Original published: M. Hofmann, G. Rünger, and T. Seifert. Transparent redirection of file-based data accesses for distributed scientific applications. In *Proceedings of the 19th IEEE International Conference on Computational Science and Engineering (CSE 2016)*, pages 1–8. IEEE, August 2016. Online available at http://dx.doi.org/10.1109/CSE-EUC-DCABES.2016.236.

Transparent redirection of file-based data accesses for distributed scientific applications

Michael Hofmann, Gudula Rünger, Tommy Seifert
Department of Computer Science
Technische Universität Chemnitz
09107 Chemnitz, Germany
Email: mhofma@cs.tu-chemnitz.de

***Abstract*—File-based data access represents one of the most common ways of providing input data and retrieving output data in scientific applications. However, in distributed computing environments, the execution platform of the application and the storage location of the data might be different. Thus, additional efforts for performing the data transfers are required either by the application programmer or user. In this article, we propose an approach for accessing non-local data files with existing applications that can be performed by a regular user. The existing file-based data access operations of an application are intercepted and redirected to dedicated storage services with a specialized communication library. The approach is transparent in the sense that the application does not have to distinguish whether the data access is redirected or not. We present an implementation that supports various POSIX file I/O operations. Performance results with a benchmark application and the mesh generator Gmsh are shown to demonstrate the good performance and the low overhead of the proposed approach.**

***Index Terms*—file accesses, distributed applications, data coupling, engineering simulation, scientific computing**

I. Introduction

Today's simulations in science and engineering are increasingly complex applications build up from individual software components. These components are often highly specialized software tools (e. g., a special-purpose programming library or application) developed independently from each other. Depending on their unique purposes, each of these components might also have individual requirements (e. g., computational power, memory or storage capacity, user interaction) regarding the utilized hardware platform. Thus, instead of implementing complex simulations as monolithic application codes, it becomes increasingly important to develop an application such that it reuses existing codes and supports distributed executions. The potentially large variety of software components and hardware platforms involved leads to several challenges for the application programmer. Especially the data coupling between the software components has to be supported in an efficient way [1].

The complex simulations which we consider are applications from mechanical engineering for optimizing lightweight structures based on numerical simulations. This class of applications is extensively studied in the research project MERGE[1]. The simulations cover the manufacturing process of short fiber-reinforced plastics with a computational fluid dynamics (CFD) application based on OpenFOAM [2]. The mechanical properties of the structures are subsequently tested by simulating specific operating load cases with an in-house finite element method (FEM) application called SPC-FEM implemented in Fortran [3]. Additional software components perform the optimization methods, distribute compute and data intensive numerical simulation jobs among HPC platforms, handle interactive user input, and store and convert simulation data. To support a flexible coupling of these individual software components in distributed computing environments, the Simulation and Data Coupling (SCDC) programming library [4] has been developed.

Existing tools and applications in scientific computing are usually not prepared for the usage in distributed computing environments. However, there exist a large variety of programming frameworks and libraries to achieve the coupling of independent software components of scientific applications. An overview of the SCDC library and a comparison with other existing approaches is given in [1]. All these approaches usually require high additional programming efforts to modify existing applications in such a way that data accesses to non-local storage locations can be performed.

In this article, we present a new approach that works without modifying an application and, thus, is applicable for a broad range of users and applications. The existing file-based data accesses of an application are intercepted and immediately redirected using the SCDC library as a communication layer. The approach is transparent in the sense that the application does not have to distinguish whether its input or output data files are stored within the local file system or redirected to a non-local storage location. Instead, the user of the application specifies where the input or output data files are stored and provides either regular file paths or specific URI-based addresses to be used by the SCDC library. We present performance results for file I/O benchmarks and for the mesh generator Gmsh [5].

The rest of this article is organized as follows: Section II discusses related work. Section III presents the approach for redirecting file-based data accesses. Section IV gives an overview of the SCDC library and Sect. V describes its usage for the redirection. Section VI shows corresponding performance results. Section VII concludes the article.

[1]MERGE Technologies for Multifunctional Lightweight Structures, http://www.tu-chemnitz.de/merge

II. Related work

Distributed file systems represent the most common approach for providing file-based data access in situations where the accessing application is executed on a different computing platform than the data storage. Widespread systems and protocols include the Lustre file system [6] and the IBM General Parallel File System (GPFS) [7] for HPC platforms, the Hadoop Distributed File System (HDFS) [8] for cloud computing, as well as the Network File System (NFS) [9] and the Andrew File System (AFS) [10] for general computer networks. Shared directories of a distributed file system are usually mounted within the hierarchy of the local file system of a computer such that any application executed on the computer can access the distributed data. However, setting up corresponding file servers and integrating distributed file systems into a specific computer usually requires extensive administration efforts and privileges that are not within the scope of a regular user of an application. Furthermore, distributed file systems usually enforce a centralized data transfer across dedicated file servers, but do not provide direct point-to-point-oriented data transfer. In comparison to that, the redirection proposed in this article represents a lightweight solution that can be used by any user of an application. Direct point-to-point-oriented data transfer is supported, because the service-oriented approach of the utilized SCDC library allows to run corresponding storage services on any computer in a distributed environment.

