Improving MPI Application Communication Time with an Introspection Monitoring Library

Emmanuel Jeannot
Inria, LaBRI, Univ. Bordeaux
Talence, France
emmanuel.jeannot@inria.fr

Richard Sartori
Inria, Bordeaux-INP
Talence, France
richard.sartori@inria.fr

***Abstract*—In this paper we describe how to improve communication time of MPI parallel applications with the use of a library that enables to monitor MPI applications and allows for introspection (the program itself can query the state of the monitoring system). Based on previous work, this library is able to see how collective communications are decomposed into point-to-point messages. It also features monitoring sessions that allow suspending and restarting the monitoring, limiting it to specific portions of the code. Experiments show that the monitoring overhead is very small and that the proposed features allow for dynamic and efficient rank reordering enabling up to 2-time reduction of communication parts of some program.**

***Index Terms*—MPI, monitoring, communication optimization, HPC**

I. Introduction

To cope with application requirements in terms of speed, precision or memory, the use of supercomputers has emerged as a dominant solution. To program parallel applications onto distributed memory systems, MPI (Message Passing Interface) is the de facto standard. MPI defines how distributed processes exchange data through point-to-point messages as well as collective or one-sided communications.

Being able to write parallel applications whose performance is close to the peak of the target machine is still a very difficult challenge. It requires to design an efficient parallel algorithm, to optimize data structure, to cope with load imbalance, etc. One of the main problems is the way data are exchanged, and allocated. This is of tremendous importance for the overall application performance as when an application (weakly) scales it spends more time in communication. Hence, among all the difficulties, understanding how the processes of the application communicate and coping with data locality is key. To do so, it requires to be able to monitor the application behavior and take decision (at launch time or at runtime) on the process mapping, the communicator composition, and the way communications are executed.

Monitoring MPI applications has been proposed in many tools [14], [16], [18]. However, these monitoring tools are designed for performance analysis, performance debugging and post-morten analysis. Having a monitoring tool to perform runtime optimization requires it to be able to perform introspection during execution. This means that this tool should be able to query the state of the monitoring during execution. In [5], a subset of the authors of this paper proposed a low-level MPI monitoring component for Open MPI [9]. Such a component is based on MPI Tool Information Interface which is a low-level API of MPI to query performance variables of the MPI runtime systems. For a given MPI Process, such interface is able to gather the number of messages and the amount of data exchanged with other ranks. However, it is very low level and requires a deep understanding of the MPI Tool Information Interface. In this paper, the main contribution is that we leverage on this previous work to provide a higher-level and more abstracted introspection library for MPI. This new library enables the application to query its specific state through a simple API during the execution. In particular the proposed library features the notion of monitoring sessions which can be suspended and resumed for monitoring only specific part of the code. Different sessions can overlap, monitor specific types of communications (point-to-point, collective or one-sided) and be attached to a specific communicator enabling a precise understanding of the behavior of the application during the execution. It provides a C and Fortran interface. Last, we monitor communication once a collective has been decomposed into its point-to-point messages: this unique feature enables to gather the affinity between processes.

The goal of this paper is to describe the proposed introspection library and to show that thanks to it, it enables to optimize the communication time of parallel MPI applications. We detail the notion of session, its usage and a specific use-case (dynamic rank reordering). We provide extensive experiments to compare it with hardware counters, assess its low overhead, and show how to optimize communications through rank reordering.

This paper is organized as follows. Related work is presented in Section II and the background in Section III. In Section IV we describe in detail the library. The dynamic rank reordering technique and algorithm are depicted in Section V. We present the experimental results in Section VI. We discuss issues and problems in Section VII and we give our concluding remarks in Section VIII.

II. Related Work

Monitoring an MPI application can be achieved in many ways but in general relies on intercepting the MPI API calls

and delivering aggregated information. We present here some examples of such tools.

PMPI is a customizable profiling layer that allows tools to intercept MPI calls. Therefore, when a communication routine is called, keeping track of the processes involved and the amount of data exchanged is possible. This approach has drawbacks, however. First, managing MPI datatypes is awkward and requires a conversion at each call. Also, PMPI cannot comprehend some of the most critical data movements, because an MPI collective is eventually implemented by point-to-point communications, and yet the participants in the underlying data exchange pattern cannot be guessed without knowledge of the collective algorithm implementation. A reduce operation is, for instance, often implemented with an asymmetric tree of point-to-point sends/receives in which every process has a different role (i.e., root, intermediary, and leaves). Known examples of stand-alone libraries using PMPI are DUMPI [12] and mpiP [20].

EZtrace [18] is a tool for analyzing and monitoring MPI programs. These tools launches the MPI executable and capture all point-to-point communication in a set of files. Each file corresponds to a process and describes its communication behavior with the other processes. However, it only allows for post-mortem and static analysis of the trace. It is not possible for the MPI program to monitor itself and change its behavior at runtime according to the communication pattern. Similarly to PMPI-based tools, this approach has an API-level granularity, and is unconcerned with detailed information about collective calls.

Another tool for analyzing and monitoring MPI programs is Score-P [16]. It is based on different but partially redundant analyzers that have been gathered within a single tool to allow both online and offline analysis. It uses Periscope and TAU, live profiling tools that evaluates performances and tries to track bottlenecks in both communication and memory accesses. Score-P relies also on Scalasca [10] and Vampir [15] for post-mortem analysis of event traces, with a graphical representation. Score-P relies on MPI wrappers and call-path profiles for online monitoring. Nevertheless, the application monitoring support offered by these tools is kept outside of the library, which means access to the implementation details and the communication pattern of collective operations once decomposed is limited.

PERUSE [14] takes a different approach, in that it allows the application to register callbacks that will be raised at critical moments in the point-to-point request lifetime. This method provides an opportunity to gather information on state-changes inside the MPI library and gain detailed insight on what type of data (i.e., point-to-point or collectives) is exchanged between processes, as well as how and when. This technique has been used in [6], [14].

