# Assignment4 Report
# CSE436, Summer 2016
**Kazumi Malhan**
**07/01/2016**

## 1. Backgrounds and Motivation

In the real world, almost 80% of high performance computer (HPC) uses cluster architecture, which is distributed memory architecture. As OpenMP or Cilk do not work in this memory system, the Message Passing Interface (MPI) plays key role. In this assignment, algorithm to find minimum number in an array is implemented using two different methods. The performances of two functions are compared with different number of processors.

## 2. Function Implementation Description

**mpi_min_v1.c (Scatter and Reduce)**

First local variables are declared. Next, MPI_Init function is called to initialize MPI. Also, MPI_Comm_size and MPI_Comm_rank are called to determine the total number of processors, and processor number, respectively. The processor 0 prints out the usage of the program, and array size N is defined by reading 1st parameter. After calculating the size of array that each processor will calculate the minimum number, whole array is allocated and initialized by processor 0. Also, other processors allocate the memory space to store local array.

Next, MPI_Scatter operation is performed. If the function is performed by processor 0, which is source processor (root), it scatters "local_N" elements of array to all other processors in "MPI_COMM_WORLD". For example, processor 0 gets 1st local_N elements, processor 1 gets 2nd local_N elements, processor 3 gets 3rd local_N elements, and so on. If the function is executed by other processors (not processor 0), then it simply receives the local_N elements of array from processor 0. Each processor will calculate the minimum number among the received list of numbers using for loop.

Once each processor figures out the minimum number among their own local arrays, MPI_Reduce is called. If the function is executed by target (processor 0) or root, the processor simply sends the minimum number, which stored in "min" to target (processor 0). If the code is executed by processor 0 or root, it will receive data from all other processors, perform specified operation, and place the result in another buffer. In this case, the function calculates the minimum number among received numbers, and place the result in "gmin". Processor 0 displays the code execution time, and calculated minimum number. Finally, local arrays are de-allocated, and MPI is terminated using MPI_Finalize.

**mpi_min_v2.c (Send and Receive)**

Similar to version 1 of the program, local variables are declared, and MPI is initialized using MPI_Init function. Total number of processors and current processor number are determined with MPI_Comm_size and MPI_Comm_rank, respectively. Status variable which has "MPI_Status" data type also declared because this is necessary parameter when MPI_Recv function is used later. The processor 0 displays the usage of program, and size of local array is calculated. The processor 0 allocates the memory space for array size N, and initializes them while other processors allocate the memory for local arrays.

For processor 0, by using the for loop, it sends the local_N elements of array to other processors one by one using MPI_Send. As this is a blocking message passing method, the function doesn't return until destination processor receives the message. For all other processors, they utilize MPI_Recv function to receive data from processor 0. The received data is placed in local_A. It is important to note that all MPI_Send and MPI_Recv should have matched pair as lack of this condition will end up in deadlock.

Each processor calculates its own minimum number among the local arrays using for loops. When computation completes, it will performed opposite of what the program did before. All processors except processor 0 sends the result back to processor 0 using MPI_Send. It is notable that different tag (1234) is used this time to differentiate the message from previously generated messages. The

processor 0 will receive all data using for loop and MPI_Recv. Again, it is important to have matching pair to avoid deadlock.

Next, another for loop is used to determine the smallest number among calculated minimum numbers by processor 0, and result is displayed on terminal. Finally, allocated memories are freed, and MPI is terminated using MPI_Finalize.

## 3. Performance Report

The code is compiled and executed on **lennon.secs.oakland.edu** Linux server via VPN connection. The **figure 1** shows that as number of processes increases for both functions, the execution time reduces. With 16 processors, the function can run almost 2.5 times faster than running with single processor. However, the efficiency of the program decreases significantly as number of processor increases. For example, when the program was executed with 16 processors, the efficiency was below 20%. The cause of lower efficiency is because more processors require more communication between the processors (overhead).

