may differ. Hard disks rotate with a fixed speed and contain more data on the cylinders with a larger circumference, therefore subsequent blocks can be accessed faster on the outer cylinders than on the inner cylinders. For an SSD, the throughput depends on the speed data is transferred between the chips and the controller, which should utilize multiple flash chips concurrently.

- *Concurrency*: Depending on the access pattern and deployed block device, it can be useful to issue multiple requests at a time to increase aggregated performance. In case of small requests, a disk scheduler might optimize throughput by reordering requests – for SATA disks this is called *Native Command Queuing* (NCQ). On SANs, multiple pending requests keep the disk(s) busy. Thus, additional network latency due to communication with the SANs does not show up in the aggregated performance, although individual operations take longer.

All technologies deploy mechanisms that enable them to tolerate errors occurring while data is accessed. For instance, a HDD verifies data written to disk by re-reading the magnetic information. Any read and write error detected will cause the disk to spin again over the same track while trying to recover the information. SSDs deploy wear-leveling algorithms to increase the life time of flash memory, but also have reserve cells to tolerate failure to a certain extent. RAID schemes on top of block storage may increase capacity or throughput, and might protect the system from failures – depending on the scheme. While all those mechanisms mitigate and protect from errors, recovering from an erroneous state requires time and resources. Thus, an error degrades performance in a real I/O subsystem, but might be transparent by the user. More information about HDD's is available in [Mey07].

Platform A computer is built with a particular processor platform in mind. The platform includes chipset, internal technology and interconnect to peripheral devices such as I/O devices. For example, *PCI-Express* is a bus-system that is deployed with newer processor platforms. Thus, a platform determines how fast data can be shipped between the different hardware components of a computer. In most cases, the performance provided by the platform suffices to saturate network, disk or memory. However, very fast networks like Infiniband put the internal bus systems under pressure.

Certain optimization techniques and technologies might also be provided by a platform. For example, *Direct Memory Access* (DMA) is an access method that enables data to be transferred between peripheral devices and main memory without copying it through the CPU. Together with the network card, the platform might enable *remote direct memory access* (RDMA). This technique transfers data directly into a remote main memory, without involving the operating system of any of those systems. Thus, data need not be copied between application buffers and kernel[20]. RDMA reduces the communication latency, but usually requires data to be aligned contiguously in memory, i.e., only a contiguous region of data can be transferred with RDMA.

2.2.3. Computation Performance

From the user's perspective, an application computes some valuable result; I/O and communication just help to get the work done. Therefore, computation performance is prime importance. The following aspects are relevant on all software layers and include the operating system. The experience of the developer is important to pick the appropriate algorithm, programming language and compiler for solving a given task. Finally, the coding in the programming language expresses how a problem is solved.

Algorithms are blueprints describing how computation will be performed to calculate the results. An algorithm determines the data dependency and thus communication patterns. Furthermore, certain data structures might be required to perform an algorithm efficiently. Therefore, the algorithmic specification restricts the formulation in a programming language. An algorithm has a computational complexity,

[20] Since data can be transferred without copying buffers is often called *zero-copy*.

approximately defining how the computation time grows with an increasing problem size, i.e., the data itself.

Overlapping of I/O, communication and computation In the best case, the application spends all its time to compute valuable results. In this sense time needed for I/O and communication are wasted. Therefore, any communication and I/O should take place concurrently to computation. Usually, communication and I/O libraries provide non-blocking calls. When such a call is invoked, it initiates communication (or I/O activity) and returns immediately. The actual activity is performed in the background. The possible speedup achievable by overlapping I/O and computation is bounded: if I/O takes the same time as the computation, a speedup of two can be achieved [BKL09]. If communication and I/O can be perfectly overlapped a speedup of three is possible. However, overlapping requires additional memory space to temporarily buffer the I/O (or communication) data until the background activity completes. Also, the programming and debugging of the application becomes usually more difficult.

Programming language The programming language is a formal notation in which the developer tells the system what to execute (in imperative paradigms) or the result he/she is interested in (in declarative paradigms). Expressibility and semantics limit possible formulations of code to solve the problem. The programming language is tightly coupled with the potential optimizations a compiler can apply.

Compiler The compiler parses a programming language and transforms it into another programming language, typically a sequence of machine code, which can be executed on the target machine. While the compilation is performed, the compiler tries to optimize the code in order to increase performance[21]. Compilers provide different sets of available optimizations. Users have to select the appropriate optimizations to achieve efficient machine code; for example, a brief evaluation of compiler optimizations for stencil operators is provided in [KN10].

