Simulating MPI execution on IITK CSE Cluster

Yash Srivastav (150839)

I. Problem Description

MPI programs might behave very differently (in terms of speed) when run on different host nodes due to the different network arrangement of the nodes. One might want to test out various different node assignments so as to check out the impact on data transfer rates / congestion. The target machines should easily be changeable without having to go through the whole runtime and network transfers again. Also a simulation might give a better idea independent of real world factors like load on the CPUs.

II. Related Work

A number of works have explored this problem before in order to come up with solutions to efficiently solve it. There are 2 major ways of tackling this problem: in-situ (online) simulation and post-mortem (offline) simulation. In-situ simulators simulate the actual running of the program and while running, perform necessary calculations / simulations. Port-mortem simulators in general collect traces during a sample program execution and then simulate the behaviour later on based on the collected traces.

[1] performs in-situ simulation by hooking onto the MPI network layer and then sending exact packet transfers to network simulator running simultaneously. Also somehow the computation time itself has to be simulated while running which requires applying a uniform slowdown to account for latency introduced due to the network simulator. A benefit of using this method is that the implemented simulator only needs to simulate packets and doesn't need to worry about internal algorithms being used.

[2] and [3] perform post-mortem simulations. They both collect traces of `MPI` events occurring during the execution and then simulate the behaviour based on the collected logs. A benefit of using this method is that long computations can be avoided during simulation as only the expected delay needs to be noted and not really executed.

[4] and [5] follow a hybrid approach and simulate on a complete reimplementation of the `MPI` Standard on the **SimGrid** simulation kernel.

The offline approaches employ either simpler network models or use older network simulation frameworks like **OmNet++**.

III. Methodology

The method we apply in this paper is an offline approach similar to [2]. The program is run with the `PMPI` interface active so as to obtain event traces of `MPI` events. Also **P**erformance **API** (PAPI) is used to measure the computation cycles between successive `MPI` events. This effectively gives us a log of events which need to be simulated. Something like the following:

```
[0] compute 18288
[0] send 1024 to 1
[1] compute 3108
[1] recv 1024 from 0
[1] compute 29708
[0] compute 127451
[0] wrapping up
[1] wrapping up
```

With the above information, its possible to actually simulate these events now on a network simulator. For this paper, we use the `ns3` simulation framework [6]. Each ranked process is treated as an application running on a node in the simulator and then simulates itself accordingly. At the end, we can obtain the total time taken till all applications reached end of execution as the simulation time.

IV. Implementation Details

There are two main components to the simulation described in this paper: The **Profiler (Trace Collector)** and the **Simulator**.

A. Profiler (Trace Collector)

The profiler is a pretty simple PMPI wrapper which allows adding hooks to every MPI function call before and after execution. Every process in an execution writes to a log file named `simpi-$rank.log`. At the start of every MPI function, the current instruction count is recorded and a difference from the previous value is taken and logged. This gives a fair idea of computation between 2 MPI function calls. This can be used as a measure of delay to be introduced while simulating. Also every MPI function call itself is logged via the important params being logged. Right now, the following functions are being traced and observed:

- `MPI_Init`: Also used to initialize profiler variables like logfile file pointer.
- `MPI_Finalize`: Close file pointer and any other wrapping up needed.
- `MPI_Send`
- `MPI_Recv`
- `MPI_Scatter`
- `MPI_Gather`
- `MPI_Bcast` (short message algorithm only)

The implementation only supports a single communicator `MPI_COMM_WORLD`. Its possible to add support for communicators by logging the respective calls and then performing

computation regarding which nodes receive which message based on that.

The implementation doesn't care about the result of any of the calls and only logs the called params. If needed, we could log the results as well.

