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/`

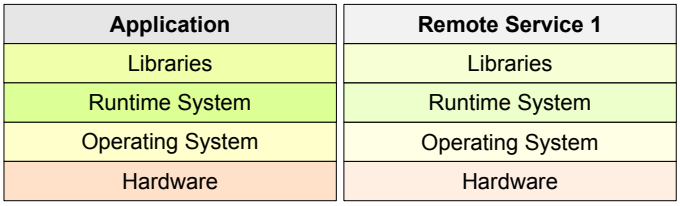


Figure 2.7.: An abstract view of layers and components involved in application execution.

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.

Chapter 2

Background and Related Work

This chapter provides solid background to a rich variety of topics[1]. After discussing each topic, the state of the art and related work are discussed. In this context recent research and several software tools are shown[2].

The first three sections provide more detail about HPC hardware and software, and their performance implications. First, the concepts of file systems and several real world representatives are introduced in Section 2.1. Then, aspects involved in the performance of parallel applications are discussed in Section 2.2. This indicates the complexity of those systems, but also enables classifying of relevant aspects. Next, the Message Passing Interface is introduced in Section 2.3 – special emphasis is put on the optimization potential within MPI.

Fast execution of an application is of main interest to the user since scientific goals should be achieved in time. To design fast programs, a user must understand the run-time behavior of the program on the system on which the program will be executed. Nowadays, users typically measure run-time behavior of their application, locate the bottleneck and try to improve performance in these long-running and thus critical code sections. There are also software-engineering concepts that can be applied during the whole software development cycle that take performance factors into account. Methodologies that lead to improved run-times in the implemented application are discussed in Section 2.4.

In Section 2.5 background about creation and verification of a model for a system is provided. The concept of simulation is introduced, too. A simulator implements a model and allows analyzing its behavior in silico. Further, several simulation tools that ease the implementation of models are introduced. While this section is not related to HPC, it provides relevant background to the art of simulation.

At last, in Section 2.6 a couple of simulators related to the simulation of cluster and I/O systems are presented. None of them allow to simulate parallel I/O and MPI at the level of detail required for this thesis.

2.1. Parallel File Systems

In contrast to a distributed file system, a *parallel file system* is explicitly designed for achieving performant and concurrent access to files. Therefore, internally data of a file is physically scattered among a subset of the available servers and their I/O subsystems. This enables those servers to participate in one I/O operation thus bundling their hardware resources to achieve higher aggregated performance.

This section extends the description of storage in Chapter 1 in three directions. First, by introducing representative enterprise and parallel file systems, important fundamental concepts are described. This enables assessing alternative I/O architectures and client-server communication protocols. Therewith, it helps us to define the scope of the architectural model and communication protocol that should be implemented in the simulation; the model should be flexible enough to represent several file systems. With the *Parallel Virtual File System* (PVFS) the architecture of one representative parallel file system is discussed in detail. Then, the I/O path of PVFS is described. This archetypal path is general enough to represent communication optimization strategies available in parallel file systems. With this knowledge, the I/O

[1] Note that a few passages are based on descriptions given in the author's master's thesis [Kun07].

[2] It is my personal opinion that all mentioned third-party software and published paper deserve a tribute because of the time spent by the authors to provide helpful tools and since they drive computer science forward. It is also my belief that software is never static and must be maintained to deal with new user and platform requirements. Such a dynamic software is probably never bug-free, and at any time some desirable features are missing. If I argue about missing features or suboptimal solutions in this chapter, then it is done in order to distinguish this thesis from existing approaches – all of the references deserve to be honored. It is not my intention to highlight or to criticize any of the papers or software mentioned. And I hope I succeeded in honoring work cited in an objective and constructive way.

path used in the simulator can be assessed better. Furthermore, it supports performance considerations which are discussed in the next section.

But first, a few basic terms:

Hierarchical namespace For convenient access of data, sequences of bytes are organized in file systems. The namespace is a concept that describes the organization scheme. Traditionally, file systems organize data in logical[3] objects: files and directories[4]. Files contain raw data. Internally, a *file* is like an array of bytes that can grow or shrink at the end. Thus, data must be serialized into a sequence of bytes to be stored. Directories structure the namespace by allowing to put other file system objects into them and give them a name, which results in a hierarchy of "labeled" objects.

This common organization scheme for file system objects is called *hierarchical namespace*. An example namespace is provided in Figure 1.3. By knowing the absolute path name within the hierarchical namespace, a logical file is unambiguously identified. Specific bytes of data can be addressed by referring to the offset within the "array" of data and the number of bytes (the size) to read or write . This type of addressing is referred to as *file-level* interface. However, the interpretation of the accessed bytes must be known to the program accessing it. For convenient access, a hierarchical namespace supports typically alternative names for individual objects, i.e., a single object can be found under different absolute file names. This is achieved by storing a reference to the original data (or directory) under an alternative name – this reference is called link. With a *global namespace*, the file system hierarchy can be accessed from multiple components in a distributed environment.

Beside the hierarchical namespace, there are other data access paradigms: In the cloud storage provided by *Amazon Simple Storage Service* (S3), data is referenced by a bucket (which can be thought of as a folder) and by using a key (similar to a file name). Arbitrary information can be stored for a given key. With the *Structured Query Language* (SQL), databases offer a high-level approach to access and to manipulate structured data. In contrast to a file-level interface data is managed on a higher level of abstraction: stored data is structured, every element of the structure has a label and datatype. Also, with SQL a user specifies the logic of the operations to perform with data and not the control flow that defines execution[5].

Client and Server Applications (and their processes) that access objects of a distributed file system are referred to as *clients*. In the context of hardware, the term describes a node hosting at least one process accessing the distributed file system. In this sense a node that provides parts of the parallel file system to a client is referred to as *server*.

(Meta)data Data refers to the raw content of a file. *Metadata* refers to the information about files and other file system objects themselves – the organization of objects in the namespace and their attributes. Attributes like timestamps or access permissions describe file system objects further. Usually, data and metadata are treated differently within the file system because of the semantic difference and the amount of stored information.

Block storage File systems use *block storage* to persist data. Block storage offers a block-level interface: Storage space is partitioned into an array of blocks that can be read or written individually. Access must be performed with a granularity of full blocks (typically 512 byte or 4 KiB), that means no block can be accessed partially. A number in a linear space specifies which block to access – this scheme is called *logical block addressing* (LBA). The relation between the blocks a file system object is made of is not defined on the block-level and must be managed by the file system.

[3] The term logical refers to the fact that this kind of object is accessed by using the file system interface. Internally, the file system might use several objects that work together to look like the logical object (e.g., file) to the user.

[4] Directories are also referred to as folders.

[5] Although there are procedural extensions, SQL is a declarative programming language.

A block device is a single hardware component that offers such a block-level interface. Multiple devices can be combined into a larger block storage that looks like a single block device.

2.1.1. Capabilities of Parallel File Systems

There are several requirements for file systems: persistence, consistence, performance, and manageability, just to name a few. *Persistence* describes that stored data can be accessed any time later. Also, the namespace should be in a correct state and all data read should also match the data that has been stored (*consistence*). Both requirements are vital because production of data is costly, and reading of corrupted (wrong) data can be disastrous. *Performance* describes the requirement that the file system should be able to utilize the underlying block storage efficiently[6]. Tools must be supported to mount the file system, to check the correctness or to repair a broken file system.

Additionally, for parallel file systems scalability, fault-tolerance and availability are of interest. *Scalability* of the file system allows deployment of the file system to provide sufficient performance for arbitrary workloads and to operate with any number of clients; performance just depends on the provided hardware resources and is not limited by the software. A *fault-tolerant* file system can tolerate transient and persistent errors to a certain extent without corrupting data, although the file system might be unavailable while errors are being corrected and it might crash when an error occurs. *Availability* describes a file system that continues to operate in the presence of hardware and software errors (usually with degraded performance).

The concepts of scalability and fault-tolerance will be briefly illustrated. In the context of this thesis other features are not relevant and therefore subsequently disregarded.

Scaling performance and capacity The architecture of most existing file systems and appliances can handle the demand of any customer ranging from small to very big (and fast) systems – these architectures are considered to be *scalable*. Furthermore, the requirements for an installed storage system might change over time. Since a storage system is costly and management of multiple systems is difficult, it is also important that the existing system can *scale* with increasing storage demands, either in performance or capacity. Otherwise, the money invested will be lost. Therefore, major enterprise file system vendors offer seamless upgrades of already installed solutions.

In the example storage system depicted in Figure 2.1 – it can be imagined that the single NAS server may be a bottleneck. With a scalable architecture, the storage system could be upgraded easily, with minimal modifications to the existing infrastructure; in the example additional NAS servers could be integrated.

There are two orthogonal principles to extend the capabilities of an existing system: *scale out* and *scale up*. The term *scale out* (or *scale horizontally*) is used to advertise modular systems which allow adding more storage nodes as they are needed – additional storage nodes add further capacity and performance at the same time. In contrast *scale up* (or *scale vertically*) means to equip the existing infrastructure with faster components; for example, by replacing a complete node or a single hardware component of the server. While upscaling is limited by the capability of available hardware technology, a scale-out solution provides high extensionability by adding more components. Both principles also apply to cluster computers.

To enable scaling, vendors typically put a distributed file system on top of storage nodes that aggregates all resources into a global namespace; data of a single file is distributed across all nodes. Thus, with an increasing number of storage nodes, the available capacity and performance can be scaled horizontally to any demand.

[6] Performance aspects of file systems are discussed explicitly in section 2.2.

Fault-tolerance Fault-tolerance as a requirement for high availability in parallel file systems is usually provided by replicating data over multiple devices in such a manner that permanent failure of a single component does not corrupt the file system integrity. Keeping a complete copy of data (mirroring) is costly as twice as much space is required. Therefore, mainly error-correcting codes such as the *RAID* levels 1 to 6, or *Reed-Solomon* codes are implemented.

Traditionally, error-correcting codes are applied on the block-level, in most cases multiple block devices are combined into a *redundant array of independent disks* (RAID). The RAID looks like a regular block device to the file system. If an error occurs and a block device of the array must be replaced, then the data of the broken device must be reconstructed by reading data from all disks and writing the lost information to the new block device.

File-level RAID is a software concept in which the file system controls the redundancy explicitly – for every file, the locations of the file blocks including the redundant data blocks are known. In contrast to block-level RAID, this has the advantage that hardware problems in the system only trigger a rebuild of the currently used space. Thus, empty (not-allocated) space of the system is not rebuilt. Failures that happen during the rebuild of a file, do not invalidate the whole RAID and thus file system – instead, the broken file can be identified and reported.

2.1.2. State of the Art

This section describes some parallel file systems that are deployed in enterprises and in a HPC center. This will allow us later to design an abstract communication protocol to simulate their interactions. Since performance of a file system depends on the communication path between client and servers, the focus of this section lies on the I/O path.

Enterprise storage In enterprise business, often a complete storage "solution" or *appliance* is purchased that is a bundle of hardware and software which can be integrated seamlessly into the existing communication network. Several commercial vendors for enterprise and datacenter storage sell *Network Attached Storage* (NAS) systems. Well-known vendors of network storage are: Panasas [NSM04], Netapp, Xyratex, BlueArc and Isilon [Kir10].

A customer connects the purchased storage system to the existing network infrastructure and accesses storage on the new system via standardized remote network protocols like the *Network File System* (NFS) or the *Common Internet File System* (CIFS)[7].

With NFS and CIFS, a remote node or workstation connects to exactly one file server (this scheme is illustrated in Figure 2.1). Clients forward file system operations to this server which executes them on a local file system. The block storage persisting this file system can be either integrated into the server, attached to it or provided in a *storage area network* (SAN)[8]. Recently, distributed file systems rely on *Object-based Storage Devices* (OSD) [Pan04] as underlying storage. Compared to low-level block devices, object-based storage provides a higher level of abstraction – access is performed by addressing file system objects directly. See Figure 1.7 and the descriptions on Page 9 for an example data access on a file system and the underlying block storage.

Panasas ActiveStor With *ActiveStor* Panasas delivers a performant and extensible system. The customer buys blade enclosures, which are equipped with a number of metadata servers (so-called director blades) and/or storage servers (so-called storage blades). Internally, the parallel file system *PanFS* distributes data across the available hardware. Metadata operations are performed on director blades, while data is stored on storage blades. The storage grows by adding further metadata and storage server blades or new

[7] An advantage of using those protocols is that they are available for most operating systems.

[8] A SAN is an additional network for block oriented storage. Devices that is part of a SAN, is usually addressed with the SCSI protocol. A single storage can be utilized from multiple servers, although not concurrently.

HPC user has a limited amount of time to adapt an application to a given system and to configure the environment in an optimal way.

Consequently, even if very efficient algorithms are implemented resources are often wasted. The following lists a few reasons why resources might be utilized suboptimally:

- Limited scalability of the algorithm adds computational and communication overhead.
- A wrong mapping of processes to hardware leads to unnecessary communication.
- An inefficient I/O access pattern degrades performance of the parallel file system.
- Load imbalance in I/O or computation causes bottlenecks and idle resources on the system.

Efficiency also depends on the system characteristics, that is the hardware characteristics of the basic components: network, storage and compute nodes; the topologies of all interconnects, software layers involved and finally, on the configuration made by the administrator and user.

In software layers the system's algorithms for communication and I/O have a large influence on the observable performance[14]. Performance of the communication library itself is influenced by a number of factors. First, it is obvious that the library should be geared towards a given system, after which the right communication algorithm must be chosen. However, the tuning is complicated because performance of each communication algorithm depends on the parameters specified by the user. For example, while the broadcasting of small messages could be faster on the given hardware by sending data sequentially from the root, a tree algorithm might perform better for larger data, and a pipelined peer-to-peer alike algorithm would be preferable for big data. While the performance of the algorithm could be optimized for itself, efficiency also depends on the context of the application, that is, the number of processes and the current and future instructions executed by the processes.

Heterogeneous systems like Grids and Clouds increase complexity of the optimization exponentially, as more inhomogeneous components are involved.

It certainly is not easy to understand the interplay between all the hardware characteristics and potential bottlenecks, and unfortunately, mechanisms designed to optimize the system make it even harder to assess achieved performance and relate that performance to the system's capability. Further, on a high level of abstraction the communication interface implementation and I/O layers use techniques that, in most cases, improve performance. This is achieved by manipulating the request, deferring the operation, or fusing operations into one compound operation.

The complexity of analyzing an application on a distributed supercomputer becomes clear when a developer tries is to understand performance of a particular sequential code, which is only executed on a single processor. On the one hand, observed performance depends on the CPU architecture – branch-prediction, caches, the translation lookaside buffer, to name just a few mechanisms for optimizing hardware. On the other hand, it also depends on the compiler, which tries to transform the given high-level code to machine instructions in the best way possible.

From the user's perspective, post-mortem performance optimization is state-of-the-art. In this process performance of an application is measured on an existing system and analyzed to identify bottlenecks. Nowadays, developers are happy to achieve 10% peak performance on a given system[15]. It is important to optimize from the most promising and performance-boosting bottleneck to the least one. When confronted with this issue an important question arises concerning how much work is necessary to tune or modify the algorithm, the code and the system, and what will be gained by these modifications.

Estimating performance Modeling the system allows assessing obtained performance and therewith estimate the performance potentially gained by optimizations. Simple approaches to optimize floating point operations are to measure them and compare them with the theoretical peak performance of the system.

[14] More information of relevant performance factors is given in Section 2.2.
[15] This information is derived from available presentations of compute centers and scientific applications.

