Sustainability through flexibility: Building complex simulation programs for distributed computing systems

Michael Hofmann*, Gudula Rünger

Department of Computer Science, Chemnitz University of Technology, 09111 Chemnitz, Germany

Abstract

Complex simulation programs in science and engineering are often built up from a diverse set of existing applications. The large variety of application codes and their high computational demands lead to an increasing utilization of distributed computing systems. Furthermore, the need for developing sustainable simulation programs, especially with regard to ever increasing data sizes, requires a profound flexibility such that application codes and hardware resources can be easily replaced or extended. In this article, we propose a methodology for building complex simulation programs for distributed computing systems. A software library specifically designed to support a client-server-based development of simulation program components is presented. An application example for the simulation and optimization of lightweight structures in mechanical engineering is used to demonstrate the approach.

Keywords: scientific computing, distributed simulations, data coupling, parallel computing

1. Introduction

Today's simulation programs in science and engineering are increasingly complex and are often built up from already existing application codes which might come from commercial or scientific sources. Since the existing application codes are diverse in many aspects, such as the mathematical simulation method, the internally and externally available data structures, the programming language and environment as well as the sequential or parallel hardware addressed, their combination includes several challenges. In addition, the cooperation of the application codes should usually be flexible in the sense that each specific scientific or engineering problem might require a different combination of these codes. Naturally, building a specific simulation program that combines different

*Corresponding author. Tel.: +49 371 531 35571
Email addresses: `mhofma@cs.tu-chemnitz.de` (Michael Hofmann), `ruenger@cs.tu-chemnitz.de` (Gudula Rünger)

application codes leads to a programming effort which is too large a burden for a quick and productive use by the scientist interested in the specific scientific question. Furthermore, the continuing trend towards multi-simulations as well as the need for an efficient utilization of distributed computing resources and future ultrascale systems leads to great challenges for application programmers.

The goal of our work is to support the building of complex simulation programs with great flexibility to ensure a sustainable development process. We propose a component-based client-server programming model with which a coarse-grained program structure can be specified. As an application example in mechanical engineering, we consider the optimization of lightweight structures within the project MERGE[1]. This complex simulation program includes application codes for computational fluid dynamics (CFD) and finite element methods (FEM), optimization codes, and customized programs to prepare the input data for the simulations and to evaluate their results.

For each application code, there might exist different variants which should be interchangeable in the complex simulation program. An interchange might be desirable when a specific functionality is required; for example, an FEM code might be interchanged by an adaptive FEM code. The execution of the simulation program should be flexibly distributed in a hardware setting with replaceable components. That means it should be possible to utilize both single computers and distributed computing systems with sequential or parallel hardware. To support these demands, we have designed a programming model in which each application code is a component which provides its functionalities as service and accesses other components as client.

The proposed programming model requires a transformation of already existing application codes into codes that are able to act as clients and/or servers. Such a transformation can be a tedious and time-consuming work. To support the application programmer, we developed a library for Simulation Component and Data Coupling (SCDC). Figure 1 gives an overview of the software and hardware environment with the SCDC library. The SCDC library provides functionalities to set up and access application codes as clients and servers. To support a flexibly distributed execution of the simulation components, the SCDC library encapsulates all data exchange operations. Depending on the distributed execution, these operations are mapped to appropriate data access methods, for example, through direct function calls or network communication. Since the data sizes of simulation programs can be very large, the SCDC library especially allows data exchanges without a limitation of size. Thus, our general contribution is twofold. We propose a programming model for large modular scientific simulation programs and we provide the SCDC library for the actual programming in such a model. This approach will lead to a more sustainable development process for simulation programs by reducing their need for ad-hoc interfaces and solutions that are only applicable for single use cases or platforms.

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

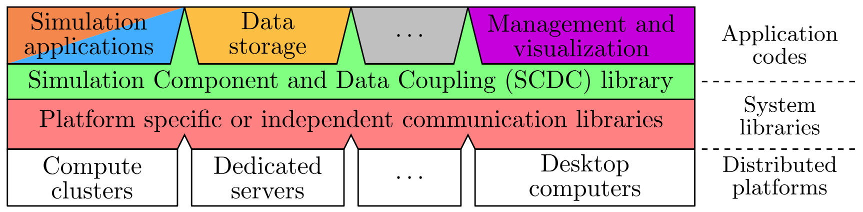

Figure 1: Overview of the software and hardware environment for complex simulations with the SCDC library.