In [5], a subset of the authors of this paper proposed a low-level MPI monitoring component for OPEN MPI [9]. This paper presented the design and evaluation of a communication monitoring infrastructure developed in the OPEN MPI software stack and able to expose a dynamically configurable level of detail about the application communication patterns. This component combines the advantages of the MPI Tool Information Interface interface to configure a flexible low-level implementation to provide efficient and dynamically configurable message-passing monitoring capabilities. As it is a component inside the OPEN MPI stack, it is able to decompose collective operations into their point-to-point expression: the monitoring component is plugged into the stack once messages are buffered to be sent to another MPI process. This is a strong advantage compared to all other approaches as it provides a better view of the actual messages exchanged during a collective communication. Moreover, all types of communications supported by the MPI-3 standard (including one-sided communications and I/O) are monitored.

In [3] the authors propose an introspection library for monitoring performance data at the application level. In this aspect, it is more general than the proposed approach. However, our approach is based on internal monitoring of the MPI runtime providing more precise information in terms of exchanged data In [2], the authors proposed an introspection library for a task-based runtime system which is a different progralming model than the message passing one of MPI.

III. BACKGROUND

The proposed library is based on a monitoring interface [5]. As presented above, this component was developed in OPEN MPI and used the MPI Tool Information Interface.

The OPEN MPI Project [9] is a comprehensive implementation of the MPI 3.1 standards [8] that was started in 2003, taking ideas from four earlier institutionally-based MPI implementations. It is developed and maintained by a consortium of academic, laboratory, and industry partners, and distributed under a modified BSD open source license. It supports a wide variety of CPU and network architectures that is used in the HPC systems. It is also the base for a number of vendors commercial MPI offerings, including Mellanox, Cisco, Fujitsu, Bull, and IBM. The OPEN MPI software is built on the Modular Component Architecture (MCA) [4], which allows for compile or runtime selection of the components used by the MPI library. This modularity enables experiments with new designs, algorithms, and ideas to be explored, while fully maintaining functionality and performance. In the context of this study, we take advantage of this functionality to seamlessly interpose our profiling components along with the highly optimized components provided by the stock OPEN MPI version.

MPI Tool Information Interface, is an interface that has been added in the MPI-3 standard [8]. It allows MPI developers, or third party, to offer a portable interface to different tools. These tools may be used to monitor applications, measure its performances, or profile it. MPI Tool Information Interface is an interface that eases the addition of external functions to a MPI library. It also allows the user to control and monitor given internal variables of the runtime system. In [5], a component was developed within this interface to precisely record the message exchanges between nodes during MPI

applications execution. This component is available within OPEN MPI since version 4.0. The number of messages and the amount of data exchanged are recorded, including or excluding internal communications (such as those generated by the implementation of the collective algorithms). This component can be activated at launch-time though `--mca pml _monitoring_enable value` on the `mpirun` command line to set the monitoring mode where `value` can be:

- 0 monitoring (and component) is disabled.
- 1 monitoring is enabled, with no distinction between user issued and library issued messages.
- ≥ 2 monitoring enabled, with a distinction between messages issued from the library (**internal**) and messages issued from the user (**external**).

However, this component is extremely low level. It requires to manipulate low-level features of the MPI Tool Information Interface and does not provide high-level semantics such as sessions or easy ways to gather monitored results from the different nodes.

Precise monitoring can be used to optimize process placement. Process placement is an optimization strategy that takes into account the affinity of processes (represented by a communication matrix) and the machine topology to decrease the communication costs of an application [11]. Various algorithms to compute such a process placement exist, one being TreeMatch [13] (designed by a subset of the authors of this article). We can distinguish between static process placement which is computed from traces of previous runs, and dynamic placement, that can be implemented by rank reordering, computed during the application execution (See experiments in Section VI).

IV. LIBRARY DESCRIPTION

A. General Description

The main interest of the MPI_Monitoring Library is to provide a higher-level interface by allowing the user to simply monitor its code and access the collected data. It mostly relies on low-level MPI Tool Information Interface features, mainly performance variables, that remain hidden to the user.

The MPI_Monitoring Library only defined one opaque datatype, **MPI_M_msid** (which stands for Monitoring Session IDentifier), that can only be used through the function of this library. They allow the user to create and act on monitoring sessions attached to a given communicator. While the session is active, the number and size of the messages exchanged between processors of the communicator are recorded and can later be obtained. Note that it also records communications that do not go through the communicator, as long as both processors belong to it. For example, a monitoring session attached to the communicator that splits even and odd processors will record all exchanges between processors 0 and 2, even if some communications use the communicator MPI_COMM_WORLD.

All these functions are prefixed by **MPI_M_** and their name does not contain any other capital letter, to respect the MPI convention. All functions are thread-safe. However they are not interrupt-safe (due to non-interrupt-safe MPI routines). They must be used in a proper environment that can be set using **init** and **finalize**, and both must be called between **MPI_Init** and **MPI_Finalize** as well. As other collective MPI routines, the MPI_Monitoring Library collective functions must be called by all processes of the given communicator. Note that **init** and **finalize** could be called multiple times as long as their environment do not overlap, but it is simpler to call them along with **MPI_Init** and **MPI_Finalize**.

Within the environment, the user can manage monitoring sessions using either **start**, **suspend**, **continue** or **reset**. It allows the user to precisely define the portion of the code to watch. The unique initial **start** put the session in its "active" state, and must match a final **suspend**. The monitoring session can be put in a "suspended" state using **suspend**, and later be put back in the "active" state using **continue**. The code is only watched while the session is in the "active" state. The function **reset** can be used on a session in the "suspended" state to put the data it contains back to zero. Note that if the session is in the "suspended" (resp. "active") state, **suspend** (resp. **continue**) cannot be called again. Another important feature is that sessions are completely independent and hence different sessions can overlap similar part of the code if necessary.

It is left to the user to properly use **free** on each started monitoring session to avoid memory leak. The recorded data can be copied into the user's buffers through **get_data**, **allgather_data** and **rootgather_data**.