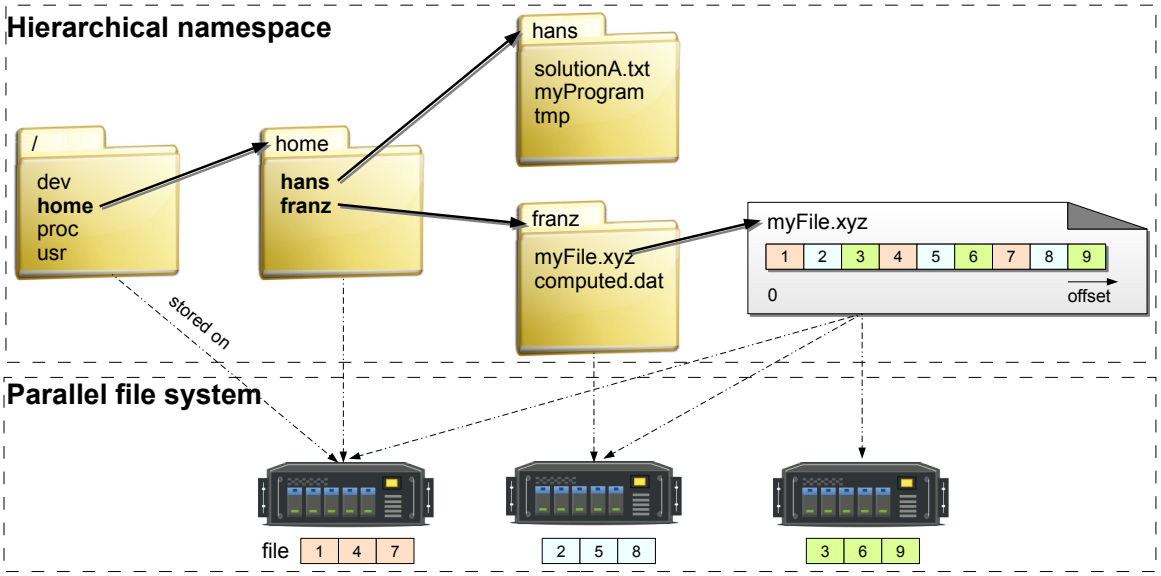


Figure 1.3.: Example hierarchical namespace and mapping of the objects to servers of a parallel file system. Here, metadata of a single logical object belongs to exactly one server, while file data is distributed across all servers.

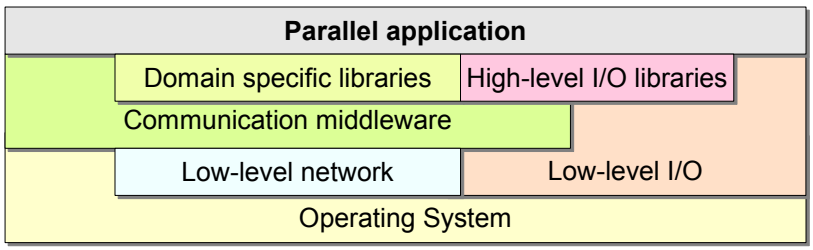


Figure 1.4.: Representative software stack for parallel applications.

storage devices is low-level. At the block-level, data can be accessed with a granularity of full blocks by specifying the block number and *access type* (read or write).

In a *Storage Area Network* (*SAN*) the block device seems to be connected directly to the server. To communicate with the remote block storage device, servers use the *Small Computer System Interface* (SCSI) command protocol. A SAN could share the communication infrastructure, or another network just for I/O can be deployed. Fibre Channel and Ethernet are common network technologies with which to build a SAN. In the latter case, SCSI commands are encapsulated into the IP protocol, that is, the so-called *Internet SCSI* (iSCSI). Therefore, the existing communication infrastructure can be used for I/O, too.

1.1.2. Software Layers

Several software layers are involved in running applications on supercomputers. A representative software stack is shown in Figure 1.4.

A *parallel application* uses a domain-specific framework and libraries to perform tasks common to most applications in that field. For example, the numerical library Atlas[11] is widely adopted by scientists.

Since collaboration between remote processes of an application requires communicating data, a conveniently programmable interface and an efficient implementation is important. The service to exchanged data is offered by the *communication middleware*. Several programming paradigms and models exist for

[11] Automatically Tuned Linear Algebra Software

each architectural type of the parallel computer, and those models often explicitly address the way communication is performed. Communication models can be classified according to the characteristics of the model. For instance, a classification made by the level of abstraction for communication distinguishes whether data exchange happens automatically whenever necessary, (i.e., is hidden from the programmer), or if data exchange must be encoded explicitly. The middleware might offer a high-level interface that abstracts from the physical location of processes, instead the user just specifies the particular communication partner.

Two models used to program the distributed memory architecture of cluster machines are MPI and PGAS. With the *Message Passing Interface* (MPI) [Mes09] the programmer embeds instructions in his code to explicitly send and receive messages. MPI also offers routines for parallel input and output of data. With *Partitioned Global Address Space* (PGAS) a process can access data that is stored in remote memory by using the syntax of the programming language, such as array access. Run-time environments ensure that, if required, data is transferred between the systems. However, the data partitioning is still encoded by the programmer. The language extensions which enable remote memory access in C and Fortran are called *Unified Parallel C* (UPC) or *Coarray Fortran* (CAF), respectively.

It is common practice to use MPI to communicate within a distributed memory architecture, and to use *Open Multi-Processing* (OpenMP) [BC07] to collaborate within a single node. Since developers may use both programming paradigms at the same time, the efficiency of this hybrid-programming model is specific to the problem being addressed and to the underlying hardware.

Input data and results are accessed either by harnessing *high-level I/O libraries* or by using low-level I/O interfaces such as the *Portable Operating System Interface* (POSIX). The *Network Common Data Form* (NetCDF) [HR08] and the *Hierarchical Data Format* (HDF5) are common high-level I/O libraries, that hide the complexity of defining low-level data formats from the user and offer features such as automatic data conversion. I/O can be performed with semantics close to the data structures used in the code. Domain specific libraries or high-level I/O libraries could be parallelized with MPI or OpenMP in order to utilize available cluster resources.

Low-level network communication enables the node to transfer data with another node. In contrast to a communication middleware, it operates directly with the network device and uses the network specific address formats. Programmers of an application usually work with the communication middleware, which provides a higher level of abstraction to address remote processes that is closely related to application logic. For example, the programmer should not have to care about the address (i.e., host name and port) on which the process might be listening. Distributed file systems provide their own *low-level I/O* interface to transmit operations and data between the storage servers and the node calling for I/O.

An *operating system* (OS) controls the local hardware and provides a software basis to run applications on a node. Often, the POSIX standard is supported by the OS. However, to reduce the overhead and complexity of background interaction a few supercomputers provide a reduced operating system. BlueGene, for example, offers a compute kernel with restricted threading support for processes [AAA+02].

The figure is representative of most applications, but theoretically a layer could use all underlying layers directly. For example, a parallel application could call the functionality of the operating system to drive a network interface, however, programming of the communication would be cumbersome.

Software layers can be provided by the vendor of the technology, a supercomputer vendor, a cluster integrator or by the open source community. For performance reasons a cluster's integrator often gears the software stack towards the cluster's hardware.

1.1.3. Example Application Execution

Consider a simple parallel MPI program running on four processors; each node has two logical processors which execute commands concurrently. Figure 1.5 shows the relevant code for each process: An input matrix is read on Process 0 by a high-level I/O library and broadcasted to all processes. Each process

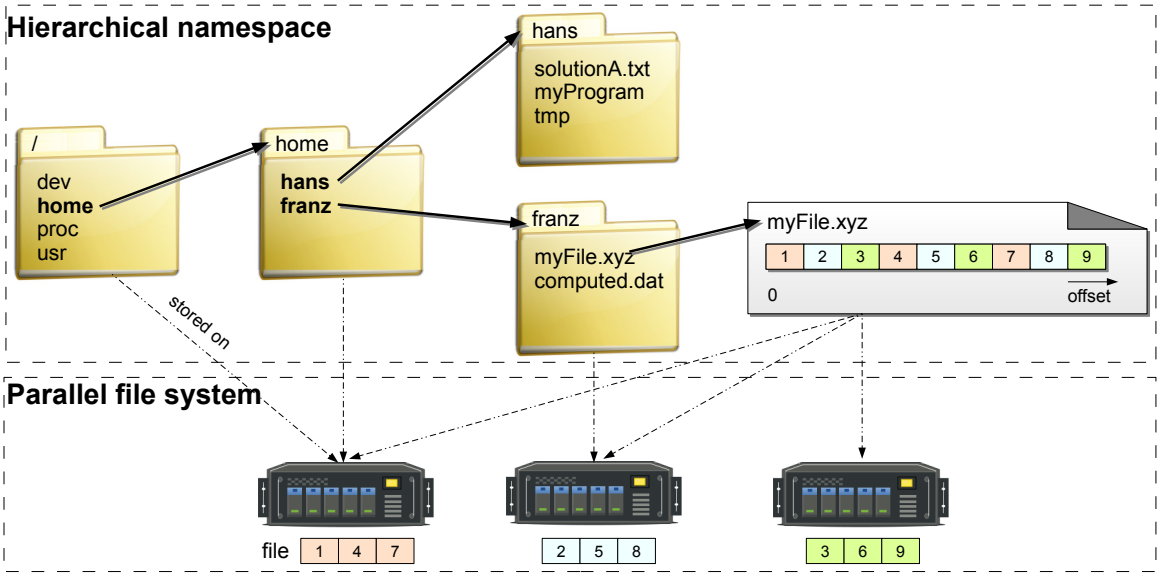


Figure 1.3.: Example hierarchical namespace and mapping of the objects to servers of a parallel file system. Here, metadata of a single logical object belongs to exactly one server, while file data is distributed across all servers.

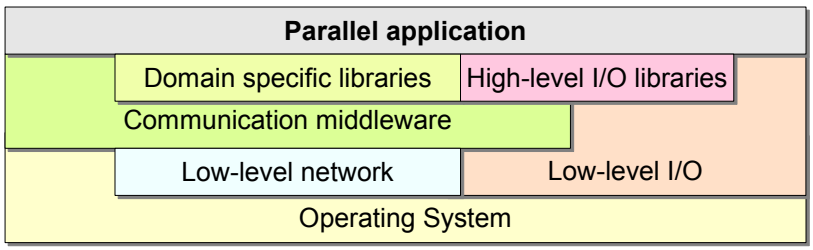


Figure 1.4.: Representative software stack for parallel applications.

storage devices is low-level. At the block-level, data can be accessed with a granularity of full blocks by specifying the block number and *access type* (read or write).

In a *Storage Area Network* (*SAN*) the block device seems to be connected directly to the server. To communicate with the remote block storage device, servers use the *Small Computer System Interface* (SCSI) command protocol. A SAN could share the communication infrastructure, or another network just for I/O can be deployed. Fibre Channel and Ethernet are common network technologies with which to build a SAN. In the latter case, SCSI commands are encapsulated into the IP protocol, that is, the so-called *Internet SCSI* (iSCSI). Therefore, the existing communication infrastructure can be used for I/O, too.

1.1.2. Software Layers

Several software layers are involved in running applications on supercomputers. A representative software stack is shown in Figure 1.4.

A *parallel application* uses a domain-specific framework and libraries to perform tasks common to most applications in that field. For example, the numerical library Atlas[11] is widely adopted by scientists.

Since collaboration between remote processes of an application requires communicating data, a conveniently programmable interface and an efficient implementation is important. The service to exchanged data is offered by the *communication middleware*. Several programming paradigms and models exist for

[11] Automatically Tuned Linear Algebra Software

each architectural type of the parallel computer, and those models often explicitly address the way communication is performed. Communication models can be classified according to the characteristics of the model. For instance, a classification made by the level of abstraction for communication distinguishes whether data exchange happens automatically whenever necessary, (i.e., is hidden from the programmer), or if data exchange must be encoded explicitly. The middleware might offer a high-level interface that abstracts from the physical location of processes, instead the user just specifies the particular communication partner.

Two models used to program the distributed memory architecture of cluster machines are MPI and PGAS. With the *Message Passing Interface* (MPI) [Mes09] the programmer embeds instructions in his code to explicitly send and receive messages. MPI also offers routines for parallel input and output of data. With *Partitioned Global Address Space* (PGAS) a process can access data that is stored in remote memory by using the syntax of the programming language, such as array access. Run-time environments ensure that, if required, data is transferred between the systems. However, the data partitioning is still encoded by the programmer. The language extensions which enable remote memory access in C and Fortran are called *Unified Parallel C* (UPC) or *Coarray Fortran* (CAF), respectively.

It is common practice to use MPI to communicate within a distributed memory architecture, and to use *Open Multi-Processing* (OpenMP) [BC07] to collaborate within a single node. Since developers may use both programming paradigms at the same time, the efficiency of this hybrid-programming model is specific to the problem being addressed and to the underlying hardware.

Input data and results are accessed either by harnessing *high-level I/O libraries* or by using low-level I/O interfaces such as the *Portable Operating System Interface* (POSIX). The *Network Common Data Form* (NetCDF) [HR08] and the *Hierarchical Data Format* (HDF5) are common high-level I/O libraries, that hide the complexity of defining low-level data formats from the user and offer features such as automatic data conversion. I/O can be performed with semantics close to the data structures used in the code. Domain specific libraries or high-level I/O libraries could be parallelized with MPI or OpenMP in order to utilize available cluster resources.

Low-level network communication enables the node to transfer data with another node. In contrast to a communication middleware, it operates directly with the network device and uses the network specific address formats. Programmers of an application usually work with the communication middleware, which provides a higher level of abstraction to address remote processes that is closely related to application logic. For example, the programmer should not have to care about the address (i.e., host name and port) on which the process might be listening. Distributed file systems provide their own *low-level I/O* interface to transmit operations and data between the storage servers and the node calling for I/O.

An *operating system* (OS) controls the local hardware and provides a software basis to run applications on a node. Often, the POSIX standard is supported by the OS. However, to reduce the overhead and complexity of background interaction a few supercomputers provide a reduced operating system. BlueGene, for example, offers a compute kernel with restricted threading support for processes [AAA+02].

The figure is representative of most applications, but theoretically a layer could use all underlying layers directly. For example, a parallel application could call the functionality of the operating system to drive a network interface, however, programming of the communication would be cumbersome.

Software layers can be provided by the vendor of the technology, a supercomputer vendor, a cluster integrator or by the open source community. For performance reasons a cluster's integrator often gears the software stack towards the cluster's hardware.

1.1.3. Example Application Execution

Consider a simple parallel MPI program running on four processors; each node has two logical processors which execute commands concurrently. Figure 1.5 shows the relevant code for each process: An input matrix is read on Process 0 by a high-level I/O library and broadcasted to all processes. Each process

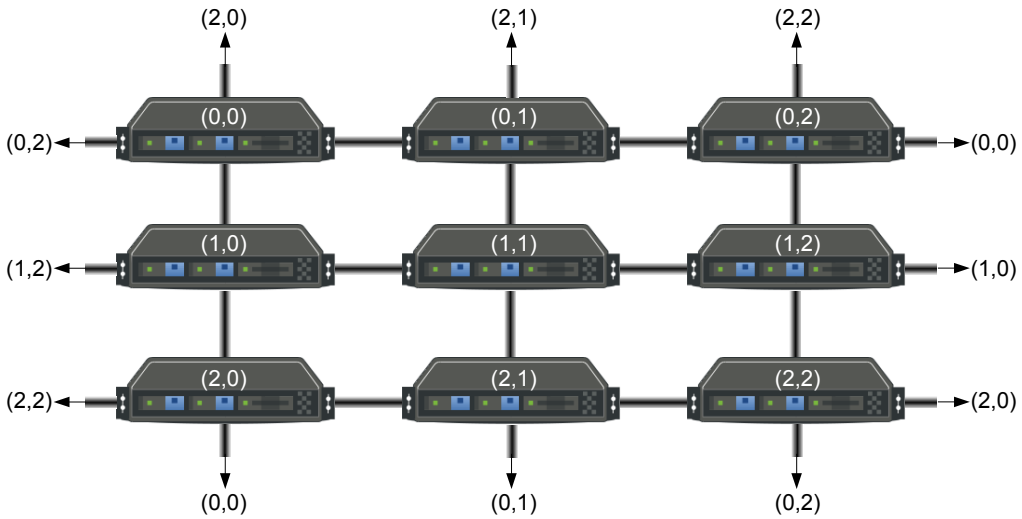


Figure 1.2.: Network topology of a 2D-Torus. Leftmost nodes (X,0) are interconnected with the rightmost nodes (X,2). Top nodes (0,X) are connected with the bottom nodes (2,X).

send a message to the center node (1,1). The routing algorithm might route both messages via node (0,1); however, the node has only one link to node (1,1).