There exists a large number of frameworks and toolkits (both application-specific and application-independent) for the coupling of independently developed software components into a single simulation application [11]. Examples include the Earth System Modeling Framework [12], the OASIS framework [13], the Model Coupling Toolkit [14], or the Multiscale Coupling Library and Environment [15]. Specialized communication libraries, such as PSMILe from the OASIS framework [13], the parallel coupler PALM [16], or the Typed Data Transfer (TDT) library [17] provide operations for data exchanges between parallel software components. However, integrating an existing application into these frameworks usually enforces significant changes for the usage of the application. Using the communication libraries in an existing application requires additional programming efforts. In comparison to that, the redirection proposed in this article represents a lightweight approach that does not require changes to the application. Additionally, the utilized SCDC library has several unique advantages such as the dynamic coupling of flexible distributed software components at runtime and the support for various data exchange methods, including direct function calls. A more detailed comparison of the SCDC library with other existing frameworks and libraries is given in [1].

III. Redirecting file-based data accesses

A typical usage scenario of component-based complex simulation codes is illustrated in Fig. 1. The user executes an existing application code either locally on its desktop computer (A) or remotely on the nodes of a compute cluster (B).

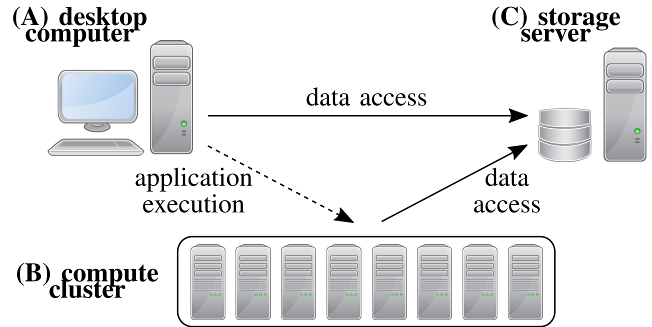

Figure 1. Overview of distributed application executions and data accesses.

The storage location of the input and output data files for the application to be executed is either the desktop computer (A) or a dedicated storage server (C). To support a large variety of applications, it is assumed that an application code is capable of reading and writing its data within the local file system of the computer executing the application. Achieving a seamless utilization of varying storage and execution locations thus requires additional efforts for transferring the data between these locations.

A redirection of file-based data accesses is required and should work in any situation of data file storage or execution platform. There are several approaches to achieve this goal for existing applications:

- The applications and their utilized programming libraries can be modified so that the application code can access data from the specific storage location. This approach requires access to the corresponding source codes, which is not always possible, e. g. for commercial application codes. Moreover, the programming efforts required for modifying the applications can be very high for an application user.
- The input and output data files can be stored temporarily on the computer executing the application code. This approach leads to a runtime overhead for storing data (that is usually only read or written once) and is often only applicable for data of limited size. In the worst case, the total data storage requirements increase by a factor of two if the input and output data files are stored at the original and the temporary location.
- A distributed file system can be used for storing the input and output data. This approach requires significant system administration efforts and is usually not allowed for a regular user of a computer system. Thus, the distributed file system has to be available on all computers to be utilized and the input and output data files have to be stored (and subsequently used) from within the distributed file system.

The approach proposed in this article replaces the existing file access operations utilized by the application by alternative versions that redirect all accesses to alternative storage locations. This solution circumvents the drawbacks and limitations of the previous approaches. However, the implementation re-

quires (1) an appropriate communication layer for performing the data transfers and (2) alternative versions of all file access operations utilized by the application.

(1) The SCDC library is used as a communication layer for performing the data transfers in a distributed computing environment. This programming library provides a unique interface for performing data transfers based on different data exchange methods, such as direct function calls, interprocess communication, and network communication. An URI-based addressing scheme is provided for selecting a specific data exchange method and storage location. These addresses are used as the file paths that a user of an application can specify as input and output data files. A description of the SCDC library and its utilized functionalities is given in Sect. IV.
(2) File accesses within scientific applications are usually based on specific data formats. Proprietary data formats, such as the native *MSH* format of the mesh generator Gmsh [5] or the *STD* format of the SPC-FEM application [3], are directly implemented within the applications. More general data formats, such as the *VTK* format of the Visualization Toolkit (VTK) [18] or the *STL* format for surface geometries [19], might be used through dedicated programming libraries. Due to the large variety of existing data formats, applications, and programming libraries involved, a format-independent approach based on plain file accesses will lead to a widely applicable solution. Thus, the proposed approach redirects the file access operations of the Portable Operating System Interface (POSIX). The specific operations supported and their implementation based on the SCDC library is described in Sect. V.

IV. Simulation and Data Coupling (SCDC) library

The SCDC library is a programming library that can be used by an application programmer to implement the data coupling between distributed program components. In the following, we give an overview of the SCDC library as well as its supported data exchange methods and data storage functionalities. A more detailed description of the SCDC library is given in [1].

