15-618 Parallel Computer Architecture and Programming

MPI as a C++ Library

Aditya Bhagwat (abhagwa2@andrew.cmu.edu)

Karthik Bhat (kbhat2@andrew.cmu.edu)

Summary

In this project, we implemented a subset of Message Passing Interface (MPI) as a C++ library to parallelize programs across multiple cores of a single node. We then evaluated the performance and scalability of our library on multiple application programs.

Specification

Here we list the functions that we expose in our library with short descriptions of their purpose and usage. The function interfaces (parameters and return values) and datatypes our library exposes are identical to those exposed by OpenMPI. This means our library is compatible with any MPI application program (provided the application is restricted to the subset we define below).

- `MPI_Datatype` is an enum denoting the datatype being passed in the message. It has the following values.
    - `MPI_INT`
    - `MPI_CHAR`
    - `MPI_FLOAT`

    - `MPI_LONG`
    - `MPI_DOUBLE`
- Miscellaneous types
    - `MPI_Comm` is a type for the communication model being used. For our simple implementation it admits only one value - `MPI_COMM_WORLD.`
    - `MPI_Op` is an enum that denotes the reduction operation being performed. It is enum with the following values - `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, `MPI_PROD`, `MPI_LAND` and `MPI_LOR` for the operators maximum, minimum, sum, product, logical AND and logical OR respectively.
    - `MPI_Status` is a type for the status returned by certain message passing functions. For our implementation, all failures are catastrophic and unrecoverable and this is used only for maintaining compatibility with the original interface.
- Basic operations
    - `MPI_Init:` Initialize data structures and global variables.
    - `MPI_Finalize:` Clean up resources and exit gracefully.
    - `MPI_Comm_size:` Return number of processes in the system.
    - `MPI_Comm_rank:` Return the calling process' rank.
- Point-to-point Synchronous Communication
    - `MPI_Send`: Send a certain amount of data to a process whose rank is specified. This is synchronous, i.e. control waits until the function returns.
    - `MPI_Recv:` Receive a certain amount of data from a process whose rank is specified. This is also synchronous.
- Collective Communication
    - `MPI_Bcast:` Root sends some data (synchronously) to all processes.

- `MPI_Scatter`: Root distributes a buffer among all processes.
- `MPI_Gather`: Root collects data from all processes into one buffer.
- `MPI_Allgather`: All processes exchange their chunk of data with all the other processes, so that everyone ends up with the same final buffer. This is functionally equivalent to a gather followed by a broadcast.
- `MPI_Reduce`: Root collects data from all processes and reduces it based on the reduction operator specified.
- `MPI_treereduce`: Same as above but with tree-shaped communication.
- `MPI_Allreduce`: Same as a MPI_Reduce but all participating processes receive reduced data instead of just the root process.
- `MPI_Wtime`: Returns the elapsed time of the process from the start of the epoch.

Approach

Broad Overview

We aimed to implement the same interface and usage as OpenMPI for the most common use case.

```
Usage:
./mpirun [-h] -n numProcs -e executable [args]
[-h]           : Print this message.
-n numProcs    : Number of Processes to create.
-e executable : Executable to run.
[args]         : Arguments for executable.
```

We expose `mpirun` as an executable that takes in the number of processes and the executable to run in each of those processes as the first two command line arguments. The executable's command line arguments follow. The program `mpirun` is responsible for spawning `N` instances of the executable. We fork `(N-1)` times and call `execve` to execute the desired user-specified executable.

The first challenge is to give each process a rank between `0` and `N-1` that they all agree on. The processes will hereafter refer to each other using this rank for message-passing. This rank can be decided at the time of forking. We stuff the rank and the total number of processes `N` in the `argv` passed to each process using `execve.` It is now the responsibility of `MPI_Init` to parse the modified `argv,` read the number of processes and rank and set these as global variables, accessible throughout. `MPI_Comm_rank` and `MPI_Comm_size` merely return the values of these global variables. `MPI_Init` is also responsible for setting the `argc` and `argv` values correctly before returning so that the user application can use them without issues.

This approach is elegant in that we follow the MPI specification exactly. A caveat of this approach is that `argc` and `argv` will have different values to what the user expects before `MPI_Init` is called. We will come back to the structure of `argv` in while outlining our approach below.

Point to Point communication : MPI_Send and MPI_Recv

Processes communicate between each other through the use of sockets. Since sockets are bidirectional, our initial implementation had only N total sockets one each for every process. All sockets would call listen on their own port during initialization in `MPI_Init`. During `MPI_Send`, the sending process would call `connect` on the destination process' port. During `MPI_Recv`, the receiving process would just call `accept`. After the connection is established, data transfer would take place.

