Noname manuscript No.
(will be inserted by the editor)

Simulating Parallel Programs on Application and System Level

Julian M. Kunkel

Received: date / Accepted: date

Abstract Understanding the measured performance of parallel applications in real systems is difficult – with the aim to utilize the resources available, optimizations deployed in hardware and software layers build up to complex systems. However, in order to identify bottlenecks the performance must be assessed.

This paper introduces *PIOsimHD*, an event-driven simulator for MPI-IO applications and the underlying (heterogeneous) cluster computers. With the help of the simulator runs of MPI-IO applications can be conducted in-silico; this includes detailed simulation of collective communication patterns as well as simulation of parallel I/O. The simulation estimates upper bounds for expected performance and helps assessing observed performance.

Together with *HDTrace*, an environment which allows tracing the behavior of MPI programs and internals of MPI and PVFS, PIOsimHD enables us to localize inefficiencies, to conduct research on optimizations for communication algorithms, and to evaluate arbitrary and future systems. In this paper the simulator is introduced and an excerpt of the conducted validation is presented, which demonstrates the accuracy of the models for our cluster.

Keywords Simulation · Tracing · MPI-IO

1 Introduction

Understanding hardware and software performance is the foundation for optimizing application and system behavior. In order to gain insight into system and application behavior either all activity can be recorded and analyzed, or modeled and simulated. Visualizing the real system activity helps analyzing the behavior and to localize regions that require most of the execution time. However, in many cases the reason for the observation cannot be identified. Further, determining whether a recorded activity is conducted optimally is not possible because it depends on the system and hardware configuration.

Efficiency depends on the system characteristics – the characteristics of basic components: network, storage and compute nodes – the topologies of all interconnects, the system's algorithms for communication, I/O and the configuration done by administrator and user.

It is not an easy task to understand the interplay between all hardware characteristics. Unfortunately, mechanisms designed to optimize the system make it even harder to assess achieved performance and relate it to the system's capability: An MPI implementation provides several collective algorithms which should achieve good performance on the underlying hardware. Further, collective I/O can improve performance by manipulating and scheduling the I/O requests of all processes.

Therefore, theoretic models of the system behavior are needed to assess the observed behavior. Modeling and simulation approximates performance and, therefore, estimates performance gains of optimizations. With the help of simulation the application behavior could also be projected to an extended or future system before such a new system is built. During the design of the new system such an evaluation can guide development to avoid later disappointments.

This paper is organized as follows: In Section 2 the state-of-the-art in simulation of compute clusters is presented. More details about the simulator and the under-

Thanks to Nathanel Hübbe for reviewing this paper.

Julian M. Kunkel
University of Hamburg
E-mail: kunkel@informatik.uni-hamburg.de

lying hardware and software models are given in Section 3. In Section 4 the performance of the simulator is assessed. The validation methodology of the simulation model is presented in Section 5. Section 6 concludes the paper and presents possible future analysis which can be conducted with the simulator.

2 State of the art

Recording of trace information in post-mortem performance optimization is state-of-the-art to localize performance bottlenecks. Popular performance analysis tools are Tau [9], Vampir [5] and Scalasca [1]. However, MPI internals and I/O activity are usually hidden from available trace environments. With *HDTrace* collective operations can be traced, as well as client and server activity in the PVFS file system [7]. In addition to the improved possibility to localize the cause of an observation, HDTrace allows assessing of simulation results by comparing them to observations on real systems. This allows us to validate, and to adapt and refine the models.

There are many simulators for distributed systems; a few that are closely related to the work of this paper are introduced in the following. In [3] trace information is altered, 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. Analytical models for message passing like $\log_x$P [10] allow to predict performance, however, neither network collisions nor I/O is modeled. LogGOPSim [4] is a simulator for this class of models. It supports a simple network collision model. Dimemas [2], a simulator, 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.

The Structural Simulation Toolkit (SST) [8] aims to provide a parallel event-driven simulator to simulate memory, computation, network and I/O at arbitrary levels of abstraction from instruction level to analytical models. To accomplish this goal it provides a modular framework and interfaces with many existing simulators. Besides performance, SST estimates parameters for power, energy, area, costs and reliability. Currently, no parallel I/O is simulated. To simulate single disk activity DiskSim is adapted and incorporated into their source tree.

