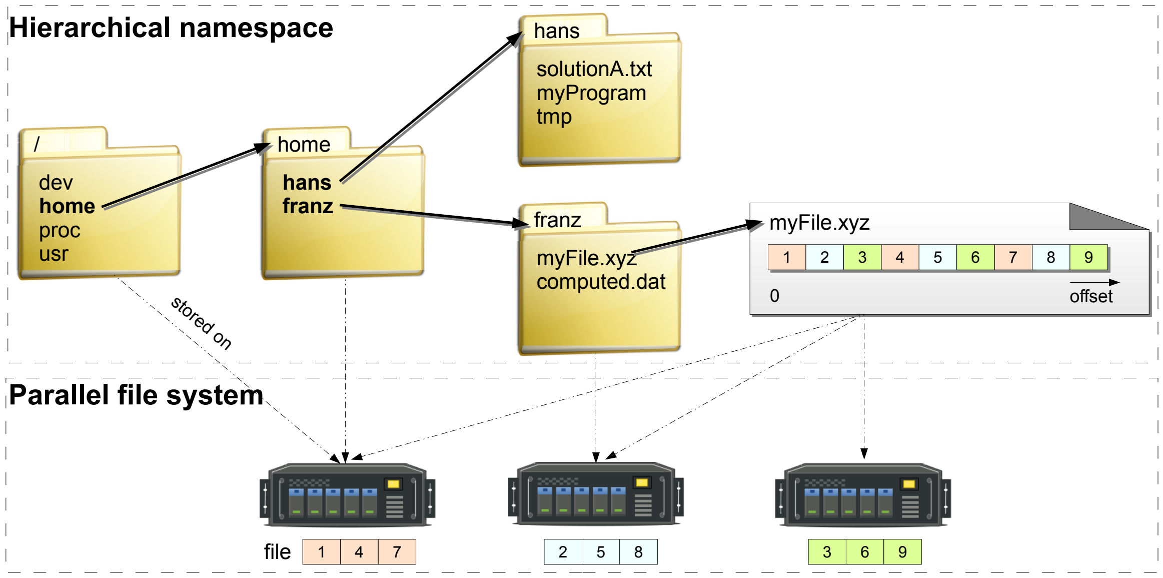

Figure 1.3.: Example hierarchical namespace and mapping of the objects to servers of a parallel file system. Here, metadata of a single logical object belongs to exactly one server, while file data is distributed across all servers.

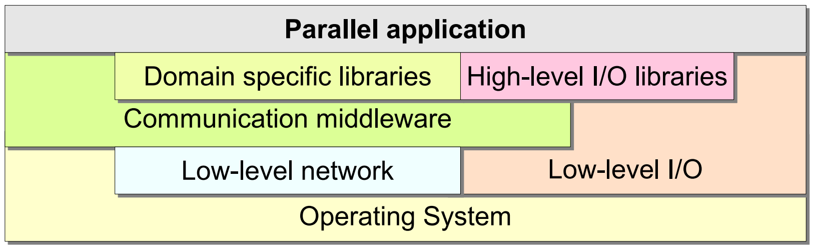

Figure 1.4.: Representative software stack for parallel applications.

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