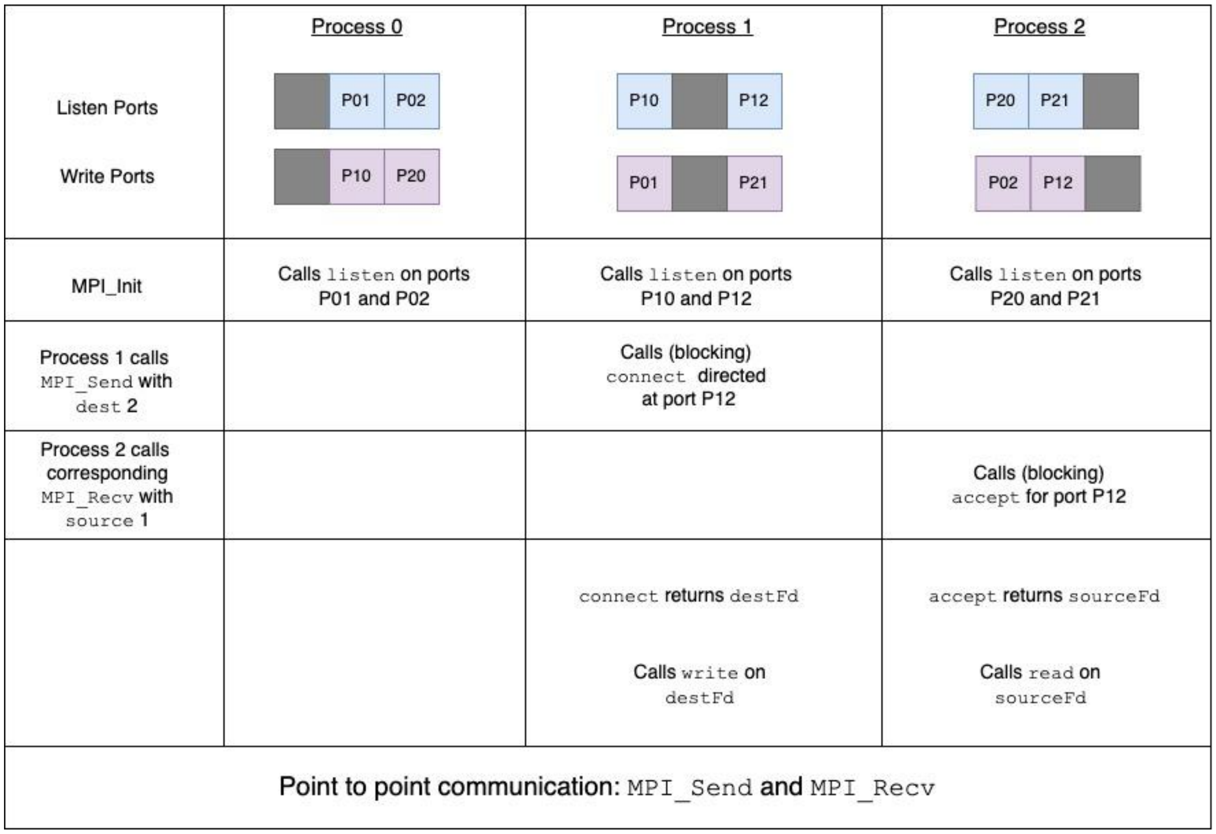
This was broadly correct, but it failed when multiple processes were sending to one process, because there is only one port for listening. There is no check in the receiving process about who is allowed to connect - it can arbitrarily accept connections from any sender who is sending and mix up data. Merely adding a check if the sender is valid will not help - because the connect requests (MPI_Send calls) can appear out of order with respect to the MPI_Recv calls in the receiver, so merely rejecting an out of order request will lead to lost data.

One way around this could be to do some kind of a handshake where process IDs are matched before data transfer but this would blow up the number of messages passed and as a result, the communication costs. An easier solution is to simply have N^2 ports - one each for all possible process combinations. (This can be reduced to ${}^{N}C_2$ ports due to the bidirectional nature of sockets, but the resulting solution becomes overly complex). The example below illustrates this choice.

Process 0	Process 1	Process 2
	`MPI_Recv(recv=buf0, source=0)`	
		`MPI_Send(data=buf, dest=1)`
	`MPI_Recv(recv=buf2, source=2)`	
`MPI_Send(data=buf, dest=1)`		

Why we need N^2 ports: Pitfalls of the accept-all approach

Our initialization function now has to start listening on `N` ports, one for each source of messages. For the moment, lets not concern ourselves with how these ports are chosen and agreed upon between processes. The below figure illustrates how point to point communication works in our system.