In contrast to the introduced systems, *PIOsimHD* covers parallel I/O and allows replay 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 localization of network congestion and to evaluate I/O optimization on client, server or disk side. With its help an analysis of several I/O schedulers and collective I/O variants has already been performed in [6]. However, that paper lacked a detailed description of the simulator and a validation of the hardware and software models.

3 PIOsimHD

PIOsimHD is a sequential discrete event simulator written in Java. Its goal is to assist MPI(-IO) research and to foster understanding of performance factors in clusters. Arbitrary network topologies can be created and relevant characteristics of the components can be adjusted freely. The specification of the model can be either explicitly programmed or read from an XML file.

Internally, a discrete event simulator processes events which are stored in a queue and sorted by start time, a global clock for the model time is incremented according to the start time of the next event [11]. An event itself can create new (future) events. Output from the simulation can be stored in trace files and compared to the original run.

Due to limited space, underlying model concepts are only explained briefly.

3.1 Hardware Model

The hardware model 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). On each node one I/O server can be placed. Each holds a cache layer, which schedules operations, and an I/O-subsystem. A network topology defines how network edges are connected to intermediate nodes. Any network graph can be created.

Each component implementation uses characteristics to simulate hardware behavior. To cope with several levels of abstraction a component can have several implementations. During the model specification the concrete model and its characteristics can be selected individually for each component. Usually, characteristics are provided in vendor specifications or obtained by benchmarking the existing system.

The current level of detail of the cluster hardware is as follows: A *Node* has an amount of memory and number of CPUs. Each CPU processes a fixed number

of instructions per second. Right now, the memory is used only for caching I/O on the server side. CPU time is shared equally among all processes in timesharing manner. A CPU is the only component which shares available resources among all pending jobs, all other components process jobs sequentially – usually in the order they were submitted. *Network edges* have a latency and a transfer rate. *Network nodes* have a maximum bandwidth to relay data, and represent store-and-forward systems. With the help of network components memory access of communication can be simulated. This permits to model local communication. The *StoreForwardMemoryNode* 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. Multiple nodes are interconnected by a central switch. In this figure, throughput and latency of all network components are given as observed on our Intel Westmere cluster consisting of 10 nodes.

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. Each file is assumed to be stored sequentially on disk. Depending on the distance to the last byte accessed within the 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.

3.2 Network Communication

One non-functional requirement to the simulator is to provide approximate best-cases for data transport over the network, but on the other hand results should be realistic. For example, a bottleneck in the network should not cause packets to pile up on its input. In the past we evaluated several transport algorithms – mechanisms like wormhole routing or buffering with packet dropping are not capable of saturating a network completely in case of a network bottleneck.

Therefore, 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, then the transmission of incoming data flows is continued. Fragmentation into chunks is done by the NI, which are then routed through the network individually. While a packet is on its path through the network graph, each intermediate node decides which outgoing edge will be used depending on a routing algorithm. Currently, packets can be either routed on the shortest path or distributed in a round-robin fashion to neighbors with the same distance to the target.

On each component the status of the flow between every source and all target nodes in the network is maintained, that is a single stream for every communication pair is kept. 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 it achieves the highest utilization of all network components for all streams, and it does not throw packets away.

3.3 Software Model

PIOsimHD allows execution of programs conforming to the MPI standard. Asynchronous communication and collective operations of MPI-3 are supported. To simulate the execution of a particular MPI function at least one implementation must be provided within the simulator. Internally, each MPI function is programmed as a state machine consisting of states (steps) and transitions. A state can issue a set of blocking send and receive operations, or spawn multiple child state machines, to allow concurrent data transport between multiple endpoints. An arbitrary instruction number can be added to each state to simulate computation. Multiple implementations for a given MPI function can be programmed and selected in the model specification (several collective calls are already implemented).

The state machine has a global view of the simulation, i.e. it is possible to see the state of other clients. For instance, this global world view allows implementing `MPI_Barrier()` without network communication at all – once all clients invoked the barrier the collective call finishes.

An abstract parallel file system defines how client and server interaction takes place. File data is partitioned among all servers as defined by a selectable distribution function. Metadata operations are not considered. Clients and servers interact in a similar fashion to the PVFS model, but the concept is universal to most parallel file systems: To write data, a client requests a write operation from the server and then starts to transfer all data. In the simulator file sizes are updated once a write operation finishes. Non-contiguous I/O requests are supported. It is also possible to add I/O forwarders to a client. A forwarder is responsible to relay data to at least a single I/O server. All data between this particular client and server is then routed via the forwarder.