The rest of this article is organized as follows. The programming model is described in Sect. 2 and the SCDC library in Sect. 3. Section 4 demonstrates the proposed development process for an application example in mechanical engineering. Section 5 discusses related work and Sect. 6 concludes the article.

2. Programming model for complex simulation programs

Advanced scientific or technical simulations, such as weather or atmospheric simulations or material design simulations, require complex simulation programs comprising several specific simulations as sub-simulations. Usually, these sub-simulations are developed in isolation by specialists with the appropriate knowledge about the specific mathematics or algorithmic properties. From these basic application codes, complex simulation programs are built.

2.1. Non-functional requirements for complex simulation programs

Complex simulations in science and engineering are usually very specific and assembled for a specific problem to be solved. Thus, a standard approach is that the scientist designs the entire task, identifies subtasks to be performed by existing application codes, e. g. an FEM code, plans the interactions of existing and newly programmed parts, and finds strategies for storing and exchanging the data used or created. Furthermore, the execution of the simulation program is adapted to a specific hardware platform to be used while keeping in mind the hardware requirements of the different application codes.

The development of complex simulation programs is usually done by the application programmer in several steps in a very individual and interactive way. Although, the experienced scientist can handle such a task, there are several drawbacks with the approach. First, this is a tedious and time-consuming work, which leads to a software that is only useful for a single scientist or research group. The development process usually has to be repeated for each new simulation program and if the problem to be solved or the hardware platform to be used changes significantly. Second, the development is strongly focused on functional properties to get a single simulation program work. Non-functional requirements, such as a reasonable flexibility of the simulation program, are usually neglected.

The intention of our work is to provide a methodology for a sustainable development of complex simulation programs. To achieve this goal, the simulation code created has to be flexible in the sense that the scientist gets all the variations he is otherwise used to implement in a manual and time-consuming way. These variations include flexible combinations of basic application codes, exchanges of basic application codes with similar functionality, diverse ways of storing data, and adaptations to and utilizations of different hardware platforms. Thus, it will be necessary to support a flexible combination of simulation components and a flexibly distributed execution on distributed platforms. The development should be based on a programming model in which the scientist can design the application in the way he is used to and at the same time, it should be possible to map the application into a running simulation code.

2.2. Designing the structure of complex simulation programs

We propose a programming model similar to component-based programming in which a program consists of well-defined components hiding the internal implementation and providing interfaces for the incoming or outgoing data. In our case, the basic application codes performing simulations or sub-simulations play the role of components. In addition, there might be components for storing the simulation data as well as for data handling, such as transformations or evaluations. Furthermore, we assume that any kind of additional program control, such as simulation loops, conditional executions, compositions of subprograms, or user interactions, is also represented by dedicated components. Since there are no further obligations or limitations for the implementation of components, this approach provides a great flexibility for the application programmer.

Given a set of components, a complex simulation program is built up in a bottom-up method driven by the application to be designed. The result is a graph-like structure in which the nodes represent basic application codes or other codes from the set of components and directed edges represent the data exchanges between them. Thus, the design of the structure of a complex simulation program as well as the cooperation and coordination between its components will be reduced to the underlying data flow. Whether or not a specific data flow between two components also represents a flow of control depends on the specific role and implementation of the components. In this model, the control flow is only a subset of the data flow in order to model situations in which, for example, the simulation control flows from component A to component B, but the data flow from A to a third component C where they are stored, processed further, or send to other components of the complex simulation program.

Figure 2 shows the graph of a simulation program example with four components L, A, B, and E. The enumerated edges represent the data flow between the components. The component L performs a simulation loop and uses data transfers along the edges (1) and (2) to start the computations of the components A and B. The edges (3)–(5) represent data dependencies between the compute components A and B and the evaluation component E. The final edge (6) represents the data transfer of the evaluation result back to the component L such that it can decide whether and how to continue the simulation loop.

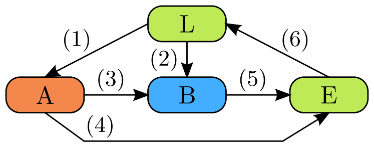

Figure 2: Data flow example for a simulation program consisting of four components.

