Project for ARCHITECTURAL CONCEPTS II 2019 – New Mexico State University

MPI Communication Bottleneck

Fabio Tardivo
New Mexico State University
Las Cruces, NM
ftardivo@nmsu.edu

Paolo Grignaffini
New Mexico State University
Las Cruces, NM
grignaff@nmsu.edu

I. Introduction

While working with HPC clusters there is often the necessity to retrieve information about the communication between the computational nodes in order to identify potential bottlenecks [1]. This work presents a tool to calculate the communication statistics, such as messages' size, source, and destination, while providing the user with a GUI to render these information.

II. Motivation

In distributed applications running on cluster the main cause of slowdowns is often due to bottlenecks inside the communication.
Specifically, it is hard to predict the communication behavior since it depends on the input data. An example is the research of a gene inside the DNA [2]. In this case, the DNA is divided between all the computational nodes, and the research proceeds in parallel on each node. When the gene is found it is possible to adopt two different approaches: (i) wait for all the nodes to finish their research, or (ii) send a broadcast message to inform all the nodes that the gene has been already identified and thus there is no need for them to finish running their processes.
Which approach is better depends on the communication latency, that in turn, depends on the network configuration. In some cases could be faster to wait the termination of each node, in other cases could be more convenient to send a message to stop the research. This tool enables the user to easily discriminate which approach can bring to faster results by providing a picture of the network status.
Nowadays frameworks to visualize this type of data and much more already exist, but they often require a purchase license to be utilized [1]. This project offers an open-source lightweight tool to provide the user with information regarding the network communication during the execution of distributed programs.

III. Design and Implementation

The main goal while designing the software architecture was to realize a fast and easily usable tool.
For this reason, an implementation choice of not using launchers and/or auxiliary scripts to help analyzing the program has been made. This decision has increased the code's complexity but at the same time has provided some benefits, such as:

- Keeping the configuration files unchanged to submit the program in a job scheduler
- Avoiding problems due to different versions of the scripting's language

The choice of the output's data format has been made following the idea that such information should be easily readable both by the user and the GUI. The first requirement constrains the decision to textual files, while the second one demands the usage of a structured format. Thus, the logical choice has been to use JSON [3], an open-standard file format that uses human-readable text that can be easily parsed by the user interface.
Since it is in the user's interest to visualize the communication for each pair of nodes in the network, the data have been encoded in a matrix format. Each matrix has dimension $n \times n$, where n represents the number of the nodes in the cluster. An entry in position n_{ij} reports the data transmission between the node i and the node j (Fig. 1). The different matrices contained in the JSON file show information of different nature.
In the following list are reported all the metrics that have been used, because considered the most significant, to embody the network's status:

`size`: number of nodes in the network
`m_recv_count`: number of messages received by the nodes
`m_recv_size`: average dimension of the messages received by the nodes
`m_send_count`: number of messages sent by the nodes
`m_send_size`: average dimension of the messages sent by the nodes
`p_recv_count`: number of packets received by the network's interface
`p_recv_size`: average dimension of the packets received by the network's interface
`p_recv_err`: number of errors reported by the network's interface regarding the packets' reception
`p_send_count`: number of packets sent by the network's interface
`p_send_size`: average dimension of the packets sent by the network's interface
`p_send_err`: number of errors reported by the network's interface regarding the packets' dispatch

```
"m_recv_count":
[
    [ 0, 0, 0, 0, 0, 0 ],
    [ 1, 0, 1, 1, 1, 1 ],
    [ 2, 2, 0, 2, 2, 2 ],
    [ 3, 3, 3, 0, 3, 3 ],
    [ 4, 4, 4, 4, 0, 4 ],
    [ 5, 5, 5, 5, 5, 0 ]
],
"m_recv_size":
[
    [ 0, 0, 0, 0, 0, 0 ],
    [ 40, 0, 40, 40, 40, 40 ],
    [ 40, 40, 0, 40, 40, 40 ],
    [ 40, 40, 40, 0, 40, 40 ],
    [ 40, 40, 40, 40, 0, 40 ],
    [ 40, 40, 40, 40, 40, 0 ]
]
```

Fig. 1. Example of data transmission statistics saved in matrix form.