3.4 Simulation Workflow

In general, there are two ways to increase insight in the interplay between system and application with *PIOsimHD*: either a running application is instrumented to generate trace files or the communication and I/O behavior can be coded explicitly in the form of Java programs. In the latter case helper classes allow explicit programming of a cluster model and application behavior. This can be used to perform small tests of I/O systems or MPI-internal communication.

An MPI-wrapper is linked into instrumented applications which intercepts all MPI-IO activities with the *PMPI* interface and records these events. If PVFS is used as the underlying file system, then additional traces for client and server activities can be included by using the instrumented version offered by *HDTrace*.

Traces of the application and optionally PVFS can be visualized using the graphical viewer *Sunshot*. To perform simulation a model must be created by the user that describes the cluster, and the mapping of the processes to available nodes. Note that the user can simulate concurrent processing of multiple applications at the same time to stress network and I/O infrastructure. Model classes read this model and the application trace files that are required for simulation, and offers these as command classes to the simulator. *PIOsimHD* performs the discrete event simulation and, if requested, stores the processing as trace files. Results of simulation can then be visualized by *Sunshot*, which enables a comparison of the recorded process and file system activities and simulation results.

4 Performance of the Simulator

Time to conduct a simulation depends on the model initialization and the event processing by the simulator core. To assess the performance a few experiments have been conducted on a laptop equipped with an Intel i7-640M (2.8 GHz). The laptop runs under Ubuntu 11.10 (64-bit) and OpenJDK 1.6_23 (IcedTea6 1.11pre).

Since the processor supports Intel's Turbo Boost technology, which could effect shorter experiments, the CPU governor is set to userspace and the frequency is fixed to 2.8 GHz.

Experiments are encoded in Java and use builder classes to set up the system model. All tests rely on the introduced system model with a StoreForwardMemoryNode to model processor sockets.

This test measures the simulator performance of inter-node communication, which is the basis for I/O as well. Performance of I/O simulation is not analyzed explicitly because measureable speed of simulated I/O depends on the chosen cache layer and its implementation; for instance, processing of N operations has an average-case complexity of $O(N \cdot \log N)$ when using a specific cache layer since it keeps operations in a heap data structure in order to fuse them.

4.1 Speed of the Event Processing

To measure the speed of the event processing a simple experiment is conducted: One process sends 100 GiB of data to another one; both are hosted on a single node. In the experiments the NIC model fragments data into chunks of 100 KiB resulting in more than 10 million processed events. On average 1.5 million events are processed by the simulator per second if debugging calls are commented out. The implementation of the simulator uses asserts to check the correctness of parameters and it offers debugging capabilities of the internal states.

The impact of removing debugging calls is tremendous – with enabled debugging only about 45,000 events/s can be processed. Note that string processing is very time consuming in Java, and the processing of asserts needs some time, too (even if they are disabled). Therefore, scripts are available that comment out asserts and the invocations of debugging messages. To evaluate the impact those scripts have been applied to clean the code.

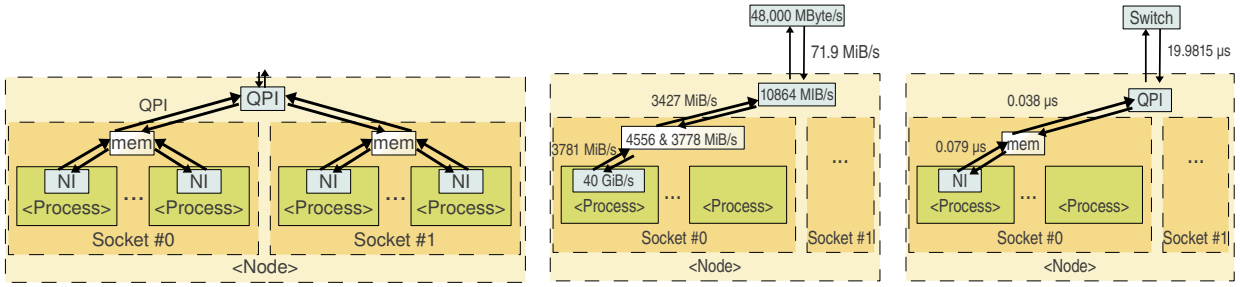
Therefore, once the model and simulator is verified it is recommended to remove debugging calls from the simulator. The impact of asserts is not measurable with OpenJDK and those should be kept. Preliminary tests with the JDK from Sun showed similar improvements, but with this JDK disabling of assertions improved performance, too.