A. Overview

The SCDC library follows a service-oriented approach where all interactions between program components are organized as data exchanges between client and service components. The general interaction scheme supported by the SCDC library is application-independent and proceeds as follows: A service component provides access to *datasets* that are managed by *data providers*. A client component interacts with services by executing *commands* on specific datasets provided by services. The corresponding functionalities of datasets and commands depend on the specific data provider. The SCDC library contains several data providers with predefined functionalities as well as a generic data provider whose functionality is specified by the programmer through hook functions.

In this work, the pre-defined *storage data provider* for accessing the local file system will be used. A dataset of a storage data provider represents a directory or file within the local file system of the execution platform on which the corresponding service component is executed. Executing commands on these datasets can be used, for example, to read or write files. More detailed information about the utilization of the storage data provider and its supported functionalities is given in Sect. IV-C. Additionally, the pre-defined *relay data provider* will be used as a bridge between client and service components that run on execution platforms which are not directly connected. The datasets of a relay data provider have the same functionalities as the corresponding datasets of the target service configured for the relay by the programmer. Executing a command on the dataset of a relay service invokes the command on the target service. The relay data provider is used for the performance experiments presented in Sect. VI.

The datasets of an SCDC service are identified with an URI-based addressing scheme:

```
<scheme>://<authority>/<base>/<path>
```

The `scheme` part identifies the data access method to be used to interact with an SCDC service and the `authority` part identifies the specific SCDC service to interact with. In Sect. IV-B, the currently supported data access methods and the corresponding specification of `scheme` and `authority` are described. The `base` part identifies the specific data provider of the SCDC service and the remaining `path` identifies the specific dataset. The following example shows a valid URI address for accessing a file of a storage service:

```
scdc+tcp://gupta/storeHDD/bench/rand1G.dat
```

Network communication with a TCP socket is used to interact with the storage service running on host `gupta`. File `bench/rand1G.dat` within the root directory of the storage data provider `storeHDD` will be accessed.

The application programmer of a client component has to use the library functions to open a dataset of a service, execute one or several commands on the dataset, and close the dataset when it is not used anymore. The execution of a command is implemented within the library in the following steps:

1) The input data of the command is transferred from the client to the service.
2) The corresponding data provider managing the dataset performs the command on the service.
3) The output data of the command is transferred from the service back to the client.

The input and output data are provided by the application programmer through dedicated *data objects* that contain at least a plain memory buffer for storing the input and output data. Furthermore, a data object can optionally contain a reference to a *next function* that supplies additional input or output data. The next function of an input data object has to be specified by the programmer and will be called by the SCDC library during the execution of the command whenever additional input data is required. The next function of an output

data object is returned by the SCDC library after executing the command and has to be called by the programmer to retrieve additional output data. This mechanism allows to transfer and process the input and output data as a data stream and is especially designed for supporting large data sizes. Neither client components nor service components have to store the entire input or output data before their processing.

The SCDC library can be utilized through C and Python programming interfaces. The library functions provide mechanisms for setting up existing application programs as services as well as to access these services from within other application programs as clients.

B. Data access methods

The SCDC library supports different data access methods to execute a command on a dataset (i. e., invoked by a client and performed by a service). Direct access is enabled as default and connects all commands executed by a client to direct function calls of a service. Further support of connection-oriented access can be enabled (and released) by a service component with the `nodeport_open/close` functions. The specific data access method used for accessing a dataset depends solely on the URI address specified by the client. All programming details of the different data access methods are transparently hidden within the SCDC library. Thus, a client component can easily switch between different datasets and data access methods without additional programming efforts. The following specifications for the `scheme` and `authority` parts of the URI address are currently supported.

- `scdc`: Access datasets within the same program component through direct function calls. The `authority` part has to be empty.
- `scdc+uds`: Access datasets of program components running on the same execution platform through inter-process communication with Unix Domain Sockets. The socket within the local file system is given by the `authority` part.
- `scdc+tcp`: Access datasets of program components running on different execution platforms through network communication with TCP sockets. The hostname of the execution platform running the service component is specified by the `authority` part.
- `scdc+mpi`: Access datasets within a distributed memory parallel program with message-passing based on MPI. The `authority` part specifies an existing MPI communicator or a port name to establish an MPI connection.

