$ to $P(next\_rank)$ ▷ The first $M_{reduction}$ is M_{rank}
4: Receive $M_{reduction}$ from $P(previous\_rank)$
5: Modify $M_{reduction}$ based on *Op* using M_{rank}
6: **end for**

2.2 Tree All-Reduce

Tree all-reduce (see fig. 1) is an operation that reduces the target message arrays in all processes (the leaves of the tree) to a single array in a process (the root of the tree) and returns the resultant array from the root to all processes.

Pseudocode of the algorithm written to perform the tree all-reduce when using 2^n processes is shown in Alg. 2. There are two significant parts in this algorithm.

In the first part, the 2^n processes can be divided into the former half group (including 2^{n-1} processes, $0 \leq rank < 2^{n-1}$) and the latter half group (including 2^{n-1} processes, $2^{n-1} \leq rank < 2^n$). Then, the latter

half group send their message arrays to the former half. Therefore, each process in the former half can receive a message array from the corresponding process and calculate a reduced result $M_{reduction}$ based on *Op*. After that, utilize the former half processes that have updating message arrays to repeat the same procedure until only one process is left.

In the second part, the algorithm starts from the last one (equal to 2^0) process which has a final reduced result. This process sends its message array to the next process, then make a group (including 2^1 processes, $0 \le rank < 2^1$) that have the final reduced result. Afterwards, processes in this group send their message to the corresponding process in the next group (including 2^1 processes, $2^1 \le rank < 2^2$) and these two groups merge into a new group. Repeat the same procedure until all the processes get the final reduced result.

To implement the tree all-reduce, whether it from the leaves to the root or from the root to the leaves, each process only send or receive a message array once at any loop. Therefore, it can be achieved using `MPI_Send` and `MPI_Recv`. For the calculation of different *Op*, a switch statement can be applied to find the corresponding calculation method.

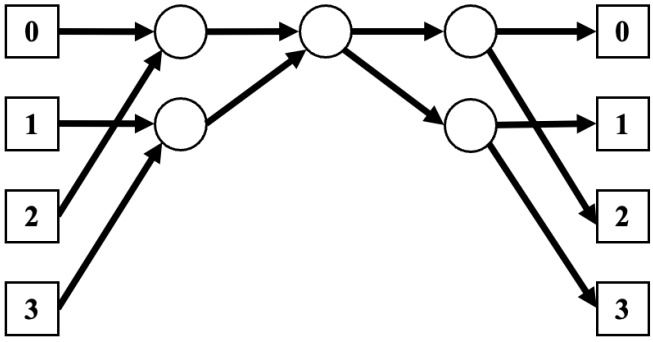

Figure 1: Tree all reduce

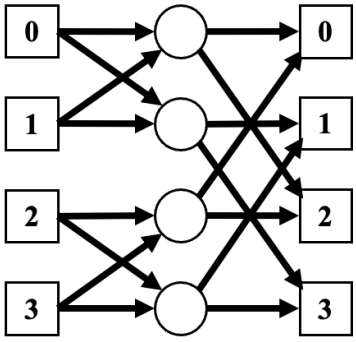

Figure 2: Butterfly all-reduce

Algorithm 2 Tree all-reduce for 2^n processes

1: **Input:** Message array M_{rank} in current rank of process $P(rank)$ and calculating option *Op*
2: **for** $i = 2^n, 2^{n-1}, \ldots, 2^1$ **do** ▷ From the leaves to the root
3: **if** $rank < i/2$ **then** ▷ Former half of *i* processes
4: Receive $M_{rank+i/2}$ from $P(rank+i/2)$
5: Modify $M_{reduction}$ based on *Op* using $M_{rank+i/2}$ ▷ The first $M_{reduction}$ is M_{rank}
6: **else if** $rank > i/2$ **and** $rank < i$ **then** ▷ Latter half of *i* processes
7: Send M_{rank} to $P(rank-i/2)$
8: **end for**
9: **for** $i = 2^0, 2^1, \ldots, 2^{n-1}$ **do** ▷ From the root to the leaves
10: **if** $rank < i$ **then** ▷ All of *i* processes
11: Send $M_{reduction}$ to $P(rank+i)$
12: **else if** $rank >= i$ **and** $rank < 2i$ **then** ▷ Other *i* processes after previous *i* process
13: Receive $M_{reduction}$ from $P(rank-i)$
14: **end for**

2.3 Native All-reduce Operation in MPI

MPI provides a function `MPI_Allreduce` to implement the operation of all-reduce. The communication structure of this function is called butterfly (see Fig. 2). For butterfly all-reduce, there are the same number of steps as a tree reduce (from the leaves to the root), but all processes fetch and combine in parallel. Therefore, it is amenable to modern high-performance interconnects with high bandwidth.

