Online Dynamic Monitoring of MPI Communications

George Bosilca, Clément Foyer, Emmanuel Jeannot, Guillaume Mercier, Guillaume Papauré

Online Dynamic Monitoring of MPI Communications

George Bosilca[1], Clément Foyer[2], Emmanuel Jeannot[2], Guillaume Mercier[23], and Guillaume Papauré[4]

[1] Univ of Tennessee, ICL
[2] Inria, Labri, CNRS Univ. Bordeaux
[3] Bordeaux INP
[4] ATOS - BULL

Abstract. As the complexity and diversity of computer hardware and the elaborateness of network technologies have made the implementation of portable and efficient algorithms more challenging, the need to understand application communication patterns has become increasingly relevant. This paper presents details of the design and evaluation of a communication-monitoring infrastructure developed in the Open MPI software stack that can expose a dynamically configurable level of detail concerning application communication patterns.

Keywords: MPI; Monitoring; Communication Pattern; Process Placement

1 Introduction

With the expected increase of applications concurrency and input data size, one of the most important challenges to be addressed in the forthcoming years is data transfer and locality (i.e., how to improve data accesses and transfers in the application). Among the various aspects of locality, one issue stems from the memory and the network. Indeed, the transfer time of data exchanges between processes of an application depends on both the affinity of the processes and their location. A thorough analysis of the application's behavior and of the target underlying execution platform combined with clever algorithms and strategies have the potential to dramatically improve the application communication time, making it more efficient and robust in the midst of changing network conditions (e.g., contention). The general consensus is that the performance of many existing applications could benefit from improved data locality [9].

Hence, to compute an optimal – or at least an efficient – process placement we need to understand the underlying hardware characteristics (including memory hierarchies and network topology) and how the application processes are exchanging messages. The two inputs of the decision algorithm are therefore the machine topology and the application communication pattern. The machine topology information can be gathered through existing tools or be provided by a management system. Among these tools Netloc/Hwloc [4] provides a (nearly)

portable way to abstract the underlying topology as a graph interconnecting the various computing resources. Moreover, the batch scheduler and system tools can provide the list of resources available to the running jobs and their interconnections.

To address the second point and understand the data exchanges between processes, precise information about the application communication patterns is needed. Existing tools are either addressing the issue at a high level and thus failing to provide accurate details, or they are intrusive and deeply embedded in the communication library. To confront these issues, we designed a light and flexible monitoring interface for MPI applications with the following features. First, the need to monitor more than simply two-sided communications—interactions in which the source and destination of the message are explicitly invoking an API for each message—is becoming prevalent. As such, our monitoring support is capable of extracting information about all types of data transfers: two-sided, one-sided (or remote memory access), and I/O. In the scope of this paper, we will focus our analysis on one- and two-sided communications. We recorded the number of messages, the sum of message sizes, and the distribution of the sizes between each pair of processes. We also recorded how these messages have been generated by direct user calls via the two-sided API or automatically generated as a result of collective algorithms, a process related to one-sided messages. Second, we provided mechanisms for the MPI applications themselves to access this monitoring information through the MPI tool information interface. This allowed the monitoring—which may involve recording only specific parts of the code or recording only during particular time periods—to be dynamically enabled or disabled, and it gave the ability to introspect the application behavior. Last, the output of this monitoring provides different matrices describing this information for each pair of processes. Such data is available both online (i.e., during the application execution) and off-line (i.e., for the post-mortem analysis and optimization of a subsequent run).

We conducted experiments to assess the overhead of this monitoring infrastructure and to demonstrate its effectiveness as compared with other solutions from the literature.

In Section 2 of this paper we present the related work; in Section 3, the required background; in Section 4, the design; in Section 5, the implementation; in Section 6, the result; and in Section 7, the conclusion.

2 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 [10] and mpiP [15].

Another tool for analyzing and monitoring MPI programs is Score-P [13]. It is based on different but partially redundant analyzers that have been gathered within a single tool to allow both online and offline analysis. 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 [12] 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 [5, 12].

Tools that provide monitoring that is both light and precise (e.g., showing collective communication decomposition) do not exist.