C. Storage data providers

A service component can have several data providers at the same time. Each data provider has to be created (and released) with the `dataprov_open/close` functions and can be individually addressed with the `base` part of the URI address (see Sect. IV-A). Creating a storage data provider within a service component thus requires to specify an individual string for the `base` part as well as the root directory that should be accessed within the local file system. A client component opens a dataset, executes commands on the dataset, and closes the dataset with the `dataset_open/cmd/close` functions. Each command is specified as a string consisting of the command name and additional optional parameters. A dataset of a storage data provider represents either a directory or file within the root directory of the storage data provider and supports the following commands:

- `cd`: Change the dataset to select the directory or file given as additional parameter.
- `ls`: List information about the selected directory or file. If a directory is selected, then a list of the directory entries (i. e., subdirectories and files) is returned as the output data of the command. If a file is selected, then the size of the file is returned as the output data of the command.
- `rm`: Remove the selected directory or file and change the dataset to the parent directory.
- `put`: Create a new file and/or write data to a file. If a directory is selected, then create a new file given as additional parameter. If a file was selected or a new file was created, then write the input data of the command to the file. In both cases, additional parameters can be used to specify the offset and the size of the data to be written.
- `get`: Read data from a file. If a directory is selected, then read the file given as additional parameter. If a file was selected, then read the selected file. In both cases, the data read is returned as the output data of the command and additional parameters can be used to specify the offset and the size of the data to be read.

V. Transparent redirection of file accesses

The goal of this work is to implement a redirection of the file access operations of existing applications to an SCDC storage service. In the following, we give an overview of the approach and describe the implementation with the SCDC library as well as the supported POSIX file I/O operations.

A. Overview

The redirection should be transparent in the sense that a user of an application is able use a local file or an URI address of an SCDC storage service as input or output data file. However, it should not be required to modify the application code itself, because otherwise only a limited number of applications might be supported. Instead, the operation of opening a file within an existing application is intercepted. Depending on the file path it is detected whether an URI address or a file within the local file system is used. All following operations (e. g., reading, writing, or closing a file) are then either performed with the SCDC library as described in Sect. IV-C or with the original operations provided by the operating system.

To redirect the file access operations, an alternative implementation for each POSIX file I/O operation utilized by the application is provided. These alternative implementations are included in a software library called `libfileio_scdc` that can be used either as a static library (i. e., an *archive* on Linux systems) or as a shared library (i. e., a *shared object file* on Linux systems). The static library can be used if the user

performs the linker step for creating the application executable itself. The shared library can be used for the execution of dynamically linked application executables. By adding the shared library `libfileio_scdc.so` to the `LD_PRELOAD` environment variable, the dynamic loader `ld-linux.so` on Linux systems loads the shared library before all other libraries are loaded [20]. This causes that the alternative implementations for file accesses are used by an application instead of the original operations provided by libraries of the operating system. Fig. 2 illustrates the redirection of file accesses to local and remote file systems with the `libfileio_scdc` library.

Fig. 3 gives an overview of the software layers involved in redirecting the file access operations of existing applications:

1) *User application layer* using POSIX file I/O operations, such as `fopen`, `fread`, `fwrite`, and `fclose`.
2) *LFIO layer* mapping POSIX file I/O operations either to the operations of the operating system or the SCDC library as described in the next subsection.
3) *Operating system layer* providing POSIX file I/O operations for local file accesses.
4) *SCDC layer* providing service-oriented accesses to remote file systems as described in Sect. IV.

B. Mapping to the SCDC library

The POSIX API supports various file access operations with overlapping functionalities. To map all these operations to the corresponding operations of the SCDC library, the `libfileio_scdc` library uses a separate layer called *LFIO*. This layer contains a small set of functions for plain byte-oriented accesses to a file within an SCDC storage service. Handles are used to distinguish between different SCDC storage files that are accessed at the same time. The LFIO handle of an SCDC storage file is a structured data type that contains an SCDC dataset representing a file within an SCDC storage provider (see Sect. IV-C) as well as the current read/write position within the file.

The `lfio_open` function uses a given URI address of an SCDC storage file to perform the corresponding `dataset_open` function of the SCDC library and returns a new LFIO handle. The following opening modes are supported:

- `READ`/`WRITE`: Open the file for reading and/or writing.
- `SYNC`: Disable an optional I/O buffering.
- `CREATE`: Create the file with a `put` command first.
- `TRUNCATE`: Empty the file by removing and recreating the file with `rm` and `put` commands.

The information about the `READ`, `WRITE`, and `SYNC` modes are stored within the handle. The required commands for the `CREATE` and `TRUNCATE` modes are performed with the `dataset_cmd` function of the SCDC library.

The `lfio_read` function reads a requested amount of data from the SCDC storage file of an LFIO handle. The data is retrieved with a `get` command using the current read/write position of the handle as offset and the given memory buffer for the output data object of the command (see Sect. IV-A). The number of bytes read is returned and the current read/write position of the handle is increased. Analogously, the `lfio_write` function writes a given amount of data to an SCDC storage file with a `put` command.

An I/O buffering is implemented within the LFIO handle so that `put` and `get` commands transfer bigger amounts of data if possible. The buffering considers only a continuous area around the current read/write position. For example, if only a few bytes should be read, a bigger amount of data is retrieved with a `get` command and this additional data is stored in a separate buffer that can be used to serve subsequent reads. Analogously, writing small amounts of data is first directed to the buffer and only transferred with a `put` command if necessary. The `lfio_sync` function can be used to synchronize the buffered data. The I/O buffering is not intended as a cache that reduces the amount of data transferred, but as a way of reducing the number of data transfers. Thus, the size of the buffer is kept relatively small (see Sect. VI-A) and the order of read and write operations is maintained.