2.3. Client-Server Interface

For the implementation and coordination of the components, we use a client-server model in which each component can play both the role of a server providing data and the role of a client requesting data from other components. This data-oriented model allows the abstraction from the large variety of components of a complex simulation program. The different tasks and purposes of components, such as executing an application code or storing or converting data, are reduced to their common functionality of accepting and providing data. The interactions between the different components are reduced to a common scheme where active client components interact with passive server components.

Since many kinds of components might exists, the client-server interface is not implemented directly, but specified with a programming library to be described in the next section. The library is used to implement different versions of the components from which the appropriate one is chosen when the complex simulation program is executed. The implementation version can be procedural on one processor as well as parallel or distributed to utilize compute resources such as HPC clusters or servers. In the case that two components get data from one another, the appropriate data access methods are used by the library.

The library provides functions to set up and run a component as a server as well as to access other components as a client. Variations within the complex simulation program will be implemented as separate server components. This includes variations, such as alternative application codes and hardware platforms or integrating different data sources and stores. However, since a client accesses all servers through the same library functions, the complex simulation program can be flexibly changed without additional programming efforts.

2.4. Deriving the execution of a simulation run

The graph structure described in Sect. 2.2 represents the data flow of a complex simulation program. Even though the graph constitutes an order for the data processing, it does not entail a unique order for the execution of the components. Therefore, performing a simulation run requires further decisions about the desired flow of control within the simulation program. We propose a request-driven invocation and execution of the components for the data-oriented client-server model. In this approach, active client components request data exchanges during their execution and passive server components react to these requests. Thus, each edge of the data flow graph has to be implemented by a request taking place between the two connected components. Two different implementations can be distinguished for each single edge of the data flow:

Push-based For the push-based implementation, a directed edge from a component X to a component Y is realized in such a way that the client component X transfers its data to the server component Y. Thus, the component X has to perform a request to which the component Y reacts, for example, by starting computations or by transferring its data to other components.

Pull-based For the pull-based implementation, a directed edge from a component X to a component Y is realized in such a way that the client component Y requests data from the server component X. Thus, the component Y has to perform a request to which the component X reacts, for example, by starting computations or by requesting its data from other components.

Figure 3 shows pseudocode for the data flow example of Fig. 2 using both push- and pull-based implementations. A data exchange is performed with the Cmd function by a client component and answered with the Do-Cmd function of a server component. The function Cmd(srv, cmd, inp) performs an arbitrary command *cmd* with input data *inp* on the server component *srv* and returns the output data of the command. A function Do-Cmd(cmd, inp) is invoked on a server component to perform the command *cmd* with the given input data *inp*. Calling the Cmd function blocks until the corresponding Do-Cmd function is finished by returning the output data of the command.

It is assumed that all components are running at the same time, but the Do-Cmd functions are only executed as reaction to calls of the Cmd function. The component L represents an initially active client component that executes a simulation loop (line 1). In each iteration, data exchanges according to the edges connected to the component L are performed. First, data is transferred to the components A and B with a push-based implementation of the edges (1) and (2) (lines 4–5). Afterwards, the result data is requested from the component E with a pull-based implementation of the edge (6) (line 6).

The components A, B, and E react to requests by executing their Do-Cmd functions. The data exchanges from the component L initiate the computations of the components A and B as asynchronous jobs (lines 4 and 5, respectively). Thus, the client component L can finish their calls to the Cmd function and continues its work while the computations of the components A and B are performed. The components B and E request data from their predecessors (line 4 of component B and lines 3–4 of component E). The components A and B react to these requests by waiting for the end of their computations to return their results (lines 6–8 of component A and lines 7–9 of component B). Furthermore, the component E reacts by evaluating the results obtained from the components A and B and returns its results (line 6 of component E). This data exchange finishes the last call to the Cmd function of the component L (line 6) such that it can continue with the simulation loop.