Run-time system The transformation of *interpreted* languages or *byte-code* into machine-specific instructions depends on the capabilities of the run-time system. Java's just-in-time compiler aggressively optimizes the code regions that are used frequently, while at its first execution, code it is translated quickly in order to reduce translation time. A PGAS language needs a run-time system to manage remote memory access; buffers are used to store local results. In this case, the strategy, such as potential background transfer of data, reduces the communication overhead. Making use of the platforms' RDMA feature of platforms is favorable as it permits modification of remote data without disturbing the remote processing.

Cache usage The ratio of memory bandwidth per floating point operation is rather low at around 1 byte per Flop, hence optimization of memory access is crucial. Caching is a standard practice in computer systems to exploit access locality of computer programs. Most programs need the same data or instruction sequence multiple times (the working set). If the data is already held on media that is faster accessible, it is not necessary to request it from a slower one. As a CPU cache is much faster to access than the main memory, the algorithm and program should operate on a small working set – the more local accesses are performed, the better. Random access to small pieces of data degrades memory performance due to the fact that memory access is performed in larger cache lines of, e.g., 64 bytes. Thus, if only one byte is needed per cache line then effectively 63/64 of the memory bandwidth is wasted. The efficiency of cache re-use depends on the characteristics of memory and the platform (see above).

[21] However, the language semantics must be obeyed in this step even if relaxed semantics would be valid for the given code. If too many optimizations cannot be applied due to the semantics, then typically the programming language is adjusted. For example, this has led to the new `restrict` keyword in C99.

2.2.4. Communication Performance

Cooperation between processes requires information exchange – inter-process communication transfers data between processes. The observable performance is restricted by the semantics of the interface, the actual communication pattern, the internal communication algorithms, memory alignment of the provided data and placement of the processes which communicate. During the communication local resources such as CPU and memory might be required.

Interface and semantics The API and semantics of the communication paradigm defines how inter-process communication is performed. Usually a rich variety of potential operations is provided; this enables the user to pick the best abstraction to realize his communication pattern. Two interaction patterns are common, either two processes communicate with each other in a *point-to-point* operation, or a set of processes exchange information together in a *collective* operation. Operations might follow rather strict semantics, sometimes more strict than needed, which degrades performance. Internally, the library and run-time system can use any (additional) provided information to gear the communication pattern towards the available hardware.

Communication pattern The communication pattern refers to the observable inter-process communication, which is the sequence of collective or point-to-point operations invoked by the program and the communication partners. In most scientific codes, the communication pattern varies between the processes. Typically, processes communicate sparsely with other peers; only rarely, all processes talk to each other.

In point-to-point communication, a late start of the receive operation or the send operation cause a ready communication partner to wait – late processes are referred to as *late sender* and *late receiver*, respectively.

The arrival pattern determines the temporal order in which processes start a collective operation. A process is referred to as *early starter* if it starts a collective operation earlier than other processes (in terms of wall-clock time); in comparison, a *late starter* joins an already started collective operation. Optimally, all processes invoke the collective operation concurrently; this *balanced start* yields the best performance[FPY07].

When a program tries to send data from multiple processes to one process at a given time, then the single receiver imposes a bottleneck. Therefore, implementations of collective operations try to reduce the *network congestion* and balance network transfer among the participants.

Algorithms Internally, the communication must be realized by implementing some inter-process communication protocol. Several algorithms can be thought of for point-to-point communication: By using a rendezvous protocol, a sender waits until a receiver is ready to start transmission, or a sender can start transfer immediately without knowing the actual state of the receiver.

Collective operations are implementable in several communication patterns; the algorithm that realizes the semantics of the collective operation has a major impact on the observable communication pattern. An efficient algorithm follows the semantics, yet utilizes provided hardware resources at any time and minimizes redundant operations. Further information about the optimization of MPI is provided in Section 2.3.4.

Memory and CPU Performance of network data transfer is limited by memory performance because data from the network card must be copied from, or to, main memory. Without *direct memory access* (DMA) capability of the platform the CPU must also move data explicitly, which implies a performance limitation for fast networks. Also, the CPU might be necessary to initiate or control communication, or it does some bookkeeping such as computing checksums for the network packets. Therefore, many network adapters

accelerate communication protocols by supporting DMA and by providing so called *offload engines*, which perform most of the required computation inside the adapter.

