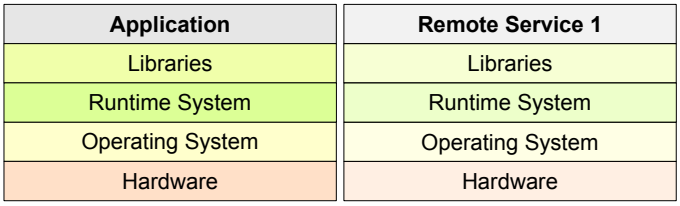

Figure 2.7.: An abstract view of layers and components involved in application execution.

Next, the hardware aspects contributing to performance are examined. Software aspects contributing to performance are discussed in more detail in Section 2.2.3.

2.2.2. Hardware

Clearly, observable performance is limited by the physical capability of the hardware to perform computation, communication and I/O. In the following, a subset of relevant hardware characteristics is provided.

Network Since a network transfers data between two independent nodes, it limits the communication capability. To share the available links among all communication streams, most technologies utilize the communication method of *packet switching*: Data is fragmented into *packets*[13], which are transported between source and target. This allows to multiplex available network links among multiple streams. Common network technologies are *Ethernet*, *Infiniband* and (for storage servers) *FibreChannel*. Many supercomputers deploy their own custom network interface.

Within the scope of this section, important hardware characteristics are[14]:

- *Latency*: the delay between the moment network transfer of a message is initiated, and the moment data is starting to be received. Latency can be referred to as *propagation delay* as well. It is measured either one-way, or *round-trip*, which is the one-way latency from source to destination plus the one-way latency from the packet destination back to the original source. Round-trip time can be easily measured between two machines by inducing low-overhead network communication. Tools like *netperf* are designed to determine the network latency and throughput on top of the network protocols. Latency has a high influence on small messages, e.g., metadata operations in file systems, but it becomes negligible for large messages. Each inter-process message involves latency, therefore it is favorable to reduce the number of messages by packing more data into one big message.
- *Bandwidth/Throughput*: In computer networking, the term bandwidth refers to the data transfer rate supported by a network connection or interface – how many bits are transferred over the wire. Bandwidth can be thought of as the capacity of a network connection. Network protocols and encoding of information add overhead to the actual transferred data and thus the effective *throughput* of the hardware is lower than the actual available bandwidth. For this reason, the term throughput is preferred in this thesis.
- *Topology*: is the physical (and logical) arrangement of a network's elements like nodes and interconnections between them. In a fully-meshed network each node is able to communicate directly with each other node. Unfortunately, realization of this topology is very expensive in reality and almost impossible for large numbers of nodes. Common topologies try to mitigate this problem by reducing the number of links, but still aim to provide a high bandwidth and a low latency. To reduce the number of communication links an indirect network topology either restricts communication to a subset of neighbor nodes[15], or it uses hierarchies such as trees.

[13] Packets on an Ethernet link are called *frames*.

[14] Further information on the topic of networking is offered in [PM04, PD11].

[15] These nodes, in-turn, are able to route the packets further towards to the destination (for instance the 2D-Torus, which has been shown in Figure 1.2).

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.

involved in providing node-local persistent storage[23]. Typically an I/O subsystem consists of the operating system which provides a file system on top of the block storage. The file system maps file system object to available blocks by using the block-level interface; it also schedules the raw block I/O. A parallel file system adds a layer on top of many independent I/O subsystems. The I/O path inside the operating system and the (bus) systems involved to transport data between memory and block storage are also addressed by the term I/O subsystem.

In the following, general considerations about the influences to I/O performance are discussed. Further information specific to local file systems is given in Section 3.6.1.

Access patterns The access pattern describes how spatial access is performed over time. With an access pattern, the I/O of a single client process can be described, but also the actual observable patterns on the I/O servers, or on a single block device. The pattern on the I/O servers is caused by all clients and defines the performance of the I/O subsystems.

An access pattern can be characterized by:

- **Access granularity:** Depending on the context, this is the amount of data accessed per I/O call or request. It is desirable to access a large amount of data per request, to reduce the influence of latency induced by I/O subsystems and network. In the best case, a single large contiguous I/O region is accessed per call.

  To access multiple non-contiguous regions of a file, either one request is issued per region, or a so-called non-contiguous I/O request bundles them into a single request. Non-contiguous I/O enables optimizations on the I/O subsystem such as scheduling, because the server has more pending operations to choose from. Also, the setup time for the initial link establishment and preparations of the I/O is reduced. Access to non-contiguous data in memory and on disk requires additional computation and may lead to in-memory aggregation of the contiguous regions into a temporal buffer. The extra computation time increases with the number of fragments while the costs of in-memory copies depend on their sizes.