	Process 0	Process 1	Process 2
Listen Ports	■ P01 P02	P10 ■ P12	P20 P21 ■
Write Ports	■ P10 P20	P01 ■ P21	P02 P12 ■
MPI_Init	Calls `listen` on ports P01 and P02	Calls `listen` on ports P10 and P12	Calls `listen` on ports P20 and P21
Process 1 calls `MPI_Send` with `dest` 2		Calls (blocking) `connect` directed at port P12	
Process 2 calls corresponding `MPI_Recv` with `source` 1			Calls (blocking) `accept` for port P12
		`connect` returns `destFd` Calls `write` on `destFd`	`accept` returns `sourceFd` Calls `read` on `sourceFd`

Point to point communication: `MPI_Send` and `MPI_Recv`

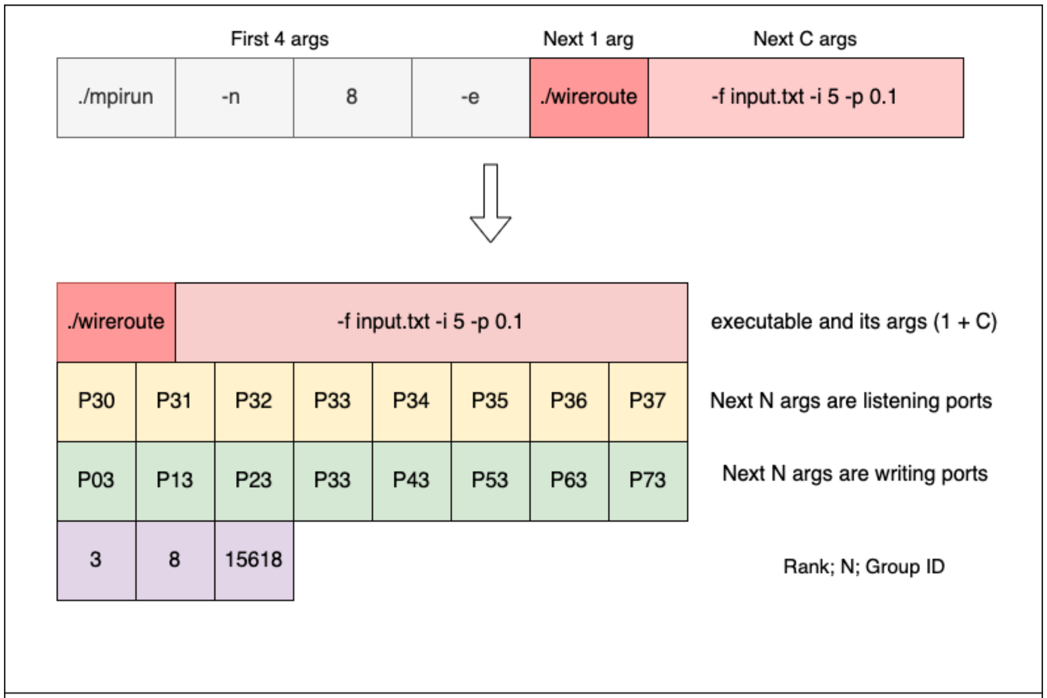
More port troubles - static vs dynamic choice of ports

So, there are `N` ports associated with each process - how do we choose them so that all processes agree on the same set of ports to enable subsequent communication? The simplest solution is a static allocation of ports i.e. each process takes a set of ports that is a simple function of its rank e.g. for 8 processes, process 2 takes ports in the range 2000 to 2008, process 3 takes the range 3000 to 3008 and so on.

This simple approach worked well for small values of `N` in scenarios where most ports on the node were unoccupied. But as soon as we run our application on busy nodes, we run into the issue that many of these pre-decided port strings would be taken. Therefore, ports cannot be statically set.

One possible approach might be for `MPI_Init` to run some kind of consensus protocol to decide every participating process' port addresses. When MPI is being run across different physical nodes, this is the only way. But in our simplified settings, there is an easier approach. We delegate this responsibility to the `mpirun` executable that is responsible for spawning processes. It iterates through all possible ports starting from an offset, trying to find a set of N^2 free ports that can be used by our application. There are 2*N ports that are relevant to each process - its N accepting ports (1 for each source), and the N (destination) ports it has to connect to while sending data to the respective processes. These are passed to it in the `argv`
array that we discussed previously.

The following figure illustrates how the relevant metadata is packed in the `argv` array. `MPI_Init` parses all these arguments into relevant global variables so that every subsequent send and receive call is appropriately directed to the right port address.