3 Background

The Open MPI Project [8] is a comprehensive implementation of the MPI 3.1 standard [7] that was started in 2003 and takes ideas from four earlier institutionally based MPI implementations. Open MPI is developed and maintained by a consortium of academic, laboratory, and industry partners and is distributed under a modified BSD open-source license. It supports a wide variety of CPU and network architectures used in HPC systems. It is also the base for a number of commercial MPI offerings from vendors, including Mellanox, Cisco, Fujitsu, Bull, and IBM. The Open MPI software is built on the Modular Component Architecture (MCA) [1], 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 has been added in the MPI-3 standard [7]. This interface allows the application to configure internal parameters of the MPI library and get access to internal information from the MPI library. In our context, this interface will offer a convenient and flexible way to access the

monitored data stored by the implementation and control of the monitoring phases.

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 [9]. Various algorithms to compute such a process placement exist, one being TreeMatch [11] (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 computed during the application execution (see the experiments in Section 6).

4 Design

Monitoring generates the application communication pattern matrix. The order of the matrix is the number of processes, and each (i, j) entry gives the amount of communication between process i and process j. Monitoring outputs several values and, hence, several matrices: the number of bytes and the number of messages exchanged. Moreover, it distinguishes between point-to-point communications and collective or internal protocol communications.

It is also able to keep track of collective operations after their transition to point-to-point communications. Therefore, monitoring requires interception of the communication inside the MPI library itself instead of relinking weak symbols to a third-party dynamic one, which allows this component to be used in parallel with other profiling tools (e.g., PMPI).

For scalability reasons, we can automatically gather the monitoring data into one file instead of dumping one file per rank.

In summary, we plan to cover a wide spectrum of needs while employing different levels of complexity for various levels of precision. Our design provides an API for each application to enable, disable, or access its own monitoring information. Otherwise, an application can be monitored without any modification of its source code by activating the monitoring components at launch time; results are retrieved when the application completes.

We also supply a set of mechanisms to combine monitored data into communication matrices. They can be used either at the end of the application (when `MPI_Finalize` is called) or post-mortem. For each pair of processes, a histogram of geometrically increasing message sizes is available.

5 Implementation

The precision required for the results prompted us to implement the solution within the Open MPI stack[5]. The component described in this article was developed in a branch of Open MPI (available at [14]) and is now available in

[5] A proof-of-concept version of this monitoring has been implemented in MPICH

the development version of Open MPI, and on all stable version after 3.0. Because we were planning to intercept all types of communications—two-sided, one-sided, and collectives—we exposed a minimalistic common API for the profiling as an independent engine and then linked all the MCA components doing the profiling with this engine. Due to the flexibility of the MCA infrastructure, the active components can be configured at runtime either via `mpirun` arguments or via the API (implemented with the MPI Tool Information Interface). All implementation details are available at [3].

To cover the wide range of operations provided by MPI, we added four components to the software stack: one in the collective communication layer (COLL), one in the one-sided layer (remote memory accesses, OSC), one in the point-to-point management layer (PML), and one common layer capable of orchestrating the information gathered by the other layers and record data. When activated at launch time (through the `mpiexec` option `--mca pml_monitoring_enable x`), this enable all monitoring components, as indicated by the comma-separated value of x. The design of Open MPI allows for easy distinctions between different types of communication tags, and x allows the user to include or exclude tags related to collective communications or to other internal coordination (these are called internal tags as opposed to external tags, which are available to the user via the MPI API).

Specifically, the PML layer sees communications after collectives have been decomposed into point-to-point operations. COLL and OSC both work at a higher level to be able to record operations that do not go through the PML layer (e.g. when dedicated drivers are used). Therefore, as opposed to the MPI standard profiling interface (`PMPI`) method where the MPI calls are intercepted, we monitored the actual point-to-point calls that are issued by Open MPI, which yields much more precise information. For instance, we can infer the underlying topologies and algorithms behind the collective algorithms (e.g. the tree topology used for aggregating values in an `MPI_Reduce` call). However, this comes at the cost of a possible redundant recording of data for collective operations when the data-path goes through the COLL and the PML components[6].

For an application to enable, disable or access its own monitoring, we implemented a set of callback functions using the MPI Tool Information Interface. The functions make knowing the amount of data exchanged between a pair of processes possible at any time and in any part of the application's code. An example of such code is given in Fig. 1. The call to `MPI_T_pvar_get_index` provides the index (e.g., the key) of the performance variable. This variable is allocated and attached to the communicator with a call to `MPI_T_pvar_handle_alloc`. This starts a monitoring phase that resets the internal monitoring state. Then, an MPI_T session is started with the `MPI_T_pvar_start` call. When necessary, the monitored values are retrieved with `MPI_T_pvar_read`. Last, a call to `MPI_Allreduce` allows each processes to get the maximum of each value.