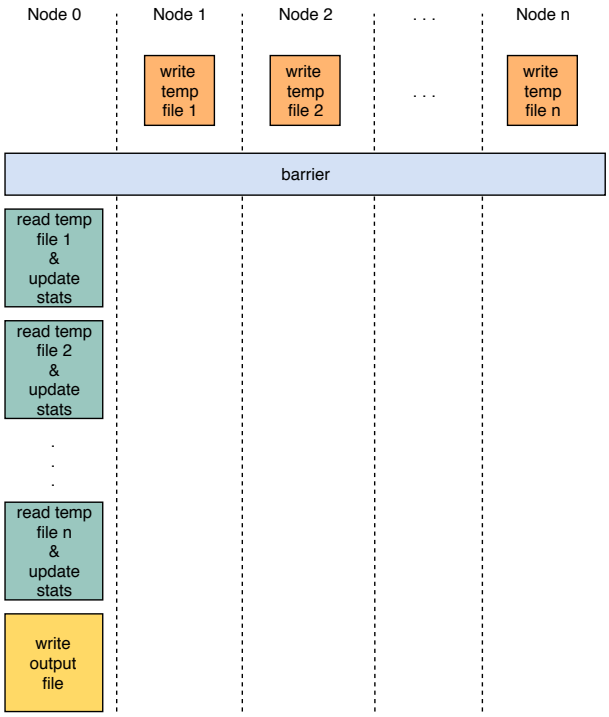

Fig. 2. Logic of the `MPI_Finalize` reimplementation.

A. PMPI

In order to collect the communication information, the profiling interface defined inside the MPI standard [4] (i.e. PMPI) has been used. Its functioning is really similar to function's overloading:

1) Given an MPI function (e.g `MPI_Send`), create a function with the same name and parameters.
2) Insert in the function's body the code needed for the profiling.
3) Instead of calling the original function (e.g. `MPI_Send`), invoke its profiling counterpart by prepending the letter "P" to the original function's name (e.g. `PMPI_Send`).

Inside the redefinition of the initialization's call (i.e. `MPI_Init`) the matrices are created and initialized. These matrices must be accessible from any point of the code where either a *send* or a *receive* instruction is performed, thus the choice to make them as global accessible. In order to avoid possible interferences with other variables inside the application, the matrices have been inserted into a *namespace*.

The redefinition of the *send* and *receive* functions (i.e. `MPI_Send`, `MPI_Recv`) and their respective asynchronous counterparts (i.e. `MPI_Isend`, `MPI_Irecv`) is relatively easy since it only needs to increment the correct values inside the matrices.

The redefinition of the finalization's function (i.e. `MPI_Finalize`) is depicted in Figure 2. Each node, except for the first one, saves its internal statistics in a temporary file and it calls a synchronization barrier. The first node calls the synchronization barrier and waits for all the nodes to reach it. Once the barrier is overtaken all the nodes terminate, except for the first one that before termination reads the temporary files, merges the results and create the output file.

The computation of the statistics relative to the network's interface has been harder than expected.

The first approach utilized and then discarded was to read the pseudo-file `/proc/net/dev` [5], which contains the network device status information such as the number of received and sent packets [6].