First 4 args
Next 1 arg
Next C args
./mpirun
-n
8
-e
./wireroute
-f input.txt -i 5 -p 0.1
./wireroute
-f input.txt -i 5 -p 0.1
executable and its args (1 + C)
P30 P31 P32 P33 P34 P35 P36 P37
Next N args are listening ports
P03 P13 P23 P33 P43 P53 P63 P73
Next N args are writing ports
3 8 15618
Rank; N; Group ID

Argument Packing : 2*N ports for sending and receiving, and other relevant metadata added

MPI_Finalize and graceful exit

The above figure shows that we pass an additional piece of information, the ‘group ID’ inside `argv`. The following explains why that is needed for a graceful exit.

`MPI_Init` opens `N` different ports for listening to connection requests. These need to be closed. MPI mandates that all message passing must happen between `MPI_Init` and `MPI_Finalize`. Therefore, all open ports opened by a process can be safely closed in MPI_Finalize.

In the event that the process is killed by the user, N orphan child processes. These processes continue to run and occupy opened ports and this adversely affects the functioning of the other processes running on the node. Our solution to this problem is to have a new signal handler that is invoked for SIGINT (Ctrl + C), SIGTSTP (Ctrl + Z). This signal handler is registered via `sigaction` at the time of `MPI_Init`. We block the delivery of all signals before `MPI_Init` completes - so that the correct handler function is invoked. Killing the parent process invokes our handler, which sends a SIGINT to all participating processes and proceeds to call MPI_Finalize.

To send SIGTERM to all the child processes, we make sure all the spawned process belong to same process group so that the signal handler can send a SIGTERM to this group. The process group ID is passed to all the processes as an argument in `argv`.

Broadcast, Scatter and Gather

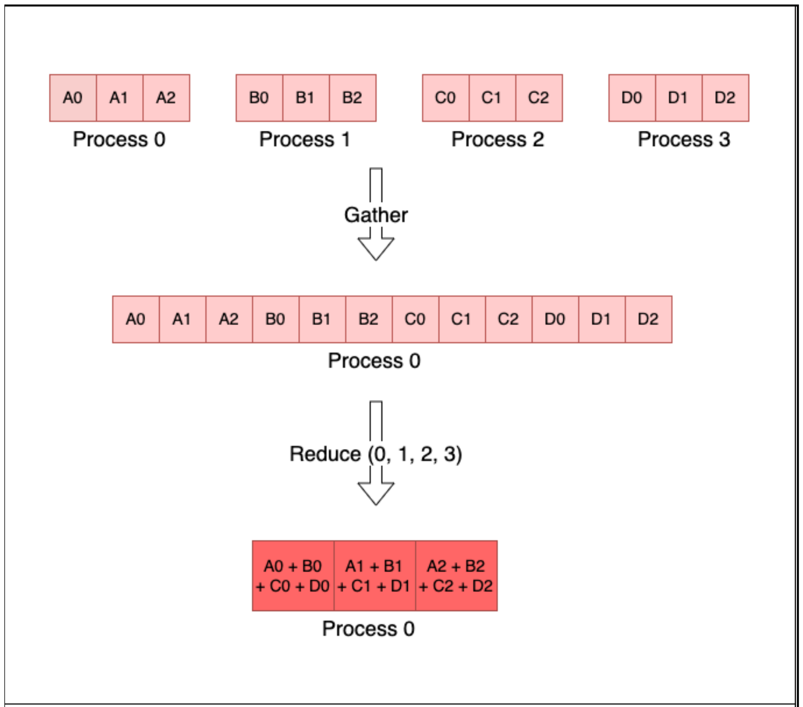
After point to point communication is successfully implemented, the collective communication protocols can be implemented by composing MPI_Send and MPI_Recv calls.
MPI_Bcast involves the root making (N-1) calls to MPI_Send, while all the other processes call MPI_Recv. Conversely, MPI_Gather involves the root making (N-1) calls to MPI_Recv while all the other processes call MPI_Send. Both of these operations can be sped up through the use of threads viz. for MPI_Gather, (N-1) threads are spawned each receiving from one of the (N-1) non-root processes.

MPI_Scatter is similar to MPI_Bcast, except the root process sends a different chunk of data to different processes depending on the process rank. All of these functions involve ingenious use of pointer arithmetic to ensure no unnecessary memory allocation or copying takes place.