- **Randomness:** Defines how close the accessed bytes are in the file (or on the block device). When multiple operations are issued in a sequence, then the locality of the accessed data matters. Disks achieve best performance with contiguous accesses where physical locations on the persistent storage are close together. For an SSD, random access is not so harmful, well designed controllers handle sequential and random workloads with the same efficiency; still access granularity should be in the order of the erasure-block size.

- **Concurrency:** Describes the number of concurrently issued I/O requests. The hardware resources must be multiplexed among all requests. Even if multiple sequential accesses to files are issued, an I/O scheduler might interleave these requests. Thus, from the perspective of the I/O subsystem it might look like a non-sequential access pattern.

- **Load balance:** This is the distribution of the workload among the servers in a parallel file system (or multiple block devices in a RAID).

  While distribution of a file among the servers is assigned by the parallel file system, the actual usage of servers depends on the access pattern. Depending on both the data distribution and the client usage, the requests to parallel or distributed file systems may access data only from a subset of the servers. This may potentially lead to a different utilization of the servers – a load imbalance. It is expected that the aggregated throughput of these requests is reduced. Intuitively, the highest throughput is achieved if data is accessed on all servers and if the amount of data accessed on each server depends on the server's current capabilities.

[23] In literature, the term I/O subsystem is often used in a relaxed manner; for example, to refer to a software layer which performs I/O. In this thesis, the term is used to highlight that performance and observable behavior is determined by more than the block device.

- **Access type:** All layers can either perform read or write accesses. There are other types of access for file servers, such as *flushes*, which cause cached data to be written to the block devices, or metadata operations. The I/O path might distinguish the different types of access; depending on incorporated optimizations and observed access patterns, other optimizations might be involved.
- **Predictability:** This measure indicates if data access follows a more or less regular pattern over time. The easier a pattern, the better predictable it is. Checkpointing[24], for example, stores a big amount of data in a well defined period and thus it is easily predictable. Predictable access patterns might be recognized and automatically enable comprehensive I/O optimizations in the operating system or file system like read-ahead, for example.

I/O strategy In general, the mechanisms introduced in this paragraph are orthogonal to the hardware and the architecture of the parallel file system. On the client-side, for instance, requests could already be tuned to improve the access pattern which will be observed on the servers. Similar to optimizations found in communication, these strategies could be applied on any layer involved in I/O.

A set of potential strategies are:

- **Caching algorithm:** Physical I/O is much slower than access to main memory. Therefore, it is favorable to hide the slow device performance by buffering I/O in main memory. Caching of requests is also a necessary prerequisite for many further optimizations like reordering or aggregation of requests.

  Write bursts fitting into the memory can be delayed – an early acknowledge to the writer permits it to proceed with processing. In the presence of bursts, this *write-behind* strategy can saturate the slower media constantly; while the writer continues with its computation, the data is slowly persisted on the storage. Unaligned writes might benefit from data already cached by allowing to combine multiple small accesses into a full block; this avoids the need to read the old block from the block device.

  Read-ahead is a strategy in which data is read in advance into a faster cache. Later reads access the cached data and avoid loading data from the slow I/O subsystem. However, in case the data is not needed in the near future it was fetched from the I/O subsystem unnecessarily. Thus, a balance of the read-ahead size is important and depends on the predictability of the access patterns and costs of the additional operations.

  Distributed systems could build a coherent *distributed cache* in which cached data not necessarily belongs to the local disk subsystem. This strategy increases the total available size of the cache at the expense of inter-process network communication. As network is often faster than the I/O subsystem, this boosts performance of applications whose working sets fit into the cache. On the client side, these strategies can be applied to avoid creation of requests to servers at all – all data is simply held locally. One difficulty with caching strategies is that it is necessary to update cached data to ensure a coherent view – this is often defined by the consistency semantics of the system that provides the caching.
- **Replication:** This strategy creates distributed copies of data, which then can be used to satisfy read requests. This effectively balances read-mostly workloads among the available components by reading the data from replicates located on idle servers. The number of copies can be varied depending on the level of load-imbalance and regarding the costs of the replicas' distribution. Replication is also a method to provide high availability: if one server crashes, data of the file is still accessible on a replica.

  The replication protocol defines the approach by which data is replicated: In a parallel file system typically either the clients or the servers are responsible for mirroring data. Also, data written could

[24] A checkpoint contains all the data that is required to continue computation later. This is especially useful to restart long-running applications that crashed due to hardware errors.

be replicated synchronously – during the write call, or asynchronously in the background. Since more data must be communicated than necessary, the chosen algorithm has a big impact on performance.

