Parallel Programming HW4 Write Up
Due Monday February 15, 2021, 11:59 pm

Connor Osborne
A01880782

Monday February 15, 2021

1 Description

For homework 4, I wrote an MPI program that finds the global sum of an integer variable across all processes, such that each process has the final sum. This program assumes power of 2 processes. It also uses 5 different methods to find the sum:

a) MPI_Allreduce to find the sum immediately.

b) only MPI_Gather and MPI_Bcast to communicate among processes.

c) only MPI_Send and MPI_Recv to send all the integers to rank 0, where the sum is calculated and the result is send back to each process.

d) only MPI_Send and MPI_Recv in a ring interconnection topology.

e) only MPI_Send and MPI_Recv in a hypercube topology.

2 Implementation

The implementation for using MPI_Allreduce sets up each process with their MPI rank and the number of processes. Then it initializes a variable for the data, which is equal to each processes rank, and the total. The MPI_Allreduce is used to sum the data variables across each process and store the answer in the total variable.

```
void allreduce(){
int rank, size;

MPI_Comm_rank(MCW, &rank);
MPI_Comm_size(MCW, &size);

int aData = rank;
int aTotal = -1;

MPI_Allreduce(&aData,&aTotal,1,MPI_INT,MPI_SUM,MCW);
MPI_Barrier(MCW);
cout<<rank<<" calculated total using MPI_Allreduce: "<<aTotal<<endl;

}
```

The implementation for using MPI_Gather and MPI_Bcast sets up each process with their MPI rank and the number of processes. Then it initializes a variable for the data, which is equal to each process's rank, an array for storing the data gathered from each process ,and a variable for the total. Next, each value in the array is set to -1 using a for loop to ensure proper data storage later. Then, MPI_Gather is used to have each process send data to be stored in the array for process 0. Then process 0 loops through the array to sum up each element. Once the total is found MPI_Bcast is used to update the total for all processes.

```
void gatherAndBcast(){
int rank, size;

MPI_Status mystatus;
MPI_Comm_rank(MCW, &rank);
MPI_Comm_size(MCW, &size);

int bData = rank;
int recvdata[size];
int bTotal = 0;
if(rank <= 0){
    for(int i=0;i<8;++i){
        recvdata[i]=-1;
    }
}

MPI_Gather(&bData,1,MPI_INT,recvdata,1,MPI_INT,0,MCW);
if(rank<=0){
    for(int i=0;i<size;++i){
        bTotal += recvdata[i];
    }
}
MPI_Bcast(&bTotal, 1, MPI_INT, 0, MCW);
MPI_Barrier(MCW);
cout<<rank<<" calculated total using MPI_Gather and MPI_Bcast: "<<bTotal<<endl;

}
```

The implementation for using MPI_Send and MPI_Receive sets up each process with their MPI rank and the number of processes. Then it initializes a variable for the data, which is equal to each process's rank, a variable for keeping track of how many times process 0 has received data,and a variable for the total. Process 0 starts a while loop to receive data from each process and add that number to the total. Once it has received data from each process it then enters a for loop to send the total to each process. Every other process besides 0 will only use MPI_Send to send the data and then use MPI_Recv to block while waiting to receive the calculated total from process 0.

```
void sendAndReceive(){
int rank, size;

MPI_Status mystatus;
MPI_Comm_rank(MCW, &rank);
MPI_Comm_size(MCW, &size);

int cData = rank;
int cTotal = 0;
int received = 1;

if (rank == 0){
    while(received < size){
```

```
            MPI_Recv(&cData, 1, MPI_INT, MPI_ANY_SOURCE, 0, MCW, &mystatus);
            cTotal += cData;
            received += 1;
        }
        for(int i=1;i<size;++i){
            MPI_Send(&cTotal, 1, MPI_INT, i, 1, MCW);
        }

    }
    else{
        MPI_Send(&cData, 1, MPI_INT, 0, 0, MCW);
        MPI_Recv(&cTotal, 1, MPI_INT, 0, 1, MCW, &mystatus);
    }
    MPI_Barrier(MCW);
    cout<<rank<<" calculated total using MPI_Send and MPI_Recv: "<<cTotal<<endl;

    }
```