3. Simulation Component and Data Coupling library

The design of complex simulation programs defines several demands concerning the actual implementation. This demands include the handling of different

Component L

```
while not finished do
    $id$ = 'XYZ'
    prepare input $inA$ and $inB$
    Cmd($A$, $put$, $\{id, inA\}$)
    Cmd($B$, $put$, $\{id, inB\}$)
    $outE$ = Cmd($E$, $get$, $\{id, \varnothing\}$)
```

Component E

```
function Do-Cmd($cmd$, $\{id, d\}$)
    if $cmd$ = get then
        $outA$ = Cmd($A$, $get$, $\{id, \varnothing\}$)
        $outB$ = Cmd($B$, $get$, $\{id, \varnothing\}$)
        $outE$ = Eval($outA$, $outB$)
        return $outE$
```

Component A

```
function Do-Cmd($cmd$, $\{id, d\}$)
    if $cmd$ = put then
        start job $id$
            $outA$ = Comp($d$)
        return OK
    if $cmd$ = get then
        wait for job $id$ to finish
        return $outA$
```

Component B

```
function Do-Cmd($cmd$, $\{id, d\}$)
    if $cmd$ = put then
        start job $id$
            $outA$ = Cmd($A$, $get$, $\{id, \varnothing\}$)
            $outB$ = Comp($d$, $outA$)
        return OK
    if $cmd$ = get then
        wait for job $id$ to finish
        return $outB$
```

Figure 3: Pseudocode for implementing the example components shown in Fig. 2.

data structures, the combination of components written in different languages and programming models, a flexible adaption to sequential or parallel hardware as well as different kinds of data transfers. Most important are data transfers without limitations of the data size and flexible recombinations of the components to perform diverse simulation runs. To meet all these requirements, we have designed a library for Simulation Component and Data Coupling (SCDC) which is based on the programming model described in the previous section.

3.1. Overview of SCDC Library design

The SCDC library provides the technical infrastructure for implementing the components of the data-oriented client-server model. Since software components of various kinds and purposes should be able to be integrated, we generalize them as *services* that can be accessed by *clients*. All interactions between components will be mapped to data access operations between clients and services. Thus, each component has to be prepared to act as an SCDC client that actively accesses data and/or act as an SCDC service that passively provides data.

The data provided by services are generalized as *datasets* and each service can individually define the functionalities of its datasets. For example, the datasets of a storage service might represent data objects stored while the datasets of a scientific simulation service might represent simulation runs. The functionalities of datasets are utilized by executing *commands* on them. The different kinds of datasets are managed by *data providers* whereas each service can have several data providers at the same time. To make an appropriate use of a service, a client has to know about the existence of the service, the functionalities of its datasets, and the format of the input and output data of dataset commands. However, each client accesses the datasets of all services in an equal way independent from aspects, such as how a service is implemented, where it is executed, and what data access methods are used. Services and datasets are identified with an URI-based addressing scheme such that switching between different services and datasets is achieved by changing only the URI address.

The SCDC library is used to implement the service functionalities of a component. Each service has to specify how clients should be able to gain access, e. g. with direct function calls or through network communication. Furthermore, each service has to implement its datasets and functionalities through data providers. The SCDC library includes data providers with predefined purposes (e. g., for accessing the local file system or a MySQL database) as well as generic data providers that execute arbitrary programs or user-defined hook functions. For example, to create a service from an existing application code, the application programmer has to use the SCDC library to let the application code or a dedicated wrapper program act as an SCDC service and configure a generic data provider that executes the instructions to run the application code.

The SCDC library is implemented with C++, but provides a public C and Python interface for the integration into application codes which allow modifications of their source code. With these interfaces, the data access can be performed directly through memory buffers provided to the library functions while at the same time the underlying data exchange is mapped, for example, to network communication. Additionally, the Python interface is used for wrapping closed-source (e. g., commercial) application codes that employ usually a file-based data access. To provide such an application as a service, a dedicated Python program will be used for setting up the service with the SCDC library, executing the specific application code, and managing its input and output files.

3.2. SCDC Library functions

The SCDC Library provides several functions for the application programmer to implement its components as SCDC clients and/or services.

3.2.1. Service functions for setting up data access methods

Implementing a software component as a service requires to set up the supported data access methods. Direct access is enabled as default and connects all commands executed by a client to direct function calls of a service. Further connection-oriented access can be enabled with the following functions:

```
np = nodeport_open(conf, ...)
nodeport_close(np)
```

The `nodeport_open` function initializes a new access method specified with the configuration string `conf` and returns a handle `np` representing the access method. Additional arguments might be given to pass specific configuration parameters to the access method. The `nodeport_close` function release an existing access method given by the handle `np`. Currently, the following access methods can be selected with the configuration string:

uds A Unix Domain Socket is set up for enabling accesses through inter-process communication. An additional argument is used as identifier of the socket within the local file system.

tcp A TCP socket is set up for enabling accesses through network communication. The additional arguments are used to (optionally) specify the network address and port to be used.

mpi Message-passing based on MPI is used for enabling accesses within distributed memory parallel programs. The additional arguments are used to either specify an existing communicator to be used or to establish a connection with the MPI operations `MPI_Open_port` and `MPI_Comm_accept`.