The function **get_data** will copy the data specific to the process that called it into a buffer of this process. Even if it seems to be a function that could be called by only one process, it must be called by all that belong to the communicator of the session. However, parameters can vary among processes, and the special value MPI_M_DATA_IGNORE can be used to get rid of the unwanted data. The function **allgather_data** is equivalent to a call to **get_data** followed with a call to **MPI_Allgather**, such that all processes receive the collected data from all processes as a 2D matrix represented by a 1D array in row major format. The function **rootgather_data** act similarly, but it takes an additional parameter, root, and only the process whose rank is root will receive the data.

It is left to the user to give large enough buffers to store the recorded data. The minimal required size can be obtained with the function **get_info**. The user can also use **flush** and **rootflush** to directly save the data in a file. Those functions act similarly to **get_data** and **rootgather_data**, but they need a proper filename instead of buffers. All functions meant to obtain data require a flag argument to specify which of kind of communication the data is wanted (point-to-point, collective, one-sided or any combination of the previous options). Note that some collective MPI routines might generate point-to-point zero-length messages.

As accessing the data uses collective MPI routines that the user does not want to record along with dynamic memory allocation, the data can only be accessed while the session is in the "suspended" state and not already freed. Note that data

is stored using arrays of *unsigned long int*, and therefore a code with a lot of communications may cause overflows.

B. Usage

To properly use monitoring sessions, one should call **MPI_M_init** right after **MPI_Init** and **MPI_M_finalize** right before **MPI_Finalize** to set a proper environment. Then, create a different MPI_M_msid variable, automatically allocated, for each monitoring session wanted, in a scope that contains both the code to monitor and where the data need to be used. These sessions are completely independent from one another, therefore monitored portions of the code can overlap. Start the recording with **MPI_M_start** and stop it with **MPI_M_suspend**. If one wants to interrupt the monitoring session and then restart it, he can use **MPI_M_suspend** and **MPI_M_continue**, in this order. Once the monitoring is done, data can be obtained through multiple functions, it depends on how the data will be used. Sessions can be freed when the data they contain is no longer needed. A simple example will follow in Listing 1.

C. Common Example

Here is an example to find out how MPI uses point-to-point communications to implement MPI_Barrier. Note that this code is not fully complete as it does not check any return value, but it gives an idea on how this library can be used.

Listing 1: Produces a file that described all point-to-point messages used to implement MPI_Barrier

```
#include <mpi.h>
#include "MPI_Monitoring.h"

int main(){
  MPI_Init(NULL, NULL);
  MPI_M_init();
  MPI_M_msid id;
  MPI_M_start(MPI_COMM_WORLD, &id);

  MPI_Barrier(MPI_COMM_WORLD);

  MPI_M_suspend(id);
  MPI_M_rootflush(id, 0,
        "barrier", MPI_M_P2P_ONLY);
  MPI_M_free(id);
  MPI_M_finalize();
  MPI_Finalize();
  return 0;
}
```

Note that the only portion of the code that is being watched is between the calls to MPI_M_start and MPI_M_suspend, and this portion could possibly contain anything else.

D. Case with Several Collectives in a Program

As said above, the MPI monitoring component sees collectives once they have been decomposed into point-to-point messages. However, it is not able to distinguish between different calls: the monitoring aggregates all the sent operation into the same MPI_T variable. However, thanks to the sessions we are able to solve the problem of being able to distinguish which send operation belong to which collective. Indeed, it is sufficient to create one session per MPI call the programmer wants to distinguish (e.g. two different collective calls). In this case, the amount of data sent will be copied and stored in different buffers within the introspection library. As the library is designed such that the sessions can overlap or be nested, any kind of situations can be monitored thanks to the session mechanism.

V. Rank Reordering with Introspection Monitoring

Here, we explain how the introspection monitoring can be used to compute an optimized communicator where ranks are reordered and that allows for optimizing communications.

Communicator reordering was proposed in [17] for the case where the application is first monitored and then re-executed. Here, the algorithm described in Figure 1, does not require to restart the application.

Assume that you have a parallel program that performs an iterative computation using a function called *compute_iteration*. This function takes two parameters the (iteration number and a communicator). The first iteration (line 4) is monitored by our tool. Then, the number of data exchanged between all the ranks (`size_mat`) is gathered on rank 0. We compute a new mapping[1] of the processes in order to minimize communication cost (line 8). The output of this call is an array –`k`– of `n` integers (`n` is the number of MPI processes: the size of the original communicator). The array `k` describes an optimized mapping – based on the topology of the machine and the gathered communication pattern of the application – in order to minimize the communications. More precisely, `k` is such that in order to minimize communication cost, Process `i` should be executed on the process/core `k[i]`. This array is then broadcast among all the MPI processes (line 10). A new communicator (`opt_comm`) is then computed such that MPI process of rank `i` in the original communicator gets rank `k[i]` in `opt_comm` (line 11). Indeed, as the second parameter (color) of `MPI_Comm_split` is the same for all ranks, all MPI processes are put in the same communicator. It might be then required to redistribute the data (line 12) before executing the remaining iterations on the optimized communicator: this requires to know the vector `k` on all ranks such that any useful data is sent from rank `k[i]` to rank `i` in the original communicator.

The tricky part of the algorithm is actually line 11. Indeed, the fact that to optimize communication, process `k[i]` is mapped by TreeMatch onto processing unit `i` is equivalent of having rank `i` in the original communicator becomes rank `k[i]` in the optimized communicator.

VI. Experimental Results

The sequential experiment of Sec VI-A was performed on two nodes having an Infiniband EDR card and a Xeon 6140 Processor at 2.3 GHz.