- **High availability (HA) support:** Mechanisms to increase availability of the computing environment imply additional expenses for hardware, and they reduce performance of the system by requiring it to write more data than necessary.

  Redundant components like hot spares and redundancy by storing data on multiple devices with error correction codes tolerate a number of failures of components or physical data. In case of a hardware failure, reconstruction of the original data with the redundant information implies a performance impact, though. Complete data replication multiplies the amount of stored data but has the advantage to provide load-balancing for files that are mostly read as well. HA could be provided just on a per server basis, or could be built into the parallel file system itself by spanning multiple servers to tolerate complete server faults.

  In large data sets the so-called *silent data corruption* [Mic09] becomes important[25]. There are methods to identify data corruption, for example, by storing checksums of the data. Another approach for fault tolerance is *end-to-end data integrity*. It validates that data accessed by the client is the same as stored on the I/O subsystem. However, transmitting the checksums and verification may reduce performance slightly.

- **I/O forwarding:** An I/O server can handle only a limited number of concurrent clients because each connection requires resources on the network interface and in memory. I/O forwarding [ACI+09] is a technique in which clients do not communicate with servers directly[26]. Instead they communicate with intermediate I/O nodes – the so-called I/O forwarders. Those nodes channel the I/O of all responsible clients to the file system and thus, reduce the burden on the file system.

  I/O forwarding could reduce performance because data must be communicated to another node, and might be a bottleneck in a system. However, this depends on the network topology and placement: In indirect network topologies an I/O forwarder can be placed on a node that is directly on the path between client and server. Thus, in such a configuration the I/O forwarding does not imply additional network communication. This is true, for example, for the network topology of a BlueGene system.

  However, optimization techniques, such as caching, aggregation or scheduling, can be implemented on the I/O forwarders that improve performance of I/O servers.

- **Aggregation:** This technique combines a set of smaller requests to a larger request containing all the data and operations. Aggregation can be performed on various layers: on the client side, on intermediate I/O forwarders, on the server or on the I/O subsystem.

  This technique is very useful, when independent operations are interleaved or overlapping, multiple operations can be combined into a bigger sequential access. It is also possible to integrate independent requests into one non-contiguous request, even across disjoint applications, if they access the same file. However, additional buffer space is required to merge the data into the new request. Also, the number of requests to decode is lowered – reducing computation time to process operations.

- **Scheduling:** A scheduler queues pending work and dispatches it according to a strategy. Requests can be deferred to avoid flooding of layers which are unable to process all the pending requests concurrently. In the meantime, the outstanding requests can be reordered or aggregated in order to optimize the access pattern.

  Scheduling algorithms can be applied on different abstraction layers. For example, with NCQ the block device incorporates reordering schemes for block accesses as discussed in the hardware section.

[25] Data corruption means read data does not match the written data any more – this can happen due to bit flips or hardware errors. Silent refers to the fact that this data corruption cannot be detected by the system. Thus, the user is not aware that data has been corrupted.

[26] Note that sometimes in literature the intermediate nodes are referred to as I/O aggregators, too.

Also, scheduling strategies could be applied not only on block-level but also on file-level.

Parallel file system Performance of a parallel file system highly depends on its design as it provides the frame for the deployed optimization strategies. Several aspects like consistency semantics also apply to higher level interfaces like domain specific I/O libraries.

The following important factors tend to vary over the available parallel file systems:

