Homework3:Report

Kai Zhang 2023316026

Tsinghua University

zhang-k23@mails.tsinghua.edu.cn

Oct 14th 2023

1. Implementation

First, I set up the basic MPI operations to create multiple processes and then copied the data as the initial value for the reduction algorithm. The code is as follows:

```
int rank, size;
// Get the current process rank
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Get the size of the communicator
MPI_Comm_size(MPI_COMM_WORLD, &size);
memcpy(recvbuf, sendbuf, count * sizeof(int));
```

For the case of a single process, there's no need for reduction. We can simply return the copied result.

Then, I calculated the loop depth for the case of dividing by 2:

```
int max_depth = 0;
int temp_size = size;
while (temp_size > 1) {
    max_depth++;
    temp_size = (temp_size + 1) / 2;
}
```

Next comes the loop for the reduce and merge operation, which is the core of our implementation:

```
for (int depth = 0; depth < max_depth; depth++) {
    // Calculate the partner process rank
    int partner = rank ^ step;
    if (rank % (2 * step) == 0 && partner < size) {
    int *received = (int *) malloc(count * sizeof(int));
    MPI_Request recv_request;
```

```
    MPI_Irecv(received, count, MPI_INT, partner, 0, MPI_COMM_WORLD,
    &recv_request);
    // Wait for the receive to complete
    MPI_Wait(&recv_request, MPI_STATUS_IGNORE);
    // Apply the reduction operation (in this case, MPI_SUM)
    for (int i = 0; i < count; i++)
        recvbuf[i] += received[i];
    free(received);
    }
    else if (rank % (2 * step) == step) {
        int dest = rank - step;
        MPI_Request send_request;
        MPI_Isend(recvbuf, count, MPI_INT, dest, 0, MPI_COMM_WORLD,
        &send_request);
        // Wait for the send to complete
        MPI_Wait(&send_request, MPI_STATUS_IGNORE);
    }
    step *= 2;
}
```

During partner process computation and reduction operations, we employed the Bitwise Butterfly algorithm, which enables the rapid calculation of communication patterns and objects in parallel computing, facilitating the execution of reduction operations.

Considering that only addition is required, our reduction process involves performing addition operations on partner process data. Subsequently, in order to expedite inter-process communication, we employed non-blocking communication operations using send and recv, as MPI_Isend() and MPI_Irecv().

With these steps, our MPI_Reduce() function is now fully implemented.

2. Time Result and Analysis

Through 10 times average experiments, we obtained the results of N=2,4, and 8

nodes on data of 64K, 1M, 16M, and 256M integers respectively, and compared them with the MPI_Reduce() function, as shown in Table 2.1.

Table 2.1 Time Results

Node Num	Data Size	Time of MPI_Reduce (μs)	Time of My_Reduce (μs)
2	64K	7193.8	5567.9
	1M	102028.9	36466.1
	16M	1517241.3	172202.9
	256M	21125222.4	1828888.9
4	64K	6164.4	8736.2
	1M	54256.5	55652.7
	16M	889048.5	325975.8
	256M	5484650.6	2821446.2
8	64K	9326.5	6581.2
	1M	42698.3	39916.0
	16M	514360.7	337341.1
	256M	6119606.7	4228270.0

It can be observed that, with an increase in the number of nodes, the time overhead for the same amount of data has significantly decreased. Regarding the test results, it can be seen that my implemented ***My_Reduce()*** function (corresponding to the ***YOUR_Reduce()*** function in the code) performs similarly to the ***MPI_Reduce()*** function, and even outperforms it, especially with larger data sets. I have marked in green the cases where it has better time efficiency compared to the standard function.

In order to more intuitively show the time results of my implementation, I also ploted a bar graph, as shown in Fig. 2.1.