[1]In the experiments, we will use the TreeMatch algorithm [13], but any other relevant algorithm can be used.

```
1  begin
2  |  MPI_M_init();
3  |  MPI_M_start(original_comm, &id);
4  |  compute_iteration(1, original_comm);
5  |  MPI_M_suspend(id);
6  |  MPI_M_rootgather_data(id, 0,
   |  MPI_M_DATA_IGNORE, size_mat,
   |  MPI_M_P2P_ONLY);
7  |  if (myrank==0) then
8  |  |  k = compute_mapping(local_topology,
   |  |  size_mat);
9  |  end
10 |  MPI_Bcast(k, n, MPI_INT, 0,
   |  original_comm);
11 |  MPI_Comm_split( original_comm, 0,
   |  k[myrank], &opt_comm);
12 |  redistribute_data(original_comm, k);
13 |  for it = 2 ... max_it do
14 |  |  compute_iteration(it, opt_comm);
15 |  end
16 end
```

Fig. 1: Reordering Algorithm for Iterative Computation

We conducted our experiments with parallel applications on an OmniPath 100 Gb/s cluster of the PlaFRIM experimental testbed. Each node features two Haswell Intel Xeon E5-2680 v3 with 12 cores (2.5 GHz) each. Each node has 128 Gb of 2133 MHz memory (5.3 GB/core).

If not detailed, for each experiments we use one MPI process per core or 24 MPI processes per node. This enables us to test the scalability and the overhead of the proposed solution.

The source, documentation and test of the library can be downloaded on the Inria gforge using this url: `svn checkout svn://scm.gforge.inria.fr/svnroot/mpi-introsp-mon`.

A. Comparison with Hardware Counters

In order to assess if the monitoring actually measures what is sent to the network, we have done the following experiment. An MPI program with 2 processes on different nodes send a random amount of data (between 1 and 800 KB) and then sleeps between 50 and 1000 ms. In the same program, a thread monitors the network traffic. We use two kinds of monitoring systems: the library we present in this paper and the hardware counters of the network card of the machine. On Linux, the number of bytes sent by an Infiniband card is available in the `/sys/class/infiniband/.../counters/port_xmit_data` file. The number read in this file has to be multiplied by the number of planes of the card (in general 4): see [1] for more details. The monitoring frequency is 10 ms and we use the reset features of the library session to monitor only what has happened between two measurements. In Fig. 2, we show the results for the Hardware counters (top) and our MPI introspection monitoring (bottom). It is a time series where the x-axis represents the time (in seconds) and the y-axis the amount of data that is monitored (in Kb).

In Fig 3, we show the same result but in a cumulative manner.

In both cases we see that the monitoring sees precisely what is actually sent to the network and the time difference is barely visible. This means that, once the introspection monitoring library has monitored some data they are almost immediately sent to the network. However, the advantage of the monitoring library compared to the hardware counter method is twofold. First, it is portable: it does not require to find the right file to be read on the target machine (in the `/sys`pseudo-filesystem), if only it exists. Second and more importantly, it provides a higher semantic as with the introspection monitoring, the rank of the sender and the receiver is attached to the sent data, which is impossible to see with the hardware counters of the network card.

B. Overhead

In order to measure the effective impact caused by the library on the monitored code, a simple test was used. It consists of a small code that is being run twice, one with and one without monitoring, both runs being timed. The code simply performs a reduce, transferring an arbitrary amount of data through MPI_COMM_WORLD. Different number of MPI processes are used 48 (2 nodes), 96 (4 nodes) and 192 (8 nodes). This test is launched 180 times to clear statistical fluctuations. The results are shown in Fig. 4. The error bar is the 95% confidence interval computed with the student T test using unpaired measures and unequal variance.

We plot only the data for small message size because the overhead can been seen only in these cases. Results show that most of the time the overhead is not statistically significant. In the worst case, the monitoring overhead is less than 10 μs.

C. Collective Optimization

In order to show the usefulness of a monitoring system being able to decompose collective communication into their point-to-point expression, we have designed an experiment that features rank reordering as explained in Section V using the communication matrix built with the monitored point-to-point messages. Here, we have taken two collective operations. A "*one to all*" collective (Broadcast) and a "*all to one* collective (Reduce). If monitored at a high-level (before the decomposition into point-to-point), it would not be possible to see the individual messages that are sent to execute the collective operation. Here, thanks to the monitoring library, each individual message is recorded, then ranks are reordered using a process placement algorithm we have developed (TreeMatch [13]). The goal is to re-arrange the ranks, such that, the ones that communicate the most are close to each other on the target machine. The results are depicted in Fig. 5. We plot the collective operation runtime versus the buffer size. Thanks to the precise monitoring, we are able to optimize

