Using Simulation to Validate Performance of MPI(-IO) Implementations

Julian M. Kunkel

University of Hamburg
Bundesstraße 45a
20146 Hamburg
julian.martin.kunkel@informatik.uni-hamburg.de

Abstract. Parallel file systems and MPI implementations aim to exploit available hardware resources in order to achieve optimal performance. Since performance is influenced by many hardware and software factors, achieving optimal performance is a daunting task. For these reasons, optimized communication and I/O algorithms are still subject to research. While complexity of collective MPI operations is discussed in literature sometimes, theoretic assessment of the measurements is de facto non-existent. Instead, conducted analysis is typically limited to performance comparisons to previous algorithms.
However, observable performance is not only determined by the quality of an algorithm. At run-time performance could be degraded due to unexpected implementation issues and triggered hardware and software exceptions. By applying a model that resembles the system, simulation allows us to estimate the performance. With this approach, the non-function requirement for performance of an implementation can be validated and run-time inefficiencies can be localized.
In this paper we demonstrate how simulation can be applied to assess observed performance of collective MPI calls and parallel IO. PIOsimHD, an event-driven simulator, is applied to validate observed performance on our 10 node cluster. The simulator replays recorded application activity and point-to-point operations of collective operations. It also offers the option to record trace files for visual comparison to recorded behavior. With the innovative introspection into behavior, several bottlenecks in system and implementation are localized.

Keywords: Simulation, MPI-IO, Performance evaluation

1 Introduction

Parallel file systems and MPI implementations aim to achieve optimal performance on all systems. The performance of communication and IO certainly depends on the hardware characteristics – the specific hardware configuration limits potential network throughput, computation power and available memory bandwidth. The selection of the optimal algorithm depends on the hardware characteristics, the network topology and application behavior. From a library's

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-642-38750-0_14

point of view, optimization can be done based on the parameters provided by the programmer. Typically, this includes the *memory datatype*, the *communicator*, *target/source rank* (for all-to-one or one-to-all operations), and the actual *amount of data* shipped with the call. Additionally, the process placement across the hardware resources is important. Therefore, MPI implementations offer several algorithms and they realize a rich variety of optimization strategies to gear algorithms towards the given system.

This adaption leads to better exploitation of available hardware resources and, ultimately, to better performance and thus application runtime. However, the interplay of hardware optimizations such as caches, the software optimizations offered by operating system and intermediate libraries result in complex behavior which make the selection of an optimal algorithm hard. With Open MPI, MPICH2, MVAPICH2, this complexity also leads to a diverse landscape of open source MPI implementations. Also, vendors and integrators offer their own proprietary solution.

Up to now, effectiveness of alternative algorithms is mainly demonstrated by comparing measured performance with performance of existing algorithms. However, observable performance is not only determined by the quality of an algorithm. At run-time performance could be degraded due to unexpected implementation issues and triggered hardware and software exceptions. Visualizing the real system activity helps analyzing the behavior and localizing regions that require most of the execution time. However, determining whether recorded activity is conducted optimally is not possible because it depends on platform and optimizations.

In this paper we propose a simulation driven systematical validation of MPI-IO performance. By applying a model that resembles the system, simulation approximates performance and, thus estimates performance of algorithms. The main contributions of the paper are 1) a performance study motivating integrated performance testing of MPI and 2) a discussion of a feasible implementation of such an approach. Without the power of simulation, many performance bottlenecks could not be found in our cluster.

This paper is organized as follows: In Section 2 an overview of the state-of-the-art is presented. In Section 3 the benefits of simulation to evaluate performance of MPI-IO implementations are described. A brief introduction to the simulator and the underlying hardware and software models is given in Section 4. Several experiments in Section 5 illustrate how theory aids to localize bottlenecks and to check for correctness. While the model is developed manually, this process could be automated to perform these steps automatically. Section 6 concludes the paper.

2 State of the art

Many algorithms were proposed to optimize collective communication. They are either directly implemented in one of the MPI implementations, e.g. [1], or provided as an external library such as STAR-MPI [2] or Magpie [3].

To our knowledge, MPI libraries lack self-awareness. There is no implementation which takes the hardware characteristics into account while determining an algorithm for collective communication or I/O. While middleware implementations ship with tests for functionality they do not automatically detect hardware characteristics. Instead, a library is shipped with empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Tuning of these parameters is time consuming, thus, the defaults might achieve only a fraction of theoretical performance. Many of these parameters exist, for example, in Open MPI, the *Modular Component Architecture (MCA)* lists more than 250 parameters on a COST Beowulf cluster.

Although MPI implementations are not considering hardware characteristics, they have become increasingly aware of the communication topology and try to utilize shared-memory communication if possible which leads to SMP-aware collective algorithms. For example, the CARTO framework of Open MPI provides topological information.