3 Benchmark and Comparison

To assess the benchmark, the run time of the ring all-reduce algorithm, the tree all-reduce algorithm and the vendor-provided all-reduce operation in MPI was measured on the Hamilton par.7 partition. In this study, two benchmarks are set based on the processes and message size.

For the first benchmark, three different implementations ran the test on a range of processes, from 2 to 128 processes. And the message size 2^1, 2^{12}, 2^{23} were chosen to compare this benchmark in the small, medium and large size of message. Similarly, for the second benchmark, the runtime of three implementations was recorded on a range of message size (including 2^1, 2^3, 2^5, …,2^{23}), then compare this benchmark in the small, medium and large number of processes (including 2, 16 and 128).

3.1 Theoretical Performance

The time of sending a message with B bytes is $T(B) = \alpha + \beta B$, where α is the message latency and β is the inverse network bandwidth. For the ring all-reduce that with P processes, the messages must be sent P-1 times to get all the way around. Therefore, the total time of ring-reduce can be defined as $T_{ring}(B) = (P\text{-}1)\ (\alpha + \beta B)$. Furthermore, use P processes to estimate the total time of tree all-reduce. The depth of the tree is $\log_2 P$ because the number of processes is divided by two each time. To reduce the message from the leaves to the root and send it back, the total time can be calculated as $T_{tree}(B) = 2\log_2 P\ (\alpha + \beta B)$. As the `MPI_Allreduce` is based on the butterfly all-reduce which has the same number of steps as a tree reduce, the time cost of butterfly all-reduce should be half of tree all-reduce.

3.2 Experimental Results

The runtime recorded for two benchmarks are shown in Tables 1 and 2. In Fig. 3, the runtime of the ring all-reduce is similar to the tree all-reduce at 2 processes. However, it becomes the worse one at 16 and 128 processes. Its scaling behaviour matches the theoretical performance, P-1 and $2\log_2 P$ are close when the number of processes is small, whereas the gap of these two variables becomes large if the number of processes increases. In addition, as the function of message size, it can be observed that the scaling of three implementations is linear. And MPI all-reduce always performs well.

As the function of the number of processes in Fig.4, no matter at which massage size, the trends of three implementations are close to their total time functions. The ring all-reduce shows the linear scaling and the tree all-reduce and MPI all-reduce show the logarithmic scaling.