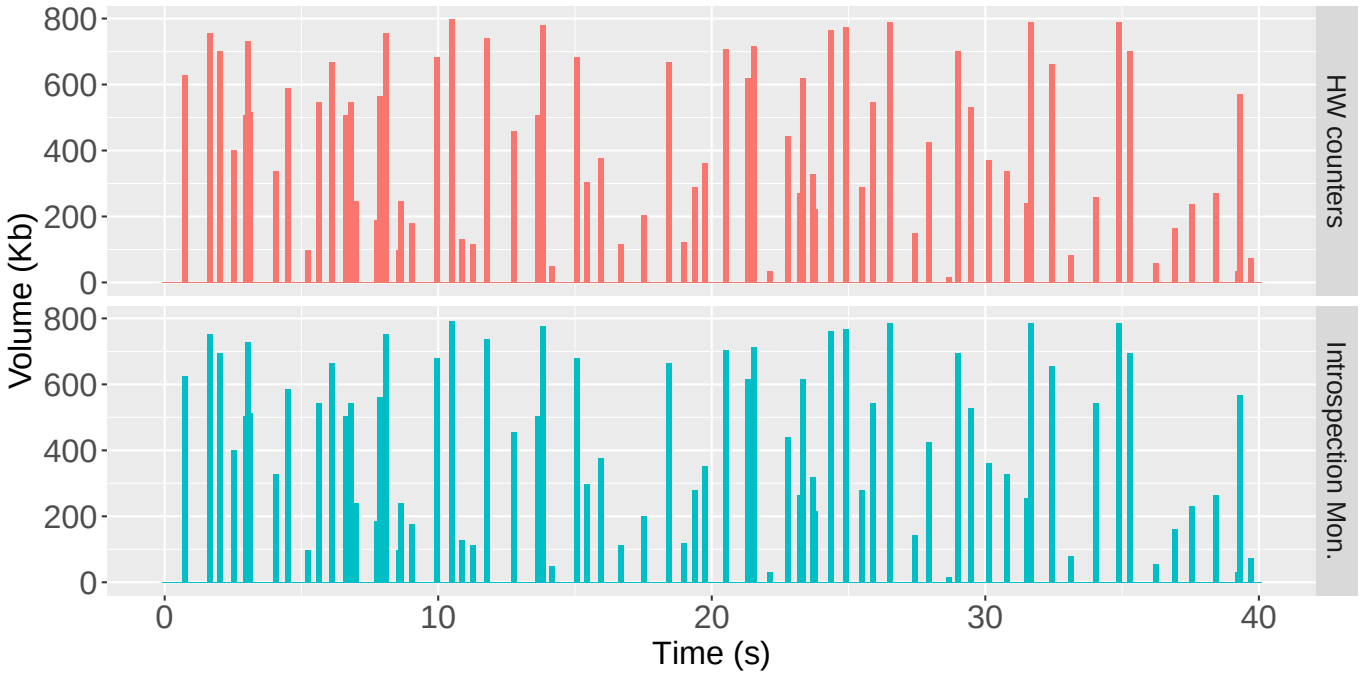


Fig. 2: Hardware Counters vs. Introspection Monitoring (time series)

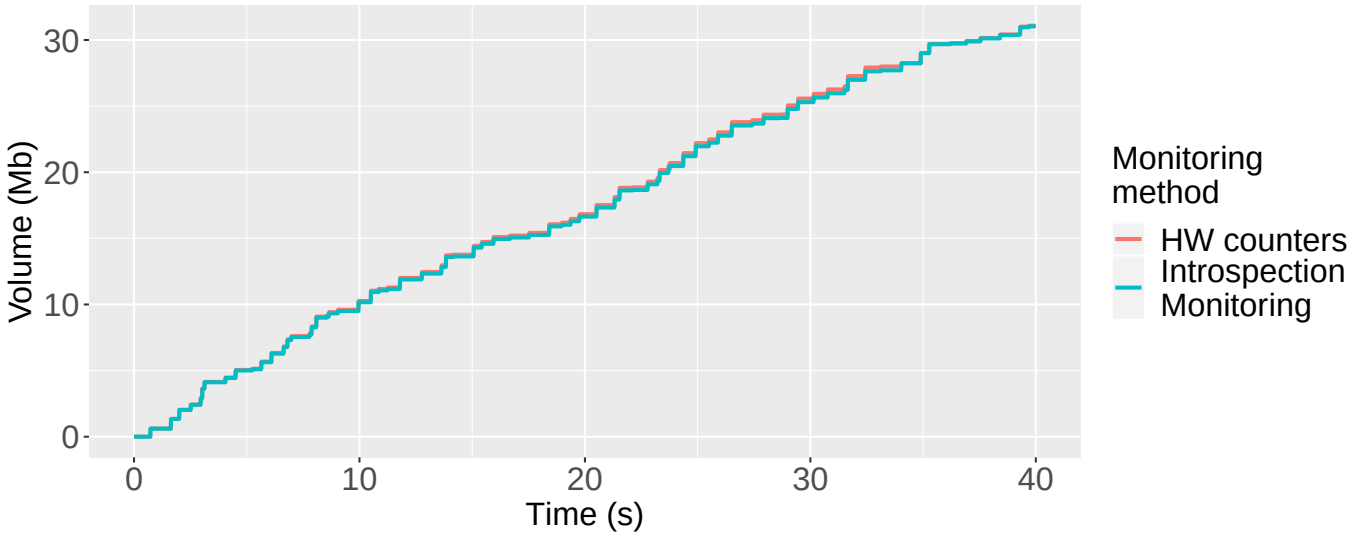


Fig. 3: Hardware Counters vs. Introspection Monitoring (cumulative)

the collective communication runtime for all the buffer size. For the reduce operation we see that, for 96 MPI processes (4 nodes), the runtime is reduced from 15.16 s to 7.57s for 2.10^8 integers. For 48 MPI processes (2 nodes), the runtime reduction is 8.28s to 5.59s for 2.10^8 integers. For the broadcast operation, the reduction is 16.34s to 10.24s for 96 ranks and 2.10^8 integers. For 48 processes, the runtime is reduced from 6.21s to 3.35s for 2.10^8 integers. For 192 MPI processes, the runtime is reduced form 11.92s to 5.01s (resp. 15.11s to 4.46s) for the reduce operation (resp the broadcast) for 2.10^8 integers.

D. Rank Reordering Micro-Benchmark

To exemplify the possibility of doing runtime optimization through the introspection monitoring library we have designed a benchmark where group of ranks perform an `MPI_Allgather` at each iteration. The processes mapping is such that for each group of ranks, their communicators span different nodes. Then, we perform a rank reordering for each group to optimize their data locality. Results are shown in Fig. 6. We display a heat map for three different number of processes (48, 96 and 192 i.e 2, 4 and 8 nodes). On the x-axis we have the size of the data (in number of integers) and on the