As algorithms must be handcrafted towards the system – for instance for a BlueGene [4] – one major problem is to pick the best algorithm for a system. Several approaches have been developed that assist in determining the best algorithm and MPI configuration. The *Abstract Data and Communication Library (ADCL)* [5] uses historic knowledge during the application run. ADCL assumes a program performs operations iteratively – in the first few iterations ADCL evaluates a set of MPI functions to determine which one is best suited for the given problem, then this function is applied to subsequent invocations. Compared to ADCL, the *Self-Tuned Adaptive Routines for MPI Collective Operations (STAR-MPI)* provides a rich set of MPI implementations for collective operations by itself [2], for instance a set of 13 algorithms is supplied for `MPI_Alltoall()`.

For parallel I/O, the problem becomes even more complex since it depends on communication. For example, non-contiguous operations and collective calls have been defined in MPI-IO which lead to a classification of data access into four levels [6]. These levels are characterized by two orthogonal aspects: contiguous vs. non-contiguous data access, and independent vs. collective calls. Depending on the level, a different set of optimizations can be thought of, for example, two-phase I/O and multiphase-collective I/O [7] aim to improve collective non-contiguous access. An adaptive approach is introduced in [8], which automatically sets hints for collective I/O based on the access pattern, topology and the characteristics of the underlying file system.

Typically, evaluation of improved algorithms is conducted by comparing performance of existing algorithms with the new algorithm. This includes improvements in the communication submodules of MPI, e.g., in Nemesis [9], or completely new MPI implementations such as Open MPI[10]. Similarly, parallel I/O research demonstrates improvements by comparing observed performance. In most cases, a baseline of expected performance is not provided. This is mainly due to the complexity of determining these baselines. There are a few exceptions to this general observation, but theoretic considerations are restricted to simple cases. For example, in [11] upper bounds for performance are provided

based on the component throughput and latency. In many cases, very coarse estimates could be computed even for complex behavior, but these are not very tight. Development of an adequate mathematical equation for complex behavior is nontrivial. Simulation of the behavior is much easier.

There are many simulators for distributed systems, most focus on communication routines, for example, the Structural Simulation Toolkit (SST) [12], LogGOPSim [13] and Dimemas [14].

Some simulators can replay previously recorded MPI(-IO) activity inside the virtual environment. For example, trace information is altered in [15], then an MPI program replays the modified trace on the original machine, which automatically enforces causality between dependencies among processors. While this approach scales well, it is not possible to simulate other hardware configurations or to gain insights into MPI. LogGOPSim is a simulator for a class of analytical models of the $\log_x$P family [16]. It supports a simple network collision model. Dimemas reads trace files and applies an analytical model to individual and collective communication. Network collisions are modeled in an abstract way by limiting the maximum throughput which can occur at a given time over a central network infrastructure.

CODES [17] and PIOsimHD [18] target parallel I/O. Built on top of the Rensselaer Optimistic Simulation System (ROSS), CODES supports parallel discrete-event simulation of queuing models. It has been successfully applied to study the role of burst buffers in systems with 100k application processes and 120 PVFS file servers. In contrast to the introduced systems, PIOsimHD covers parallel I/O and allows replaying of recorded MPI traces on a high level of abstraction – commands are implemented in the simulator to react on system conditions. The event-driven nature of PIOsimHD allows localizing of network congestion and to evaluate I/O optimization on client, server or disk side. For example, an analysis of several I/O schedulers and collective I/O variants has been performed using PIOsimHD in [19]. While simulation has been used to evaluate what-if scenarios, to our knowledge it has not been used to systematically validate measured MPI-IO performance in order to identify hidden bottlenecks. Instead, complex simulation parameters are introduced and fitted to meet observations.

3 Using Simulation to Validate MPI-IO Performance

Simulation aids in validating MPI-IO performance in two ways: First, by comparing observed run-time and theoretical run-time estimate quantitatively, implementation issues and unexpected bottlenecks can be identified. This is especially useful for validation of complex operations such as collective operations. Second, a complex sequence of operations, such as the behavior of real applications, can be inspected visually and qualitatively compared to a simulated run of the application. Therewith, unexpected behavior of individual operations can be identified and assessed. For both scenarios, simulation parameters can be varied to study the impact of certain hardware characteristics, for example by turning off computation.