[6] Nevertheless, a precise monitoring is still possible with the use of the monitoring API.

Furthermore, the final summary dumped at the end of the application gives us a detailed output of the data exchanged between processes for each point-to-point, one-sided, and collective operation. The user is then able to refine the results.

Internally, these components use an internal process identifier (`ids`) and a single associative array employed to translate sender and receiver ids into their `MPI_COMM_WORLD` counterparts. Our mechanism is, therefore, oblivious to communicator splitting, merging, or duplication. When a message is sent, the sender updates three arrays: the number of messages, the size (in bytes) sent to the specific receiver, and the message size distribution. Moreover, to distinguish between external and internal tags, one-sided emitted and received messages, and collective operations, we maintain five versions of the first two arrays. Also, the histogram of message sizes distribution is kept for each pair of ids, and goes from 0 byte messages to messages of more than 2^{64} bytes. Therefore, the memory overhead of this component is at maximum 10 arrays of N 64 bits elements, in addition to the N arrays of 66 elements of 64 bits for the histograms, with N being the number of MPI processes. These arrays are lazily allocated, so they exist for a remote process only if communications occur with it.

In addition to the amount of data and the number of messages exchanged between processes, we keep track of the type of collective operations issued on each communicator: one-to-all operations (e.g., `MPI_Scatter`), all-to-one operations (e.g., `MPI_Gather`) and all-to-all operations (e.g., `MPI_Alltoall`). For the first two types of operations, the root process records the total amount of data sent and received respectively and the count of operations of each kind. For all-to-all operations, each process records the total amount of data sent and the count of operations. All these pieces of data can be flushed into files either at the end of the application or when requested through the API.

6 Results

We conducted out the experiments on an Infiniband cluster (HCA: Mellanox Technologies MT26428 (ConnectX IB QDR)). Each node features two Intel Xeon Nehalem X5550 CPUs with 4 cores (2.66 GHz) per each CPU.

6.1 Overhead Measurement

One of the main issues of monitoring is the potential impact on the application time-to-solution. As our monitoring can be dynamically enabled and disabled, we can compute the upper bound of the overhead by measuring the impact with the monitoring enabled on the entire application. We wrote a micro benchmark that computes the overhead induced by our component for various kinds of MPI functions and measured this overhead for both shared- and distributed-memory cases. The number of processes varies from 2 to 24, and the amount of data ranges from 0 up to 1MB. Fig. 2 displays the results as heatmaps (the median of a thousand measures). Blue nuances correspond to low overhead, and yellow

```
MPI_T_pvar_handle count_handle;
int count_pvar_idx;
const char count_pvar_name[] = "pml_monitoring_messages_count";
size_t *counts;
int count; /*size of the array*/

/* Retrieve the proper pvar index */
MPI_T_pvar_get_index(count_pvar_name, MPI_T_PVAR_CLASS_SIZE,
                     &count_pvar_idx);

/* Allocating a new PVAR in a session will reset the counters and set count */
MPI_T_pvar_handle_alloc(session, count_pvar_idx, MPI_COMM_WORLD,
                        &count_handle, &count);

counts = (size_t*)malloc(count * sizeof(size_t));

/* start monitoring session */
MPI_T_pvar_start(session, count_handle);

/* Begin communications */
/* [...] */
/* End communications */

/*
   Retrieve the number of messages sent to each
   peer in MPI_COMM_WORLD
*/

/* get the monitored values */
MPI_T_pvar_read(session, count_handle, counts);

/*
   Global reduce so everyone knows the maximum
   messages sent to each rank
*/
MPI_Allreduce(MPI_IN_PLACE, counts, count, MPI_UNSIGNED_LONG,
              MPI_MAX, MPI_COMM_WORLD);

/* Operations on counts */
/* [...] */

free(counts);

MPI_T_pvar_stop(session, count_handle);

MPI_T_pvar_handle_free(session, &count_handle);
```

Fig. 1: Monitoring Code Snippet.

colors to higher overhead. As expected, the overhead was more visible on a shared memory setting, where the cost of the monitoring is more significant compared with the decreasing cost of data transfers. Also, as the overhead is related to the number of messages and not to their content, the overhead decreases as the size of the messages increased. Overall, the median overhead is 4.4% and 2.4% respectively for the shared- and distributed-memory cases, which proves that our monitoring is cost effective.