Message size	**Test Case Runtime (s.)**								
Implementations:	Ring All-reduce			Tree All-reduce			MPI All-reduce		
Num. process:	2	16	128	2	16	128	2	16	128
2	5.9×10^{-7}	7.5×10^{-6}	5.4×10^{-4}	7.9×10^{-7}	3.5×10^{-6}	8.2×10^{-5}	7.9×10^{-7}	1.4×10^{-6}	3.8×10^{-5}
8	7.8×10^{-7}	8.5×10^{-6}	5.7×10^{-4}	9.0×10^{-7}	3.7×10^{-6}	8.6×10^{-5}	7.5×10^{-7}	1.4×10^{-6}	4.8×10^{-5}
32	9.7×10^{-7}	9.8×10^{-6}	6.5×10^{-4}	1.2×10^{-6}	4.3×10^{-6}	8.7×10^{-5}	7.1×10^{-7}	1.7×10^{-6}	4.4×10^{-5}
128	1.9×10^{-6}	2.0×10^{-5}	1.1×10^{-3}	2.0×10^{-6}	7.4×10^{-6}	1.1×10^{-4}	1.0×10^{-6}	2.0×10^{-6}	5.9×10^{-5}
512	4.1×10^{-6}	4.8×10^{-5}	2.4×10^{-3}	4.5×10^{-6}	1.6×10^{-5}	1.7×10^{-4}	1.4×10^{-6}	3.4×10^{-6}	5.8×10^{-5}
2048	1.5×10^{-5}	1.7×10^{-4}	8.1×10^{-3}	1.6×10^{-5}	5.2×10^{-5}	4.7×10^{-4}	3.0×10^{-6}	9.4×10^{-6}	9.2×10^{-5}
4096	2.9×10^{-5}	3.2×10^{-4}	1.6×10^{-2}	3.0×10^{-5}	9.6×10^{-5}	8.7×10^{-4}	5.4×10^{-6}	1.5×10^{-5}	2.6×10^{-4}
8192	5.7×10^{-5}	6.3×10^{-4}	3.2×10^{-2}	5.5×10^{-5}	1.9×10^{-4}	1.7×10^{-3}	8.4×10^{-6}	2.3×10^{-5}	3.1×10^{-4}
32768	2.3×10^{-4}	2.6×10^{-3}	0.13	2.2×10^{-4}	7.2×10^{-4}	6.6×10^{-3}	3.3×10^{-5}	6.5×10^{-5}	6.4×10^{-4}
131072	9.2×10^{-4}	1.1×10^{-2}	0.59	8.7×10^{-4}	2.8×10^{-3}	2.7×10^{-2}	1.4×10^{-4}	2.6×10^{-4}	4.6×10^{-3}
524288	3.7×10^{-3}	5.3×10^{-2}	3.27	3.4×10^{-3}	1.3×10^{-2}	0.12	5.9×10^{-4}	1.7×10^{-3}	1.5×10^{-2}
2097152	1.4×10^{-2}	0.23	13.07	1.4×10^{-2}	5.2×10^{-2}	0.50	3.3×10^{-3}	6.7×10^{-3}	5.5×10^{-2}
8388608	6.9×10^{-2}	0.93	54.63	6.3×10^{-2}	0.22	2.07	1.8×10^{-2}	2.6×10^{-2}	0.27