The `lfio_seek` function can be used to set the current read/write position of an LFIO handle relative to either the start of the file, the current position, or the end of the file. Setting the position relative to the end of the file is achieved by determining the size of the file with an `ls` command. Finally, the `lfio_close` function releases an LFIO handle by writing the remaining buffered data with a `put` command and performing the corresponding `dataset_close` function.

C. Emulation of POSIX file I/O operations

POSIX file I/O operations reference open files either with a `FILE` handle (i. e., an opaque data structure) or a file descriptor (i. e., an integer value). When the `libfileio_scdc` library intercepts an invocation of a POSIX file I/O operation, it is required to decide depending on the `FILE` handle or the file descriptor whether the mapping to the SCDC library or the original operation of the operating system should be used. Since application programmers only work with pointers to `FILE` handles, a `FILE` handle can be replaced with an LFIO handle without any notice by the application programmer. However, the LFIO handle contains a specific marker such that the `libfileio_scdc` library can decide whether a given pointer references a `FILE` handle or an LFIO handle. For the file descriptors, a special range of integer values is used to represent open SCDC storage files (without notice by the application programmer). If the given file descriptor is within this special range, then a static table is used to map this SCDC storage file descriptor to its corresponding LFIO handle. The reverse mapping (i. e., from LFIO handle to file descriptor) is supported by intercepting the POSIX operation `fileno`.

Opening SCDC storage files is supported by intercepting the `fopen` and `open` operations and depending on the file path, either the `lfio_open` function or the original operation of the operating system is called. The given modes and flags of the POSIX operations are mapped to the corresponding opening modes described in the previous subsection. Additionally, the `libfileio_scdc` library also intercepts the `fdopen`, `freopen`, and `creat` operations. Closing SCDC

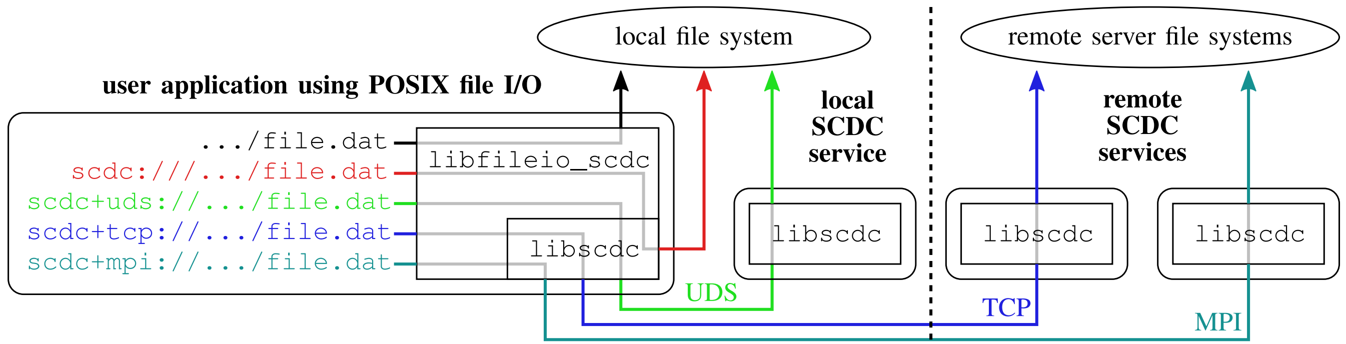

Figure 2. Overview of the redirection of file accesses to local and remote file systems with the `libfileio_scdc` library.

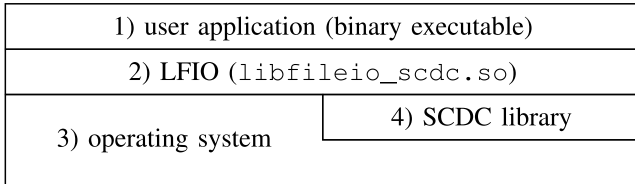

Figure 3. Overview of the software layers involved in redirecting the file access operations of existing applications.

storage files is supported by intercepting the `fclose` and `close` operations and calling the `lfio_close` function. Reading and writing SCDC storage files is supported by intercepting the following POSIX file I/O operations and calling the `lfio_read` and `lfio_write` functions.

- Byte I/O with `fread`, `fwrite`, `read`, and `write`.
- Single character I/O with `fgetc` and `fputc`.
- String I/O with `fgets` and `fputs` (using a loop to search for a newline as the end of the input for `fgets`).
- Formatted I/O with `f[v]scanf` and `f[v]printf` (using in-memory files created with `fmemopen`).

Synchronizing or discarding the optional I/O buffering is supported by intercepting the `fflush` and `fsync` operations and calling the `lfio_sync` function. Changing the current read/write position is supported by intercepting the `fseek` and `lseek` operations and calling the `lfio_seek` function. Finally, the `libfileio_scdc` library also intercepts auxiliary operations such as `ftell`, `feof`, `ferror`, and `clearerr`, file management operations such as `remove` and `stat` as well as additional variants of the POSIX operations (e. g., 64-bit variants).