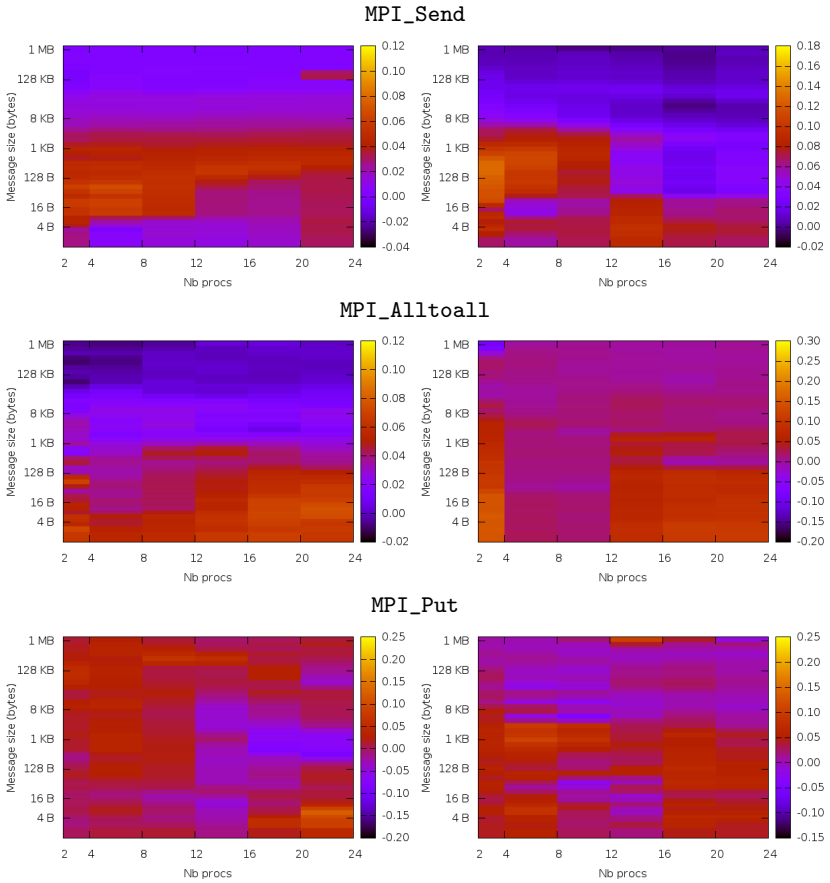


Fig. 2: Monitoring overhead for `MPI_Send`, `MPI_Alltoall` and `MPI_Put` operations. Left: distributed memory, right: shared memory.

To measure the impact on applications, we used some of the NAS parallel benchmarks—namely BT, CG and LU. These tests have the highest number of MPI calls, and so we chose them to maximize the potential impact of the monitoring on the application. Table 1 shows the results, which are an average of 20 runs. Shaded rows mean that the measures display a statistically significant

difference (using the `Student's t-Test` on the measures) between a monitored run and non-monitored one. Overall, we see that the overhead is consistently below 1% and on average around 0.35%. Interestingly, for the LU kernel, the overhead seems lightly correlated with the message rate, meaning the larger the communication activity, the higher the overhead. For the CG kernel, however, the timings are so small that it is hard to see any influence of this factor beyond

