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