We propose systematic validation of performance achieved with MPI-IO functions using simulation. Imagine an MPI-IO implementation which does not only run functional tests after installation but also performance benchmarks and assesses the results. First, it could run simple point-to-point benchmarks and create a system model. Then complex benchmarks could be run and their performance could be assessed automatically for soundness. For example, a bi-directional message transfer of a large message with `MPI_Sendrecv()` should require the same time as a unidirectional communication. The basic system model is of interest for performance optimization by itself, as it illustrates expected communication overhead. An administrator could compare these performance characteristics with micro-benchmarks to ensure that the basic communication routines extract the performance as anticipated.

This becomes more interesting for complex operations, as their performance cannot be understood easily. If expected and observed performance diverge too much, the system should raise a warning. Then the administrator has a starting point for investigating performance degradation which could be due to MPI-internal overhead, kernel, or external libraries. A result telling the administrator MPI performance behaves as anticipated is valuable too, as it reduces the chance to experience unexpected performance loss in production. Finally, by determining hardware characteristics at installation time, these values could be used at run-time to determine well-suited collective algorithms without manual intervention and ultimately allow a self-aware MPI implementation.

To conduct such a validation, it is mandatory for the simulation to mimic the expected behavior of the experimental system. Thus, basic model parameters should have similar characteristics as the real system. Since simulation should help identifying inefficiencies, it is not constructive to mimic the real system perfectly as we could not spot differences and thus unexpected behavior. In both cases, it helps if activities of an application can be recorded and replayed by the simulator because this reduces the effort to validate the execution. By this means theoretically any MPI benchmark can be run and its results can be easily compared to our expectations. For later analysis, it is also useful if the simulator can create trace files which can be compared to real traces.

4 PIOsimHD

The goal of the sequential discrete event simulator PIOsimHD is to assist MPI-IO research and to foster understanding of performance factors in clusters. PIOsimHD performs a discrete event simulation and, if requested, stores the processing as trace files. It can also read activity from recorded trace files. HDTrace is an experimental tracing environment which also provides tools to instrument existing applications and to record activity of PVFS and MPI internal communication. Simulation results can then be visualized by Sunshot, which enables a comparison of the recorded process and file system activities and simulation results.

PIOsimHD offers a hardware model which reflects the common sense of a cluster computer: Several compute resources (CPUs) are hosted on a node which is connected to one or several networks via a network interface (NI). Arbitrary network topologies can be created. On each node one I/O server can be placed, each holding a cache layer which schedules operations, and an I/O-subsystem.

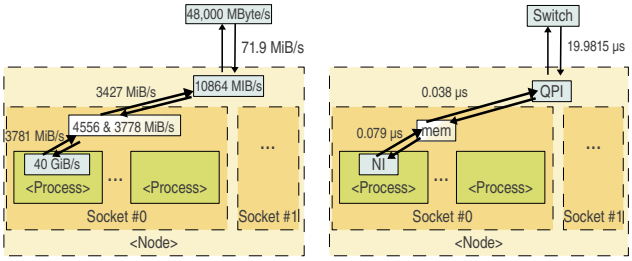
To cope with several levels of abstraction, a component can have several implementations. Component implementations are parameterized with certain characteristics. Usually, characteristics are provided in vendor specifications or obtained by benchmarking the existing system. The level of detail of the cluster hardware covers basic information describing an Ethernet based cluster. With the amount of memory and number of CPUs, a node offers shared resources for hosted processes. Each CPU processes a fixed number of instructions per second. The memory is used for caching I/O on the server side.

The simulator permits the user to create arbitrary network graphs representing store-and-forward systems. Network edges have a latency and a transfer rate. Network nodes have a maximum bandwidth to relay data. With the help of network components, memory access of communication can be simulated which permits modeling of local communication. A special node adds the local throughput as an additional parameter, which is used when two direct neighboring components of this network node exchange data. An example model of a dual-socket node is given in Figure 1. In this figure, throughput and latency of all network components are given as observed on our Intel Westmere cluster consisting of 10 nodes.

To utilize the network well, a network flow model was designed in which messages are fragmented into packets of a maximum size, which flow from source to target in a stream. When data is transferred from one component to another, the transmission of incoming data flows is continued. The maximum number of packets in flight for every stream is limited by the bandwidth-delay-product of the given link. While many concepts can be found in real systems, the data flow differs because this concept achieves the highest utilization of all network components for all streams, and it does not throw packets away.

A hard-disk as an *I/O-subsystem* is modeled by a sequential transfer rate, an average access time, track-to-track-seek time and RPM. Depending on the distance to the last byte accessed within a file, a disk will either perform no seek, will seek to the neighboring track or will apply the average access time. Access to other files always enforces an average seek.

An abstract parallel file system defines the interaction between client and server. The abstract model describes many parallel file systems because they work similarly. Clients and servers interact in a similar fashion to the PVFS model, but the concept is universal to most parallel file systems: File data is partitioned among all servers as defined by a selectable distribution function. To write data, a client requests a write operation from the server and then starts to transfer all data. File sizes are updated once a write operation finishes. Metadata operations are currently not considered since these depend on the specific file system. More details can be found in [18].