Kernel	Class	NP	Monitoring time	Non mon. time	#msg/proc	Overhead	#msg/sec
bt	A	16	6.449	6.443	2436.25	0.09%	6044.35
bt	A	64	1.609	1.604	4853.81	0.31%	193066.5
bt	B	16	27.1285	27.1275	2436.25	0.0%	1436.87
bt	B	64	6.807	6.8005	4853.81	0.1%	45635.96
bt	C	16	114.6285	114.5925	2436.25	0.03%	340.06
bt	C	64	27.23	27.2045	4853.81	0.09%	11408.15
cg	A	16	0.1375	0.1365	1526.25	0.73%	177600.0
cg	A	32	0.103	0.1	2158.66	3.0%	670650.49
cg	A	64	0.087	0.0835	2133.09	4.19%	1569172.41
cg	B	8	11.613	11.622	7487.87	-0.08%	5158.27
cg	B	16	6.7695	6.7675	7241.25	0.03%	17115.0
cg	B	32	3.8015	3.796	10243.66	0.14%	86228.33
cg	B	64	2.5065	2.495	10120.59	0.46%	258415.32
cg	C	32	9.539	9.565	10243.66	-0.27%	34363.87
cg	C	64	6.023	6.0215	10120.59	0.02%	107540.76
lu	A	8	8.5815	8.563	19793.38	0.22%	18452.14
lu	A	16	4.2185	4.2025	23753.44	0.38%	90092.45
lu	A	32	2.233	2.2205	25736.47	0.56%	368816.39
lu	A	64	1.219	1.202	27719.36	1.41%	1455323.22
lu	B	8	35.2885	35.2465	31715.88	0.12%	7190.08
lu	B	16	18.309	18.291	38060.44	0.1%	33260.53
lu	B	32	9.976	9.949	41235.72	0.27%	132271.75
lu	B	64	4.8795	4.839	44410.86	0.84%	582497.18
lu	C	16	72.656	72.5845	60650.44	0.1%	13356.19
lu	C	32	38.3815	38.376	65708.22	0.01%	54783.24
lu	C	64	20.095	20.056	70765.86	0.19%	225380.19

Table 1: Overhead for the BT, CG and LU NAS kernels

measurements noise.

We have also tested the *Minighost* mini-application [2] that computes a stencil in various dimensions to evaluate the overhead. An interesting feature of this mini-application is that it outputs the percentage of time spent to perform communication. In Fig. 3, we depict the overhead depending on this communication ratio. We ran 114 different executions of the MiniGhost application and split those runs into four range categories depending on the percentage of time spent

in communications (0%-25%, 25%-50%, 50%-75% and 75%-100%). A point represents the median overhead (in percent) and the error bars represent the first and third quartile. We see that the median overhead is increasing with the percentage of communication. Indeed, the more time you spend in communication the more visible the overhead for monitoring these communications. However, the overhead accounts for only a small percentage.

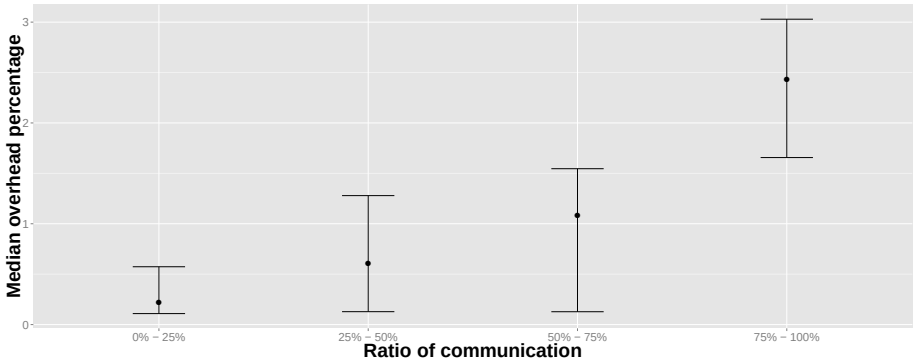
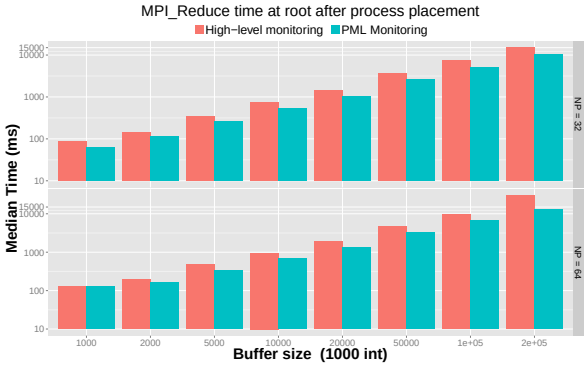


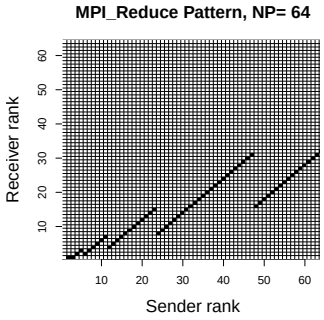
Fig. 3: Minighost application overhead as a function of the communication percentage of the total execution time.