VI. Performance results

In this section, we present performance results for redirecting the file accesses within a benchmark application as well as a scientific application. We also compare these results with alternative approaches based on a distributed file system and temporary file copies.

A. Experimental setup

For the benchmark applications, two identical compute nodes `ws1` and `ws2` of a compute cluster are used. Each node has two 6-core Intel Xeon X5650 processors with 2.66 GHz, 12 GiB main memory, and a 1 TB Western Digital hard disk. The nodes are connected with a 1 Gigabit Ethernet network and a 10 Gigabit InfiniBand network for MPI communication. For the scientific applications, a dedicated storage node `gupta` within the compute cluster is used. The storage node has two 14-core Intel Xeon E5-2683 v3 processors with 2.00 GHz, 128 GiB main memory, and two 480 GB solid state disks. Additionally, a desktop computer with an Intel Core i7-3770 processor with 3.40 GHz, 8 GiB main memory, and a 240 GB solid state disk is used. The nodes of the compute cluster can only be accessed through a dedicated login node that is connected with a 1 Gigabit Ethernet network to the desktop computer. The data size for the I/O buffering within the LFIO layer (see Sect. V-B) is set to 1 MB. The following results represent file access operations performed without and with redirection through the SCDC library. Bandwidth and latency results achieved for data transfers with the SCDC library (i. e., without file access operations) for the different networks are given in [1].

B. Benchmark application

The benchmark application is executed on the compute node `ws1` and uses the `fopen` operation for opening a file within the local file system or a file redirected with the SCDC library. Results for the local file system access either a local hard disk (HDD) or a shared directory of a distributed NFS file system provided by the login node. Results for the redirection with the SCDC library access either a file on the same compute node `ws1` through direct function calls (direct) or Unix Domain Sockets (UDS) or on compute node `ws2` through network communication with TCP or MPI. Both compute nodes `ws1` and `ws2` execute a corresponding SCDC storage service as the target of the redirection.

Figure 4 shows bandwidth results for reading a file of size 24 GB with the `fread` operation and for writing a file of size 12 GB with the `fwrite` operation. The file accesses to the local hard disk (HDD) achieve bandwidths of about 135 MB/s for reading and about 100 MB/s for writing. These results represent the corresponding limits of the local file system being utilized. Redirecting the file accesses with the SCDC library within the same compute node (SCDC direct,

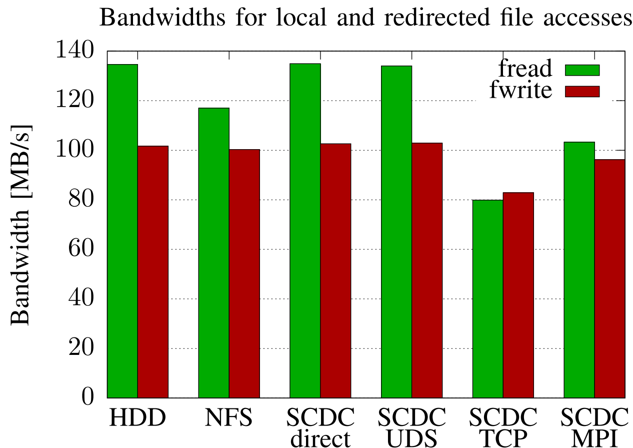

Figure 4. Bandwidth results for reading and writing files without and with redirection through the SCDC library.

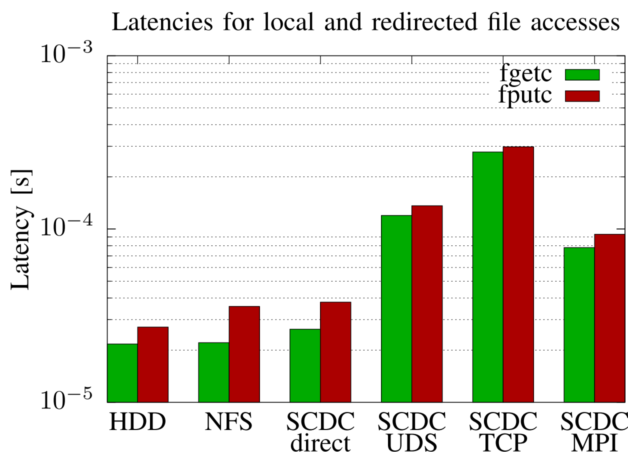

Figure 5. Latency results for reading and writing files without and with redirection through the SCDC library.

SCDC UDS) leads to the same results, thus showing that the overhead caused by the redirection does not reduce the data access performance. The usage of the distributed NFS file system leads to a lower bandwidth for reading, which is caused by the 1 Gigabit communication network. Redirecting the file accesses to the compute node `ws2` leads to bandwidths of about 80 MB/s with TCP and about 100 MB/s with MPI. The differences between reading and writing are smaller in both cases, thus showing that the achieved data access performance is limited by the data transfers. However, since the differences between TCP and MPI are significantly smaller than between the bandwidths of the communication networks, there might be still potential for optimizing the SCDC library.