(a) Throughput. **(b)** Latency of edges.

Fig. 1. Network topology model for the working groups cluster. Throughput of intra-socket communication is slightly higher

4.1 Experiments

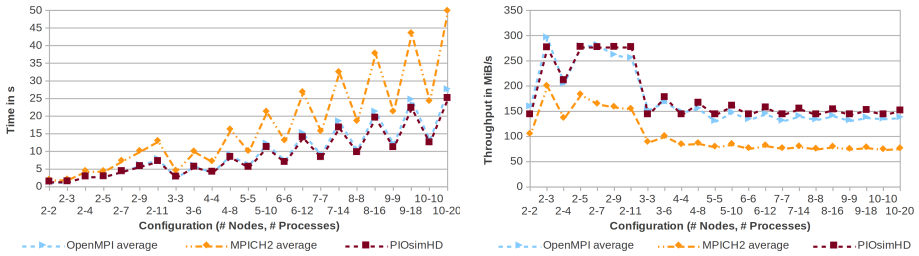
During the validation of the simulator, several unexpected bottlenecks in hardware and software could be identified. An excerpt of interesting results demonstrating the benefit of validating the soundness of observed MPI-IO performance is given in the following. Measurements are executed on our 10 node Ubuntu cluster; interconnected via. Gigabit-Ethernet, each node is equipped with two *Intel Xeon 5650* processors providing 12 cores for the experiments and 12 GByte of memory. Used software versions are: *Open MPI* 1.5.3, *MPICH2* 1.3.1, and Orangefs-2.8.3. The conducted validation is described in detail in [18].

To conduct complex validation runs, recorded activities are replayed in the simulator. Thus the same sequence of compute, network and I/O activity is executed. While a compute job takes the exact time as recorded in a validation run, execution time of parallel I/O and communication is computed by the simulator using the virtual file system and network models.

Parameterization To parameterize the simulator for a validation run, the hardware characteristics must be determined. Throughput and latencies for the network links have been measured using MPI point-to-point operations (values are annotated in Fig. 1[1]). An HDD is characterized by a track-to-track seek time of 1.1 ms, an average seek time of 9 ms, a sequential transfer rate of 96 MiB/s and 7200 RPM. The hardware model uses the fast seek time for accesses to the same file to an offset which is within a window of 1 MiB to the last access.

Communication Before analysis of collective results is conducted, a simple example of a suboptimal point-to-point communication pattern is given. In this experiment, each process exchanges a 100 MiB message with Rank 0 by calling `MPI_Sendrecv()` – the whole experiment is repeated 9 times. The average measured time is plotted for a variable number of nodes and processes in Figure 2a

[1] These values have been validated with network benchmarks such as Iperf. The issues with the network are discussed in Section 4.1.



(a) Time **(b)** Aggregated throughput

Fig. 2. Sequential data exchange of Rank 0 and all other processes using `MPI_Sendrecv()` for several configurations and 100 MiB messages

and the achieved throughput is illustrated in Figure 2b. It can be observed that PIOsimHD approximates Open MPI very well, but MPICH2 needs more time than anticipated by simulation. Without a simulation tool, the performance could be approximated manually, for example, using the following few considerations: Since our network allows bi-directional communication, about 140 MiB/s can be achieved over the single node hosting Rank 0, this performance can be seen for many configurations. However, already this simple pattern shows that computation is not this simple. Due to shared memory intra-node communication, processes hosted by the same node achieve much higher throughput. This can be observed in Figure 2b for Open MPI and PIOsimHD. Thus, while a manual computation of the expected throughput is possible, it is non-trivial. By comparing simulation results with the results of MPICH2, the unexpected slow-down become visible and could be subject for further investigation[2].

Examples for collective communication are given in Figure 3. Experiments with 10 KiB message transfers are repeated 10,000 times and for larger messages at least 9 times. Figures show the quartiles for the small messages to account for deviation, and minimum and average values for larger messages (typically, the slowest time is much higher than the average). In Figure 3a and Figure 3b it can be seen that `MPI_Allgather()` is well approximated by PIOsimHD, and thus observed performance is consistent with our theoretic expectations. In comparison, the intra-node algorithm of Open MPI achieves a better performance for configurations with 2 nodes. For small messages, Figure 3a shows much better times for configurations 4-8 and 8-16 than for other configurations. Without the simulation result, one question might be whether this behavior is due to the system's characteristics. Since the simulator executes the exact same communication pattern and results in similar performance, we can conclude the communication algorithm changes and leads to this behavior[3].