The profiler compiles to a dynamic library and which can then be specified in `LD_PRELOAD` and then any MPI program executed would be profiled and traced as needed. Sample trace collection:

```
$ export LD_PRELOAD=$HOME/papi-install/lib
    ↪ /libpapi.so:$HOME/mpich/lib/libsimpi
    ↪ .so
$ mpirun -np 2 ./asgn1-1.x 1
[0] compute 18304
[1] compute 3124
[0] send 1024 to 1
[1] recv 1024 from 0
[1] compute 29665
[0] compute 177282
[0] wrapping up
[1] wrapping up
Runtime = 0.000067
$ cat *.log
0 1 18305
0 3 1024 1
0 1 31557
1 1 3124
1 2 1024 0
1 1 29664
```

B. Simulator

The simulator consists of various different modules:

- Parser
- Topology Generator
- `MPINode`

Sample simulator run:

```
$ ./run-simulator.sh 16 ./test/test-
    ↪ hostfile ./test/test-logs-16
+0.023341061s
```

C. Topology Generator

Generates a topology exactly as per the CSE Cluster as shown in Figure 1. Every link is a 1Gbps Connection with CSMA. The two switches are actually `ns3::Bridge` in ns3 terminology connected to each other via a link having the same specifications.

D. Parser

The parser parses event logs generated by the tracer and creates a rank to events mapping. Every event in this mapping consists only of the following events:

- Compute
- Recv

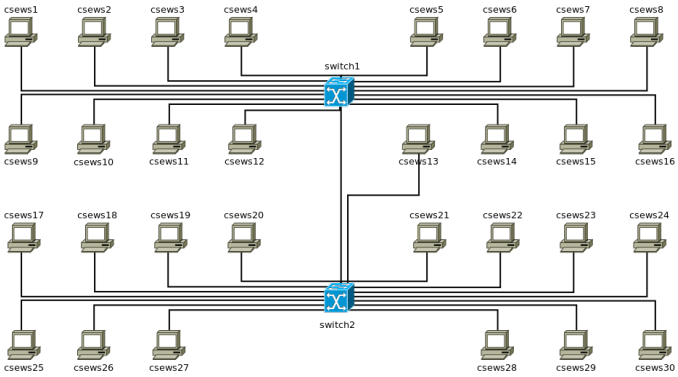

Fig. 1. CSE Cluster Topology

- Send

All other traced events are converted to a mixture of these as per the respective algorithms implemented in MPICH 3.2.1.

E. MPINode

An `ns3::Application` which is essentially a TCP server + client. Runs on a host node as per host allocation. Takes as initialization parameters the following values:

- Rank (`uint16_t`): The rank of this process as per the global rank allocation. Used to determine host allocation of the process.
- Events (`vector<simpi_event_tagged_t>`): A list of events supposed to be simulated on this ranked process. `simpi_event_tagged_t` is custom defined type which can be either a **Compute** event or a **Recv** event or a **Send** event. These are the only three events supported by the `MPINode` itself.
- Addresses (`vector<Address>`): A list of addresses of every host on which the `MPI` program is being run.

Events are processed sequentially as soon as the previous one finishes. Once there are no more events left, the application is stopped. Once all applications stop we can claim that the simulation has ended.

If a Compute event is encountered, the application is scheduled to process the next step after a delay corresponding to `num_instructions / MPI_NODE_CPU_IPS`. `MPI_NODE_CPU_IPS` is the number of instructions processed per second by the host CPU. I am using the value of **BogoMIPS** from `lscpu`.

If a Recv event is encountered, the node starts listening on a fixed port calculated from `rank / ppn`. It accepts any connection and keeps reading from the accepted socket till the required size has been read.

If a Send event event is encountered, the node connects to the address and port of the target rank `MPINode`. It sends all possible values via the socket.

While sending/receiving messages, I have assumed a header area in the packet other than the main data. The header is just a field containing necessary data for the packet transfer such as TAG value, etc. The size of the header is determined by `MPI_HEADER_SIZE`.