Table 1: Test case runtime, for 2 to 2^{23} message size at fixed processes

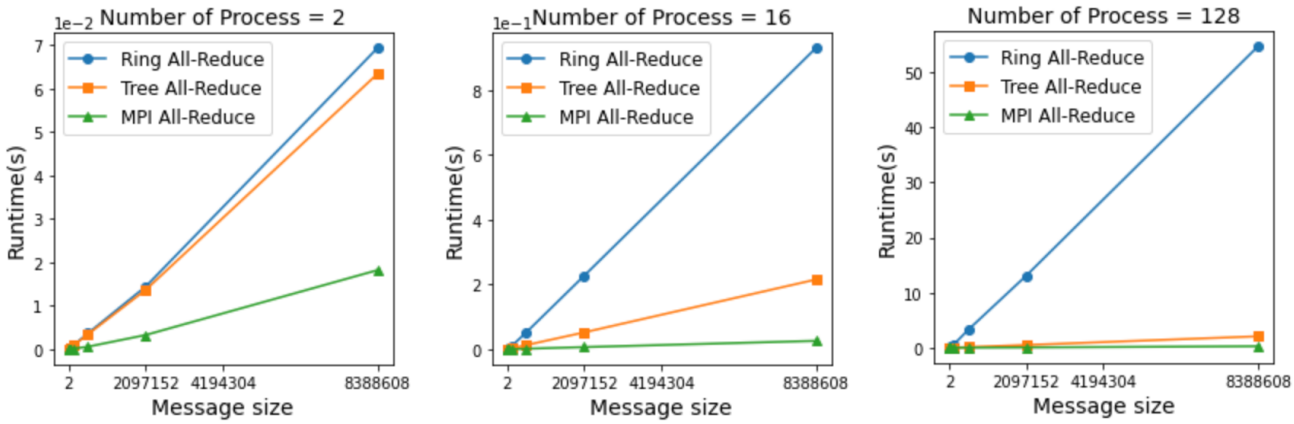

Figure 3: Message size vs. runtime at fixed num. process

Num. Process	**Test Case Runtime (s.)**								
Implementations:	Ring All-reduce			Tree All-reduce			MPI All-reduce		
Message size:	2	4096	8388608	2	4096	8388608	2	4096	8388608
2	5.9×10^{-7}	2.9×10^{-5}	6.9×10^{-2}	7.9×10^{-7}	3.0×10^{-5}	6.3×10^{-2}	7.9×10^{-7}	5.4×10^{-6}	1.8×10^{-2}
4	1.7×10^{-6}	6.7×10^{-5}	0.17	1.8×10^{-6}	5.0×10^{-5}	0.10	1.1×10^{-6}	9.4×10^{-6}	1.7×10^{-2}
8	3.7×10^{-6}	1.5×10^{-4}	0.41	2.6×10^{-6}	7.2×10^{-5}	0.15	1.1×10^{-6}	1.2×10^{-5}	2.7×10^{-2}
16	7.5×10^{-6}	3.2×10^{-4}	0.93	3.5×10^{-6}	9.6×10^{-5}	0.22	1.4×10^{-6}	1.5×10^{-5}	2.6×10^{-2}
32	3.5×10^{-5}	1.3×10^{-3}	4.06	1.5×10^{-5}	2.1×10^{-4}	0.50	5.4×10^{-6}	4.4×10^{-5}	9.3×10^{-2}
64	1.4×10^{-4}	4.0×10^{-3}	14.06	3.2×10^{-5}	3.7×10^{-4}	0.91	2.1×10^{-5}	9.9×10^{-5}	0.12
128	5.4×10^{-4}	1.6×10^{-2}	54.63	8.2×10^{-5}	8.7×10^{-4}	2.07	3.8×10^{-5}	2.6×10^{-4}	0.27

Table 2: Test case runtime, for 2 to 128 processes at fixed message size

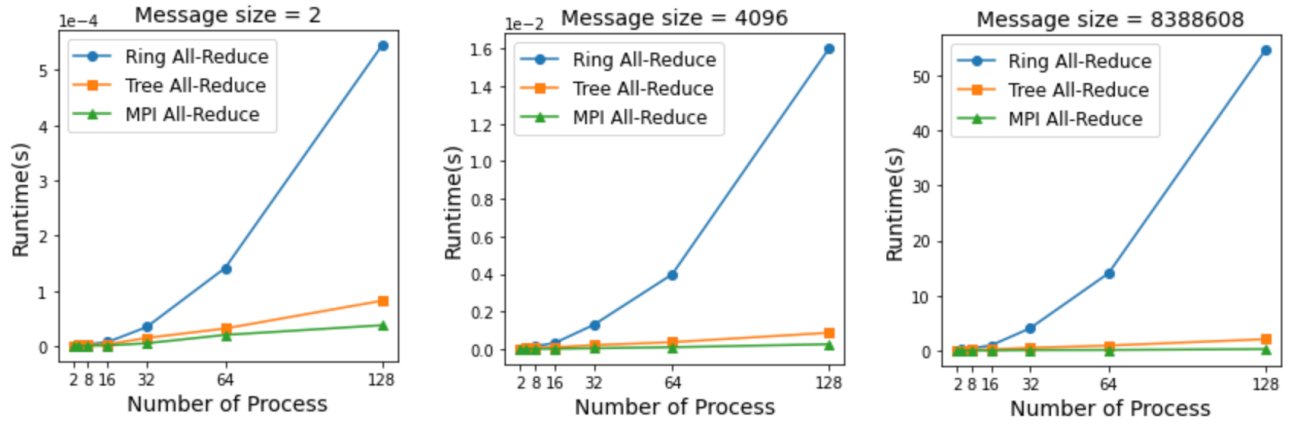

Figure 4: Num. process vs. runtime at fixed message size

From the scaling behaviour in the figure, the runtime of MPI all-reduce is almost half of tree all-reduce that prove the effectiveness of the butterfly algorithm. When the number of processes is quite small, three implementations all have a good performance. But only tree all-reduce and MPI all-reduce can maintain low runtime at the medium and large number of processes. As the algorithm of MPI all-reduce is more efficient than others, there is no doubt that MPI all-reduce is the best choice.

4 Conclusion

Ring all-reduce, tree all-reduce and MPI all-reduce have different characteristics based on their algorithms. As the function of message size, the scaling of them is linear means the message size is not the main control variable in the total time function. Tree all-reduce and MPI all-reduced have the same algorithmic scaling as a function of the number of processes. After comparison between three implementations, MPI all-reduce has the better performance at any number of processes.