The implementations for the ring interconnection topology and the hyper cube topology functions are the same as shown in class where the variables for the process information are initialized and a variable is created for deciding the destination process for a process to send data to. For the ring function the destination is set to the next ranking process while accounting for wraparound so that the highest ranking process will send to process 0. The destination for the cube function is determined based on a bit-wise XOR operation done between a process' rank and the mask variable with the dimension variable passed to the function when it's called. These two functions then call the MPI_Send and MPI_Recv functions to send to their destination and receive data being sent to them.

```
    //section d (using ring topology)
void ring(int *data){
    int rank, size;
    int dest;


    MPI_Status mystatus;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    dest = (rank+1)%size;
    MPI_Send(data,1,MPI_INT,dest,0,MCW);
    MPI_Recv(data,1,MPI_INT,MPI_ANY_SOURCE,0,MCW,&mystatus);

}

// section e (using hypercube topology)
void cube(int *data, int m){
    int rank, size;
    int dest;
    unsigned int mask=1;
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);

    dest = rank^(mask<<m);

    MPI_Send(data,1,MPI_INT,dest,0,MCW);
    MPI_Recv(data,1,MPI_INT,MPI_ANY_SOURCE,0,MCW,MPI_STATUS_IGNORE);
```

```
    return;
}
```

The implementation for calling the ring and cube functions is included in the main function that calls each function in turn and uses barriers and sleep(1) to ensure proper reporting of data. For the ring function once again variables for the data and the total are made. Then a for loop is used to have use process pass it's current data to the next process while adding it to its total variable. Barriers are included before and after the call to prevent issues with race conditions. The usage of the cube function is similar in its creation of data and total variables. However it first creates a variable pwr for keeping track of what power of 2 the number of processes being used equates to. It also uses a variable equal to the number of processes running in a for loop to divide the number by two until the variable is equal to one while it increments the pwr variable. Then in a for loop from i = 0 to i = pwr, the cube function is called using i as the dimension variable so that each process will send their current total to the proper processes such that every process will have the same total at the end of the loop.

```
int main(int argc, char **argv){
    int rank, size;

    MPI_Status mystatus;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MCW, &rank);
    MPI_Comm_size(MCW, &size);
    MPI_Barrier(MCW);


    // section a
    allreduce();
    MPI_Barrier(MCW);
    sleep(1);

    // section b
    gatherAndBcast();
    MPI_Barrier(MCW);
    sleep(1);

    // section c

    sendAndReceive();
    MPI_Barrier(MCW);
    sleep(1);


    // section d
    int dData = rank;
    int dTotal = 0;

    for (int i = 0; i < size; ++i){
        MPI_Barrier(MCW);
        ring(&dData);
        MPI_Barrier(MCW);
        dTotal += dData;
    }
    MPI_Barrier(MCW);
    cout<<rank<<" calculated total using a ring interconnection topology: "<<dTotal<<endl;
    sleep(1);
```

```
    // section e

    int pwr = 0;
    int numProcesses = size;
    while(numProcesses > 1){
        numProcesses /= 2;
        pwr += 1;
    }
    int eData;
    int eTotal = rank;

    for(int i = 0; i < pwr; ++i){
        eData = eTotal;
        MPI_Barrier(MCW);
        cube(&eData, i);
        MPI_Barrier(MCW);
        eTotal += eData;
    }
    MPI_Barrier(MCW);
    cout<<rank<<" calculated total using a hyper cube topology: "<<eTotal<<endl;


    MPI_Finalize();

    return 0;
}
```

3 Input and Output with 8 processes