6.2 MPI Collective Operations Optimization

In these experiments we have executed an `MPI_Reduce` collective call on 32 and 64 ranks (on 4 and 8 nodes respectively), with a buffer that ranged from 1.10^6 to 2.10^8 integers and a rank of 0 acting as the root. We took advantage of the Open MPI infrastructure to block the dynamic selection of the collective algorithm and instead forced the reduce operation to use a binary tree algorithm. Because we monitored the collective communications after they have been broken down into point-to-point communications, we were able to identify details of the collective algorithm implementation and expose the underlying binary tree algorithm (see Fig. 4b). This provided a much more detailed understanding of the underlying communication pattern compared with existing tools, where the use of a higher-level monitoring tool (e.g., PMPI) completely hides the collective algorithm communications. With this pattern, we used the TreeMatch algorithm to compute a new process placement and compared it with the placement obtained using a high-level monitoring method (that does not see the tree and hence is equivalent to the round-robin placement). Results are shown in Fig. 4a. We see that the optimized placement is much more efficient than the one based on high-level monitoring. For instance with 64 ranks and a buffer of 5.10^6 integers the walltime is 338 ms vs. 470 ms (39% faster).



(a) `MPI_Reduce` (MPI_MAX) walltime (x and y log-scale) for 32 and 64 ranks and various buffer sizes. Process placement based on our PML monitoring vs. high-level monitoring (RR placement)



(b) Communication pattern as seen by our monitoring tool (once the collective communication is decomposed into point-to-point communications).

Fig. 4: `MPI_Reduce` Optimization.

6.3 Use Case: Fault Tolerance with Online Monitoring

In addition to the usage scenarios mentioned above, the proposed dynamic monitoring tool has been demonstrated in our recent work. In [6]we used the dynamic monitoring feature 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. To this end, the authors used the TreeMatch [11] algorithm to recompute the process mapping onto the available resources. The algorithm decides how to move processes based on the application's gathered communication matrix: the more two processes communicate, the closer they are remapped onto the physical resources. Gathering the communication matrix was performed online using the callback routines of the monitoring: such a result would not have been possible without the tool proposed in this paper.

6.4 Static Process Placement of applications

We tested the TreeMatch algorithm for performing static placement to show that the monitoring provides relevant information allowing execution optimization. To do so, we first monitored the application using the proposed monitoring tool of this paper. Second, we built the communication matrix (here using the number of messages) and then applied the TreeMatch algorithm on this matrix and the topology of the target architecture. Finally, we re-executed the application using the newly computed mapping. Different settings (kind of stencil,

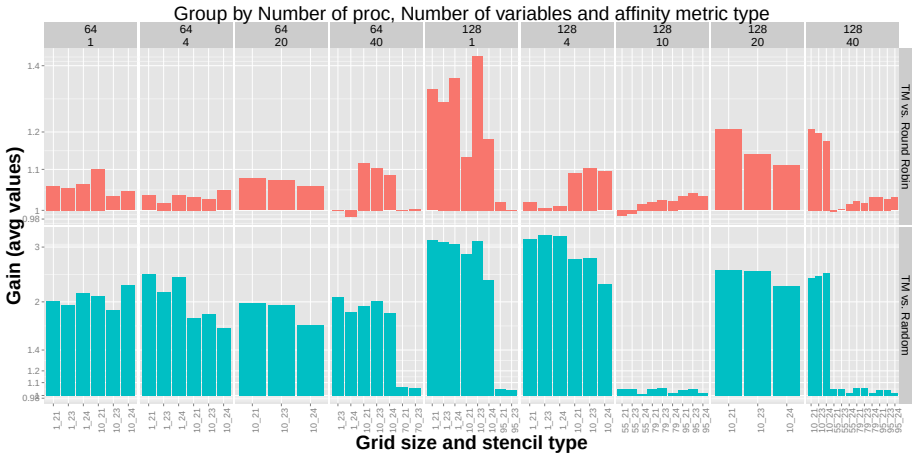


Fig. 5: Average gain of TreeMatch placement vs. Round Robin and random placements for various MiniGhost runs.

the stencil dimension, number of variables per stencil point, and number of processes) are shown in fig. 5. We see that the gain is up to 40% when compared with round-robin placement (the standard MPI placement) and 300% against random placement. The decrease of performance is never greater than 2%.