Figure 5 shows latency results for reading single characters with the `fgetc` operation and for writing single characters with the `fputc` operation (1 GB in total in both cases). The lowest latencies are achieved for the file accesses to the local hard disk (HDD), thus representing the corresponding limits of the local file system being utilized. Due to the asynchronous operation of the distributed NFS file system, there is only a small increase of the latencies. Redirecting the file accesses with direct function calls through the SCDC library within the same compute node (SCDC direct) leads to a small increase in comparison to the HDD results. Thus, the minimum overhead introduced by the SCDC library is only small. Using a redirection with UDS and TCP causes a significant increase of the latencies of about an order of magnitude. This can be attributed to the socket-based communication that leads to an additional overhead from the operating system. In comparison to that, using a specialized HPC communication library such as MPI causes significantly lower overhead.

C. *Scientific applications*

The mesh generator Gmsh [5] is used to demonstrate the redirection of file accesses within a scientific application. Results are shown for using the proposed redirection based on the SCDC library and for an alternative approach that uses temporary files and file transfers with Secure Copy (SCP). The Gmsh application is executed on the desktop computer and uses either local files or files of the storage node `gupta` as input and output data files. Since the desktop computer is not part of the compute cluster, there is no direct network connection to the storage node available. Thus, the login node is set up for tunneling the data transfers using the pre-defined relay data provider of the SCDC library or a separate SSH connection for the file transfers with SCP. A mesh file with about 75,000 vertices and 151,000 elements is used in four different data formats: binary MSH and STL ($\approx$ 7.5 MB), text MSH ($\approx$ 9.3 MB), text STL ($\approx$ 26.7 MB). Depending on the specific data format, the Gmsh application utilizes different file access operations (e. g., plain bytes or formatted strings).

Figure 6 shows Gmsh runtimes for converting mesh files between the MSH and STL formats using local and redirected input and output data files with data transfers through SCP or the SCDC library. In all situations, there are only small differences when using local input and output data files (L2L). This shows that there is almost no overhead caused by the LFIO layer if no redirection is used (see Sect. V-A). With binary data formats, the SCDC redirection is always faster than using SCP. Furthermore, the results show that file I/O represents a large part of the runtime when performing data conversions with Gmsh. For example, with the SCDC redirection, the runtime increases about a factor of two between entire local (L2L) and entire redirected (R2R) input and output data files.

With text formats, the runtimes for all conversions with Gmsh increase. Furthermore, the differences between the runtimes with the SCDC redirection and with SCP are very small in all situations. This behavior is caused by the specific file access operations utilized by Gmsh for reading and writing text format MSH and STL files (i. e., `fgets`, `fscanf`, and `fprintf`). Even though these operations read and write only small amounts of data at once, the buffering used within the LFIO layer (see Sect. V-B) leads to fewer data transfers with large amounts of data. Thus, the runtime caused by the data transfers as well as by the specific functionality of the data

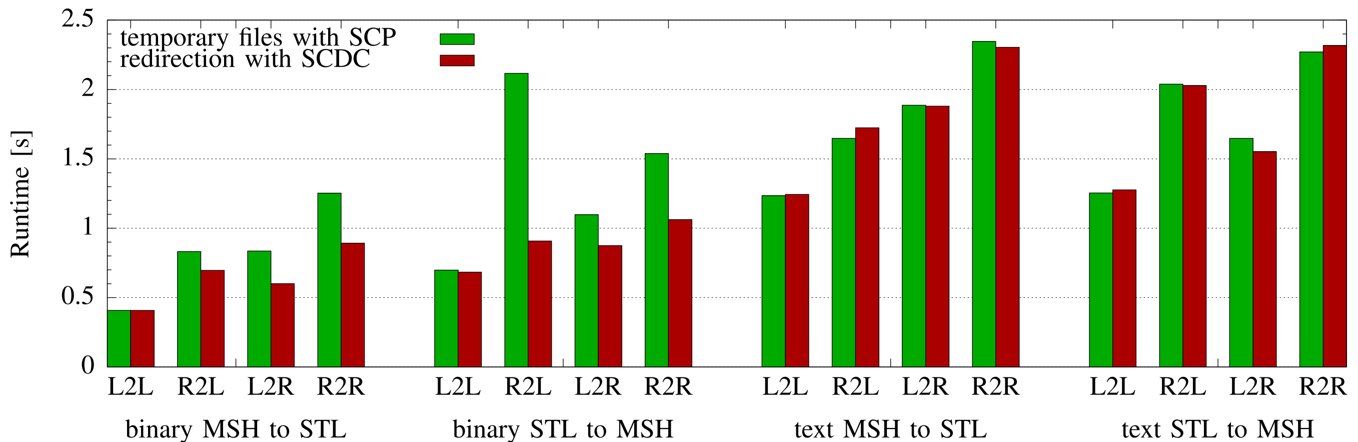

Figure 6. Gmsh runtimes for converting mesh files with local input and output (L2L), redirected input and local output (R2L), local input and redirected output (L2R), and redirected input and output (R2R). Data files are either transferred with SCP or with the SCDC library.

access operations (i. e., search for line ending in `fgets`, text parsing in `fscanf`, text formatting in `fprintf`) is the same with the SCDC redirection and with SCP.