Inside the cluster where the experiments have been conducted the aforementioned file gets updated only after the program's termination, making impossible to analyze the values' variations during the execution. In order to avoid the utilization of scripts/launcher an approach based on the syscall `fork()` has been used. In this previous implementation, the child process reads the updated values once the parent process terminates. At first this method seemed to work, but further testing showed that such mechanism was not reliable enough since the update of the data was not always immediate. Furthermore, the addition of a 5–10 seconds delay was not sufficient to mitigate the problem, probably due to the fact that `/proc/net/dev` is not a real file but an interface for kernel-handled statistics. As opposed to the implementation described in the above paragraph, the implemented solution does not require the creation of further processes and it works by reading the values of the counters provided in: `/sys/class/infiniband/<network interface>/ports/1/counters/`. Unfortunately, MPI does not offer any information regarding the network's interface in usage, and because of that this solution does not provide a general method to identify it. Therefore this task remains an operation to be executed by hand and it can be accomplished by changing the definition of the macro `NETWORK_INTERFACE` inside the source file.

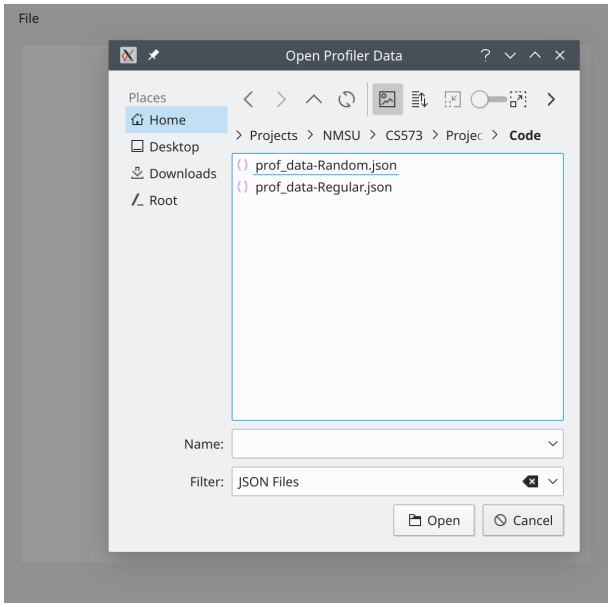

Fig. 3. File chooser dialog

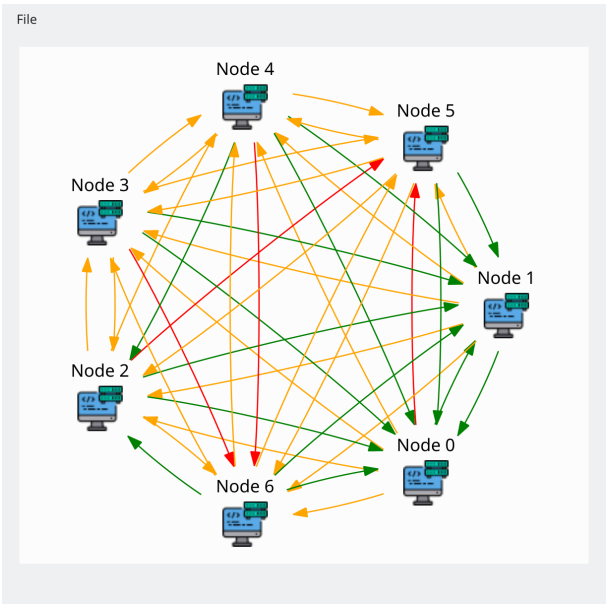

Fig. 4. Main window of the Graphical User Interface

B. GUI

In order to make the tool as easy to use as possible, the computed statistics are showed through a GUI.

The starting idea was to have a program that, once read the JSON file, was able to generate a graph through the `dot` application provided by the GraphViz package [7].

For the GUI realization, the initial choice was to use Qt [8] and its Python bindings (i.e. PyQt) to display the network on-screen, given the apparent ease of its functioning. Subsequently, the lack of support to such library has caused the project to shift to the C++ language.

The Qt-based project QGV [9] has been discovered during the documentation phase performed to get a better understanding of the GraphViz interaction with the software, and it has been of great help for the implementation. This program was already implementing most of the functionalities needed by this work, and thus its source code has been adopted as a baseline for the interface, which has been expanded to obtain the desired behavior.