- **Design:** Software architecture of a parallel file system defines the I/O paths within the file system and the distribution of data among the available servers. Naturally, the parallel file system should be able to saturate all participating components (network, servers and I/O subsystem); even with a low number of concurrent clients. Therefore, it is necessary to utilize all components to the best extent. This requires to perform operations on disk and network at the same time. An efficient path for performing I/O on the client side to the server and back is crucial. The higher the abstraction layer of the file system the more intelligent optimizations could be integrated, and the more background information a user can provide the better potential for optimization exists.
- **Implementation:** Usually, computation is not the bottleneck of the file system as CPU's are much faster than the I/O subsystem. However, it is important to be able to saturate both network and disk subsystem; to schedule and optimize operations, data structures are required. Efficient basic data structures like hashes or trees, which scale well with the number of concurrent requests, are necessary to reduce the overhead of the operations. Regarding compute performance, in principle, all the aspects mentioned in Section 2.2.3 apply.
- **Resource consumption:** The parallel file system itself needs some of the available hardware resources to operate. On the client side, these resources are not available for the application and considered overhead. The amount of memory consumed for internal data structures and allocated buffer space reduces the available I/O (and metadata) cache. The number and complexity of instructions which have to be processed in order to prepare or serve requests increase utilization of a CPU – thus, a perfect overlapping of computation and I/O may not be possible. Examples for communication overhead are additional control information transferred within the request and response messages, or server-to-server communication needed for replication.
- **Communication:** Performance factors of the communication between client and server are the steps necessary to establish a connection, to provide data encoding into a common format (in order to support heterogeneous environments), data transfer and the communication protocol. Additional control information must be transferred to the server to initiate the request. Protocols that permit bursts of commands to be transferred in a single request reduce the processing overhead. Support of parallel streams between client and server can improve performance. Several aspects like non-contiguous communication are also discussed in Section 2.1.4.
- **Data distribution:** Distribution of data among the available servers defines the maximum concurrency and highly influences the potential usage of the I/O subsystems. Large contiguous data access is preferable, hence data must be stored in large chunks on a server, yet the number of servers involved should be high to provide a good aggregated performance by utilizing all of them concurrently. Adaptive striping depending on the file size meets both requirements. In most cases automatic re-striping is not supported by the file systems, instead the user (and system administrator) can define the way data is distributed on file creation to match the expected access patterns.
- **Metadata handling:** Regular file system operations involve metadata about the objects, as it must be acquired before I/O can be performed. Metadata performance is crucial to many workloads. Metadata is rather small, therefore it has to be organized in a manner that allows bulk transfers between the servers and persistent storage. Network and I/O subsystem should be utilized, which requires a large number of metadata operations to be initiated per request.

  On the server side, an efficient metadata management is necessary. For example, logical files could

be pre-allocated to already contain all information; thus avoiding additional communication to data servers during file creation. PVFS supports some bulk operations: when listing a directory, for example, a large number of directory entries are transferred in one response. Also, PVFS incorporates read-only client side caches for metadata.

- **Consistency semantics:** The semantics of a file system define how the API calls are translated to file system manipulations and accesses. Of special interest is how and when a server (and a client) realizes concurrent modifications made to file system objects. Relaxed guarantees open the potential for additional optimizations. For instance, when a client does not need to realize all modifications made in the past, it could save communication that would refresh the state of cached objects. Also, operations could be deferred on a client to bundle them with future requests, then other clients would not realize modifications until they were communicated to the servers.

  Assume a client which tries to create a number of files; this client could request file creation of every file individually, or it could avoid some of the communications by assuming the state of the directory is still valid. With strong consistency semantics, locking is mandatory to perform bulk operations because with a lock modifications of the file system status could be prevented. For example, Lustre allows to issue a set of metadata operations at once by locking metadata, and Lustre pre-creates files, too.

  With relaxed consistency semantics, modifications will be lost, if client or server crash prior to execution and thus persistence is not ensured. However, in this case, locking mechanisms might become costly, too; the crash has to be detected and the lock must be released to continue operation.

  Consistency semantics are defined by the application program interface. POSIX for example has very restrictive semantics since I/O operations have to be applied in the same order as issued, even if they are non-overlapping. Thus, communication with the I/O subsystem must be serialized to guarantee proper handling. This is a major drawback of the POSIX interface. Most programs could be adapted to loose semantics to exploit parallelism. PVFS does not provide guarantees for I/O data in case of concurrent operations, data could be a mixture of data blocks of different requests. However, with PVFS all metadata operations are ordered in a specific way to guarantee metadata consistency of objects in the accessible namespace[27].

  Further information on concurrent access is given when MPI-IO semantics are discussed (see Page 49).

- **Locking strategy:** Locking as a mechanism can be used to prevent concurrent access to one (or multiple) file system objects. Depending on the required file system semantics, measures must be taken to ensure that clients access the current state of the file system. While GPFS and Lustre provide a locking protocol, PVFS does not offer locking; hence, when using PVFS the application must avoid concurrent access to file regions. In presence of a locking mechanism, the granularity of the lock defines the potential concurrency with related data and metadata – a lock could be valid for a file region, a logical file itself or whole directories. Different lock types could be available – depending on the type of the lock, another access type could be permitted. A particular lock, for instance, could prevent modifications to a file but allow concurrent reads, while another one may restrict the type of access possible. Locking algorithms can become expensive in a distributed environment, since they require a consistent view.

2.3. Message Passing Interface

The Message Passing Interface [Mes09] is a standard to enable inter-process communication. With MPI the user explicitly encodes the sending and receiving of data in the source code as function calls to MPI.

[27] There could be broken objects not yet linked to the namespace and inaccessible, though. These broken objects can be identified and cleaned by performing a file system check.