Another frequently deployed class of network topologies are *hierarchical networks* which add intermediate layers of hardware that relay the data; a switch connects many links and forwards incoming data into the direction of the intended receiver. This approach provides a higher aggregated bandwidth between the nodes, but increases the latency of the communication. One hierarchical network topology is a *Clos* network, where a high number of links between switches offers the maximum available network bandwidth between all connected hosts. This network topology is often deployed with Infiniband technology. Refer to [AK11] for more details on network technologies and topologies.

Storage Data management in bigger clusters is performed by distributed file systems, which scale with the demands of the users. Usually, multiple file systems are deployed to deal with disjoint requirements.

In a typical setup two file systems are provided: a fast and large scratch space to store temporary results, and a slower, but highly available volume to store user data (e.g., for home directories). Looking at a particular distributed file system, a set of storage servers provides a high-level interface to manipulate objects of the namespace. The *namespace* is the logical folder and file structure the user can interact with; typically it is structured in a hierarchy. Files are split into parts which are distributed on multiple resources and which can be accessed concurrently to circumvent the bottleneck of a single resource. Replication of a part on multiple servers increases availability in case of server failure. Truly *parallel file systems* support concurrent access to disjoint parts of a file; in contrast, conventional file systems serialize I/O to some extent[10].

An example of the *hierarchical namespace* and its mapping to servers is given in Figure 1.3. The directory "home" links to two subfolders which contain some files. In this case, directories are mapped to exactly one server each, while the data of logical files is maintained on all three servers. Data of "*myFile.xyz*" is split into ranges of equal size and these blocks are distributed round-robin among the data servers. Each server holds three ranges of the file.

Storage devices are required to maintain the state of the file system in a persistent and consistent way. A storage device can be either directly attached to a server (e.g., a built in hard disk), or the server controls devices attached to the network. In contrast to the interface provided by file systems, the interface to

[10]More details on file systems in HPC systems are provided in Section 2.1.

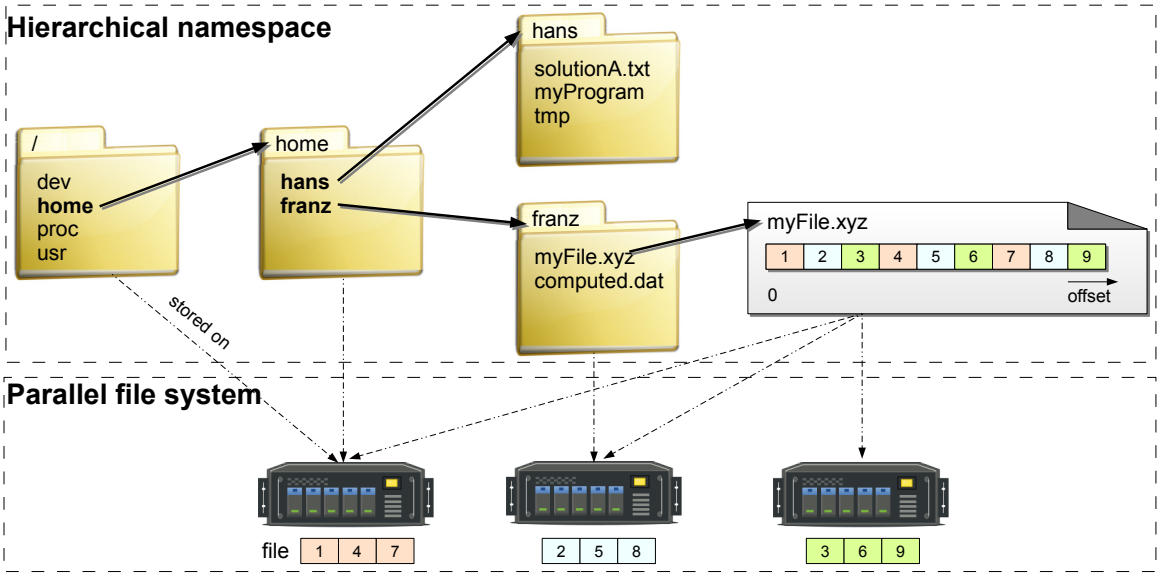


Figure 1.3.: Example hierarchical namespace and mapping of the objects to servers of a parallel file system. Here, metadata of a single logical object belongs to exactly one server, while file data is distributed across all servers.

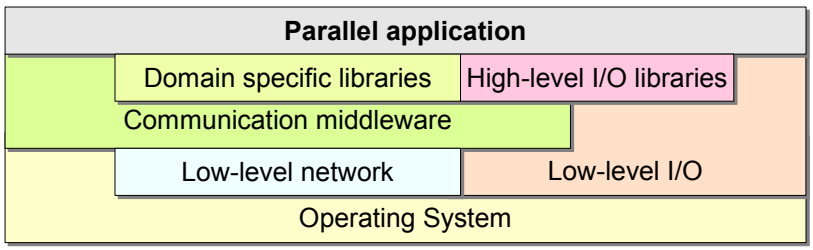


Figure 1.4.: Representative software stack for parallel applications.

storage devices is low-level. At the block-level, data can be accessed with a granularity of full blocks by specifying the block number and *access type* (read or write).

In a *Storage Area Network* (*SAN*) the block device seems to be connected directly to the server. To communicate with the remote block storage device, servers use the *Small Computer System Interface* (SCSI) command protocol. A SAN could share the communication infrastructure, or another network just for I/O can be deployed. Fibre Channel and Ethernet are common network technologies with which to build a SAN. In the latter case, SCSI commands are encapsulated into the IP protocol, that is, the so-called *Internet SCSI* (iSCSI). Therefore, the existing communication infrastructure can be used for I/O, too.

1.1.2. Software Layers

Several software layers are involved in running applications on supercomputers. A representative software stack is shown in Figure 1.4.

A *parallel application* uses a domain-specific framework and libraries to perform tasks common to most applications in that field. For example, the numerical library Atlas[11] is widely adopted by scientists.

Since collaboration between remote processes of an application requires communicating data, a conveniently programmable interface and an efficient implementation is important. The service to exchanged data is offered by the *communication middleware*. Several programming paradigms and models exist for

[11] Automatically Tuned Linear Algebra Software

during the open call. All other participating processes receive the information required to access the file by a broadcast from the root and thus the burden of the metadata server is reduced.

The collective optimization of the *Two-Phase* protocol is discussed in [CCC+03]. With Two-Phase, processes exchange their spatial access pattern and coordinate amongst themselves, which process will access a sequential file domain – data to be accessed is partitioned among all clients. Then, (for a read operation) the clients repeat two phases: a set of the processes reads the assigned file domains sequentially, then during the communication phase data is shipped to the clients which needs it[35].

Multiple-Phase Collective I/O, an extended version of the Two-Phase protocol, is presented in [SIC+07, SIPC09]. With Multi-Phase I/O, the communication phase is split up in several steps, in which pairs of clients communicate with each other in parallel. These multiple steps are used to progressively increase the locality of the data to be accessed by aggregating more operations into larger blocks. Then, during the I/O phase, sequential blocks can be accessed; the block size of the sequential access depends on the number of steps performed.

In [HYC05], the collective I/O scheme is adjusted to exploit the features of Infiniband.

A cooperate cache is integrated in [PTH+01], this allows GPFS to cache read and defer write operations. With a write-behind strategy, write operations are buffered. Effectively, data updates and storing the changed data on the file system happens concurrently. To ensure consistency, every physical data block is assigned to exactly one client which performs all I/O operations.

In [Wor06] an adaptive approach is introduced which automatically sets hints for collective I/O, based on the access pattern, topology and the characteristics of the underlying file system.

Higher-level I/O optimizations The following two approaches are not optimizing MPI-IO by itself, instead they provide a layer above MPI that enables further interesting optimizations.

Initially, *SIONlib* [FWP09] was developed to deal with I/O forwarding and communication topology of a BlueGene system. This library channels I/O operations to a logical file from POSIX via MPI, i.e., in a program regular POSIX (e.g., `fwrite()`) calls are used, which are mapped to a set of shared files by using the MPI-IO interface. Depending on the underlying system, one or multiple files are generated to optimize performance. Conflicts on parallel access of a shared file are reduced, yet, the number of files is less than the number of processes which minimizes metadata overhead on the underlying file system. For instance, on a BlueGene, all processors routing to a particular I/O aggregator can be mapped to one physical file. Hence, the I/O forwarder does not have to share access to the file with other aggregators. The mapping from logical to physical files is hidden behind the library and transparent to the application. The user just specifies the number of files in the library open call and whether or not collective calls should be performed.

The *Adaptable IO System* (ADIOS) [LKK+08, LZKS09] provides an abstract I/O API and library that decouples application logic from the actual I/O setting. In an XML file, the desired I/O method can be selected and parameterized – I/O operation can be realized either with HDF5, MPI (collective or independent), POSIX or several asynchronous staging methods. With ADIOS, each I/O performed in a C (or Fortran) program is annotated with a name that can be referred to in the XML file. The amount of data accessed, datatype and further attributes of the call are defined in the XML[36]. An advantage of decoupling the underlying I/O procedure is that the best-fitting implementation can be selected for a group of files.

Settings for the implementation, e.g., buffer size, can be defined without changing code. Moreover, data could be forwarded to a *visualization system*, simply discarded, or even multiple I/O methods can be selected to visualize and store data at the same time. Similar to SIONlib, the system is able to either write a shared file or to split logical I/O into several file system objects. With ADIOS, the *BP* file format is proposed that improves data locality for a single process and minimizes collisions between the processes.

[35] The communication and exchange phases are swapped for write operations.
[36] Elementary datatypes or arrays of arbitrary dimension are supported.

The API provides functions to the programmer to indicate when the computation starts or ends, or when the scientific application main loop occurs. On the one hand, this enables efficient communication to the servers without disturbing application communication. On the other hand, the pace in which data is created and written back is announced to the library. Concluding, ADIO provides a completely new API in which the programmer is forced to deal with I/O related aspects consciously – but due to the XML, system optimizations are possible without source-code modifications.

In our paper [KMKL11], the ADIOS interface is explained in detail. In this paper the interface is extended to offer visualization capabilities and improved energy efficiency.

Tuning library settings MPI libraries like IBM's *Parallel Environment* or Open MPI offer a rich set of environment specific parameters to tune the library internals towards a system or application. For example buffer sizes for the eager message protocol can be adjusted to the network characteristics. In Open MPI, the *Modular Component Architecture (MCA)* provides more than 250 parameters on a COST Beowulf cluster. The libraries provide empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Thus, the defaults might achieve only a fraction of theoretical performance. To provide a starting point for application specific tuning of those values, an administrator should provide appropriate values for the given system.

Chaarawi et.al. developed the *Open Tool for Parameter Optimization (OTPO)* for Open MPI which uses the automatic optimization algorithm from ADCL to determine the best settings of available MCA parameters for a given cluster system [CSGF08]. OTPO could be configured and run by administrators to set up efficient cluster defaults.

2.4. Performance Analysis and Tuning

In computer science, *performance analysis* refers to activity that fosters understanding in timing and resource utilization of applications. In terms of a single computer, the CPU, or to be more formal, each functional unit provided by the CPU, is considered to be a resource. Therefore, understanding resource utilization includes understanding run-time behavior and wall-clock time. For parallel applications, the concurrent computation, communication and parallel I/O increase the complexity of the analysis. Therefore, many components influence the resource utilization and run-time behavior; those have been discussed in Section 2.2.1.

Computational complexity theory is the field of computer science that provides methods to classify and estimate algorithm run-time depending on the problem size. Theoretical analysis of source code is usually infeasible as utilization of hardware at run-time can only be roughly estimated. Therefore, in practice, theoretical analysis is restricted to small code-pieces or clear application kernels, and typically software behavior is measured and assessed.

Programs can be classified according to their utilization characteristics and demand – important algorithms are categorized into 13 *motifs* [ABC+06]. Most applications could be thought of as a combination of the basic functionality required by those motifs. But even so, the characteristics of each real program must be analyzed individually.

In this section, it is first shown how application design and software engineering can assist in developing performance-demanding applications (Section 2.4.1). Those methods focus on integrated and automatic development to achieve efficient and performant applications.

As scientific programs usually require a huge amount of resources, one could expect them to be especially designed for performance. Unfortunately, that is not the case. One reason is that many scientific codes evolved over decades at a time when performance has been of low priority. Usually performance is analyzed after the correctness of the program has been evaluated. At this late stage, a functional version of the

Analyzing data Users analyze the data recorded by the monitoring system to localize optimization potential. Performance data is either recorded during program execution and assessed after the application finished, this approach of *post-mortem* analysis is also referred to as *offline* analysis. An advantage of this methodology is that data can be analyzed multiple times and compared with older results. Another approach is to gather and assess data *online* – while the program runs. This way feedback is provided immediately to the user, who could adjust settings to the monitoring environment depending on the results.

Due to the vast amount of data, sophisticated tools are required to localize performance issues of the system and correlate them with application behavior and finally identify source code causing them. Tools operate either manually, i.e., the user must inspect the data himself; a *semi-automatic tool* could give hints to the user where abnormalities or inefficiencies are found, or try to assess data automatically. Tool environments, which localize and tune code automatically, without user interaction, are on the wishlist of all programmers. However, due to the system and application complexity those are only applicable for a very small set of problems. Usually, tools offer analysis capability in several *views* or *displays*, each relevant to a particular type of analysis.

2.4.3. Available Tools for Analysis of Sequential Programs

There exist plenty of tools that assist in performance analysis and optimizations of sequential code, a handful of tools of different categories are briefly introduced: *GNU gprof* generates a profile of the application in user-space. *OProfile* can record and investigate application and system behavior including activity of the Linux kernel. CPU counters can be related to the individual operations. *PAPI* is a library which accesses CPU counters and provides additional hardware statistics. *Likwid* is a lightweight tool suite that reads CPU counters for an application. *LTTng* traces and visualizes activity of processes and within the kernel. However, compared to OProfile symbolic information of the application program is not supported.

All tools mentioned are licensed under an open and free license. The state of the latest stable versions available is discussed as of February 2011.

GNU gprof

GNU compilers can be instructed to include code into a program that will periodically collect samples of the program counter. During run-time profiles of function call timings and the *call graph*[40] are stored in a file. This profiling data can then be analyzed by the command line tool `gprof`.

To demonstrate the application of the tool, an excerpt of the `gprof` output for a run of the `partdiff-seq` PDE solver is given in Listing 2.1. In the *flat profile* (up to Line 12), the time is shown per function. While 6 functions have been invoked once (Lines 6 to 11), all run-time is spent in the function `calculate()` (Line 6). The textual representation of the call graph is also provided starting with Line 15 of the output. In Line 20 and Line 21, it is shown that the function `calculate()` is called from main once, additional invocations from different functions would generate further sections. The main function calls all 6 subroutines (Lines 24-30).

Most platforms provide tools alike to `gprof` to analyze performance of sequential programs. To analyze a parallel application, a profiler must be aware of the parallelism and provide an approach to handle it, for example, by generating one output for each of the spawned processes.