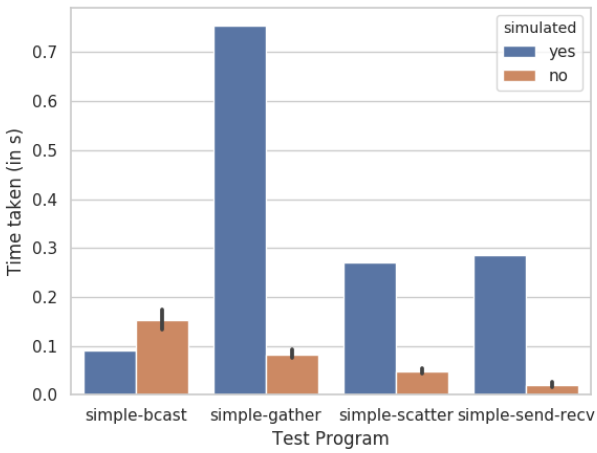

Fig. 2. Simulation vs Actual

While sending/receiving a message from the same network node, we just send a single byte message (for synchronization) to the target rank. A receiver simulates an additional delay of `MPI_COPY_DELAY_US` microseconds for the memory copy.

Right now, due to issues with understanding the socket framework of ns3, there is an issue when there is a lot of contention on a single link.

V. Experimental Results

A. Experimental Setup

The target programs are run 10 times on the CSE Lab Cluster (Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 1Gbps links) and then once on the simulator. The predicted time as per the simulator is compared against the average runtime of the 10 programs. The values of various important variables used during the simulation are given below:

Variable	Value
`MPI_NODE_PPN`	8
`MPI_NODE_LINK_MTU`	1460
`MPI_NODE_CPU_IPS`	6384000000
`MPI_COPY_DELAY_US`	20
`MPI_HEADER_SIZE`	20

All programs are executed with `nprocs=64`.

B. Results

The comparison between actual runtimes and simulated runtime for simple testcases is in Figure 2.

VI. Future Work

The following can be done to improve the current implementation:

- Add support for different communicators.
- Add support for handling `MPI` function errors during tracing and logging them as well for use during simulation.
- Remaining collectives and Async variants of `MPI` functions and associated calls like `MPI_Wait`
- Vector variants of `MPI` collectives (like `MPI_Allgatherv`)
- Allow greater send and receive sizes (fix socket + packet sending issue).
- Comparison with other `MPI` simulation frameworks developed as part of other works.
- Distributed trace collection: The trace collection itself could be run parallelly on multiple machines for faster trace collection and simulation.

VII. Conclusions

The above results show that simulation is a viable option for predicting performance of MPI program. Once most operations are implemented, its possible to use this for actually simulating any MPI program.

References

[1] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler, "Mpi-netsim: A network simulation module for mpi," in *2009 15th International Conference on Parallel and Distributed Systems*, Dec 2009, pp. 464–471.

[2] H. Casanova, F. Desprez, G. S. Markomanolis, and F. Suter, "Simulation of mpi applications with time-independent traces," *Concurr. Comput. : Pract. Exper.*, vol. 27, no. 5, pp. 1145–1168, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1002/cpe.3278

[3] P. Bédaride, A. Degomme, S. Genaud, A. Legrand, G. S. Markomanolis, M. Quinson, M. Stillwell, F. Suter, and B. Videau, "Toward better simulation of mpi applications on ethernet/tcp networks," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Cham: Springer International Publishing, 2014, pp. 158–181.

[4] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. L. Stillwell, and F. Suter, "Simulating MPI applications: the SMPI approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, p. 14, Aug. 2017. [Online]. Available: https://hal.inria.fr/hal-01415484

[5] P. Clauss, M. Stillwell, S. Genaud, F. Suter, H. Casanova, and M. Quinson, "Single node on-line simulation of mpi applications with smpi," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 664–675.

[6] G. Carneiro, "Ns-3: Network simulator 3," in *UTM Lab Meeting April*, vol. 20, 2010, pp. 4–5.