Data must be copied at least between network device and memory. However, often data cannot be transferred directly between the network device and the buffer that is specified in the application. Therefore, multiple copies are needed and the available memory throughput is degraded further. On the one hand, this might be necessary because the user space does not have direct access to the network device. On the other hand, protocols such as TCP embed further control data, the Internet Protocol, for example, requires post-processing to compute and to verify checksums which are embedded in the data packets. *Zero-copy* is a feature of the platform which enables direct data transfer between network device and user-space [KT95]. Without zero-copy capability of the platform, the OS must copy data between an OS internal buffer, in which the data is exchanged with the network device, and the buffer of the process. In the best case, data is copied from the memory of a process to the memory region of the remote process via RDMA.

Alignment of provided buffers Hardware technology and te platform can offer the capability to transfer contiguous data via RDMA which enables zero-copy. Typically, transfer of multiple non-contiguous regions in memory either requires to pack them in additional memory buffers for RDMA, or to send them without RDMA support. On some systems, copying data in memory from (or into) a fragmented buffer takes a considerable amount of time. Also, buffers not aligned to cache lines might waste memory bandwidth.

Software to hardware mapping Processes and threads of a parallel application must be assigned to available CPUs and nodes. Typically, the inter-process communication between some pairs of processes is more efficient than between others. The reason for this may be the network topology, or that intra-node communication is faster than communication via the network interface. In most cases in HPC, the user will start more processes than fitting onto one node, hence multiple nodes must be used. The communication path between two communication partners defines the performance of the inter-process communication.

Assigning the processes to the processors in an appropriate way yields high potential for optimizing applications with well-known communication and I/O behavior. Two processors which communicate often should exchange data in an efficient way. Especially, for tightly coupled applications[22], the inter-process communication should be as efficient as possible.

Complexity and potential of the mapping in the context of MPI is discussed on Page 51.

2.2.5. I/O Performance

There are several aspects involved in delivering high I/O performance to parallel applications. While given hardware characteristics are discussed on Page 37, this section focuses on the methods applicable to manipulate workloads to improve achievable performance.

Since many file systems such as PVFS2 and Lustre use local file systems like Ext4 to store their data, they are influenced by the mapping of logical blocks to physical blocks on the block devices. To ensure persistency and consistency, file systems write additional data to the block device. Those add overhead when modifying data or metadata. Further, file systems are implemented in the operating system which deploys strategies to improve performance such as scheduling, caching and aggregation.

Therefore, the observable I/O performance depends on more than the capabilities of the raw block device. In this thesis, the term *I/O subsystem* is used to refer to all the hardware components and the software layers

[22] Tightly coupled applications are programs which processes must communicate frequently with each other due to data dependency. In contrast, an *embarrassingly parallel* problem can be solved by processes which compute their result independent from other processes.

Communication algorithm Under the assumption that the placement is fixed, i.e., either because it is optimal or pre-defined, then there is potential to adjust the MPI internal algorithm towards the given application and network topology. To cope with the complex interplay of system and application, many different algorithms must be implemented in each MPI library. However, choosing the right algorithm for a given function call remains difficult. Presumably, suboptimal performance can be observed on every system (for at least some applications).

Point-to-point operations are of simple nature, as just two processes participate; therefore, the low-level network driver has a major impact on the performance. From the algorithmic point of view an MPI implementation has only a few options if it sticks to the standard's semantics. The semantics of `MPI_Send()` permit a completion of the operation, even if the receiver did not yet get the message; this allows the implementation to buffer the message either on the side of the sender or the receiver. Many variants of the point-to-point communication exist which enforce a certain buffering or synchronization behavior.

Collective operations are of much more interest. Related work shows the impact of the arrival pattern on performance and that tuning for individual message size is important [FPY07]. From a library's 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:

- The memory datatype is relevant because contiguous datatypes can be transferred via RDMA. Datatype on sender and receiver side can be be different, requiring to pack or unpack data.
- The communicator defines the participating processes in a given communication, even if an algorithm is optimal for one defined group of processes, for another set of processes a different algorithm might be faster. The problem of choosing the appropriate algorithm for a given group of processes is linked closely to network topology.
- The target rank is relevant because the implementation must perform the last operations on the target rank, therefore, for instance a spanning tree algorithm should put its root on the target rank. Similarly, for a one-to-all operation the source rank provides some information which must be transferred to all other clients.
- Depending on the amount of data the communication time varies. A basic consideration is that latency becomes more important for small messages, while larger messages are dominated by the bandwidth between the communication partners.