Listing 2.1: Excerpt of a `gprof` output for partdiff-seq

```
1  Flat profile:
2
3   Each sample counts as 0.01 seconds.
4    %   cumulative   self              self     total
5   time   seconds   seconds    calls   s/call   s/call  name
6  100.05     30.95    30.95        1    30.95    30.95  calculate
7    0.00     30.95     0.00        1     0.00     0.00  AskParams
```

[40] The call graph is a directed graph providing information about function invocation, the nodes of the graph represent functions and the edges function calls.

```
  0.00     30.95     0.00        1     0.00     0.00  DisplayMatrix
  0.00     30.95     0.00        1     0.00     0.00  allocateMatrices
  0.00     30.95     0.00        1     0.00     0.00  displayStatistics
  0.00     30.95     0.00        1     0.00     0.00  initMatrices

[...]

                     Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.03% of 30.95 seconds

index % time    self  children    called     name
                30.95    0.00       1/1           main [2]
[1]    100.0   30.95    0.00       1         calculate [1]
-----------------------------------------------
                                                  <spontaneous>
[2]    100.0    0.00   30.95                 main [2]
                30.95    0.00       1/1           calculate [1]
                 0.00    0.00       1/1           AskParams [3]
                 0.00    0.00       1/1           initMatrices [7]
                 0.00    0.00       1/1           allocateMatrices [5]
                 0.00    0.00       1/1           displayStatistics [6]
                 0.00    0.00       1/1           DisplayMatrix [4]
```

OProfile

OProfile[41] provides a sophisticated system-level profiling for the Linux operating system. Compared to `gprof`, OProfile gathers information from all running processes at the same time. Also, kernel internals are captured, and a configurable set of hardware performance counters. Profiling must be enabled and started by the super user, then all activities are recorded. Several tools are provided that analyze data recorded from user-space.

An example for system-level profiling of a desktop system running our PDE is given in listing 2.2. In this profile, the concurrent activities can be identified, also the time spent in kernel space[42] and in certain libraries becomes apparent. For each application, the user can create an individual profile, covering activities of the particular application as well as activities triggered by library and system calls.

A very handy feature of the OProfile system is that source code (and additionally assembler) can be annotated with the performance observations. This makes it easier to localize the time-consuming lines. In Listing 2.3, the source code of `calculate()` is shown with the sample count. The time spent in individual code lines becomes apparent, for example, 12% of the run-time is spent in Line 20, showing the optimization potential within this line.

Accessible hardware counters on the desktop system are shown in Listing 2.4[43].

Listing 2.2: Excerpt of a system-wide OProfile output while running partdiff-seq

```

CPU: Intel Architectural Perfmon, speed 1199 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
samples  %        image name               app name                 symbol name
1068951  71.0457  partdiff-seq             partdiff-seq             calculate
185685   12.3412  no-vmlinux               no-vmlinux               /no-vmlinux
26927     1.7896  libgstflump3dec.so       libgstflump3dec.so       /usr/lib/gstreamer-0.10/libgstflump3dec.so
16738     1.1125  libspeexdsp.so.1.5.0     libspeexdsp.so.1.5.0     /usr/lib/libspeexdsp.so.1.5.0
13980     0.9292  Xorg                     Xorg                     /usr/bin/Xorg
12617     0.8386  libQtGui.so.4.7.0        libQtGui.so.4.7.0        /usr/lib/libQtGui.so.4.7.0
12110     0.8049  libQtCore.so.4.7.0       libQtCore.so.4.7.0       /usr/lib/libQtCore.so.4.7.0
10177     0.6764  libxul.so                libxul.so                /usr/lib/firefox-3.6.13/libxul.so
8375      0.5566  libmozjs.so              libmozjs.so              /usr/lib/firefox-3.6.13/libmozjs.so
8311      0.5524  libflashplayer.so        libflashplayer.so        /usr/lib/flashplugin-installer/libflashplayer.so
6670      0.4433  libdrm_intel.so.1.0.0    libdrm_intel.so.1.0.0    /lib/libdrm_intel.so.1.0.0
6097      0.4052  libpulsecommon-0.9.21.so libpulsecommon-0.9.21.so /usr/lib/libpulsecommon-0.9.21.so
4185      0.2781  oprofiled                oprofiled                /usr/bin/oprofiled
4111      0.2732  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_lock
3819      0.2538  libgstreamer-0.10.so.0.26.0 libgstreamer-0.10.so.0.26.0 /usr/lib/libgstreamer-0.10.so.0.26.0
3255      0.2163  libpthread-2.12.1.so     libpthread-2.12.1.so     pthread_mutex_unlock
```

Listing 2.3: Excerpt of the annotated partdiff-seq source code

```
             :/* ********************************************************************** */
```

[41] Visit `http://oprofile.sourceforge.net/` for further information.

[42] By providing the kernel symbol table, all activity inside the kernel is accounted in a fine-grained manner to the kernel-internal symbols; in the listing the `no-vmlinux` symbol aggregates all kernel activities.

[43] The output was created by using `opcontrol -list-events`.

```
2                :/* calculate: solves the equation                                    */
3                :/* ******************************************************************* */
4                :void calculate(void)
5                :{ /* calculate total: 1068951 99.9979 */
6                :  int i,j;                                    /* local variables for loops  */
7                :
8                :  while(term_iteration>0)
9                :  {
10      2 1.9e-04 :    maxresiduum=0;
11   5088  0.4760 :    for(i=1;i<N;i++)                                  /* over all rows     */
12               :    {                                                 /*                   */
13  74459  6.9655 :      for(j=1;j<N;j++)                                /* over all columns  */
14               :      {
15 484776 45.3497 :        star=                    -Matrix[m2][i-1][j]
16               :           -Matrix[m2][i][j-1]   -Matrix[m2][i][j+1]
17               :                                 -Matrix[m2][i+1][j]  +4.0*Matrix[m2][i][j];
18               :        residuum=getResiduum(i,j);
19     315  0.0295 :        korrektur=residuum;
20  134782 12.6086 :        residuum = (residuum<0) ? -residuum:  residuum; // if (residuum<0) residuum=residuum*(-1);
21  135906 12.7137 :        maxresiduum = (residuum < maxresiduum) ? maxresiduum : residuum;
22               :
23   66943  6.2624 :        Matrix[m1][i][j]=Matrix[m2][i][j]+korrektur;
24               :      }
25               :    }
26               :    stat_iteration=stat_iteration+1;
27      56  0.0052 :    stat_precision=maxresiduum;
28               :    checkQuit();
29               :  }
30               :}
31               :
```

Listing 2.4: Available OProfile events (accessible hardware counters) on an Intel Nehalem system

```
1
2  OProfile: available events for CPU type "Intel␣Architectural␣Perfmon"
3
4  See Intel 64 and IA-32 Architectures Software Developer's Manual
5  Volume 3B (Document 253669) Chapter 18 for architectural perfmon events
6  This is a limited set of fallback events because oprofile doesn't know your CPU
7  CPU_CLK_UNHALTED: (counter: all)
8          Clock cycles when not halted (min count: 6000)
9  INST_RETIRED: (counter: all)
10         number of instructions retired (min count: 6000)
11 LLC_MISSES: (counter: all)
12         Last level cache demand requests from this core that missed the LLC (min count: 6000)
13         Unit masks (default 0x41)
14         ----------
15         0x41: No unit mask
16 LLC_REFS: (counter: all)
17         Last level cache demand requests from this core (min count: 6000)
18         Unit masks (default 0x4f)
19         ----------
20         0x4f: No unit mask
21 BR_INST_RETIRED: (counter: all)
22         number of branch instructions retired (min count: 500)
23 BR_MISS_PRED_RETIRED: (counter: all)
24         number of mispredicted branches retired (precise) (min count: 500)
```

PAPI and Likwid

There are several approaches that access CPU counters. *PAPI*, the *Performance API* [MCW+05, TJYD09], is a portable library which enables programs to gather performance events of all modern x86 and Power processors.

PAPI evolved from an interface for CPU counters to the component-oriented PAPI-C [TJYD10], which extends the original PAPI to capture counters from a multitude of sources – ACPI, `lm_sensors` for temperature, or specific network interfaces (Myrinet). PAPI-C leverages the existing inhomogeneous vendor (and software) interfaces.

An alternative library and user space program to profile hardware counters for an application is Likwid [THW10]. Likwid is a user space tool that profiles hardware counters for the whole application execution. When the tool is started, it initializes the counters, then starts the application, and once the application terminates, the counters are stopped and a brief report is created. A small API is offered that allows a developer to restrict the measured code regions and, furthermore, it supports to split execution into phases that can be assessed individually.

An exemplary profile of the floating point group of `partdiff-seq` is given in Listing 2.5. In this example the Intel processor operated on an average clock of 3.427 GHz (Line 28), the number of *cycles per*

instruction(CPI) is 0.52, which means every other cycle one instruction is *retired*[44] on the core. The number of double precision floating point operations per second is about 1.7 GFlop/s. One can easily conduct from the average clock speed and CPI that if we execute one instruction every other cycle, then effectively 1.7 Gigainstructions are executed per second, which means most operations were floating point instructions. This little example already demonstrates the power of hardware counters and simple theoretic considerations.

Available groups for Likwid are shown in Listing 2.6. The group to measure is selected upon startup of Likwid. Note that the memory group is aggregated for all cores on a given chip.

Listing 2.5: Excerpt of the likwid output for partdiff-seq

```
1
2  -------------------------------------------------------------
3  -------------------------------------------------------------
4  CPU type:       Intel Core Westmere processor
5  CPU clock:      2.79 GHz
6  Measuring group FLOPS_DP
7  -------------------------------------------------------------
8  /opt/likwid/bin/likwid-pin -c 1 ./partdiff-seq 0 2 100 1 2 10000
9  [likwid-pin] Main PID -> core 1 - OK
10
11 [...]
12 < program output >
13 [...]
14
15 +--------------------------------------+-------------+
16 |                Event                 |   core 1    |
17 +--------------------------------------+-------------+
18 |           INSTR_RETIRED_ANY          | 2.02167e+11 |
19 |         CPU_CLK_UNHALTED_CORE        | 1.04973e+11 |
20 |         CPU_CLK_UNHALTED_REF         | 8.55369e+10 |
21 |    FP_COMP_OPS_EXE_SSE_FP_PACKED     | 1.16896e+06 |
22 |    FP_COMP_OPS_EXE_SSE_FP_SCALAR     | 5.22953e+10 |
23 | FP_COMP_OPS_EXE_SSE_SINGLE_PRECISION | 1.0281e+07  |
24 | FP_COMP_OPS_EXE_SSE_DOUBLE_PRECISION | 5.22862e+10 |
25 +--------------------------------------+-------------+
26 +-------------------------+-----------+
27 |          Metric         |  core 1   |
28 +-------------------------+-----------+
29 |       Runtime [s]       |  37.5892  |
30 |       Clock [MHz]       |  3427.22  |
31 |           CPI           |  0.51924  |
32 | DP MFlops/s (DP assumed) |  1696.62  |
33 |      Packed MUOPS/s     |  0.037923 |
34 |      Scalar MUOPS/s     |  1696.55  |
35 |        SP MUOPS/s       |  0.333533 |
36 |        DP MUOPS/s       |  1696.25  |
37 +-------------------------+-----------+
```

Listing 2.6: Available Likwid groups on an Intel Nehalem system

```
1  BRANCH: Branch prediction miss rate/ratio
2  CACHE: Data cache miss rate/ratio
3  CLOCK: Clock of cores
4  DATA: Load to store ratio
5  FLOPS_DP: Double Precision MFlops/s
6  FLOPS_SP: Single Precision MFlops/s
7  FLOPS_X87: X87 MFlops/s
8  L2: L2 cache bandwidth in MBytes/s
9  L2CACHE: L2 cache miss rate/ratio
10 L3: L3 cache bandwidth in MBytes/s
11 L3CACHE: L3 cache miss rate/ratio
12 MEM: Main memory bandwidth in MBytes/s
13 TLB: TLB miss rate/ratio
14 VIEW: Double Precision MFlops/s
```

LTTng

The *Linux Trace Toolkit*[45] provides a user-space and a kernel-space tracer [FDD09]. The user space tracer (UST) provides an API by which a developer can instrument the source code. Also, the *GNU Project De-*

[44] Many modern CPUs execute operations that might depend on operations that are still in the pipeline, for example, speculative execution of branches before the branch condition is actually evaluated; if the prediction is correct, the results of the executed instructions are stored, otherwise these results are invalid and must be discarded. When there are no conflicts, a processor retires the instruction allowing write back of the results. For the speed of the execution the speculatively executed instructions are irrelevant, therefore, they are not covered by the CPI metric.

[45] Documentation of the *Linux Trace Toolkit* including all tools is elaborate and available on `http://lttng.org/`. A quick-start tutorial can be found here: `http://omappedia.com/wiki/Using_LTTng`.

bugger (`gdb`) can use UST to record GDB tracepoints. The kernel tracer captures activity for all processes and the system.

Traces are recorded in the *Common Trace Format* (CTF)[46]. The user-space tracer records events with zero-copy[47], resulting in a low overhead of 700 ns per event (according to the documentation). The overhead for the kernel-tracer is even lower.

LLTV is the *LTTng Viewer*, a tool providing statistical, graphical and text-based *views* of the recorded traces. Internally, the individual views are realized as modules. It has been shown capable of handling traces with a size of 10 GiB. Another viewer is incorporated into the *eclipse*[48] IDE as a plugin. With *LTTng*, several traces can be visualized concurrently; this technique has been used to view traces of a virtual machine together with a trace of the XEN host system [DD08].

Exemplary screenshots of LLTV for the desktop system running `partdiff-seq` are discussed next. In Figure 2.12, the *event view* and *control flow view* are presented. The *event view* – shown in the upper half of the screenshots – lists individual events in a textual representation while the *control flow view* – visible in the lower half of the screen – shows the activity in a timeline for each individual process. Graphical representation of the trace encodes the activity of each process in colors; green, for instance, encodes that the process on the left executes in user space (this is *partdiff-seq*).

The *statistical view* and *resource view* are shown in Figure 2.13. In the statistical view, aggregated information of all kinds of activity is provided. In the resource view, the activity on each CPU and interrupt can be observed, white color encodes user-space activity dispatched on a CPU. For easy analysis, the screenshot has been modified to show the likely[49] execution of *partdiff-seq* on the CPUs with green color. The vertical line marks the start of the program, then the single threaded program is migrated between all 4 cores on the quad-core system.

Consequently, with the tool, interactions between processes and system activity at a given time can be analyzed extensively. Note that while the overhead of the tracer is very low compared to other tools tracing 50 seconds of system activity created a set of files with the aggregated size of 57 MiB.

The tool provides capabilities to filter events, which is unfortunately not available for the graphical trace representation.

2.4.4. Available Tools for Analysis of Parallel Programs

Compared to the tools for sequential programs, tools for parallel applications are more involved. In the following, a few tools are presented which are aware of the programming and execution models of parallel applications, hence, they explicitly support the analysis of parallel applications. *VampirTrace* records execution behavior of HPC applications in the *Open Trace Format*[50]. Popular post-mortem performance analysis tools that analyze these trace files are *TAU*, *Vampir* and *Scalasca*. Those tools provide several ways to assist a developer in assessing application behavior.

An experimental tool environment which should be named is PIOviz [LKK+06a]. This environment is capable of tracing not only parallel applications, but also triggered activity on the *Parallel Virtual File System* version 2 (PVFS). In the following, brief introductions and screenshots are provided for all the mentioned tools.

[46] `http://www.efficios.com/ctf`

[47] Zero-copy is a technique in which data is communicated without copying it between memory buffers – in this case no copy between kernel-space and user-space is necessary.

[48] `http://www.eclipse.org/`

[49] It is not possible with the tool to actually relate applications with the resource view, therefore, the execution is guessed by the author based on the information provided by the control flow view.

[50] Further information is provided on Section 2.4.5.

bugger (`gdb`) can use UST to record GDB tracepoints. The kernel tracer captures activity for all processes and the system.

Traces are recorded in the *Common Trace Format* (CTF)[46]. The user-space tracer records events with zero-copy[47], resulting in a low overhead of 700 ns per event (according to the documentation). The overhead for the kernel-tracer is even lower.

LLTV is the *LTTng Viewer*, a tool providing statistical, graphical and text-based *views* of the recorded traces. Internally, the individual views are realized as modules. It has been shown capable of handling traces with a size of 10 GiB. Another viewer is incorporated into the *eclipse*[48] IDE as a plugin. With *LTTng*, several traces can be visualized concurrently; this technique has been used to view traces of a virtual machine together with a trace of the XEN host system [DD08].