[2] Actually, our version of MPICH2 extracted the same performance numbers as for uni-directional communication.

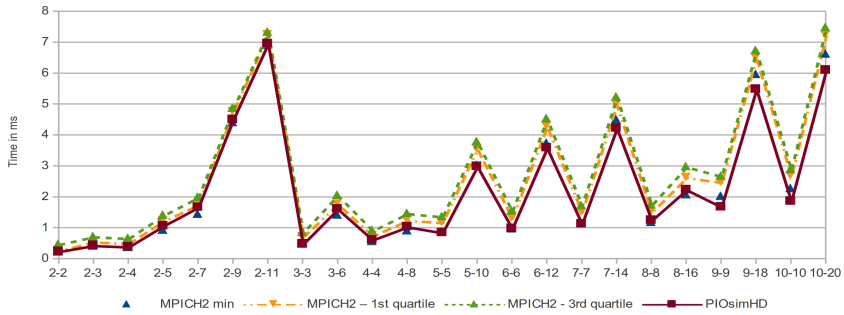
[3] Actually, the trace files can be inspected demonstrating correctness of this theory.

Similarly, an analysis of `MPI_Allreduce()` shows interesting behavior (see Figure 3c). While the measurements of a single configuration fluctuate much more, the simulation still recreates the overall pattern. The complexity of the analysis can be observed for larger messages in Figure 3d. While Open MPI shows a completely different behavior than MPICH2, none of the algorithms is optimal in all cases. In this example, PIOsimHD estimates better times than MPICH2, which reasons should be analyzed further. Thanks to simulation, the impact of certain factors can be studied. For example, the impact of computation has been investigated – as it turns out, the required time only improves slightly when recorded computation is not simulated.

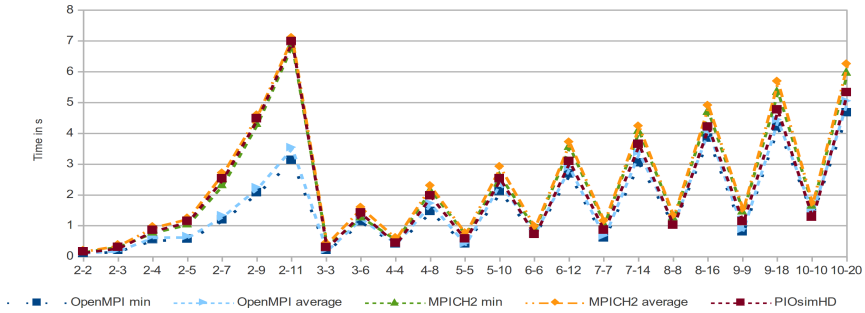
Visual inspection of application behavior Finally, a run of our Jacobi PDE solver is evaluated with the simulation. While the simulation recreates the overall pattern quite well (refer to [18]), communication in the final phase takes longer than anticipated. Timelines of the cleanup phase are given in Figure 4. Rank 0 receives selected lines of the PDE from all processes (users might inspect them to validate the run). Several data transfers need 0.2 ms although the sender and receiver is ready. Since only 400 KiB of data is transferred, a performance of only 2 MiB/s is achieved. This problem has been found by first comparing trace profiles, then a visual inspection of the individual communications has been performed. During the iterations, message exchange behaves as anticipated by the simulation. Without theoretic considerations, an assessment of the performance in terms of overall achieved time and the individual operations would be difficult. However, estimating run-time for an complex application is cumbersome.

Parallel I/O With Parabench [20] the four levels of access have been investigated for several setups. Results for independent contiguous reads are given for a variable number of clients and servers in Figure 5. In these experiments, each client reads 100 KiB (and 100 MiB) records – a total of 1 GiB of data is accessed per client. Client records are distributed in round-robin over the logical file, i.e. the first record of the file is accessed by Rank 0, then by Rank 1 and so forth. Data is stored on tmpfs, thus there is no slow I/O device involved and performance is expected to be limited by the network. The simulator approximates performance for 100 MiB records well as shown in Figure 5a. However, it overestimates performance of 100 KiB records significantly as illustrated in Figure 5b. Since the model uses measured latency and bandwidth as characteristics, these hardware factors cannot be the reason. A detailed analysis revealed timing effects in the real system leading to congestion of individual servers while most servers are idle. Once requests of multiple clients are pending on a single server, all clients must wait for data stored on this server but since the server multiplexes the NIC, data transfers of all responses take longer. With a slight variation in the simulation characteristics, these effects can also be reproduced in-silicon.