Note that the speed of the simulator, which means the number of events processed per second, is in the order of state-of-the-art simulators such as the *LogGOPSim* [4].

4.2 Scalability of the Simulator

To test the scalability an `MPI_Bcast()` of 100 MiB of data is performed with an increasing number of nodes; each node hosts exactly one process. Both the time to create the complex cluster model with a socket per node and the time to execute the simulation are measured. The implementation of the broadcast uses a binary tree algorithm to distribute data among all processes.

The simulation of the network traffic is evaluated for three cases: with the congestion model, with the analytical NIC and by transferring a single large packet. When the congestion model is applied, the number of events



(a) Topology of a single node. (b) Component throughput. (c) Latency of edges.

Fig. 1: Schematic network topology of the cluster model. The topology is annotated with performance characteristics for the working groups cluster. The memory node transfers messages faster between processes of the local socket. Performance is given next to the uni-directional edge and inside the nodes.

depends on the message size and the network granularity. The analytical NIC model reduces the number of events because the route is computed once per packet. Many other simulators use analytical models to compute communication time. Therefore, by using a large transfer granularity and the analytical NIC model a similar setup is generated.

The number of simulated nodes and processes is increased until the laptop can no longer handle the simulation. If more than 1024 processes are created, then the amount of available memory will not suffice. This means Java terminates with an out of heap memory error and with 1024 processes approximately 1.5 GiB of main memory is filled up.

In Figure 2 the time to build the model with the builder classes and to execute the broadcast operation is listed. Since the number of components grows linear with the increasing numbers of processes, the time to build the model increases, too (see Figure 2a). Surprisingly, building larger systems is a bit faster and with 1024 nodes, an average of 4.3 s are needed, for 128 processes 0.92 s – the large system does not need 8 times the amount of time for 128 processes. This is probably induced by the way the test is conducted: All configurations are evaluated in a single by a loop. When Java gets executed for every single configuration, that means the test is not executed in a loop, then the time doubles. The reason is probably the memory allocation; larger configurations need approximately twice as much memory as the previously run test. Therefore, half the memory is already preallocated in a subsequent test. Also the class loader fetches and prepares the Java bytecode in the first iteration, which adds a bias to the first experiment.

The total number of executed events roughly doubles with the number of processes and so does the execution time (see Figure 2b). Performance of the analytical model is higher by approximately a factor of 5. When just a single packet is sent, then execution time is much faster because the number of packets is decreased by a factor of 1000 (compared to the transfer granularity of 100 KiB).

Overall, building cluster model of 1000 processes needs a few seconds and scales linearly with the number of processes. The `MPI_Bcast()`, which is basically the transmission of 100 MiB of data for all processes takes much longer (25 s) when an adequate network granularity of 100 KiB is used. Since program execution requires simulation of multiple commands, the time for building the model is not so important. Further, to simulate a large number of processes the appropriate configuration of the simulation must be picked.

5 Validation

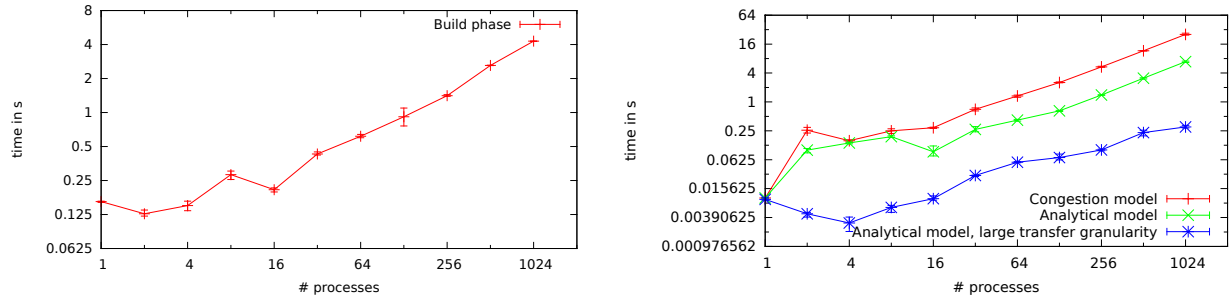
A careful validation of the hardware and software model has been done for our cluster system. The conducted experiments are described in this section, but due to the page limits only an excerpt can be discussed[1].