Exemplary screenshots of LLTV for the desktop system running `partdiff-seq` are discussed next. In Figure 2.12, the *event view* and *control flow view* are presented. The *event view* – shown in the upper half of the screenshots – lists individual events in a textual representation while the *control flow view* – visible in the lower half of the screen – shows the activity in a timeline for each individual process. Graphical representation of the trace encodes the activity of each process in colors; green, for instance, encodes that the process on the left executes in user space (this is *partdiff-seq*).

The *statistical view* and *resource view* are shown in Figure 2.13. In the statistical view, aggregated information of all kinds of activity is provided. In the resource view, the activity on each CPU and interrupt can be observed, white color encodes user-space activity dispatched on a CPU. For easy analysis, the screenshot has been modified to show the likely[49] execution of *partdiff-seq* on the CPUs with green color. The vertical line marks the start of the program, then the single threaded program is migrated between all 4 cores on the quad-core system.

Consequently, with the tool, interactions between processes and system activity at a given time can be analyzed extensively. Note that while the overhead of the tracer is very low compared to other tools tracing 50 seconds of system activity created a set of files with the aggregated size of 57 MiB.

The tool provides capabilities to filter events, which is unfortunately not available for the graphical trace representation.

2.4.4. Available Tools for Analysis of Parallel Programs

Compared to the tools for sequential programs, tools for parallel applications are more involved. In the following, a few tools are presented which are aware of the programming and execution models of parallel applications, hence, they explicitly support the analysis of parallel applications. *VampirTrace* records execution behavior of HPC applications in the *Open Trace Format*[50]. Popular post-mortem performance analysis tools that analyze these trace files are *TAU*, *Vampir* and *Scalasca*. Those tools provide several ways to assist a developer in assessing application behavior.

An experimental tool environment which should be named is PIOviz [LKK+06a]. This environment is capable of tracing not only parallel applications, but also triggered activity on the *Parallel Virtual File System* version 2 (PVFS). In the following, brief introductions and screenshots are provided for all the mentioned tools.

[46] `http://www.efficios.com/ctf`

[47] Zero-copy is a technique in which data is communicated without copying it between memory buffers – in this case no copy between kernel-space and user-space is necessary.

[48] `http://www.eclipse.org/`

[49] It is not possible with the tool to actually relate applications with the resource view, therefore, the execution is guessed by the author based on the information provided by the control flow view.

[50] Further information is provided on Section 2.4.5.

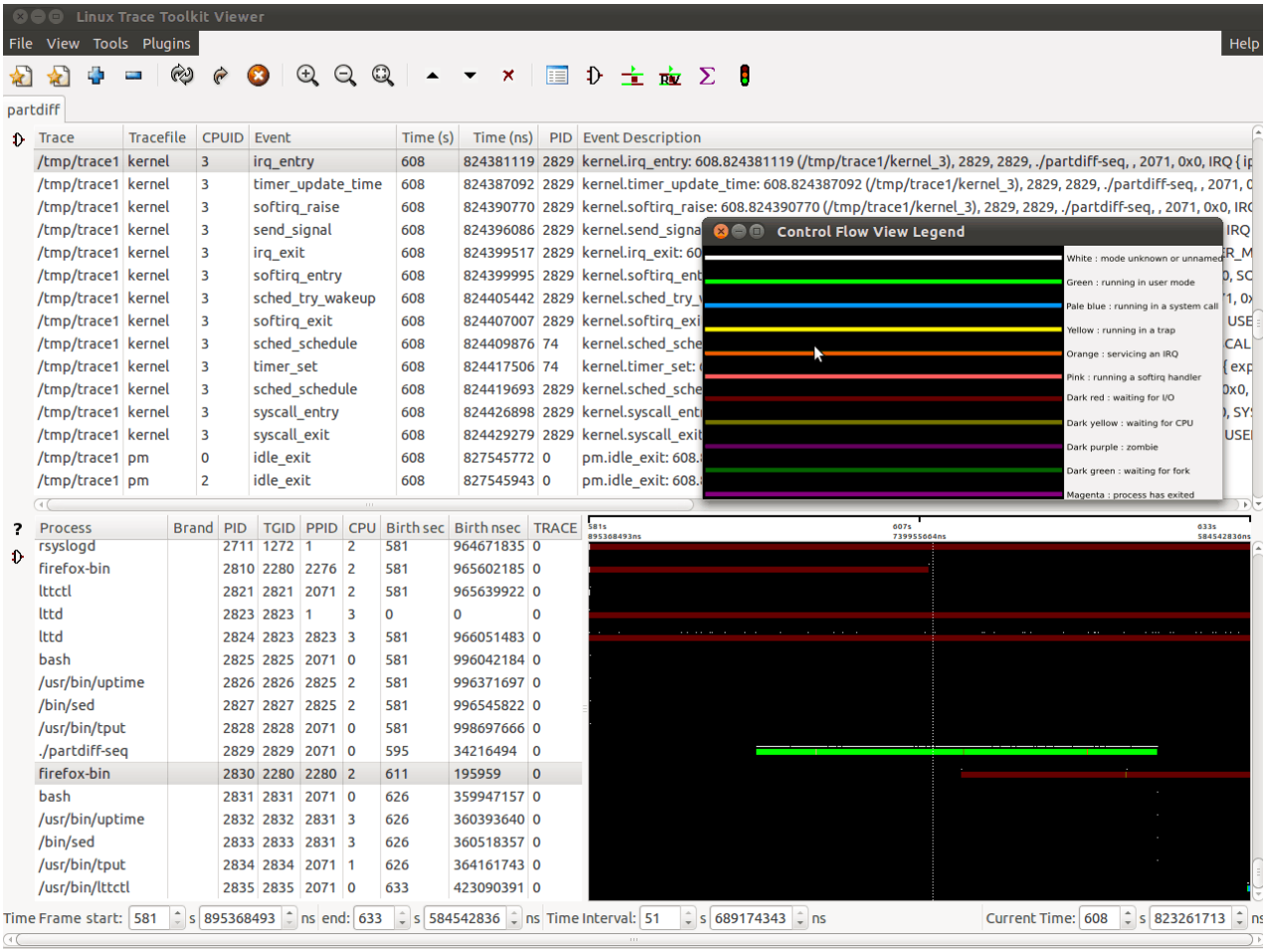


Figure 2.12.: Screenshot of the LTTV viewer for a trace of the system – event view and control flow view.

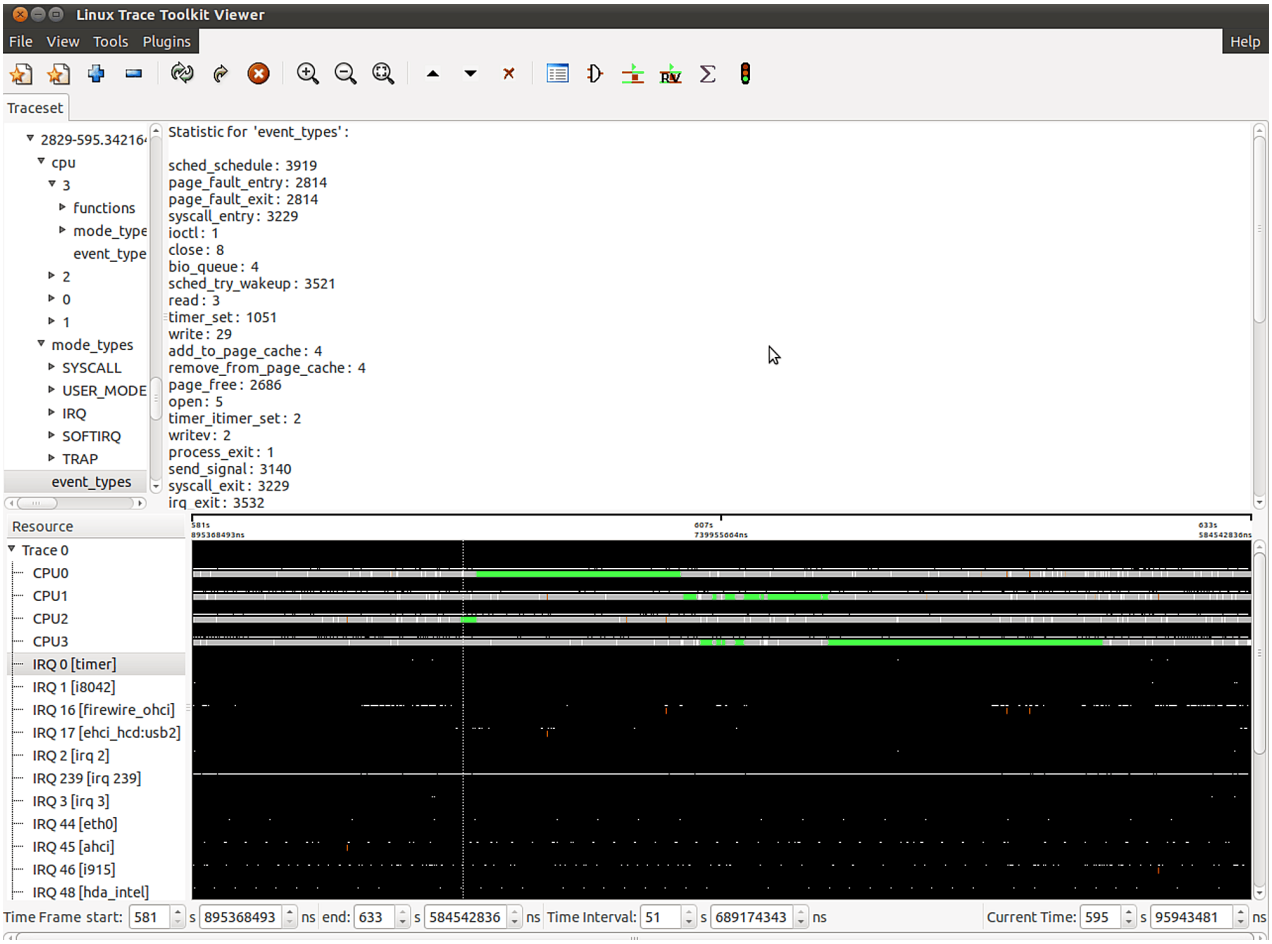


Figure 2.13.: Screenshot of the LTTV viewer for a trace of the system – statistic view and resource view. The likely scheduled execution of the program on the CPUs is marked in green.

VampirTrace

VampirTrace [MKJ+07, KBD+08] is a library and tool set which generates traces or profiles for MPI and OpenMP applications. It instruments applications to write files in the *Open Trace Format* (OTF) at runtime. Depending on the type of instrumentation, function calls of the application, MPI and OpenMP activity are recorded. Additionally, low-level POSIX I/O calls and POSIX Threads are intercepted, CPU counters are supported. Additional counters can be supplied by plugins [STHI10].

VampirTrace utilizes *MPI Profiling Interface* (PMPI), a profiling interface provided by MPI[51]. VampirTrace stores a subset of parameters and context information from MPI calls. This enables identifying communication partners and provides information about the exchanged data, e.g., the message size. OpenMP provides an interface that eases instrumentation, the *OpenMP Pragma and Region Instrumentor* (OPARI) [MMSW02], which is used by VampirTrace.

Furthermore, it supports performance counters provided by PAPI. In case the tracing of performance counters is enabled by the user, selected counters are recorded for every generated event.

To perform source code instrumentation, the application must be recompiled using provided build scripts. These are wrappers for the compiler and include a source-to-source compiler which modifies functions in order to generate events at every function entry and exit. *Dyninst*[52] [BH00] can be used to instrument binaries, too. However, the binary instrumentation has the drawback that only limited information about application internals are accessible.

TAU

The *Tuning and Analysis Utilities* [SM06], are the swiss army knife for performance analysis. They support not only many languages (C, C++, Java, Python, CUDA, ...), but also a rich variety of capabilities like profiling and tracing, sampling, throttling of event generation, extended information for POSIX I/O and communication and hardware counter support via PAPI.

Several tools are shipped with TAU which include: converters to export data to other trace or profile formats, automatic instrumentation for source code, and a visualizer for trace profiles (*ParaProf*). *Jumpshot* [ZLGS99] or *Vampir* can be used to visualize traces. The Jumpshot viewer[53], shipped with TAU, is originally part of the *MPI Parallel Environment* (MPE) and visualizes the *SLOG2* trace format.

ParaProf is a Java GUI which displays profiling results. Compared to the tools for sequential analysis, like *gprof*, with ParaProf all processes and threads of a parallel application can be analyzed together. The tool focuses on analyzing behavior of the processes and threads for a single metric at a time. Values for the selected metric can be compared, additionally average and standard deviation is provided. The metric under investigation can be chosen – for example, time or floating point operations, and the data is provided either for the executed functions or a group of functions. Additionally, derived metrics like Flop/s can be created. To analyze time-dependent behavior, TAU provides an API that can be used in a program to divide execution into phases, which are profiled and analyzed independently.

The available views in ParaProf are histograms, a communication matrix showing the interaction between the processes, textual or a 3D visualization. The 3D visualization shows the value of a selected metric for each pair of thread and function.

Performance results of experiments can be archived, either directly in ParaProf or in a (remote) database. Metrics can be compared across experiments to monitor performance over the application development, or

[51] The MPI Profiling Interface defines that every MPI function is exported with two names, an MPI prefixed call and a PMPI prefixed version. Users use the MPI call; by providing an instrumented version of the MPI function, all user calls can be intercepted. Instrumented functions can perform appropriate logging and they usually call PMPI functions that are directly provided by the MPI library.

[52] `http://www.dyninst.org/`

[53] See Page 77 for further information on Jumpshot.

run-time behavior of various parameter sets can be assessed. *PerfExplorer* is another tool shipped with TAU, it eases statistical analysis of profiles such as clustering and correlation across experiments. It can be used to create diagrams, for example to illustrate achieved speedup. Furthermore, application performance can be assessed over multiple runs, and even a longer time period.

In order to demonstrate a few of the available views, a profile of our PDE was generated for 4 processes and visualized in ParaProf. In Figure 2.14, a few windows are shown. In the upper window, the experiment is chosen and information about run-time environment characterization is given which identifies the experimental setup and run-time settings. The legend windows (on the left) show the color-code and name of the recorded events and available groups. To improve user interaction, the events and function calls are grouped into sets – in the example, the MPI group can be selected to assess communication overhead quickly. In TAU, the call-graph is just encoded into the event names. Thus, the number of events in the function legend is rather large; in the example, calculate invokes several functions.

The main data window (in the center) with the unstacked bars in the middle of the figure displays a metric for all functions and processes. In the screenshot the *time* metric has been chosen and the mouse hovers over the calculate function. By clicking an interesting function, a window pops up that just shows this metric for all threads. In the figure, the window below shows the time for `MPI_Sendrecv()` which is called by the exchange function.

Interaction between the 4 processes can be analyzed with a communication matrix, in Figure 2.15 the number of calls is encoded in a heat-map. In the figure, a diagonal communication pattern of the PDE can be observed – a process communicates with its two neighbors. A few communications from the ranks to Rank 0 are seen, this is the finalization step in which the results are gathered.

Text statistics of Rank 0 for I/O and messages are provided in Figure 2.16, showing the number of I/O calls and amount of data.

Note that the standard automatic instrumentation, which intercepts every function entry and exit, causes serious overhead; by tracing, the wall-clock run-time of the PDE increases from 13 s to 576 s. In the same run, more than 1.8 GiB of trace data will be generated per process, if tracing is enabled. Consequently, this demonstrates the need to filter unimportant events during the instrumentation or at run-time. TAU provides tools to automatically filter events, and it throttles an event when it is fired too often.

For more information about visualization and usage refer to the *TAU User Guide* [Tau10].

Vampir

Vampir [MKJ+07, KBD+08] is a mature commercial trace analysis tool, which visualizes files that have been generated with VampirTrace. However, it has a proprietary code base and cannot be extended by third parties.

Large trace files may not fit into the memory of one machine and take a long time to visualize and preprocess, therefore, the software called *VampirServer* extends the performance of Vampir by outsourcing analysis capabilities and trace handling to a set of remote processes. The number of server processes can be scaled to match the size of the trace file and the required performance.

In Vampir, multiple displays, each performing a specific visualization, can be arranged to a *workspace*. Displays can be configured, e.g., to show a particular process, to filter information or to visualize another metric. Zooming on a timeline is propagated to all displays.