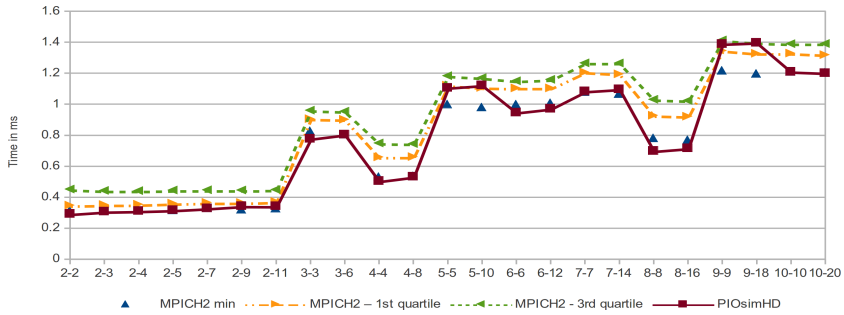
One experiment was conducted that changed the packet size of the store-and-forward network. The simulated network relays packets of 100 KiB size; a lower packet size of 10 KiB can be chosen, which improves concurrency of the components and the theoretic performance.



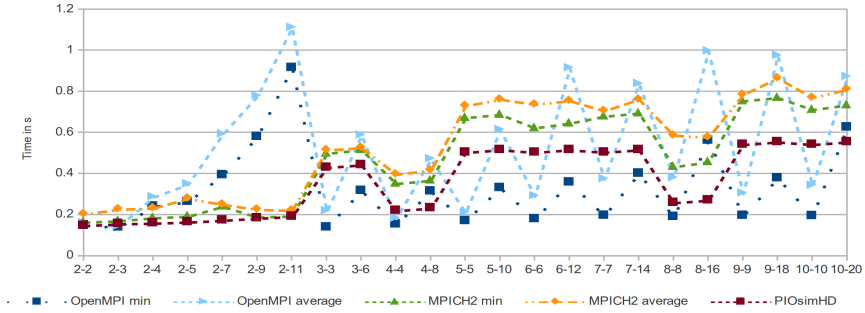
(a) `MPI_Allgather()`, 10 KiB of data



(b) `MPI_Allgather()`, 10 MiB of data

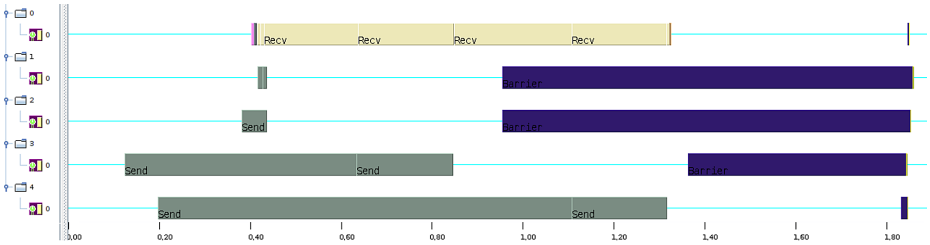


(c) `MPI_Allreduce()`, 10 KiB of data

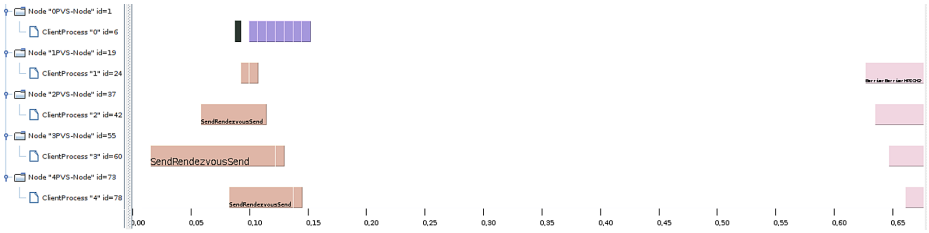


(d) `MPI_Allreduce()`, 10 MiB of data

Fig. 3. Simulation of inter-node collective communication for a variety of configurations (# of nodes, # of processes)



(a) Observed



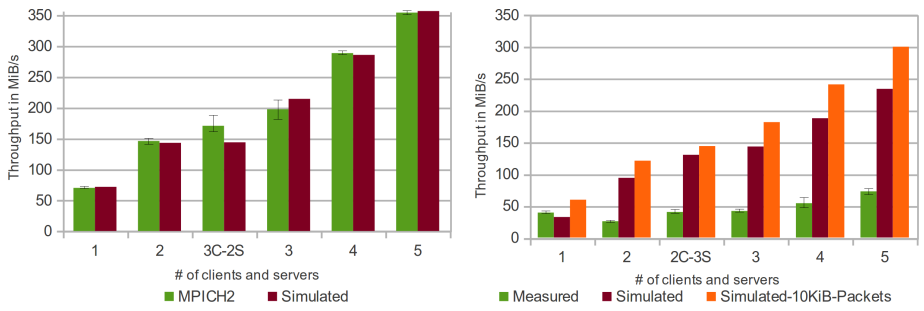
(b) Simulated

Fig. 4. Final phase of a Jacobi PDE solver – traces of observation and simulation