For what concerns the management of the JSON file, both in the GUI and in the profiler, the Json library [10] has been chosen amongst the multiple others available on-line because of its excellent integration with the C++ standard library and because of its intrinsic easiness, which makes parsing a JSON file as simple as reading data from an array. Furthermore, in order to use this library is sufficient to add a single header file inside the project.

Once the GUI is invoked, the interface allow the user to choose which profiling information file display on screen, as depicted in Figure 3. Once the profiling information file is opened by the program, the matrices are parsed and the data for each node are collected inside the relative node's object. The network is then displayed through a fully-connected graph representation, as depicted in Figure 4.

The color of the arrows represents the status of the communication between each node, where a red color asserts that the numbers of packets sent by a node is above a given $threshold_M$, to indicate that the communication could suffer from bottlenecks, while a green color asserts that the number of packets sent is below a given $threshold_m$, to indicate that the connection is not likely to affect the performance of the application. Every communication that uses a number of packets inside the range given by $[threshold_m, threshold_M]$ is displayed in yellow, to represent a situation in the middle of the previous two.

The values for $threshold_M$ and $treshold_m$ are calculated through eq. 1, 2:

$$threshold_M = (usage_M - usage_m)\frac{4}{5} \tag{1}$$

$$threshold_m = (usage_M - usage_m)\frac{1}{5} \tag{2}$$

where:

$$usage_M = \max_{0 \le j \le i < n} \{send\_size_{ij} * send\_count_{ij}\} \tag{3}$$

$$usage_m = \min_{0 \le j \le i < n} \{send\_size_{ij} * send\_count_{ij}\} \tag{4}$$

IV. Experiments and Results

The tests have been performed inside the cluster "Rocks" freely available to use for students in the Computer Science department of NMSU [1]. The cluster is composed of 7 computational nodes connected through infiniband.Each node is equipped with:

[1]The "Rocks" cluster was preferred over "BigData" or "Discovery" since on them the waiting time was bigger than the testing time.

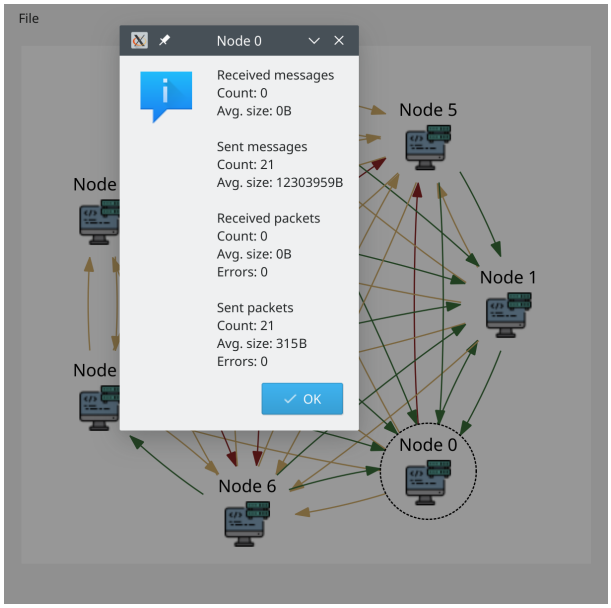

Fig. 5. Node's detailed network statistics.