For a PDE run, a workspace containing all displays has been created and is shown in Figure 2.17. Here, only a fraction of time is actually traced as the overhead with the default instrumentation that traces invocations of all functions is overwhelming[54].

[54] The PDE for the trace runs 100 iterations instead of the 10.000 that have been executed with an instrumented program for the other tools.

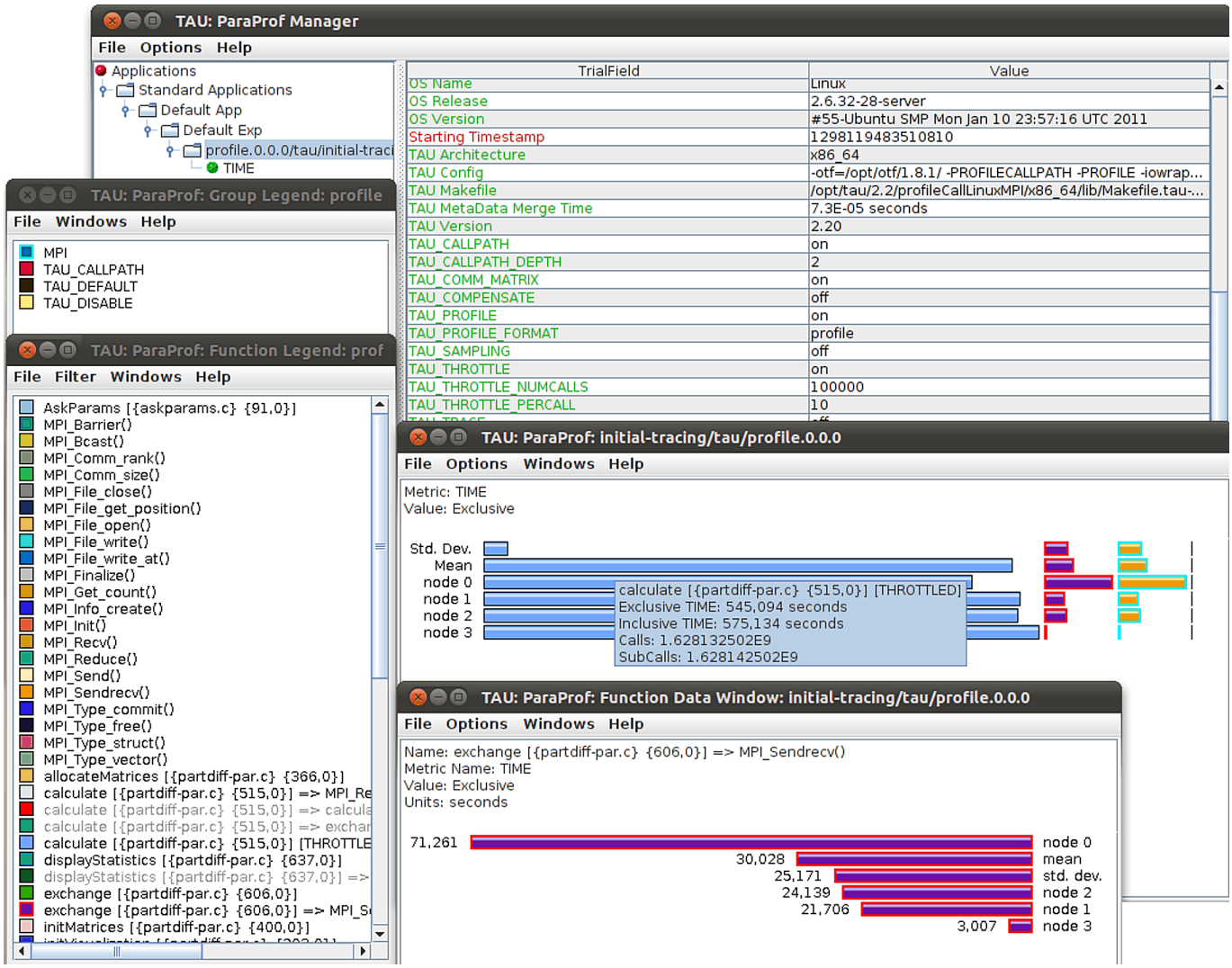


Figure 2.14.: Screenshot of ParaProf – PDE profile including experimental information.

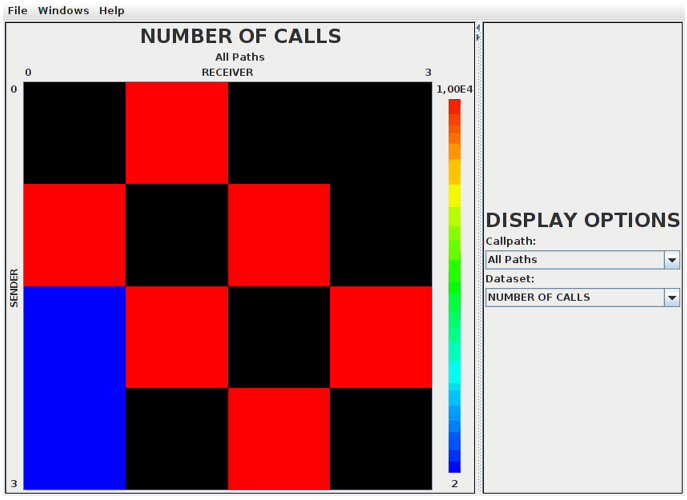


Figure 2.15.: Screenshot of ParaProf – PDE communication matrix.

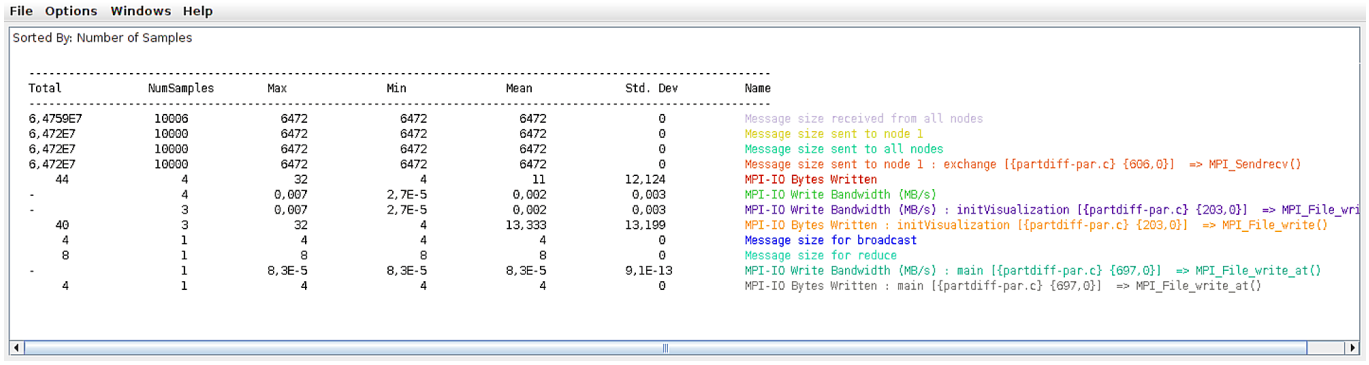


Figure 2.16.: Screenshot of ParaProf – *user event statistics* for Process 0 including MPI-IO statistics.

As Vampir has a huge number of displays, they are enumerated in the screenshot and described individually:

1. Next to the symbol toolbar is an overview timeline, which is visible all the time and does not change with zooms. This timeline shows the observed activity of all processes over time in a condensed form. The height of color in the bar encodes the number of processes that execute an operations of a given kind at a given time. Functions are grouped during the tracing and represented by different colors, example groups are I/O, communication or application. By default, MPI functions are colored in red whereas time spent in the application is colored in green. If all processes perform operations of the same group, then a column contains one color, for example at the beginning all processes perform MPI calls. To add a certain display, a user can click on the specific icon in the toolbar.

2. The *Master Timeline* shows the activity for each individual thread in colors according to the group. Inter-process messages are visualized by black lines. In Vampir there is no concept which associates the processes to the hardware topology. Therefore, the user must know the mapping of the processes to hardware.

3. In the *Process Timeline*, the call-stack of an individual process is given (here Process 0).

4. A *Counter Timeline* shows the values of one PAPI counter for one process. The minimum, maximum and average values can be plotted into one graph. In this display, the total number of instructions which were performed is plotted. The observed spike during startup might be an artifact caused by an overflow of the counter.

5. This is also a counter timeline that visualizes the Flop/s for Process 0. Note that there could be as many replicates of the displays as fit on the screen.

6. For one selectable metric, the *Performance Radar* encodes the counter values in a given color, similar to a heat map. In contrast to the counter timeline, all processes can be visualized together. Unfortunately, due to the counter overflow in the example, all processes are drawn in blue.

7. The *Call Tree* shows the call-graph together with a profile for the visualized time interval.

8. In the *Context View*, more information of the selected object is provided – most visible entities, such as an event, function or process, can be selected. In the example, the whole function group is clicked.

9. The *Function Summary* provides information on a set of processes. In the left window, the exclusive time of all processes is accumulated while the right window prints the number of invocations per group.

10. Available groups and their color scheme are listed in the *Function Legend*. By instrumenting the application manually with VampirTrace, more groups can be created.

11. An overview of the number of messages or message sizes is given in the *Message Summary*.

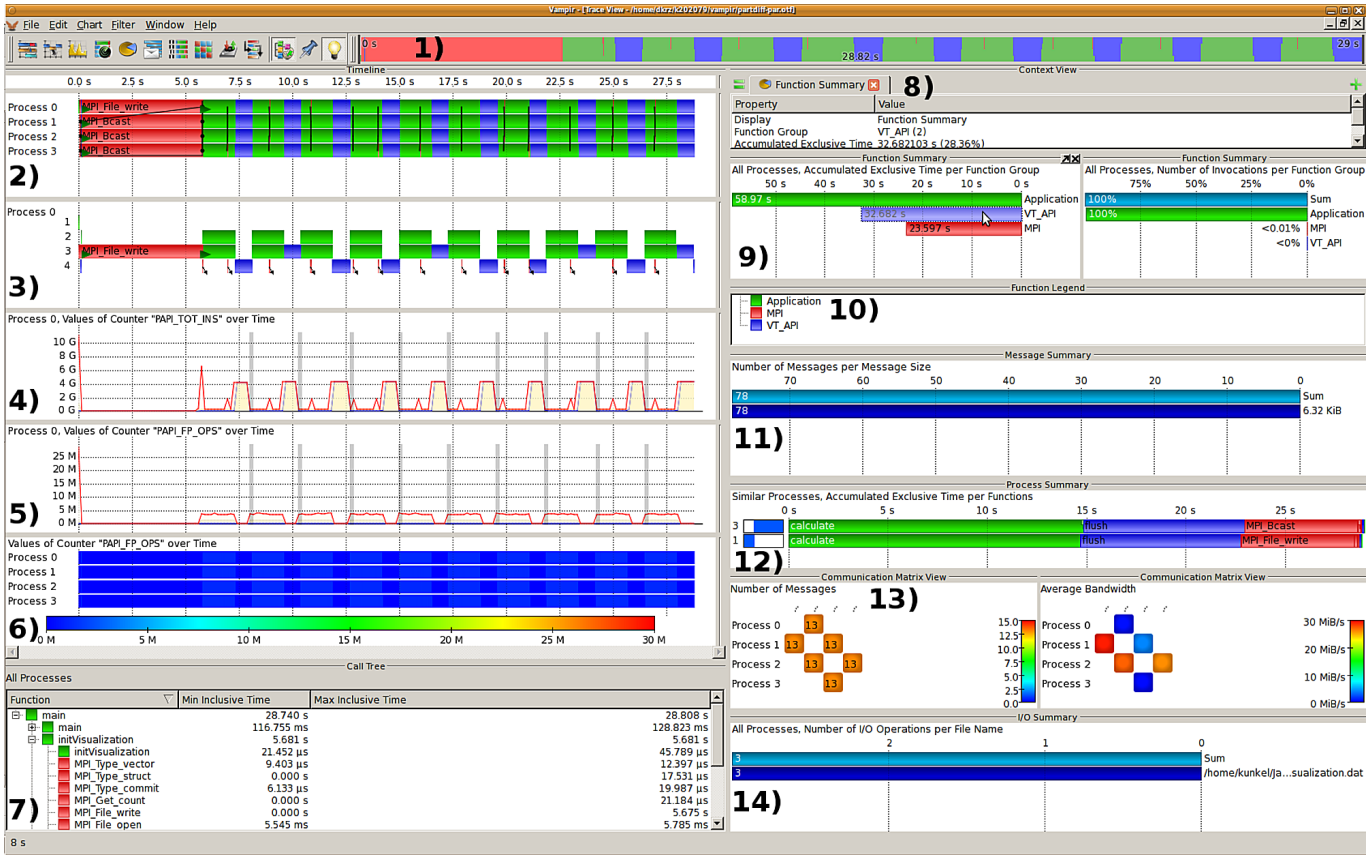


Figure 2.17.: Screenshot of a Vampir workspace.

12. In the *Process Summary*, a profile is generated and visualized for every process. If the space does not suffice, the display automatically clusters similar processes. In the example, it detected three processes with the upper function profile and one with the lower function profile – the master process which performs the actual I/O.

13. Similar to TAU, the inter-process behavior is visualized in the *Communication Matrix View*. Two displays have been created, the left window just shows the number of messages exchanged between two processes, while the right window indicates the average bandwidth.

14. The *I/O Summary* finishes our tour through the available displays: Several metrics are available and related to the file name: accessed data volume, number of operations or bandwidth.

Scalasca

Scalasca [GWW+10], is a performance analysis toolset which automatically analyzes large-scale parallel applications that use the MPI, OpenMP or a hybrid MPI/OpenMP programming model. It has been successfully applied to applications running with 200.000 processes on a BlueGene/P system [WBM+10].

Scalasca can be run in two modes, either a summary of the parallel program is created at run-time, or the application activity can be recorded in the *EPILOG* trace format and then analyzed post-mortem.

In the trace mode, Scalasca searches automatically for a common class of run-time communication bottlenecks, for example, for late senders. Scalasca ships with the sequential analyzer `expert` [WM03] and the parallel analyzer `scout` that identify wait-state patterns. The parallel analyzer runs with the same number of processes as the original application. While the analysis is performed, it replays the communication pattern of the original program and updates statistics accordingly. Therefore, `scout` scales similarly to the original application. However, the sequential tool `expert` detects more inefficient communication patterns.

Scalasca can instrument the application automatically, either by using compiler options, by transformation of the source code or by linking the program with an already instrumented library. Similarly to the other tools, an API for manual instrumentation is provided.

In contrast to previously referenced tracing tools, with Scalasca the application is typically started with an additional software monitoring system. After the application terminates the system can automatically run `scout` to perform the parallel trace analysis.

Process statistics and identified bottlenecks are displayed in the *Qt* application *Cube3*, which allows browsing through the analysis results.

To assess the features, the PDE configuration with 4 processes is instrumented with Scalasca and instructed to generate summaries.

A screenshot of the summary is provided in Figure 2.18. The view is split into three columns: the first column shows the available metrics, the second column displays all functions in the call-graph and the contribution to the metrics, the last column shows the contribution of every process of the supercomputer to the value of the function (and metrics). In the analysis session, the user selects a metric on the left, then localizes the relevant function and, at last, analyzes the distribution of the selected metrics among the processes. In the given example, the number of send operations is selected in the metrics tree; sends are invoked in the call-graph by `exchange()` which is called by `calculate()` which in turn is called by `main()`.

The view on the left aggregates the metrics among all functions and processes. Also, note that in the hierarchical view, a collapsed node of a column aggregates the values of all children. That means each node shows the inclusive metric, while an expanded node displays the *exclusive metric*, i.e., the contribution to the metrics which is not caused by any of the child nodes. Child nodes show their share by themselves (compare the *Execution* node and children in the left column). A color scheme between blue and red encodes the values similar to a heat map in all three columns. This assists in spotting bottlenecks in the metrics, code and unbalanced processes on the hardware.