5.1 Parameterization

First, the conceptual cluster model must be parameterized for the existing cluster. Therefore, several independent micro-benchmarks are run, and results are compared to ensure correctness of the benchmark. Vendor information and theoretical considerations prove the validity of the observations.

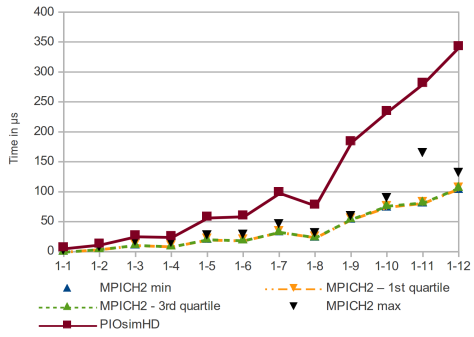
Figure 1 shows the determined throughput and latencies for the network components. Most values have been determined with MPI point-to-point benchmarks;

[1] The mentioned experiments are described and assessed in the Ph.D. thesis of the author, which will be published 2012.

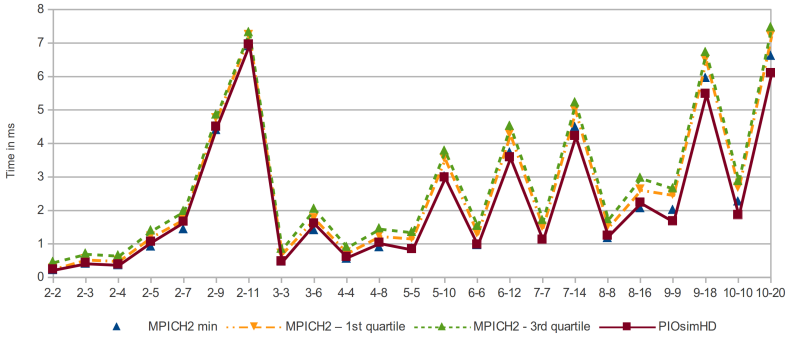


(a) Model build phase (b) Execution

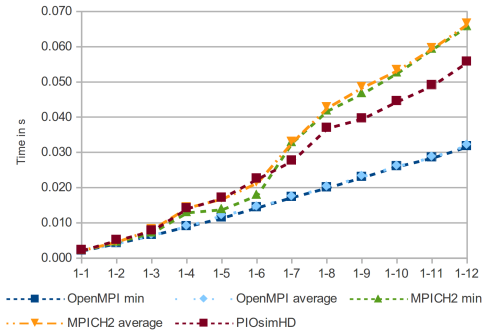
Fig. 2: Scalability of the simulated broadcast operation.



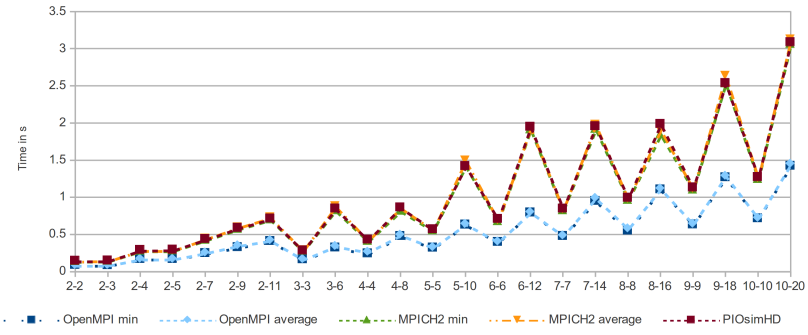
(a) `MPI_Allgather()`, 10 KiB of data – local communication.



(b) `MPI_Allgather()`, 10 KiB of data – inter-node communication.

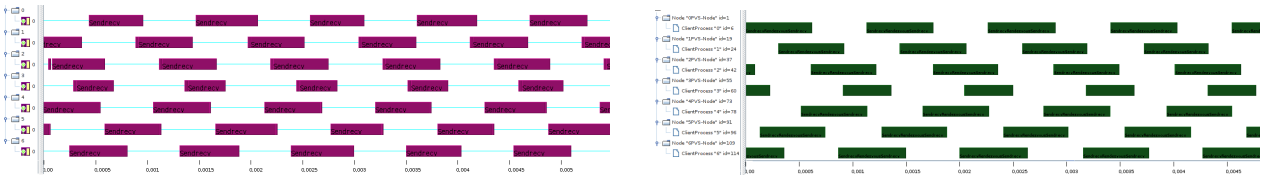


(c) `MPI_Scatter()`, 10 MiB of data – local communication.