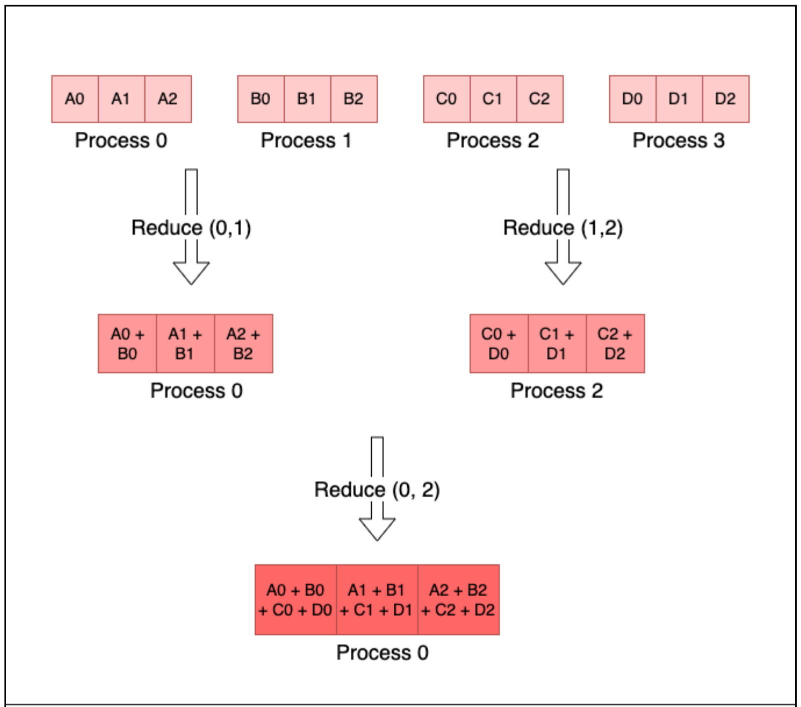
Reduce

MPI_Reduce involves reduction across processes e.g. 4 processes have a size 100 array, the result of all of them participating in MPI_Reduce is that the root has the elementwise sum of all the 4 arrays. For the sake of completion and compatibility with the original API, we implement 6 reduction operations and support reduction across all data types as long as they are compatible with the specified reduction operator.

There are many strategies for reduction - we explore two of them. The first strategy which we call ‘gather-based reduction’ involves a two step process. The root gathers all the data in an extended buffer. Then accumulates the corresponding elements from different processes using the reduction operation specified. The second strategy is referred to as ‘tree-shaped reduction’. It involves distributing the reduction computation across all the processes and parallelizing communication and computation. The two strategies are illustrated by the figures below.



Gather-based reduction



Tree-shaped reduction

The first strategy involves a call to gather and collective communication involves synchronization between all processes. This is why we expect it to have high overheads and be slower than tree-shaped reduction which involves point to point that happens in parallel. The gains from this will be compounded for arrays of large size especially when we are reducing across a large number of processes.

Experimental results and analysis

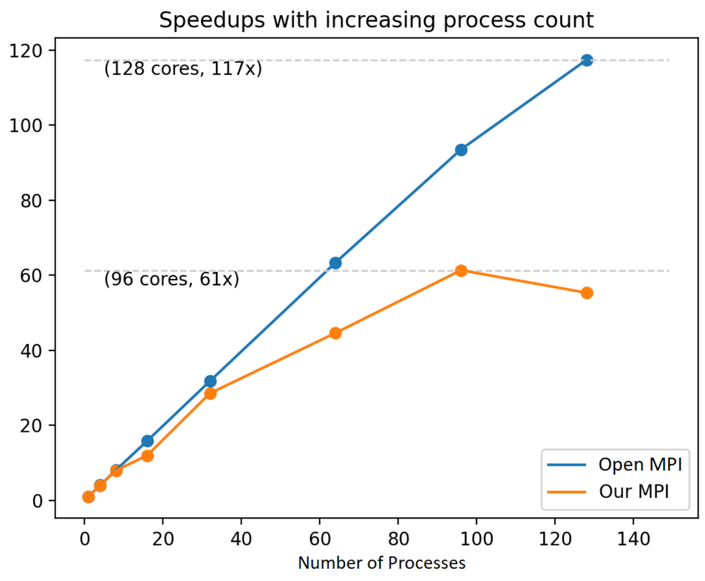
We conduct a series of experiments to analyze the performance of our system for different workloads.

Note : We compare our performance against "Open MPI" in experiments I, II and III. These are not apples-to-apples comparisons for the primary reason that when "actual" MPI is run on a single physical node it defaults to using shared memory for communication since this has far smaller overheads. Our attempts to get it to use TCP sockets did not bear fruit. Nevertheless we report these numbers to convey how well our library performs against the most efficient version of the off-the-shelf solution.