For figure Figure 2.18, 60.000 `MPI_Sendrecv()` functions have been invoked from the `exchange()` function. As the inner processes transfer twice as much messages, the four processes call the function 10.000 times, 20.000 times, 20.000 and 10.000 times, respectively. The right view visualizes either the topology of the machine – here a flat topology – or the numerical values as shown in Figure 2.19.

By instrumenting all user functions, the wall-clock time increased from 12 s to 307 s, by instrumenting just MPI with the provided PMPI library, the overhead is not measurable. With Cube, the bottleneck can be identified by looking at the (run-)time metric of the functions and processes in Figure 2.19. A total of 1158 s for all 4 processes is divided into 68 s MPI activity and the remaining time is spent for user activity. In the call-graph, 558 s is spent in `getResiduum()` and another 600 s in the calculate function itself; the right column shows that the load is almost balanced across all processes. As `getResiduum()` has a tiny function body, the overhead of the measurement system dominates run-time. In a real scenario, this function should not be instrumented and therefore would be filtered; Scalasca provides tools to filter events.

For the existing metrics, a short description is provided in the online help that assists the user in understanding them. For example, a screenshot of the metrics for computational imbalance between processes and the corresponding help is given in Figure 2.20.

Periscope [GO10] and PerfExpert [BKD+10] are other automatic tools. They scan performance properties at run-time; appropriate metrics are measured and evaluated automatically. Ultimately, as in Scalasca, this assists in automatic localization of certain types of bottlenecks. However, while all those automatic tools provide some hints, the user might be forced to use a visualization tool such as Vampir to really understand the behavior of the application.

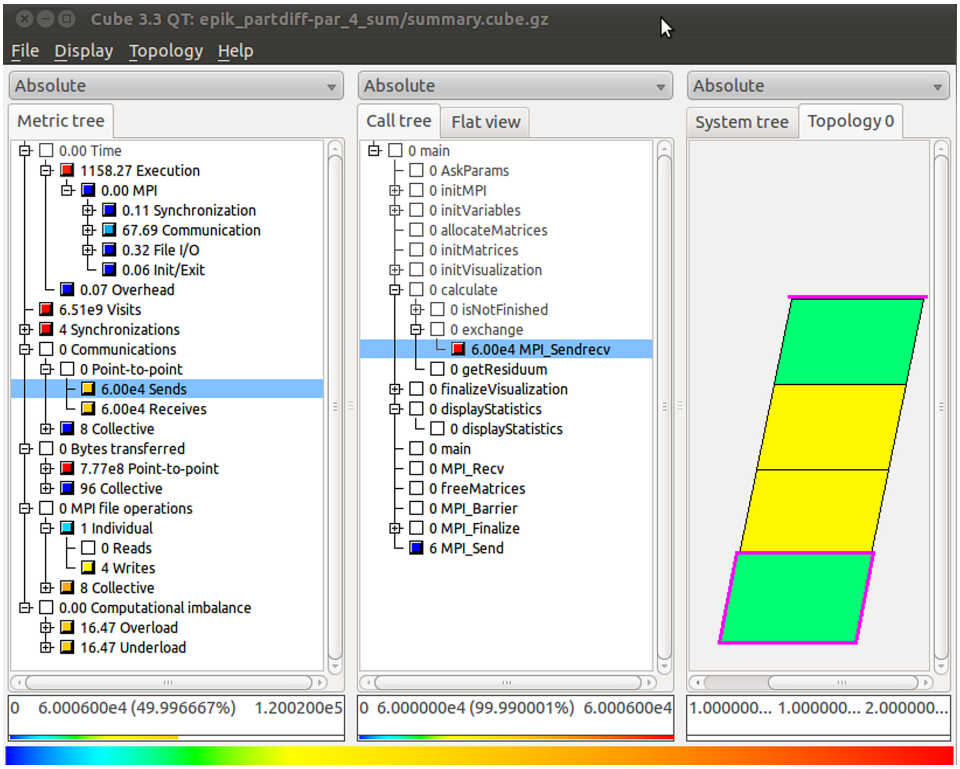


Figure 2.18.: Scalasca's Cube3 browser – the left column shows available metrics, the middle column assigns the metric's values to functions of the call-graph, the right column shows the contribution of every process to the function.

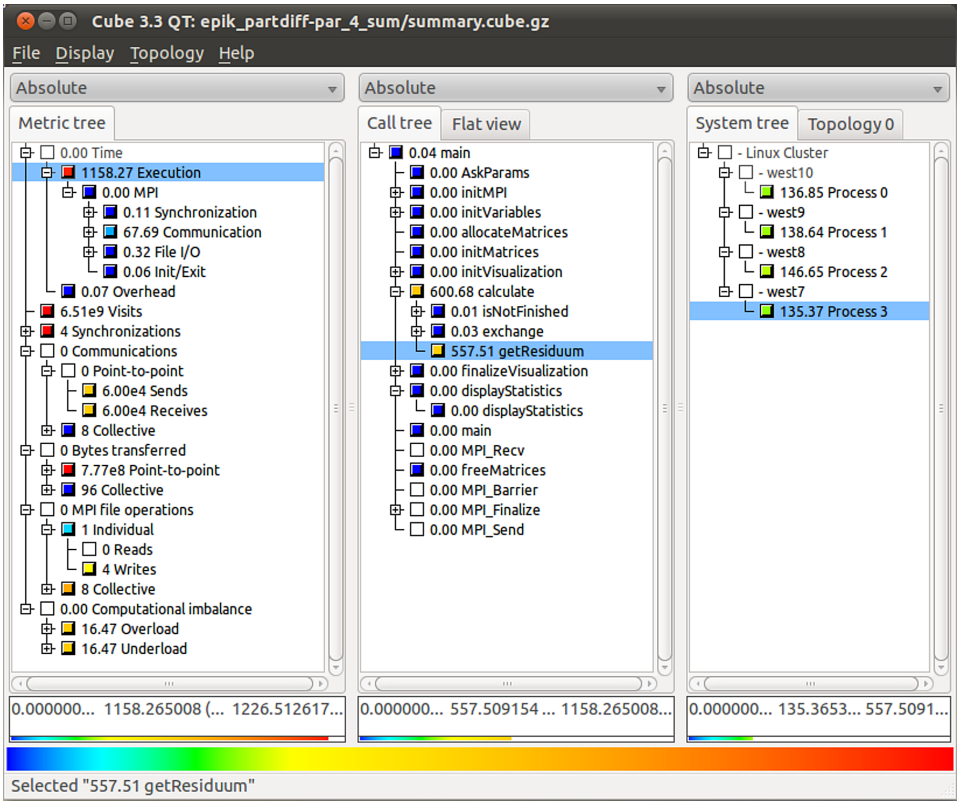


Figure 2.19.: Scalasca's Cube3 browser – identifying the computational overhead in `getResiduum()`.

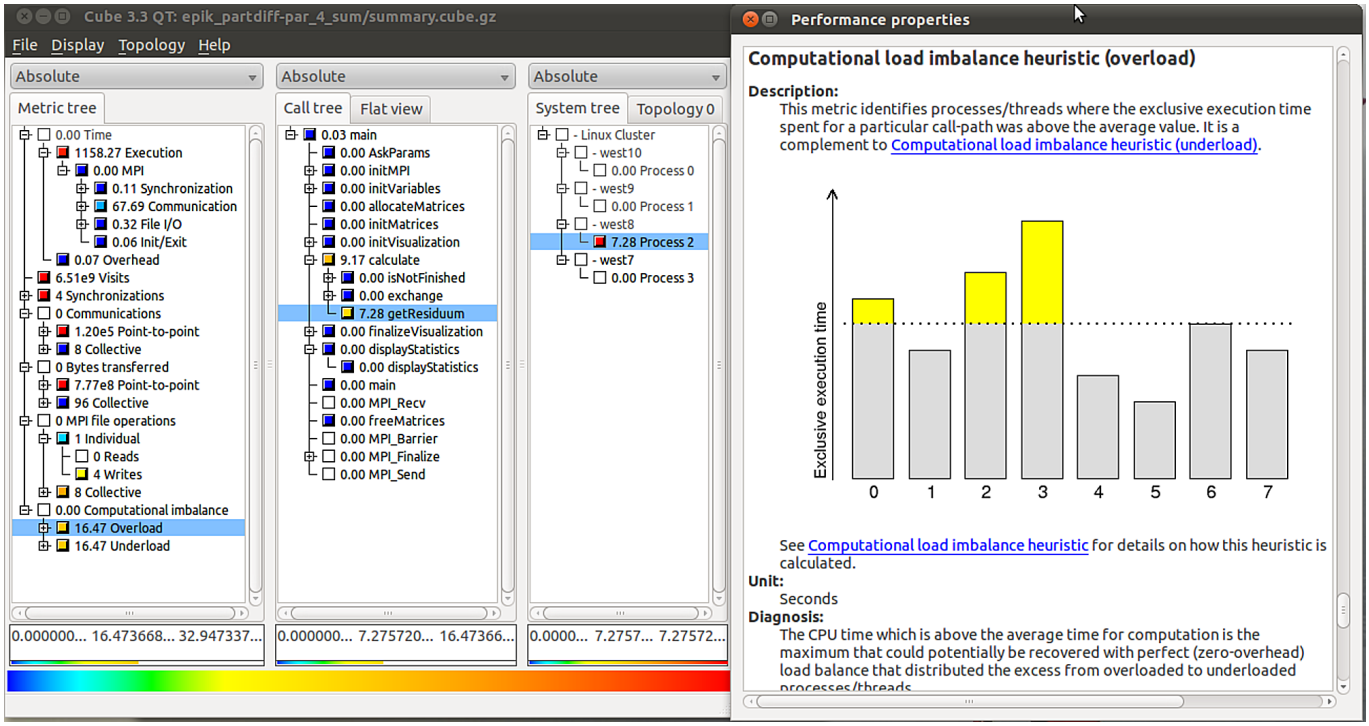


Figure 2.20.: Scalasca's Cube3 – assessing load imbalance and the online help for this metric.

MPE

The *MPI Parallel Environment* (MPE) is a loosely structured library of routines designed to support the parallel programmer in an MPI environment. It includes performance analysis tools for MPI programs, profiling libraries, graphical visualization tools and the trace visualization tool *Jumpshot* [ZLGS99]. MPE is shipped with MPICH-2 and contains different wrapper libraries, which use the PMPI profiling interface of MPI to replace the MPI calls with new functions.

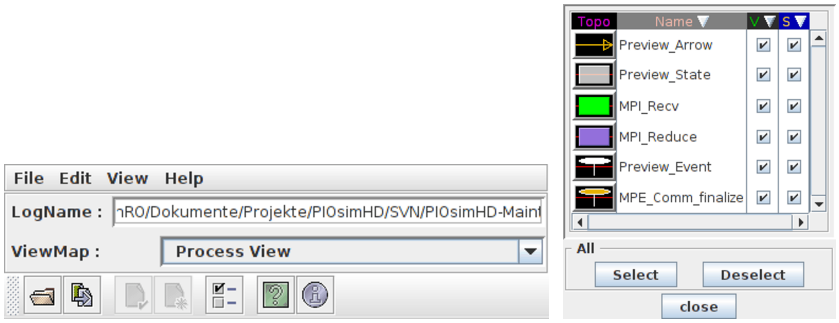
The trace visualizer of MPE is *Jumpshot*, which reads files in the SLOG2 format. SLOG2 tries to be scalable by storing aggregated information about intervals directly inside the format – further information is provided in [ZLGS99].

Upon startup of Jumpshot, the main window is shown in which a trace file can be loaded, a screenshot is given in Figure 2.21a. Jumpshot distinguishes between three types of entries: an *event*, a *state* that has a well defined start and end, and arrows which mark causal relations between states. A trace entry belongs to one named *category*. All available categories, assigned colors and whether they shall be visualized or searchable, are listed in the *legend window* (Figure 2.21b).

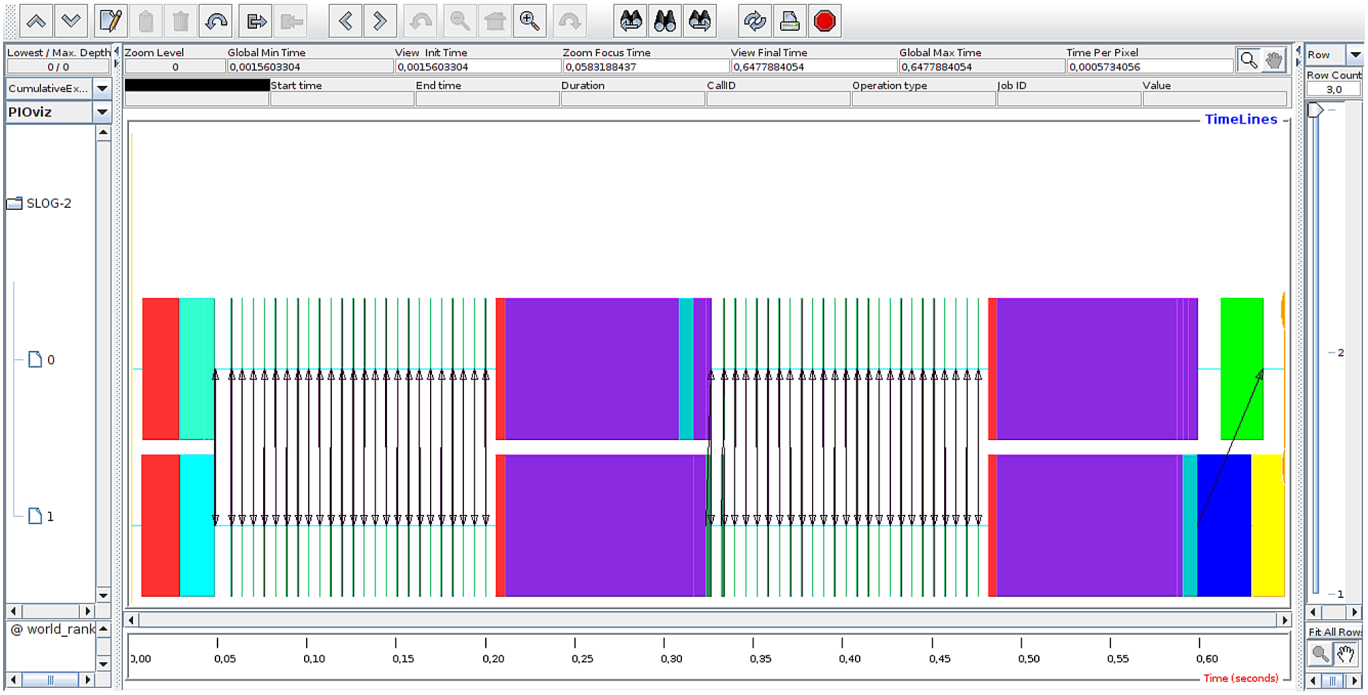
The *timeline window* shows the activity of each process over time. Processes are enumerated in a tree view and mapped to the timelines according to a *ViewMap*. A screenshot of the timeline window is given in Figure 2.21c. On the left side, the tree view shows two processes, the activity is drawn in the center. A horizontal timeline renders the activity of one processor in one row, the activity is encoded with colors as defined in the legend window. White areas in the activity correspond to computation by the client processes, the colors show the MPI function; violet, for example, represents `MPI_Reduce()`.

A user can select an interval and open a *profile window*, which aggregates the time of the states over each category and process. An example is provided in Figure 2.21d; the violet color indicates that most time is spent in `MPI_Reduce()`.