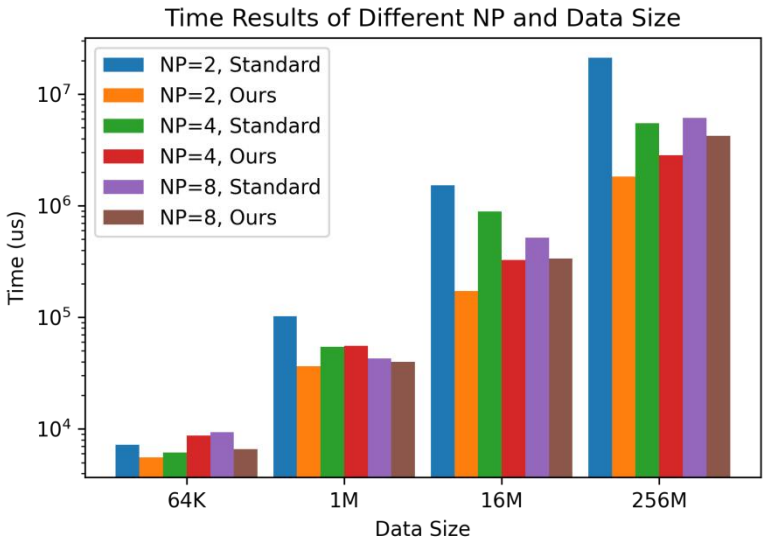

Fig 2.1 Time Results of Different NP and Data Size

From the graph, we can also observe that our implementation has achieved results that are largely consistent with, and in many cases, even superior to the standard implementation. However, the experimental results are not stable. So for every experiment, I did it for 10 times and get the average results.

3. Bonus:Multi-threaded Version

I found that in each reduction operation, a summation of large arrays is required, and this operation can be parallelized using OpenMP. Therefore, we added OpenMP statements to the internal summation operation, which resulted in a multi-threaded version of MPI_Reduce(). The code is as follows:

```
#pragma omp parallel for
for (int i = 0; i < count; i++)
    recvbuf[i] += received[i];
```

I compared the performance of the multi-threaded implementation with the single-threaded version, using data 256M only, as shown in Table 3.1 and Fig 3.1:

Table 3.1 Time Results of Multi-threaded Version

Node Num	Thread Num	Time of My_MultiThread_Reduce (μs)
	1	1529685
	2	1490661

2	3	1465264
	4	1519337
4	1	2906701
	2	2843233
	3	2813979
	4	2702313
8	1	4643249
	2	3973864
	3	3970851
	4	3733107

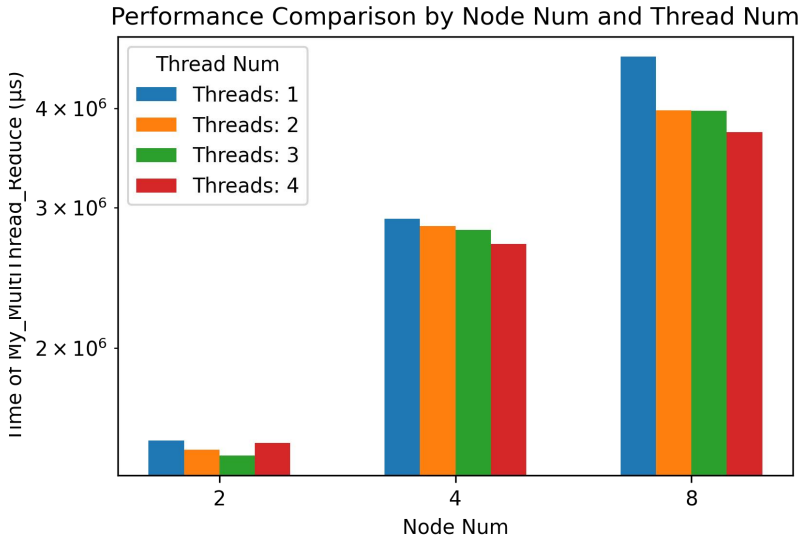

Fig 3.1 Time Results of Different NP and Thead Num

We can see that on larger datasets, for most cases, as the number of threads increases, the program's execution time indeed decreases, achieving parallel acceleration.However, due to the instability of hardware multi-threading, there may be fluctuations and errors in the test data.

4. Conclusion

I have implemented an MPI_Reduce()(Only for Sum Operation) version based on

the divide-and-conquer algorithm and conducted several tests. The results indicate that the implementation's performance is close to the standard version. Lastly, we further optimized MPI_Reduce() using multi-threading techniques(Using OpenMP learned in the first two weeks), which achieved improved results. A comprehensive report containing the complete experimental results and code details is provided in this report.