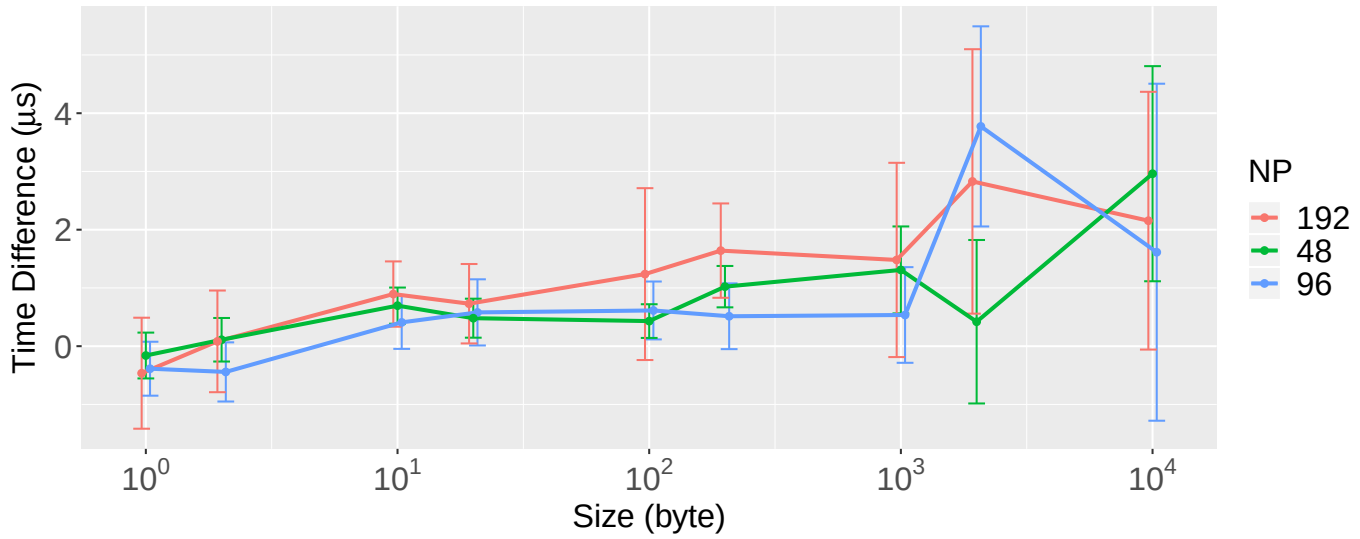


Fig. 4: Impact of the library on the monitored code (x log-scale): time difference between monitoring execution and non-monitoring execution. (positive values mean that execution with monitoring is slower than without). Each point is the difference of the average of 180 measurements and the error bar is the 95% confidence interval (unpaired T test with unequal variance).

(a) `MPI_Reduce` (MPI_MAX) walltime (x and y log-scale) for 48, 96 and 192 ranks and various buffer sizes. Binary Tree algorithm.

(b) `MPI_Bcast`walltime (x and y log-scale) for 48, 96 and 192 ranks and various buffer sizes. Binomial Tree algorithm.

Fig. 5: MPI Collective Optimization

y-axis the number of iterations. We measure the time $t1$ for n iterations then the time $t2$ for the reordering the process and the time $t3$ of n iterations after the reordering[2]. Here, to show the gain of the reordering we measure only the communication time and we compute the percentage of gain, taking into account the reordering overhead, as $100(t1-(t2+t3))/t1$.

As expected, we see that when the number of iterations is low or when the buffer size is small, the percentage of gain is lower than 0%. This means that the time to reorder plus the time to execute loops after reordering is greater than the time without reordering. In all these cases, the communication time is very small (never greater than 2 ms) and much higher than the reordering cost. However, as soon as the buffer size is large enough or the number of iterations is high, the reordering cost is amortized enabling a better execution time than the non-reorder case alone. In the best cases, the gain is more than 95% (almost a 2-time improvement): this is the case for 48 processes and 100 iterations or more for the 100 000 MPI_INT buffer size (or 1000 iterations or more for the 10 000 buffer size).

[2]timings are the average of the maximum time on all the ranks of 6 runs

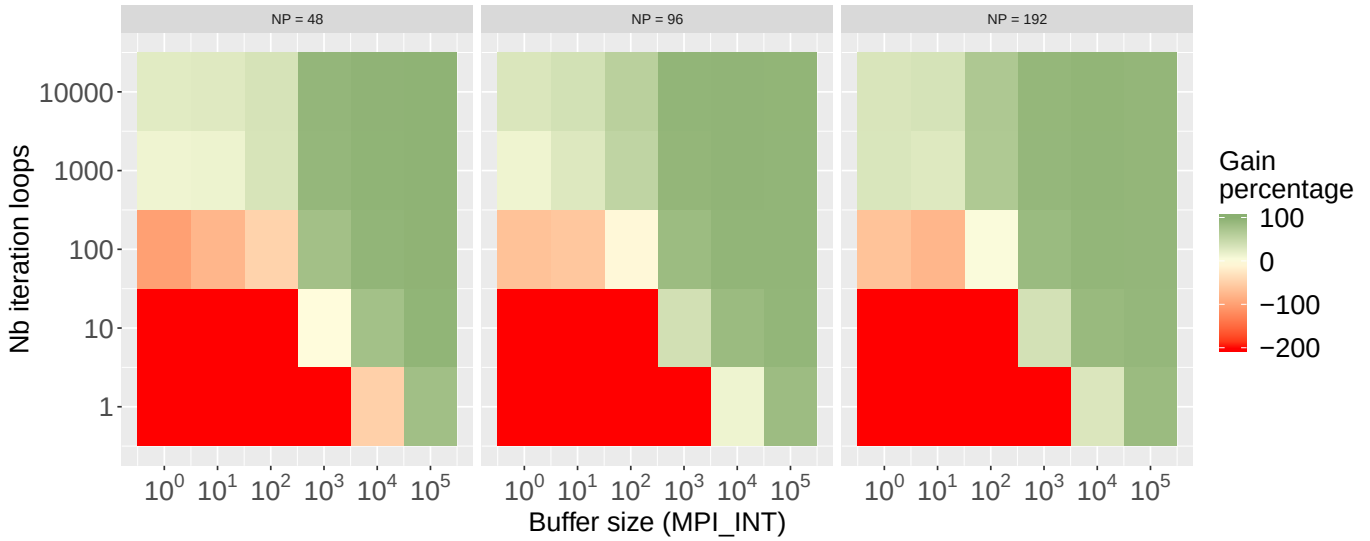


Fig. 6: Heatmap of the gain (in percent) of using reordering while varying the number of iterations and the buffer size. Green values: reordering pays off. Red values: reordering overhead is too high.