Experiment I - Averaging

We consider the simple problem of summing over the entries of a huge array. The serial version of the code iterates over all entries of the array, adding them and dividing by the size of the array. For the parallel version, each process works on its own chunk, computes the chunk average and sends it over to the root process. The root averages the chunk averages to compute the final average. Note that, all initializations are random and the root doesn't have to communicate the initial value of the array. In summary, the workload is computation-intensive with only one instance of communication at the end of local processing making it “embarrassingly parallel”.

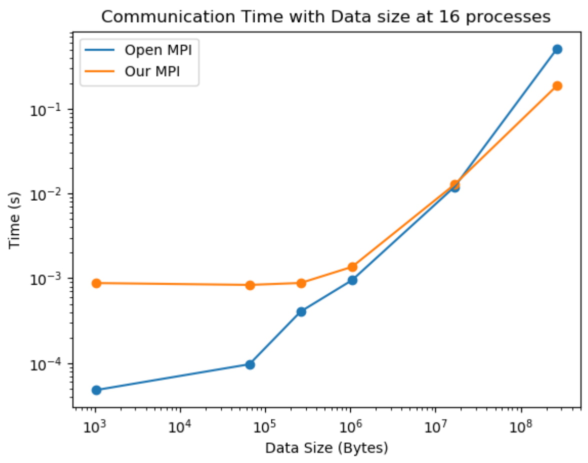
Our MPI library achieves very impressive speedups as shown in the graph below. We get a nearly perfect speedup of 28x when we use 32 cores. Our speedups increase sublinearly from 32 cores to 96 cores to give a maximum speedup of 61x on 96 cores. In comparison, the actual MPI library stays very close to the theoretical maximum speedup throughout, achieving 117x speedup for 128 cores.

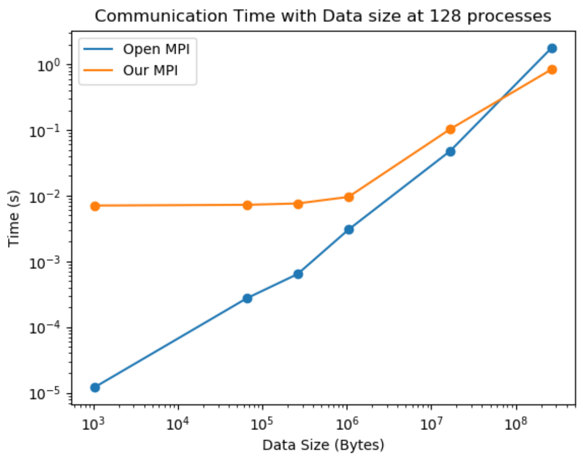


This illustrates that although our library enables our application programs to scale with an increasing number of cores, we start to encounter greater communication overheads with increasing process counts and start to diverge from the Open MPI library’s performance.

Experiment II - Communication scaling with data size

To further analyze the reason behind difference in the speedups between our MPI implementation and the Open MPI, we developed a benchmark to compare the time taken for communication with increasing size of data and with different process counts. In our benchmark, we chose to broadcast data from a root process to the rest of the processes. The datasize was varied from 2^{10} bytes to 2^{28} bytes. The experiment was run with 16 processes and 128 processes. In both of these process counts, it was observed that our implementation had a much higher communication time with smaller data sizes while at higher data sizes, it performed better than Open MPI. This can be explained by the fact that Open MPI chooses the most efficient and best available communication method on a node.

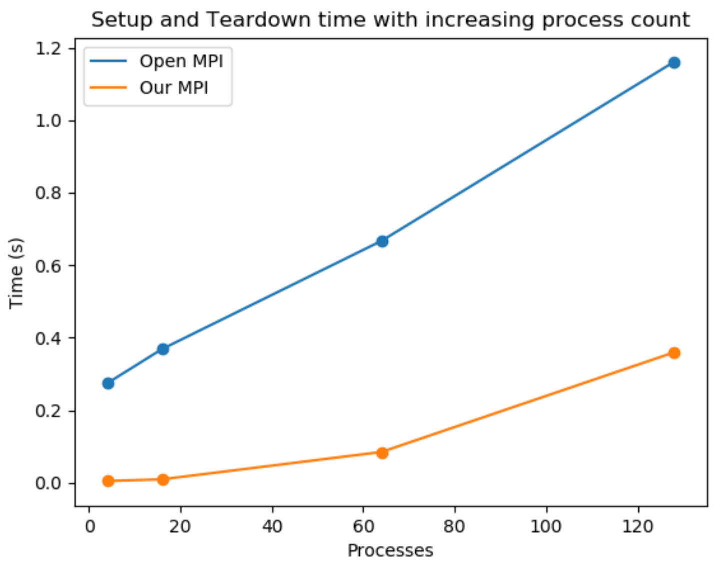




The results from this experiment suggest that with the socket implementation, majority of the time is spent in the socket handshake protocol rather than the data communication itself especially when the size of the data transferred is small.