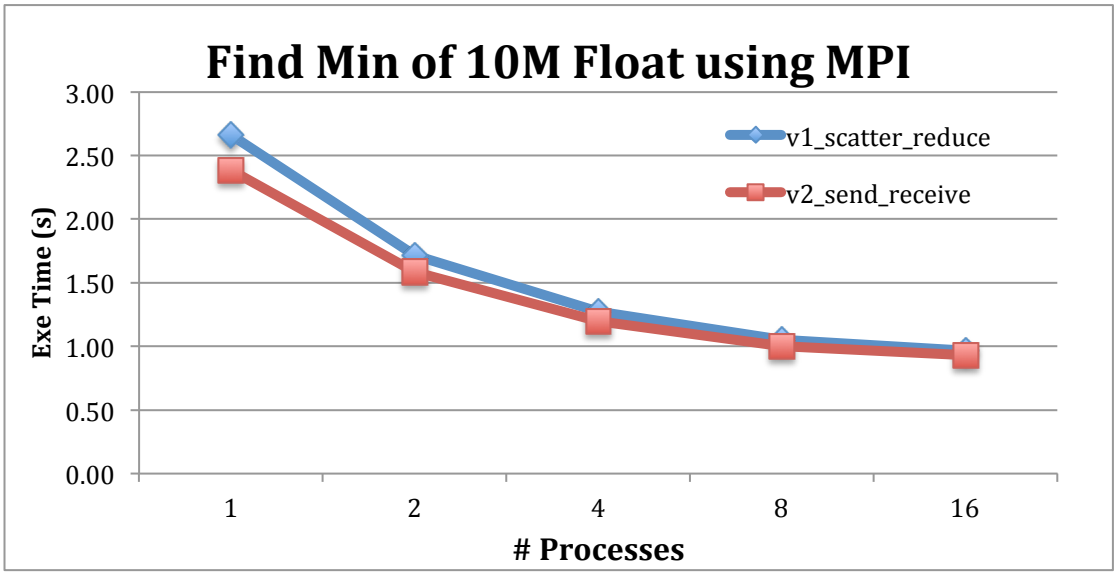


**Figure 1:** Performance difference between two functions with various numbers of processors.

Between the two functions, send/receive function was faster than scatter/reduce function, but the performance difference got smaller as number of processor increases. One cause of the difference is that scatter/reduce operation includes synchronization or barrier inside the function. This causes all processor to wait for other processors at certain point in the code. As scatter/reduce operation is internally using send/receive function + alpha, it takes more time to execute the function.

| # of Processors | | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| Speedup | Version1 | 1.55096098 | 2.09520063 | 2.5386082 | 2.75672878 |
| | Version2 | 1.50378788 | 1.98665555 | 2.37724551 | 2.56129032 |
| Cost | Version1 | 3.434 | 5.084 | 8.392 | 15.456 |
| | Version2 | 3.168 | 4.796 | 8.016 | 14.88 |
| Efficiency | Version1 | 0.77548049 | 0.52380016 | 0.31732602 | 0.17229555 |
| | Version2 | 0.75189394 | 0.49666389 | 0.29715569 | 0.16008065 |
| Overhead | Version1 | 0.771 | 2.421 | 5.729 | 12.793 |
| | Version2 | 0.786 | 2.414 | 5.634 | 12.498 |

**Table 1:** Performance matrix for both version of code.

The **table 1** above summarizes the performance of both functions with various number of processors compared to serial run (single processor execution). The amount of overhead more than doubles every time the number of processors doubles. The speedup section shows the continuous speedup, but the efficiency of the program is very low. This indicates that the most of computation power is consumed by overhead.

## 4. Conclusion

The MPI enables the program to run on multiple processors on distributed memory architecture, however, it comes with the cost. When number of processor increases, the speedup continues to improve. With 16 processors it can execute the code 2.6 times faster than serial run, but only 10% improve from execution time with 8 processors. The efficiency dropped to below 20%. This is because cost of data communication is significantly higher than shared memory programming models. When comparing the two functions, Send/Receive function runs slightly faster than Scatter/Reduce function. The main reason is that scatter/reduce contains internal synchronization point, where some processors needs to wait for other. Also, scatter/reduce functions performs more operation. The difference gets smaller when number of processors increases. The assignment demonstrated the potential of MPI, its cost, and the performance difference among different functions.

References

MPI Broadcast and Collective Communication
(http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/)