E. Rank Reordering on Conjugate Gradient

The conjugate gradient algorithm is an iterative algorithm that computes the solution of a system of linear systems whose matrix is symmetric and positive-definite. Such algorithm is perfectly suited for the reordering use-case as the communication pattern of each iteration is the same. Hence, we can monitor the first iteration, build the communication matrix, compute a new mapping and apply a rank reordering as explained in Sec. V.

In this paper, we have taken the conjugate gradient code from the NAS parallel benchmark 3.3[3] called CG. We have designed two functions one to start the monitoring and one to compute the reordering. The CG code uses only the MPI_COMM_WORLD communicator. In order to apply the reordering we have changed this to a global variable which is assigned to MPI_COMM_WORLD at the beginning of the program and to the new computed communicator after the reordering phase has been done. To avoid redistributing the data, we have used the fact that the CG code has an initialization phase that does one iteration of the conjugate gradient algorithm. We monitor this initialization phase to compute the optimized communicator. In order to be fair, the time of the reordering is added to the whole timing of the application.

In the NAS suite, the number of processes of the CG code is a power of two. We use three values (64, 128, 256) and 3, 6 and 11 nodes respectively. As the number of cores per node is 24, some cores are spared. Hence, we use and compare three different initial mappings: a random mapping, a round-robin mapping (RR) where rank i is mapping on the i^{th} leftmost core and standard where no binding is used.

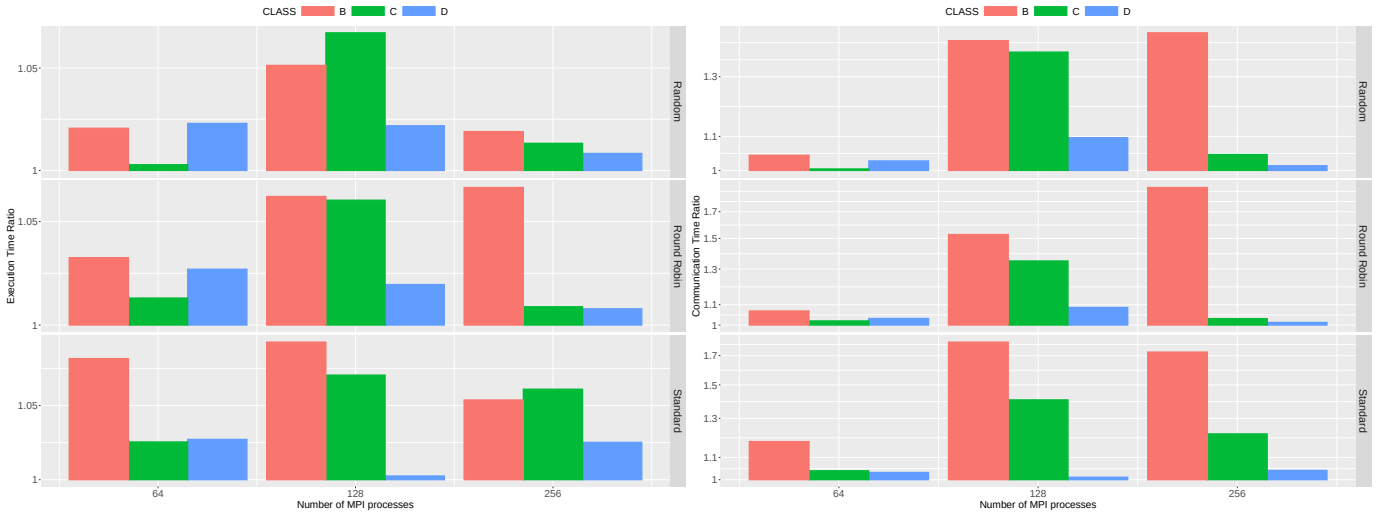
Also, to avoid interference with the other running applications, we have used nodes that are on the same 100 GB/s switch.

Results are depicted in Fig 7. We show the gain of the reordering vs. without reordering (ratio greater than 1 shows a gain for the reordering case). On the X-axis, we vary the number of MPI processes (from 64 to 256). Each bar is different class from B (small problem size) to D (large problem size)[4]. Each graph has three rows that show three types of initial mapping described in the previous paragraph (Random, Round-Robin or Standard).

In Fig. 7a the y-axis is the ratio of execution time. We see that all the ratios are greater than 1, meaning that the reordering is beneficial. In general the ratio is decreasing with the problem size (the class), this is due to the fact that the larger the problem the longer the execution time and, even if the gain difference is larger, the smaller the ratio. To see what is the actual gain in terms of communication, we have measured the gain of the reordering but only for the communication time to do so we have added a timer that measures the time spent by rank 0 in MPI calls. The results are shown in Fig 7b. In this case the ratios are much greater (both timings are reduced by the same amount) and show, in some cases, up to a 1.9x improvement. Another interesting fact is that in case of the random mapping the gain is not better than the round-robin mapping. This is due to the fact that the remapping algorithm we use (TreeMatch) is sensitive to the initial mapping: in the case of the random initial mapping it is not able to provide a reordering as good as with the round robin initial mapping. This is an issue out of the scope of the paper and will be tackled in future work.

[3]https://www.nas.nasa.gov/publications/npb.html

[4]We do not show the class A as the timing are very small (less than 0.1 s) and the reordering is not useful



(a) Gain in execution time

(b) Gain in communication time

Fig. 7: NAS benchmark conjugate gradient reordering gain. The gain is the ratio of the non-reordered case vs the reordered case: ratio greater than one means that the reordering is faster. Y-axis in log-scale. We plot different class of the CG kernel (B to D) and each row is a different initial mapping (random, round-robin or standard).