Concepts to optimize I/O Due to the complexity of I/O, that involves communication as well as server activity, there exists a rich variety of opportunities to optimize access patterns and metadata operations. First, a few hardware considerations are recapitulated from section 2.2.2. Those help in assessing the mentioned optimizations. From the perspective of the servers it is important that clients access as much data as possible in one I/O request, because it keeps the pipelines on the server busy and, furthermore, it enables sophisticated server-sided optimizations. A server can access data on block-level in a granularity of full blocks – modification of a few bytes of a block requires reading the full block – then modify it and write it back. Hence, performance is wasted. Due to the characteristics of hard disk drives, random access achieves only a small fraction of available performance (at least on hard disk drives) – the mentioned optimizations are all designed having disks as persistent storage in mind. Processing of a request requires some CPU and memory resources. As requests must travel over a network from client to server, processing involves additional delay.

To tackle the issues mentioned, the non-contiguous operations and collective calls have been defined in MPI-IO. In [TGL02] access to data with MPI is classified into *four levels of access*. The levels are characterized by two orthogonal aspects: contiguous vs. non-contiguous data access, and independent vs. collective calls. Thus, the levels are: contiguous independent, contiguous collective, non-contiguous independent and non-contiguous collective. Depending on the level a different set of optimizations can be thought of.

Chapter 9 IX

Future Works

In this chapter, features of and extensions to the presented work are described that could not be realized during this thesis. Besides the introduced optimizations[1], extensions to MPI are imaginable that have yet to be researched thoroughly. A few of those concepts are discussed in Section 9.1. Prior to implementation in a real system, they could be evaluated with PIOsimHD to assess their potential benefit.

There are three general ideas for extending the presented work:

- Performance evaluation of more cluster environments: The simulator has been applied successfully to the working group's cluster system. PIOsimHD could help in analyzing other supercomputers to identify bottlenecks in the system architecture and it would foster understanding of relevant system characteristics. For example, the Blizzard supercomputer of the DKRZ could be evaluated with a similar methodology. At the same time, the system model of the simulator could be extended to incorporate aspects relevant to this supercomputer, for example it is expected that RDMA has an impact on communication performance.
- Porting to an existing trace format: With HDTrace, an alternative trace format has been developed; its development has been guided by the needs for simulation and tracing of client-server activity that initially could not been recorded with existing trace formats. During this thesis other trace formats have evolved – now it seems that all the required information for the simulation can be included in these formats, although that may not be comfortable. While HDTrace still offers several capabilities beyond other trace formats, it is possible to modify PIOsimHD and Sunshot to rely on trace formats such as OTF. An advantage of this strategy would be that these tools can be applied in typical environments without forcing users to link to another trace environment. Also, it would eliminate the necessity of porting the trace environment to other supercomputers.
- Access pattern repository: A global repository of application behavior might be valuable for the community. Such a repository could contain trace files for relevant scientific applications and it could be open for researchers to provide and to use the available traces. In combination with a replay mechanism this would allow application specific benchmarking – researchers could evaluate application performance without the need for running the application. This is especially valuable for developers of middleware such as communication and I/O libraries. Further, by comparing performance across supercomputers, the system that is best suited for the communication and I/O pattern can be identified. With the student projects Parabench and Paraweb, a first attempt towards this goal has been made.

There are several minor modifications that would improve HDTrace and PIOsimHD:

- Improved build system: The C components mainly rely on the *GNU build system* (also known as *Autotools*). Correct and portable usage of the GNU build system is complicated, also execution of the configuration process is slow. Additionally, the configuration of the experimental intercepting library, e.g., for POSIX calls, is not automatized yet. One task for diligent work is to automatize the whole build process with *Waf*[2]. Waf is much easier to use and maintain, and the configuration process is much faster. The *TraceWriting C Library* has already been ported to Waf.
- Improved analysis of MPI-IO datatypes: Sunshot independently visualizes the MPI datatypes Vector, Contiguous and Struct for every process. More MPI constructors could be supported, also, for collective operations, a view could illustrate the accessed data for each process. It is envisioned to extend the solution by aggregating datatypes of collective calls into one view – showing the accessed file regions of all participants, each encoded with a different color. Thus, the overall activity of the

[1] See Section 2.3.4 and Section 2.3.5.
[2] `http://code.google.com/p/waf/`