Experiment III - Setup and teardown time

In this experiment, we compare the overall time taken by a program to perform the logistical tasks of MPI - setting up the ranks and cleaning up before exit. To perform this experiment, we used the timer provided by the `perf stat` tool on the respective mpirun executable. The code does not perform any communication or useful operations. To increase the consistent timing observations, a dead loop is performed for 100 million times.



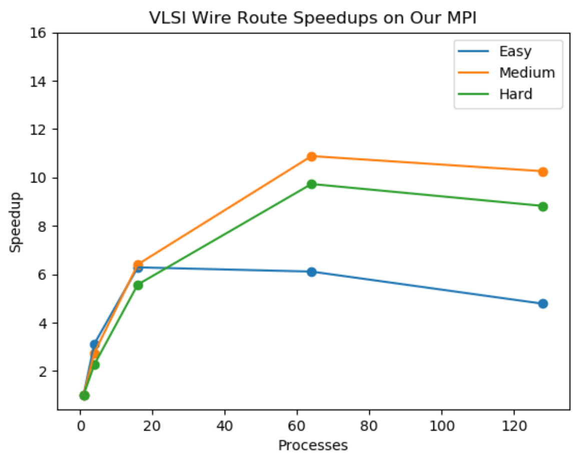
From the data obtained, it is clear that our setup and teardown time is much lesser than the Open MPI implementation. This trend could be attributed to our simpler implementation of MPI.

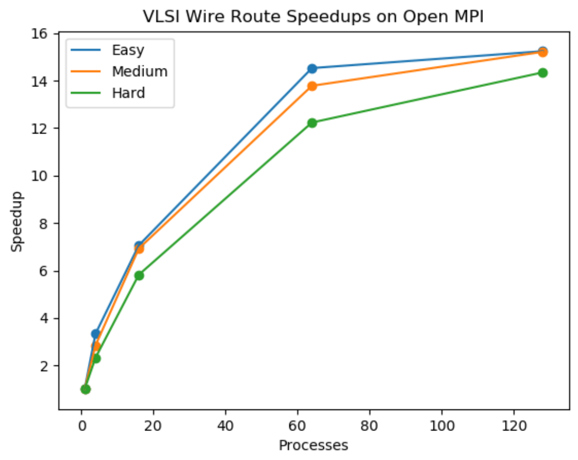
Experiment IV - Routing Wires

Here, we used our Assignment 4 program for VLSI wire routing and compared the speedups between our MPI and Open MPI for increasing process counts. Note that our algorithm for this problem increases the number of messages passed with increasing process count to maintain high accuracy of the routing. Additionally, the size of the messages also decreases.

We achieve a maximum speedup of 10.88x at 64 processes for the routing problem of medium difficulty. We note that this is lower than the speedup we are able to achieve with the “actual” MPI library which is 15.21x at 128 processes.

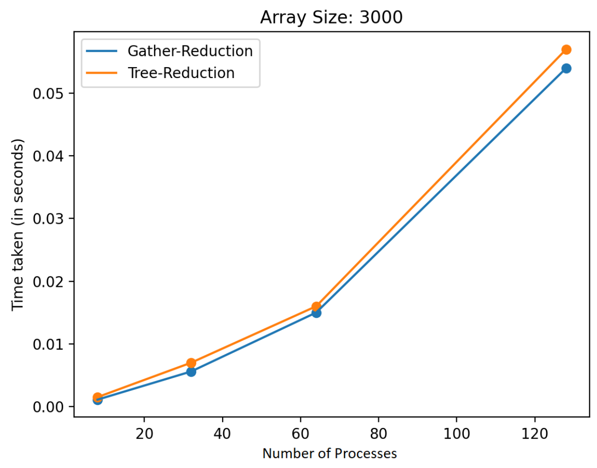
It can be seen from the graph that there is a general increase in speedups with increasing process counts (upto 64 processes). Our performance on lower process counts (2 to 8) rivals that achieved using Open MPI (not shown in the graph). But, we also see that the speedup drops (sometimes sharply) when we go from 64 processes to 128 processes.