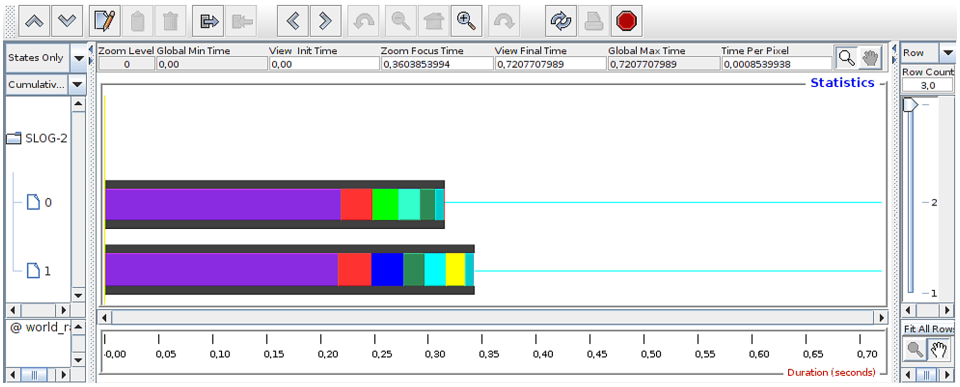
Both windows offer functionality to zoom and scroll in the window, this is provided by the icons in the toolbar. Timelines can be enlarged with the slider on the right of the windows. Additionally, individual timelines can be deleted or moved around; to move a timeline, it must be cut by the user and inserted after



(a) Main window (b) Legend window



(c) Timeline window



(d) Profile window

Figure 2.21.: Jumpshot windows.

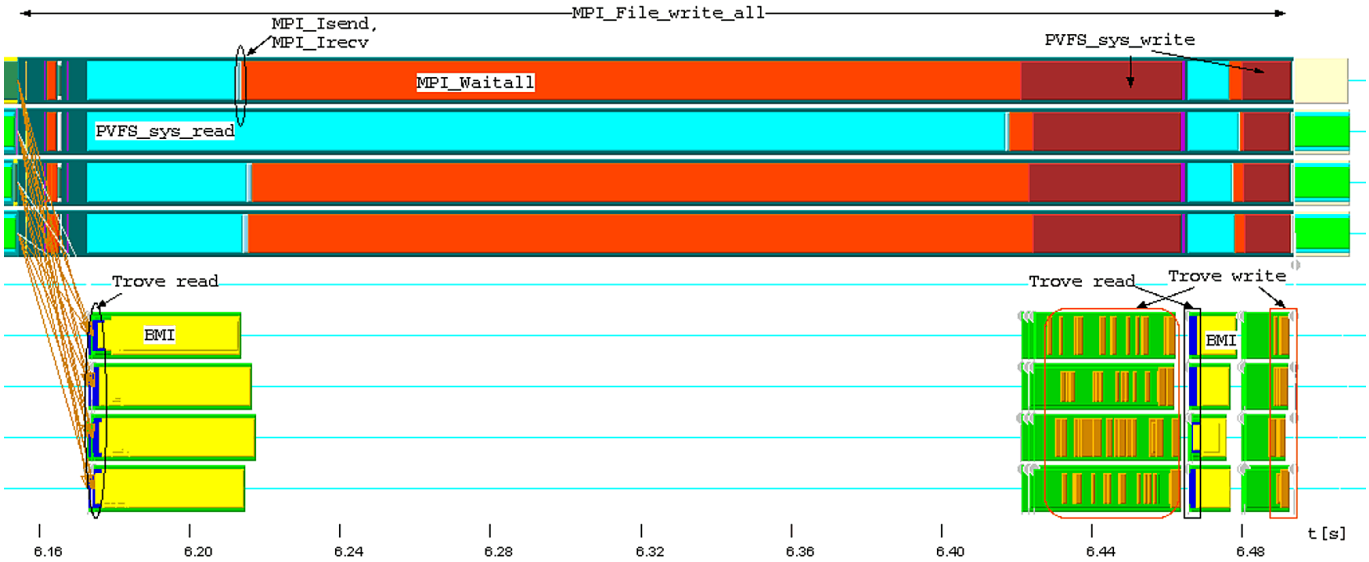


Figure 2.22.: Modified screenshot of PIOviz visualizing the interaction between 4 clients and 4 servers with explanations [KTML09].

another timeline.

PIOviz

The *Parallel Input/Output Visualization Environment* [LKK+06b, KTML09] (PIOviz) is able to trace and visualize activities on the servers of the parallel file system PVFS in conjunction with the client events triggering these activities. PIOviz correlates the behavior of the servers with program events. Developers can use these features to analyze and optimize MPI-IO applications together with PVFS. Additionally, PIOviz also collects device statistics, such as network and disk utilization, from the operating system, and computes a few PVFS-internal statistics [KL08].

The PMPI wrapper provided in MPE is used to trace the MPI function calls. However, with the original MPE only MPI activity is recorded and analyzed. PIOviz extends its capabilities by recording communication inside MPI calls, client-sided PVFS activity, and corresponding operations in PVFS servers. Compared to other tools, PIOviz is considered experimental, however, it provides novel capabilities that are not available elsewhere.

In brief, the environment consists of a set of user-space tools, modifications to the I/O part of MPICH2 and MPE, and logging enhancements to PVFS. To relate client and server activity, the following changes are made: PIOviz modifies the MPE logging wrapper to add a call-ID to each I/O request. Patches to MPI-IO and PVFS transfer this ID to the server and through the different layers, and record interesting information. The call-ID allows us to associate the MPI call with the PVFS operations triggered by this call. Also, the environment introduces additional user-space tools that transform SLOG2 files depending on this extra information. It contains modifications to Jumpshot providing additional information in the viewer.

To assess performance in the workflow, independent trace files are created on client-side and server-side once a user executes an MPI(-IO) program. Then a set of tools post-process these files and finally merges them into a single file containing all enriched information about client and server activities. PIOviz uses MPICH's SLOG2 format, therefore a user can analyze trace information with Jumpshot.

An example screenshot of PIOviz is given in Figure 2.22. Client process activity is given in the upper four timelines, and the lower five timelines show the activity for one PVFS metadata server and four data servers.

By looking at the screenshot it can be observed that the second process spends about 20 ms more time in the `PVFS_sys_read()` operation of the `MPI_File_write_all()` and thus the process waits for the read operation to complete. However, the server activity finished already, therefore, the servers are waiting for requests from clients and are not the cause of the inefficiency. Instead we claim the client library caused the observed behavior. Without knowledge of the server activity, a hypothesis for a potential bottleneck could not exclude the network, the server, or the client-server protocol.

Besides PIOviz, to our knowledge, there is no trace environment available which can gather information of client and server activity and correlates them – a recent funded project aims to extend TAU towards this goal [BCI+10], though.

2.4.5. Trace Formats

There are several trace formats available because many performance analysis tools rely on their own trace format. Usually, command line tools are provided to convert the tool-specific trace format to other well-known trace formats. Therefore, already existing tools can be applied to process the (converted) trace. Scalasca and TAU, for example, provide converters into the *Open Trace Format* (OTF).

A few general aspects in designing a trace format and its interface are discussed at first, the list is a lose collection of aspects and does not aim for completeness. Then, the concepts behind OTF are introduced.

Basically, all trace formats have been created with specific design goals in mind, however, several goals are common to most of them. The following abstract requirements and concepts represent the author's view; they are defined after looking at several trace formats and existing tools[55]:

- Recording of all *relevant information* for post-mortem analysis must be possible. This includes the possibility to record arbitrary data that is necessary to characterize an event in detail. A *context* provides more information about the recorded events or the utilized resources. For example, timestamps are required to understand the temporal causality; an identifier can specify the processor a thread is (currently) executed on. Timestamps are especially difficult: Since local clocks are not as accurate as a primary reference clock, timestamps of different components are slightly incorrect thus sorting events by the locally created timestamp can lead to a wrong order of events. Therefore, either all clocks must be synchronized accurately with one reference clock, or mechanisms are provided that are able to fix incorrect ordering. *Metadata* describes the invariant properties of the environment in which the trace is recorded. This kind of description of system configuration and experiment is important when traces are kept for a long time.
- In heterogeneous environments *portability* between machine architectures becomes relevant.
- Methods to *reduce the trace file size* and to handle large traces should be available. One simple approach that reduces the file size is permit selective activation and deactivation of the run-time tracing, that means either the application activates tracing for the relevant area or the system deactivates itself automatically after a threshold is reached. Another method is to support compression. If that does not help to reduce the file size, then tools should be capable to process large trace files.
- Required post-processing of the trace files should be of *low overhead*. For example, the PIOviz environment relies on several post-processing stages in which the trace data is read completely. For larger traces this is infeasible.
- Analysis of the trace files should require *limited resources*. A subset of the data should be loadable. Loaded information might be a subset of the processes or the record types, or just a restriction of the time interval.
- *Efficient* parallel access. Technically, this can be achieved by splitting the trace information into multiple files which can be accessed independently to prevent locking and synchronization between

[55] The inspected trace formats are RLOG2, SLOG2, TAU trace files and OTF. To encourage further reading in the design goals, two literature references are provided explicitly.: The attempt for CTF [Des10] and design considerations for OTF [KBB+06].

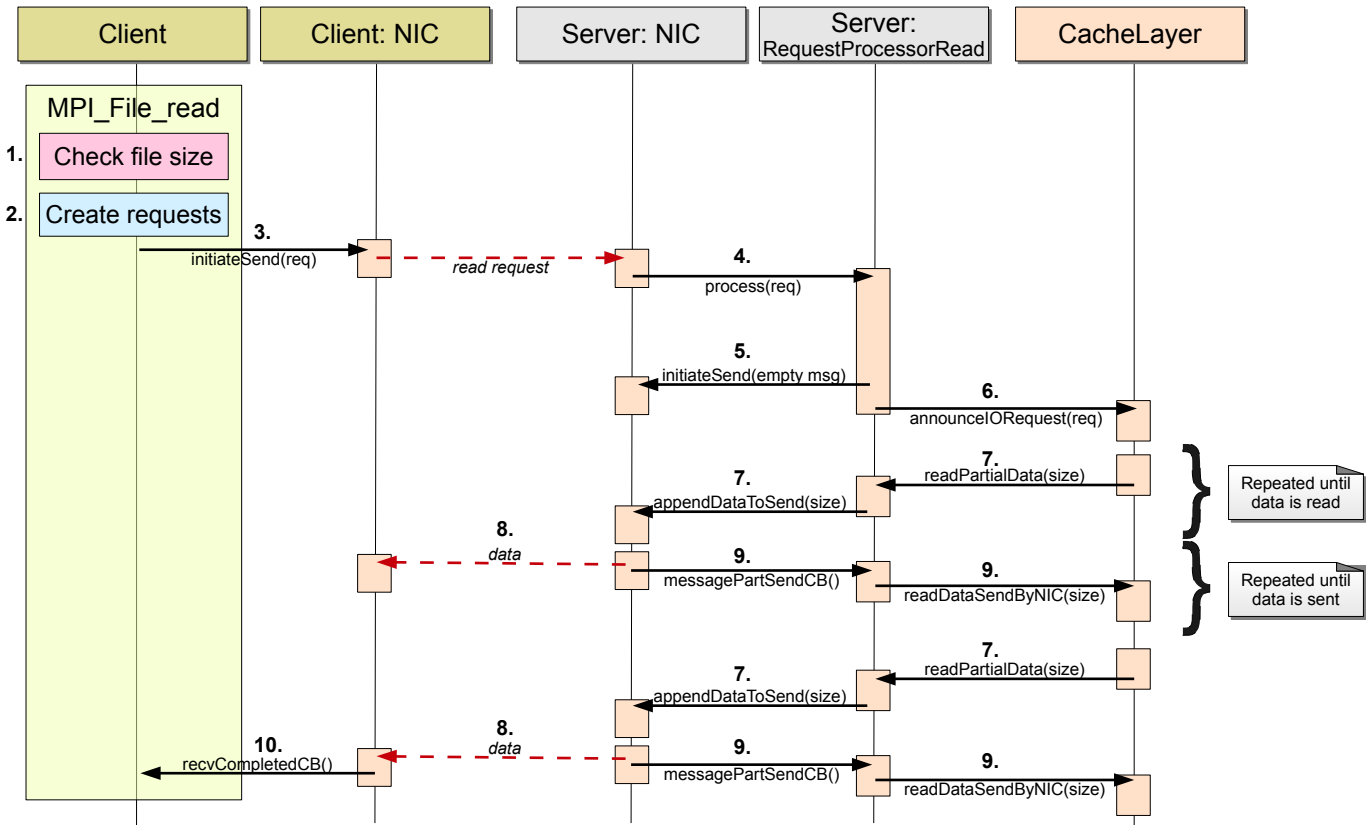


Figure 5.15.: Illustration of the client-server communication protocol – read path.

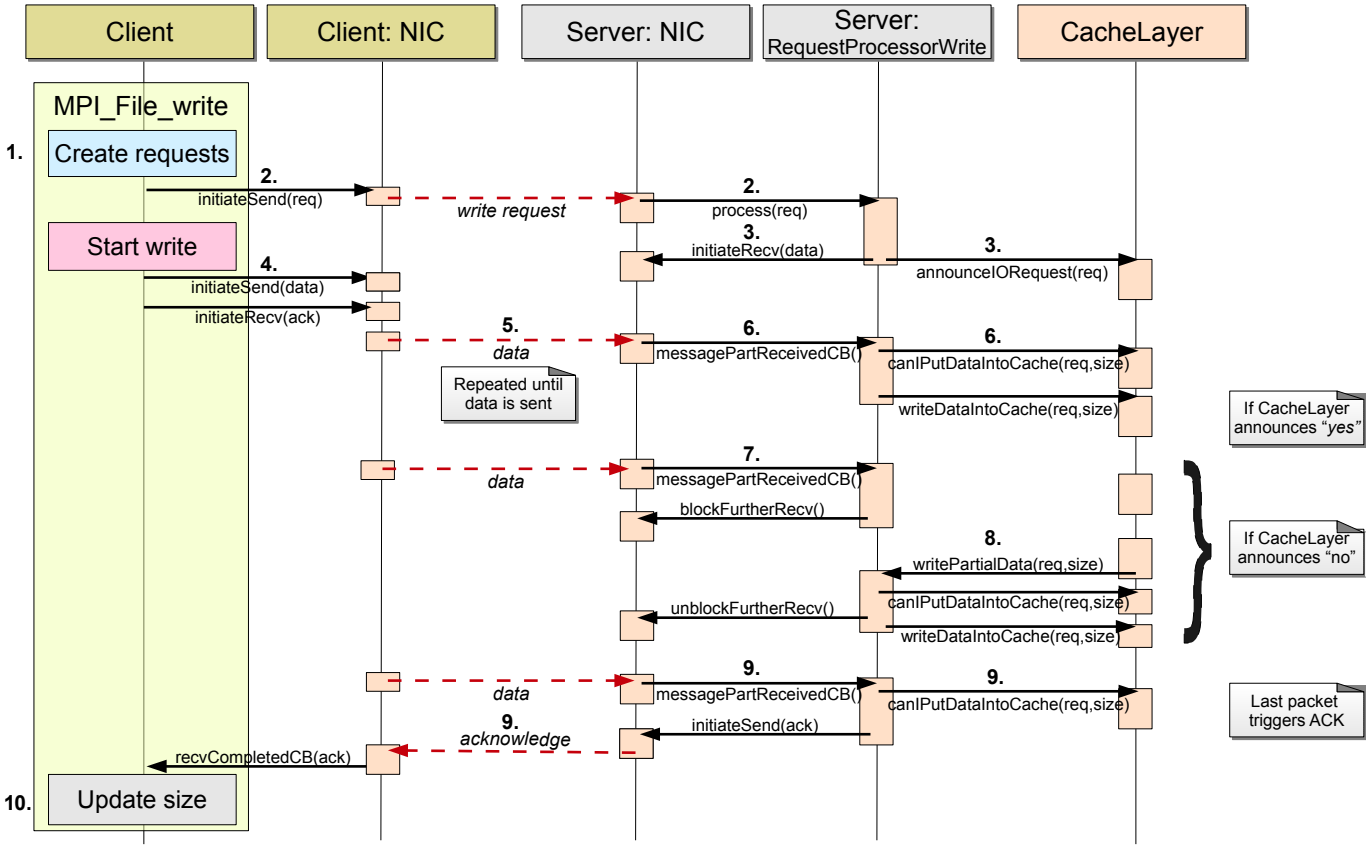


Figure 5.16.: Illustration of the client-server communication protocol – write path.