During the validation of the Jacobi PDE, an inefficiency in the PVFS module of MPI-IO could be identified and resolved. The PDE outputs the matrix diagonal for later inspection as a sequential 64 KiB data block. Internally, a memory datatype is used to address the matrix diagonal in one write call. However, the write takes about 70 ms while the simulator estimates 2 ms[4]. Using HDTrace, a detailed analysis of client and server activities has been made revealing that PVFS split the 64 KiB block into 512 bytes requests. The reason is the handling of non-contiguous datatypes by ROMIO. Since ROMIO does not use an additional buffer to store data, every non-contiguous region in memory is normally accessed with an individual operation. With ListIO, PVFS supports encapsulating to 64 non-contiguous operations in one request. Since the matrix diagonals are 8 byte, 512 byte requests are created. For the application, the problem could be fixed by setting the undocumented hint *romio_pvfs2_listio_write* which enables handling of memory and file datatypes in ROMIO. By setting the hint, the average time for a single write is reduced to 3.4 ms which is close to the estimation.

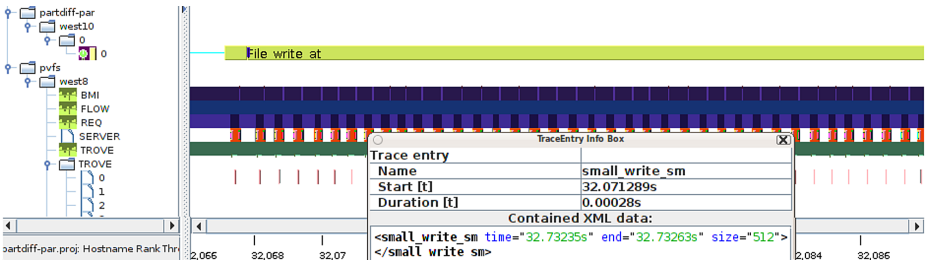
A screenshot of the obtained traces for one iteration of a write phase are shown in Figure 6. In the default operation, server and trove timeline show many small operations. With the applied hint, one large write operation can be seen in the timeline (the additional small write operation left updates the header of the file in both cases).

[4] The actual time depends on the current activity on the accessed servers.

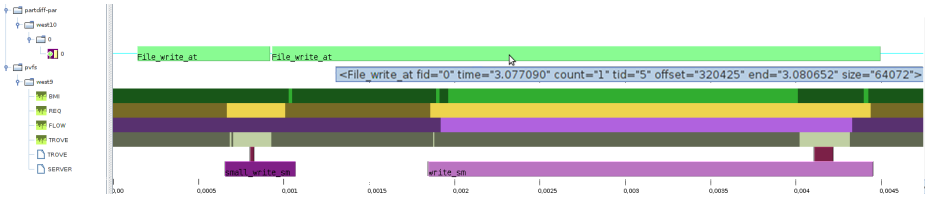


(a) 100 MiB records **(b)** 100 KiB records

Fig. 5. Performance of independent contiguous I/O with a variable number of clients and servers. Data is stored on tmpfs



(a) Default operation; details of one request are shown



(b) With supplied hint

Fig. 6. Screenshot of the PDE's data exchange for one client and one server

Difficulties to identify causes of performance degradation The intention of performance analysis is not only to localize but to resolve the causes of potential performance degradation. However, as it turned out the identification of triggered issues is non-trivial. We invested several month trying to localize the general issues in our network stack and to identify and fix the performance issue with PVFS. The latter issue could only be analyzed in detail thanks to the detailed tracing mechanisms of HDTrace. The fix involved communication with the developers, but also detailed code inspection of PVFS and MPI-IO.

Unfortunately, debugging of the network issue showed little success. As it turned out, operations sometimes take much longer than expected (10 times the average time, an overhead of about 0.2 s), and there was the network limitation of 67 MiB/s. To identify the reason for the performance degradation, several regulating knobs have been evaluated on our production system: TCP-tuning, alternative MPI implementations (MPICH2 and Open MPI), different Linux kernel and also CentOS as an alternative distribution. Also, existing network tools and benchmarks have been used to validate the observed performance.

The insight of all this effort is: Performance could only be improved a little by testing many alternative sets of TCP-options. With newer kernel versions, the throughput improved to 71 MiB/s and finally with kernel 3.5 to 117 MiB/s. Interestingly, by using CentOS, the variance of network packets stabilizes and throughput is good. As these issues disappeared by using newer kernels and another distribution, a detailed analysis of involved libraries and kernel is required.