(d) `MPI_Scatter()`, 10 MiB of data – inter-node communication.

Fig. 3: Validation of collective communication. For a variable configuration of nodes and processes collective calls are invoked with the same parameters and a fixed data size per process. Note the scaling of the y-axis.



(a) Observed behavior. (b) Simulated behavior.

Fig. 4: Trace excerpts for a PDE experiment.

latency is measured with a Ping-Pong kernel transferring empty messages; throughput is determined with large message transfers of a Ping-Pong and a bi-directional SendRecv kernel. Performance of intra-socket, inter-socket and inter-node communication is measured. With those values performance of internal edges and nodes can be derived, e.g. by subtracting the average intra-socket latency from the inter-socket latency, the hidden edge between the *mem* and the *QPI* nodes is parameterized.

5.2 Validation of Mathematical Models

Validation is performed on multiple levels of abstraction. First, the underlying mathematical models of network and I/O-subsystem are compared with the observations and this shows the discrepancy inherent to the models. Since the simulator implements the models with some slight modifications, it cannot do better than the mathematical models. For a qualification of the network model PingPong and SendRecv kernels are executed between two processes and with a variable message size. The comparison showed a good match for the network models. However, caching inside the processor L3 cache speeds up transfer of smaller messages. This can not be handled by the simulation since it does not track cache locality of the accesses.

Performance of the I/O-subsystem is measured with IOZone and another disk benchmark and compared to the disk model. Note that the benchmarked I/O-subsystem includes the overhead for an Ext4 file system and the optimizations done by the Linux kernel. The measured behavior showed the complex interplay between file system, kernel and hard disk characteristics. Therefore, the simplified mathematical model can predict performance only to a limited extent – for non-cached I/O it works well.

5.3 Validation of the Simulator

On the one hand, PIOsimHD is validated against the mathematical models to discuss the discrepancy between implementation and theory. The comparison showed a good match of the network flow protocol and the simulation of the block I/O. However, the transfer granularity, which defines the size in which the packets are split, should not be too large compared to the average message size – since messages are fragmented into packets of this size and multiplexed, this seems natural.

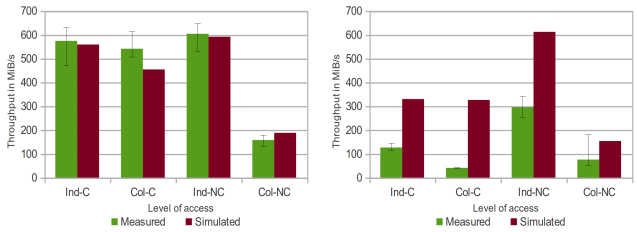
On the other hand, benchmarks and simple parallel applications are executed and their results are directly compared with the simulation. The following MPI collective calls are evaluated: Barrier, Reduce, Allreduce, Bcast, Gather and Scatter. Further, the following point-to-point communication schemes are supported: Ring – every process sends data to its right neighbor and receives from the left neighbor, Paired – an even process exchanges data with the next odd process by using `MPI_Sendrecv()`, SendToRoot – all processes send data to the root process, which receives data in order and SendrecvRoot – processes use `MPI_Sendrecv()` to exchange data with root.

The benchmark is used to measure collective and point-to-point communication times for payloads of 1, 10, 100 and 1000 MiB. A variant of the benchmark measures payloads of 10 KiB; in this case, communication time is much lower. Therefore, more repeats must be conducted. This small payload helps assessing the latency model while larger payloads are increasingly affected by the network throughput characteristics – for 100 MiB the influence of the latency is expected to be rather low on our network.

A huge number of configuration combinations is evaluated. Every MPI function is called 4 times and the whole benchmark is restarted 4 times to increase robustness. Configurations are defined by node count and process count, the node numbers range from 1 to 10, and up to 20 processes are deployed. MPICH2 is run to distribute the processes in round-robin fashion among the available nodes and the sockets within. To illustrate the placement scheme consider the Configuration *2–5*. Here 5 processes are placed on the two nodes (and the four sockets) as follows: ((0,4),(2)),((1),(3)), that is the first socket on the first node hosts Process 0 and 4, and the second socket on the same node hosts just Process 2 and so forth.