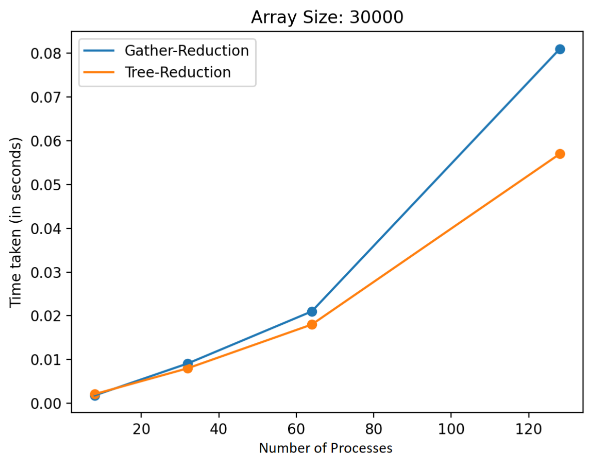


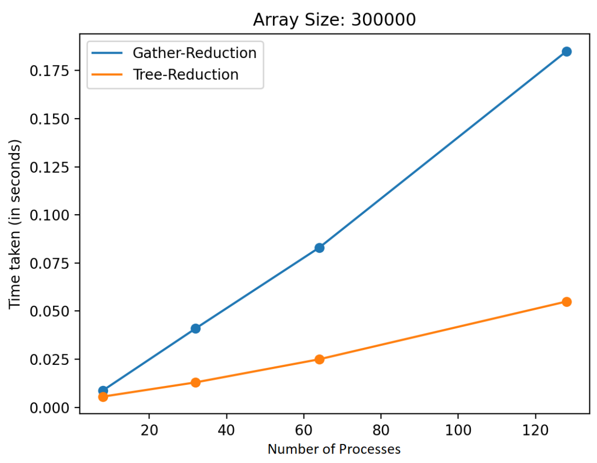


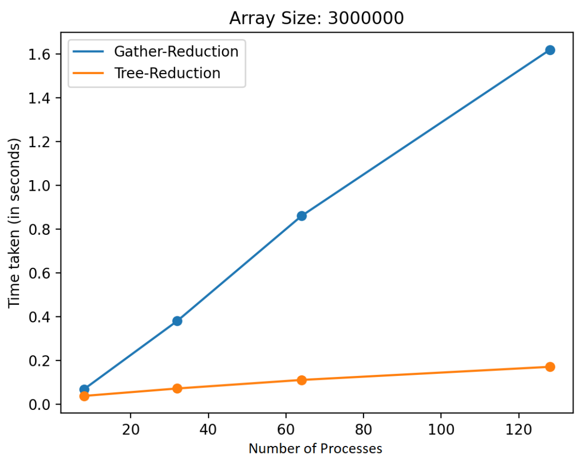
Observations from Experiment I, II and IV suggest that our collective communication scales poorly with increasing process counts after 64 because establishing connection between the processes is more expensive operation in our implementation.

Experiment V - Reduction









The observations here closely mirror our hypothesis from the previous section. For arrays of small size, tree-shaped reduction performs at par or is slightly slower than the gather-shaped reduction. But as the size of the array being reduced increases the gap between gather-based reduction and tree-shaped reduction widens. For the largest array with 3 million entries and 128 processes, we observe a gain of almost 10x from tree-shaped reduction.

The fact that the gain increases with array size is because the amount of sequential computation performed by the root process of the gather-based version increases with array size, this computation happens parallely at different processes in the tree-shaped version. The gain increases with number of processes because the root of the gather-based version can only connect to one process at a time - the communication time serially adds up in this case as opposed to being parallelized in the case of tree-shaped reduction.

References

- Bryant, R. E., & O'Hallaron, D. R. (2016). "*Computer systems: A programmer's perspective*."

  We referred to functions for opening sockets, connecting to sockets and reading and writing safely to socket file descriptors. This has also been used in the starter code for "Proxy Lab" of CMU's "15-213 Introduction to Computer Systems" course.
- MPICH documentation https://www.mpich.org/static/docs
- OpenMPI discussion forums
- S. Pellegrini, et al., "A Lightweight C++ Interface to MPI," 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2012, pp. 3-10
- S. Ghosh, et al., "Towards Modern C++ Language Support for MPI," 2021 Workshop on Exascale MPI (ExaMPI), 2021, pp. 27-35

Individual Contributions

Credit for this work should be evenly split between both team members. All the code written for the project was written over a dozen pair programming sessions with both members alternating between being the *driver* and *navigator*. We rarely, if ever, worked in isolation.

Acknowledgement

We would like to thank our mentor for a fruitful design discussion at the beginning of our project. We would also like to thank Prof. Beckmann for his guidance and nudges towards the right direction whenever we reached out to him with questions during the course of this project.