Assignment 2

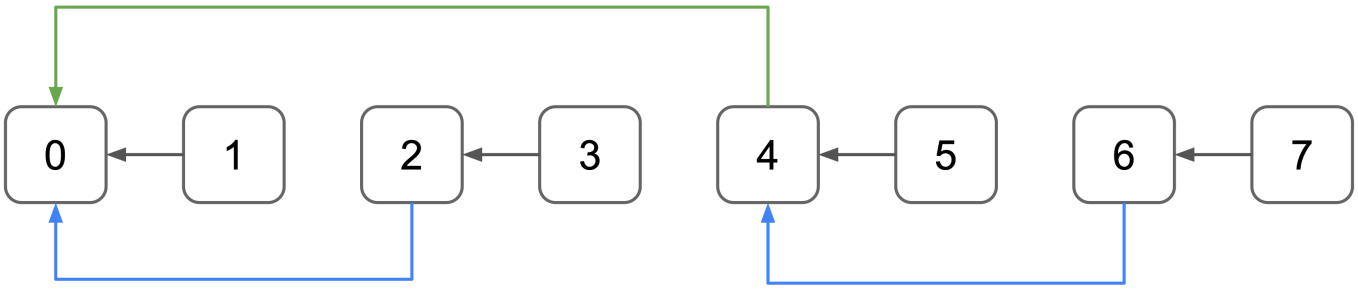
Sahand Sabour - 山姆 - 2022380024

Introduction

In this assignment, we are tasked with implementing the MPI_Reduce function using functions provided by the MPI API, such as MPI_Recv and MPI_Send.

Implementation

The main idea behind this implementation is illustrated in the below figure. For this assignment, the maximum number of nodes is 8 (rank 0-7). Let each square in the figure represent a node.



Since the total number of nodes is either 2, 4, or 8, the first step of the parallelization is sending the array from odd-numbered nodes to even-numbered nodes and summing up the results in the even-numbered nodes (as shown by the grey arrows). For the sum, we initialize a temporary array for storing the incoming results and add this array to the current node's buffer (sendbuf) and store the results of the sum in the receive buffer (recvbuf).

```
// Odd-numbered nodes send their buffer to the previous even-numbered node
if (rank % 2 != 0)
{
  dest = rank - 1;
  MPI_Send(sendbuf, count, MPI_INT, dest, 0, MPI_COMM_WORLD);
}
// Even-numbered nodes receive the arrays and perform the sum
else
{
  // Initialize array to hold the received buffer
  int *tempbuff;
  tempbuff = (int *)malloc(MAX_LEN * sizeof(int));
  // Receive buffer from the next odd-numbered node
```

```
  source = rank + 1;
  MPI_Recv(tempbuff, count, MPI_INT, source, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  // Sum the received buffer with the current buffer
  Sum_Buffers(sendbuf, tempbuff, recvbuf, count);
```

Where the Sum_Buffers functions as follows:

```
// Function for adding elements of the two buffers one by one
void Sum_Buffers(const int *buffer_a, int *buffer_b, int *recvbuf, int count)
{
    for (int i = 0; i < count; i++)
        recvbuf[i] += buffer_a[i] + buffer_b[i];
}
```

The second step of the parallelization would be to send the results of the sum of the two buffers to the previous even-numbered node (blue arrows). For 2 nodes, this is an unnecessary step. For 4 nodes, the results of node 2 are sent to 0 for the final sum.

```
// This step is only necessary for more than 2 nodes
if (size > 2)
{
  // For nodes 2 and 6
  if (rank % 4 != 0)
  {
    dest = (rank / 4) * 4; // 2 -> 0 and 6 -> 4
    MPI_Send(recvbuf, count, MPI_INT, dest, 0, MPI_COMM_WORLD);
  }
  else
  {
    // For node 4
    if (rank != 0)
    {
      // Receive the buffer from node 6 and sum the results
      MPI_Recv(tempbuff, count, MPI_INT, 6, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      int *temp = (int *)malloc(MAX_LEN * sizeof(int));
      Sum_Buffers(temp, tempbuff, recvbuf, count);
      // Send the results to node 0 for final sum
      MPI_Send(recvbuf, count, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    // For node 0
    else
```

```
    {
      // Receive results from nodes 2 (if size = 4 or 8) and 4 (if size = 8)
and calculate sum
      for (int j = 1; j < log2(size); j++)
      {
        source = j * 2;
        MPI_Recv(tempbuff, count, MPI_INT, source, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        int *temp = (int *)malloc(MAX_LEN * sizeof(int));
        Sum_Buffers(temp, tempbuff, recvbuf, count);
      }
    }
  }
}
```