For the simulation the point-to-point operations are recorded and executed by the simulator – optionally, the simulator can replay the computation time in the collective calls. Thus, the simulator recreates the real communication pattern and should achieve similar results. Four diagrams show the observed and simulated time of scattering 10 MiB of data, and of calling `MPI_Allgather()` with 10 KiB of data (Figure 3)[2]. It can be seen that the simulator achieves similar times for intra-node communication – other collectives are quite similar. Time of the estimate is typically at least around 80% of the observed time, and often matches well, see for example `MPI_Scatter()`. While on the real network the observations vary due to the Gigabit Ethernet, the model uses a fixed timing. Thus, for example, the dips for the Configurations *4–8* and *8–16* in Figure 3b are

[2] Measurements are made with MPICH2 1.3.2 and Open-MPI 1.5.3 as a reference.



(a) Reading 100 MiB records.

(b) Writing 100 KiB records.

Fig. 5: I/O performance of 8 clients transferring 1 GiB of data per process. Servers are placed on the same nodes and provide 2 GiB of main memory for caching writes.

inherent to the communication pattern. Timing of local communication is estimated well for large messages, but often underestimated by a factor of 3 to 4 for small messages, probably due to processor caching. In a few cases the simulator underestimated performance by a factor of 10, which indicates a problem in the MPI implementation. This shows the importance of theoretic estimates.

Besides the communication, the performance of the four levels of access including collective I/O is evaluated with PVFS. Two of the generated graphs are given in Figure 5. In most cases the simulator achieves similar results, proving the correctness of the simple models. However, for small record sizes the simulator estimates a better performance. This is due to the improved scheduler in the I/O servers.

At last, a Jacobi PDE which offers checkpointing is evaluated for multiple problem sizes and configurations. In this case the application trace is fed into the simulator, which simulates all calls with its modular implementations. An excerpt of the measured and the simulated timelines is given in Figure 4. The simulation of the trace and profiles are surprisingly accurate – also in many cases communication patterns can be observed in the simulation like on the real system.

6 Summary and Future Work

In this paper the simulator PIOsimHD is described and the conducted approach to validate the implemented hardware and software models is introduced. The simulator can be fed with traces to rerun the parallel program on any configuration of the cluster system. This feature has been used to validate collective calls and a Jacobi PDE for many configurations.

HDTrace is an environment which allows tracing of MPI-IO behavior. With HDTrace observations can be compared with simulation results which simplifies assessing the observations.

With the determined parameters for our Westmere cluster the validation showed an astonishing match with the observations. Due to the capability of the simulator local communication and I/O can be assessed in silico, too. This allows us to identify inefficiencies, to conduct research on new algorithms, and to evaluate future systems.

In order to evaluate and improve communication for cluster systems more experiments will be conducted in the future. Also more I/O capabilities will be evaluated, like the experiments we conducted in [6].

References

1. Geimer, M., Wolf, F., Wylie, B.J.N., Becker, D., Böhme, D., gs, W.F., Hermanns, M.A., Mohr, B., Szebenyi, Z.: Recent Developments in the Scalasca Toolset. In: Tools for High Performance Computing, Proceedings of the 3rd International Workshop on Parallel Tools. Springer (2009)
2. 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, pp. 39–46. Springer-Verlag, London, UK (2000)
3. 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, pp. 78–84. IEEE Computer Society, Washington, DC, USA (2009)
4. 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 '10, pp. 597–604. ACM, New York, NY, USA (2010)
5. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools, pp. 139–155. Springer (2008)
6. Kuhn, M., Kunkel, J., Ludwig, T.: Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations. In: PDP 2012. Munich Network Management Team, IEEE (2012)
7. Kunkel, J.: HDTrace – A Tracing and Simulation Environment of Application and System Interaction. Tech. Rep. 2, Research Group: Scientific Computing, University of Hamburg (2011)
8. 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 '06. ACM, New York, NY, USA (2006)
9. Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006)

10. 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, pp. 133–139. IEEE Computer Society, Washington, DC, USA (2009)
11. Wolfgang Kreutzer, B.P.: The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java. Shaker Verlag (2005)