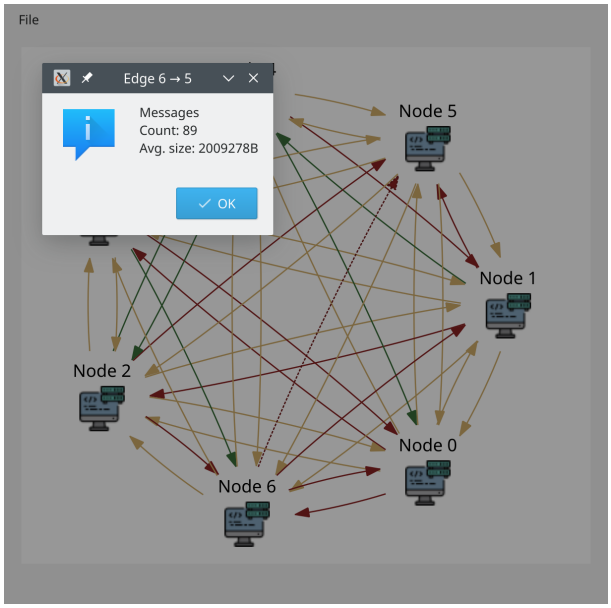

Fig. 6. Link's detailed network statistics.

CPU: AMD Opteron 6376, 1.4GHz, 16 cores
RAM: 128GB
SO: Rocks Linux (CentOS based)
Kernel: 3.10
MPI: OpenMPI 1.10 (MPI version 3.0)

The source code is paired with a makefile that allows to compile the profiling library and a couple of test programs as follow:

```
$ make all

mpic++ -std=c++11 -shared -fPIC Lib.cpp
-o "libMpiSimpleProfiler.so"

mpic++ -std=c++11 -L ... -o "testRandomMSP"
TestRandom.cpp -l"MpiSimpleProfiler"

mpic++ -std=c++11 -L ... -o "testRegularMSP"
TestRegular.cpp -l"MpiSimpleProfiler"
```

The "regular" test consists of a distribuited program where each node i sends j times a 4MB message to the node j if $i \neq j$. For example, node 2 will send 0 messages to node 0, 1 message to node 1, 3 messages to node 3, 4 messages to node 4, etc. The resulting statistics are presented in Figure 5.

The "random" test consists of a distribuited program where each node sends a random number of packets between 1 and 100 of a random size between 4B and 4MB. The resulting statistics are presented in Figure 6.

V. Conclusion and Future Works

In conclusion, it can be asserted that the project meets the goals that have been set in the proposal, namely to implement a simple profiler to enable the developer to quickly identify potential communication bottlenecks inside applications that make use of the standard MPI.

The gathered metrics include both high-level information (e.g number of sent messages) and low-level information (e.g. number of packets received), in order to represent the communication in the most reliable manner.

Furthermore, the proposed work offers the opportunity to consult the data both through a textual format and through a GUI, in order to cover the two major development workflows: by IDE (graphic environment), and by remote shell (without graphic environment).

In the future, this profiler could be expanded in different ways:

- Real-time statistics: By adding to the GUI the possibility to connect to the cluster and by modifying the profiler to transmit data via socket instead of via file, it is possible to update the interface in real time
- Timeline analysis: By recording the starting and finishing time of each MPI call is possible to draw a timeline to represent the different phases of the computation during the communication

References

[1] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "Vampir: Visualization and analysis of mpi resources," 1996.

[2] C. Del Fabbro, F. Tardivo, and A. Policriti, "A parallel algorithm for the best k-mismatches alignment problem," in *2014 22nd euromicro international conference on parallel, distributed, and network-based processing*, pp. 586–589, IEEE, 2014.

[3] Wikipedia, "JSON — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=JSON&oldid=895923828, 2019. [Online; accessed 08-May-2019].

[4] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, (New York, NY, USA), pp. 878–883, ACM, 1993.

[5] Wikipedia, "Synthetic file system — Wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Synthetic%20file%20system&oldid=873668526, 2019. [Online; accessed 08-May-2019].

[6] *proc(5) Linux User's Manual*, July 2013.

[7] A. Research, "Graphviz - graph visualization software," 2008.

[8] Nokia Corp., "Qt : cross-platform application and ui framework," 2012.

[9] B. Nicolas, “Interactive qt graphviz display.” https://github.com/nbergont/qgv. [Online; accessed 08-May-2019].
[10] N. Lohmann, “Json for modern c++.” https://github.com/nlohmann/json. [Online; accessed 08-May-2019].