7 Conclusion

Parallel applications tend to use a growing number of computational resources connected via complex communication schemes that naturally diverge from the underlying network topology. Optimizing the performance of applications requires any mismatch between the application communication pattern and the network topology to be identified, and this demands a precise mapping of all data exchanges between the application processes.

In this paper we proposed a new monitoring framework to consistently track all types of data exchanges in MPI applications. We implemented the tool as a set of modular components in OPEN MPI that allow fast and flexible low-level monitoring (with collective operation decomposed to their point-to-point expression) of all types of communications supported by the MPI-3 standard (including one-sided communications and I/O). We also provided an API based on the MPI Tool Information Interface standard for applications to monitor their state dynamically, with a focus on only the critical portions of the code. The basic use of this tool does not require any change in the application nor any special compilation flag. The data gathered can be provided at different granularities, either as communication matrices or as histograms of message sizes. Another significant feature of this tool is that it leaves the PMPI interface available

for other usages, allowing additional monitoring of the application using more traditional tools.

Microbenchmarks show that the overhead is minimal for intra-node communications (over shared memory) and barely noticeable for large messages or distributed memory. After being applied to real applications, the overhead remain hardly visible (at most, a few percentage points). Having such a precise and flexible monitoring tool opens the door to dynamic process placement strategies and could lead to highly efficient process placement strategies. Experiments show that this tool enables large gain for dynamic or static cases. The fact that the monitoring records the communication after collective decomposition into point-to-points allows optimizations that were not otherwise possible.

Acknowledgments

This work is partially funded under the ITEA3 COLOC project #13024, and by the USA NSF grant #1339820. 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. Barrett, B., Squyres, J.M., Lumsdaine, A., Graham, R.L., Bosilca, G.: Analysis of the Component Architecture Overhead in Open MPI. In: Proceedings, 12th European PVM/MPI Users' Group Meeting. Sorrento, Italy (September 2005)
2. Barrett, R.F., Vaughan, C.T., Heroux, M.A.: Minighost: a miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Sandia National Laboratories, Tech. Rep. SAND2011-5294832 (2011)
3. Bosilca, G., Foyer, C., Jeannot, E., Mercier, G., Papauré, G.: Online Dynamic Monitoring of MPI Communications: Scientific User and Developper Guide. Research Report RR-9038, Inria Bordeaux Sud-Ouest (Mar 2017), https://hal.inria.fr/hal-01485243
4. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in hpc applications. In: Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on. pp. 180–186. IEEE (2010)
5. Brown, K.A., Domke, J., Matsuoka, S.: Tracing Data Movements Within MPI Collectives. In: Proceedings of the 21st European MPI Users' Group Meeting. pp. 117:117–117:118. EuroMPI/ASIA '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2642769.2642789
6. Cores, I., Gonzalez, P., Jeannot, E., Martín, M., Rodriguez, G.: 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 (Jun 2016), to appear
7. Forum, M.P.I.: MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/ (September 2012)

8. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. pp. 97–104. Budapest, Hungary (September 2004)
9. Hoefler, T., Jeannot, E., Mercier, G.: An overview of topology mapping algorithms and techniques in high-performance computing. High-Performance Computing on Complex Environments pp. 73–94 (2014)
10. Janssen, C.L., Adalsteinsson, H., Cranford, S., Kenny, J.P., Pinar, A., Evensky, D.A., Mayo, J.: A simulator for large-scale parallel computer architectures. Technology Integration Advancements in Distributed Systems and Computing 179 (2012)
11. Jeannot, E., Mercier, G., Tessier, F.: Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. IEEE Transactions on Parallel and Distributed Systems 25(4), 993–1002 (Apr 2014)
12. Keller, R., Bosilca, G., Fagg, G., Resch, M., Dongarra, J.J.: Implementation and Usage of the PERUSE-Interface in Open MPI, pp. 347–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11846802_48
13. Knüpfer, A., et al.: Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir, pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
14. Open MPI Master Repository (2017), https://github.com/open-mpi/ompi
15. Vetter, J.S., McCracken, M.O.: Statistical scalability analysis of communication operations in distributed applications. In: ACM SIGPLAN Notices. vol. 36, pp. 123–132. ACM (2001)