VII. Discussion

In most of our experiments, we have 1 MPI process per core. However, having many MPI processes per nodes does not help to exhibit communication optimization. Hence, we think that our results would show even more gain in the case where we use more nodes and less MPI processes per nodes.

Apart from rank reordering, being able to understand how the application communicates can be useful in many other cases. For instance, in [7] we used the dynamic and introspection monitoring to compute the communication matrix during the execution of an MPI application. The goal was to perform elastic computations in case of node failures or when new nodes are available. The runtime system migrated MPI processes when the number of computing resources changed: the placement of such processes was computed according to the topology and the communication matrix. Recently, in [19], we use the introspection monitoring to detect and predict network usage using machine learning technique. Here, the goal is to determine when the network is under-utilized in order to fetch checkpoint to the storage.

Reordering is interesting if the mapping algorithm is fast enough. In the experiments we have shown that reordering 256 MPI Processes has a negligible impact on the duration (up to 0.02 seconds). One might wonder what happens in the case of a larger number of processors. In table I we display the mapping computation time of TreeMatch for very large settings (up to a communication matrix of order 65 536). We see that even for such large input size the time to compute the reordering is less than 100s.

Com Matrix size	8 192	16 384	32 768	65 536
Reordering time in s	2.6	6.3	20.9	88.7

TABLE I: Reordering computation time for large input size

VIII. Conclusion

Being able to query the state of the MPI software stack is very important as it enables runtime optimization. In particular this enables to optimize the way communications are carried out on a distributed memory parallel machine. In this paper we have proposed an introspection library. This high-level library features sessions that allows for watching specific part of the application, is able to see how collectives are decomposed into point-to-point communications, provide a C as well as a Fortran API and is freely available. We have carried-out experiments that show that the library captures precisely what is sent to the network card with a very small overhead. Thanks to its ability to see how collective are decomposed in point-to-point, we have been able to optimize tree-based collective at runtime. Last, we have presented a dynamic rank reordering algorithm and show that, as long as the communication cost is large enough, the reordering cost is amortized leading to almost 2-time performance improvement.

This library is based on a monitoring module available only in Open MPI (version 4.0 or later). The advantage of such a library is that it hides the low-level MPI tool variables and command. Hence, if a monitoring module would be developed in another MPI implementation (such as MPICH), our proposed library could easily be ported to such implementation enabling a better portability.

ACKNOWLEDGMENT

We would like to thank Guillaume Mercier for fruitful discussion on the reordering strategy. The PlaFRIM experimental testbed is being developed with support from Inria, LaBRI, IMB, and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux, and CNRS.

REFERENCES

[1] Understanding mlx5 Linux Counters and Status Parameters. https://community.mellanox.com/s/article/understanding-mlx5-linux-counters-and-status-parameters, December 2018.

[2] Xavier Aguilar, Herbert Jordan, Thomas Heller, Alexander Hirsch, Thomas Fahringer, and Erwin Laure. An on-line performance introspection framework for task-based runtime systems. In *International Conference on Computational Science*, pages 238–252. Springer, 2019.

[3] Xavier Aguilar, Erwin Laure, and Karl Fürlinger. Online performance data introspection with ipm. In *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 728–734. IEEE, 2013.

[4] Brian Barrett, Jeffrey M. Squyres, Andrew Lumsdaine, Richard L. Graham, and George Bosilca. Analysis of the Component Architecture Overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[5] George Bosilca, Clément Foyer, Emmanuel Jeannot, Guillaume Mercier, and Guillaume Papauré. Online Dynamic Monitoring of MPI Communications. In Springer, editor, *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing*, volume 10417 of *LNCS*, pages 49–62, Santiago de Compostela, Spain, August 2017.

[6] Kevin A. Brown, Jens Domke, and Satoshi Matsuoka. Tracing Data Movements Within MPI Collectives. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 117:117–117:118, New York, NY, USA, 2014. ACM.

[7] Iván Cores, Patricia Gonzalez, Emmanuel Jeannot, María J. Martín, and Gabriel Rodriguez. An application-level solution for the dynamic reconfiguration of mpi applications. In *12th International Meeting on High Performance Computing for Computational Science (VECPAR 2016)*, Porto, Portugal, June 2016. To appear.

[8] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/, September 2012.

[9] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[10] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[11] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. An overview of topology mapping algorithms and techniques in high-performance computing. *High-Performance Computing on Complex Environments*, pages 73–94, 2014.

[12] Curtis L Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P Kenny, Ali Pinar, David A Evensky, and Jackson Mayo. A simulator for large-scale parallel computer architectures. *Technology Integration Advancements in Distributed Systems and Computing*, 179, 2012.

[13] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, April 2014.

[14] Rainer Keller, George Bosilca, Graham Fagg, Michael Resch, and Jack J. Dongarra. *Implementation and Usage of the PERUSE-Interface in Open MPI*, pages 347–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[15] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[16] Andreas Knüpfer and et al. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[17] Guillaume Mercier and Emmanuel Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In *Proceedings of the 16th International EuroMPI Conference*, LNCS 6960, pages 39–49, Santorini, Greece, September 2011. Springer Verlag.

[18] François Trahay, Elisabeth Brunet, Mohamed Mosli Bouksiaa, and Jianwei Liao. Selecting points of interest in traces using patterns of events. In Masoud Daneshtalab, Marco Aldinucci, Ville Leppänen, Johan Lilius, and Mats Brorsson, editors, *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, pages 70–77. IEEE Computer Society, 2015.

[19] Shu-Mei Tseng, Bogdan Nicolae, George Bosilca, Emmanuel Jeannot, Aparna Chandramowlishwaran, and Franck Cappello. Towards Portable Online Prediction of Network Utilization using MPI-level Monitoring. In *EuroPar'19: 25th International European Conference on Parallel and Distributed Systems*, Goettingen, Germany, August 2019.

[20] Jeffrey S Vetter and Michael O McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Notices*, volume 36, pages 123–132. ACM, 2001.