Regarding 8 nodes, results for nodes 6 and 2 are sent to nodes 4 and 0, respectively. However, results from node 4 are directly sent to node 0 (green arrow). The reason for this design decision was that sending the results from node 4 to node 2, then to node 0, would add an unnecessary step.

```
// Clear the receive buffer for each node
for (int i = 0; i < count; i++)
  recvbuf[i] = 0;
```

Lastly, as shown above, the receive buffer for each node is cleared at the beginning of their operation to make this buffer suitable for storing the results of the sum.

Multi-Threading

Since each MPI process only uses 1 CPI core, as a bonus, we leverage the OpenMP API to parallelize the program on multiple threads. In our implementation, the for loop in the sum function (Sum_Buffers) could be multi-threaded as follows:

```
#define NUM_THREADS 1 // set to 1, 2, 3, and 4 to report results
#pragma omp parallel for num_threads(NUM_THREADS) schedule(guided)
for (int i = 0; i < count; i++)
  recvbuf[i] += buffer_a[i] + buffer_b[i];
}
```

We used guided scheduling, as this type of scheduling demonstrated the best results in the previous assignment. However, since the computational complexity for different

iterations would be fairly similar, static scheduling could also be used.

Results

The obtained results for the main part of this assignment are provided in the below table.

# Nodes	Array Size	MPI_Reduce (μs)	Your_Reduce (μs)
2	64k	5,601	2,587
2	1M	87,065	21,427
2	16M	1,349,710	252,403
2	256M	20,974,621	4,051,229
4	64k	4,596	4,108
4	1M	40,070	30,804
4	16M	614,130	503,069
4	256M	11,341,200	7,535,841
8	64k	8,951	8,493
8	1M	38,116	53,820
8	16M	538,987	624,006
8	256M	8,616,184	9,540,725

Accordingly, the results of multi-threading for array size of 256M are provided in the below table.

# Nodes	# Threads	MPI_Reduce (μs)	Your_Reduce_MultiThreaded (μs)
2	1	14,875,459	3,795,050
2	2	15,431,387	3,644,490
2	3	13,795,250	2,798,630
2	4	16,066,923	2,649,538
4	1	10,479,700	6,532,334
4	2	9,316,657	5,616,887
4	3	10,597,322	5,547,743
4	4	10,599,139	5,237,466
8	1	9,813,315	9,486,927
8	2	10,066,120	7,493,879
8	3	9,563,048	6,634,345
8	4	8,894,055	6,390,358

Discussion

As shown in the first table, we can observe that for a smaller number of nodes and smaller arrays, the runtime for Your_Reduce is considerably smaller than the original MPI_Reduce. However, as the number of nodes or the size of arrays increases, the difference between the runtimes of the two methods decreases significantly. We believe this happened because the computational complexity of the overhead and the parallelization logic for the original implementation is considerably more complicated than our implementation. Hence this overhead slows down the reduction process in fewer nodes and smaller arrays but it is highly beneficial for larger arrays and more nodes.

Regarding multi-threading, the second table demonstrates that distributing the processing workload among different CPU cores increases speedup; as we increase the number of threads, the runtime relatively decreases. We can also observe the same finding as above: the difference between the runtimes of the two methods decreases as the problem size increases.

Conclusions

In this assignment, we implemented the MPI_Reduce function using the MPI API. Accordingly, we used the OpenMP API to parallelize each node's process across multiple CPU cores. Lastly, we provided the obtained experimental results and discussed our findings.