VII. Conclusion

In this article, we have proposed an approach for redirecting the existing file-based data access operations of scientific applications to arbitrary storage locations within a distributed computing environment. By emulating the specific POSIX file I/O operations utilized by the applications, a transparent redirection without modifying the applications was achieved. Due to the large number of existing POSIX file I/O operations, a separate software layer was introduced to ease the mapping to the communication library utilized for the data transfers. The reading and writing of data is buffered within this software layer to avoid extensive transfers of small amounts of data. Performance results with a benchmark application demonstrated the low overhead introduced by the redirection approach. The results with the mesh generator Gmsh showed that our approach is either faster or at least as fast as an alternative approach with explicit transfers of temporary files.

Acknowledgment

This work was performed within the Federal Cluster of Excellence EXC 1075 “MERGE Technologies for Multifunctional Lightweight Structures” and supported by the German Research Foundation (DFG). Financial support is gratefully acknowledged.

References

[1] M. Hofmann and G. Rünger, “Sustainability through flexibility: Building complex simulation programs for distributed computing systems,” *Simulation Modelling Practice and Theory, Special Issue on Techniques And Applications For Sustainable Ultrascale Computing Systems*, vol. 58, no. 1, pp. 65–78, 2015.

[2] H. Jasak, A. Jemcov, and Z. Tukovic, “OpenFOAM: A C++ library for complex physics simulations,” in *Proc. of the Int. Workshop on Coupled Methods in Numerical Dynamics (CMND’07)*, 2007, pp. 1–20.

[3] S. Beuchler, A. Meyer, and M. Pester, “SPC-PM3AdH v1.0 - Programmer’s manual,” *Preprint SFB/393 01-08, TU-Chemnitz*, 2001.

[4] M. Hofmann, F. Ospald, H. Schmidt, and R. Springer, “Programming support for the flexible coupling of distributed software components for scientific simulations,” in *Proc. of the 9th Int. Conf. on Software Engineering and Applications (ICSOFT-EA 2014)*. SciTePress, 2014, pp. 506–511.

[5] C. Geuzaine and J.-F. Remacle, “Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities,” *Int. J. for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[6] *Lustre Software Release 2.x: Operations Manual*, http://lustre.org/, 2016.

[7] *IBM General Parallel Filesystem (GPFS)*, https://www-03.ibm.com/systems/storage/spectrum/scale/, 2016.

[8] *Hadoop Distributed File System (HDFS)*, https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2016.

[9] R. Arpaci-Dusseau and A. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015, ch. Sun’s Network File System (NFS).

[10] ——, *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015, ch. The Andrew File System (AFS).

[11] D. Groen, S. Zasada, and P. Coveney, “Survey of multiscale and multiphysics applications and communities,” *Computing in Science & Engineering*, vol. 16, no. 2, pp. 34–43, 2014.

[12] C. Hill, C. DeLuca, V. Balaji, M. Suarez, and A. da Silva, “The architecture of the earth system modeling framework,” *Computing in Science & Engineering*, vol. 6, no. 1, pp. 18–28, 2004.

[13] R. Redler, S. Valcke, and H. Ritzdorf, “OASIS4 – A coupling software for next generation earth system modelling,” *Geoscientific Model Development*, vol. 3, no. 1, pp. 87–104, 2010.

[14] J. Larson, R. Jacob, and E. Ong, “The Model Coupling Toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models,” *Int. J. of High Performance Computing Applications*, vol. 19, no. 3, pp. 277–292, 2005.

[15] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, C. P.V., and A. Hoekstra, “Distributed multiscale computing with MUSCLE 2, the Multiscale Coupling Library and Environment,” *J. of Computational Science*, vol. 5, no. 5, pp. 719–731, 2014.

[16] A. Piacentini, T. Morel, A. Thévenin, and F. Duchaine, “O-PALM: An open source dynamic parallel coupler,” in *Proc. of the IV Int. Conf. on Computational Methods for Coupled Problems in Science and Engineering*, 2011, pp. 1–11.

[17] C. Linstead, *Typed Data Transfer (TDT) User’s Guide*, https://www.pik-potsdam.de/research/transdisciplinary-concepts-and-methods/tools/tdt/typed-data-transfer-tdt-user-s-guide, 2004.

[18] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object-oriented Approach to 3D Graphics*. Kitware, 2006.

[19] *Stereolithography Interface Specification*, 3D Systems Inc., 1988.

[20] *Linux Programmer’s Manual, ld.so, ld-linux.so**, http://man7.org/linux/man-pages/man8/ld-linux.so.8.html, 2015.