5 Summary & Conclusions

In this paper, we describe the benefit of using simulation for validating performance of MPI-IO implementations. To estimate performance, the simulator executes the activity of a parallel program on a virtual cluster with similar characteristics as the experimental system. In many cases, a very good match to observations is achieved, which validate that system and implementation behavior is consistent. However, an excerpt of experiments is given in which performance gaps become visible. For example, observed performance of `MPI_Sendrecv()` and `MPI_Allreduce()` fall behind the expectations which indicate a demand for further investigation. Automatic performance assessment could be an integral part in a test-suite – a simulator complements existing benchmarks by creating run-time estimates. Shipped with MPI implementations, these tests would spot unexpected performance inefficiencies directly. While we tried to identify the causes of the network performance degradation in kernel, libraries and system hardware, we did not succeed so far. Nevertheless, without systematic testing we would be unaware of these inefficiencies, showing the necessity of automatic tools and the involvement of developers.

Suboptimal behavior during parallel I/O has been investigated but it is much more complex than collective I/O. For example, timing effects may overload individual servers. Finally, a performance problem in an application could be identified and with the help of advanced tracing of client and server behavior,

the reasons could be identified and fixed by applying an MPI hint. Overall, the virtual laboratory of HDTrace allows us to identify inefficiencies and to study behavior of communication and file system, to conduct research on new algorithms, and to evaluate future systems. In the future, we will try to build the envisioned system for automatic validation of MPI-IO behavior.

Acknowledgements

I want to thank the PVFS development team to help resolving the performance degradation of writing the non-contiguous matrix diagonal.

References

1. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. International Journal of High Performance Computing Applications **19**(1) (February 2005) 49–66
2. Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In: Proceedings of the 20th annual international conference on Supercomputing. ICS, New York, NY, USA, ACM (2006) 199–208
3. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP, New York, NY, USA, ACM (1999) 131–140
4. Miller, S., Kendall, R.: Implementing Optimized MPI Collective Communication Routines on the IBM BlueGene/L Supercomputer. Technical report, Iowa State University (2005)
5. Gabriel, E., Huang, S.: Runtime Optimization of Application Level Communication Patterns. In: International Parallel & Distributed Processing Symposium. IPDPS, IEEE (2007) 1–8
6. Thakur, R., Gropp, W., Lusk, E.: Optimizing Noncontiguous Accesses in MPI-IO. Parallel Computing **28** (2002) 83–105
7. Singh, D.E., Isaila, F., Pichel, J.C., Carretero, J.: A Collective I/O Implementation Based on Inspector–Executor Paradigm. The Journal of Supercomputing **47**(1) (2009) 53–75
8. Worringen, J.: Self-adaptive Hints for Collective I/O. In: PVM/MPI. (2006)
9. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on. Volume 1. (2006) 10–pp
10. Graham, R., Shipman, G., Barrett, B., Castain, R., Bosilca, G., Lumsdaine, A.: Open MPI: A high-performance, heterogeneous MPI. In: Cluster Computing, 2006 IEEE International Conference on. (2006) 1–9
11. Kunkel, J., Ludwig, T.: Performance Evaluation of the PVFS2 Architecture. In: PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Euromicro (2007) 509–516
12. Rodrigues, A.F., Murphy, R.C., Kogge, P., Underwood, K.D.: The Structural Simulation Toolkit: Exploring Novel Architectures. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. SC, New York, NY, USA, ACM (2006)

13. Hoefler, T., Schneider, T., Lumsdaine, A.: LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing. HPDC, New York, NY, USA, ACM (2010) 597–604
14. Girona, S., Labarta, J., Badia, R.M.: Validation of Dimemas Communication Model for MPI Collective Operations. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag (2000) 39–46
15. Hermanns, M.A., Geimer, M., Wolf, F., Wylie, B.J.N.: Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. (2009) 78–84
16. Tu, B., Fan, J., Zhan, J., Zhao, X.: Accurate Analytical Models for Message Passing on Multi-core Clusters. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing. (2009) 133–139
17. Cope, J., Liu, N., Lang, S., Carns, P., Carothers, C., Ross, R.: CODES: Enabling Co-design of Multilayer Exascale Storage Architectures. In: Proceedings of the Workshop on Emerging Supercomputing Technologies 2011. (2011)
18. Kunkel, J.: Simulating Parallel Programs on Application and System Level. Computer Science – Research and Development (online first) (May 2012)
19. Kuhn, M., Kunkel, J., Ludwig, T.: Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations. In: 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing. (2012) 562–566
20. Mordvinova, O., Runz, D., Kunkel, J., Ludwig, T.: I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. Procedia Computer Science (2010) 2119–2128