The implementation of the SCDC library uses a self-designed communication protocol to perform requests with the connection-oriented access methods. The operations for establishing these connections and for sending and receiving data are implemented in separate C++ classes that encapsulate the necessary UDS-, TCP-, and MPI-related commands. After the initialization, each access method can be started and stopped temporarily with the following functions:

```
nodeport_start(np, mode)
nodeport_stop(np)
```

The `mode` parameter specifies whether the access method should be started in a blocking way to keep the service running. Otherwise, the `nodeport_start` function returns immediately such that it is possible to enable further access methods or to keep the service running by the application itself.

3.2.2. Service functions for setting up data providers

Data providers of a service component are set up with the following functions:

```
dp = dataprov_open(base_path, conf, ...)
dataprov_close(dp)
```

The `dataprov_open` function initializes a new data provider specified with the configuration string `conf` and returns a handle `dp` representing the data provider. Additional arguments might be given to pass specific configuration parameters to the data provider. Each data provider of a service is accessed by a client through its individual base path `base_path` within the URI-based addressing scheme. The `dataprov_close` function release an existing data provider given by the handle `dp`. The functionality of a data provider depends strongly on its type that is specified with the configuration string. Currently, the following predefined data providers are supported:

fs Access to the local file system of a service is provided, whereas an additional argument specifies the root directory. The corresponding datasets represent the directories and the commands are used to navigate through the file system and to access single files.

store A nonhierarchical folder-oriented storage is provided either within the local file system (**store:fs**) or a MySQL database (**store:mysql**). An additional argument is used to specify either the storage directory or the database access credentials. In both cases, the corresponding datasets represent the folders and the commands are used to store and retrieve the data items of the folders.

relay An arbitrary mapping of the access through a path of the service to the URI address of another service is provided. Such a data provider might be used, for example, to connect distributed services across distinct communication networks or to hide several back-end services behind a single front-end service. Mapping accesses is also supported within a single service as well as between several services in series. The corresponding datasets represent the target datasets of the mappings and support their individual commands. Additional commands are provided to register or remove single mappings. Thus, it is possible to modify the mappings locally from within a service component as well as remotely through the access of another client or service component.

jobrun A job-oriented execution of arbitrary programs is provided. Additional arguments are used to specify the program and the maximum number of jobs that can be executed in parallel. The corresponding datasets represent the jobs and the commands are used to submit a job with its input data as well as to wait for the completion of a job and to retrieve its output data.

jobrun_relay A specialized mapping for the job-oriented execution based on jobrun data providers is provided. This data provider supports the same functionality as the relay data provider and can be used, for example, to mediate the accesses to a large number of compute components through an intermediate component. At the same time, the corresponding datasets and commands perform the execution of jobs identical to a jobrun data provider. If a job is submitted without giving a path that selects a registered mapping, then the job is assigned automatically to one of the registered jobrun data providers (i. e., currently, in a round-robin way).

hook A mechanism for executing user-defined hook functions is provided whenever a dataset is accessed. An additional argument is used to specify the hook functions, whereas it is mandatory to specify at least a hook function to execute dataset commands. Further hook functions for creating and destroying datasets and for initializing and releasing the data provider can be (optionally) given. The functionalities of the corresponding datasets and commands are solely defined by the hook functions and can be used, for example, to integrate application specific codes or auxiliary tools, such as data conversions.

Despite their different functionalities, all data providers support a common way for accessing their configuration parameters as special datasets through the path `<base_path>/CONFIG`. These special datasets provide commands for listing the configuration parameters as well as for retrieving or setting the parameter values. Thus, the client functions for accessing datasets with any of the supported access method can also be used configure of a data provider.

3.2.3. Client functions for accessing datasets

The following client functions are used to access the datasets of a service:

```
ds = dataset_open(uri)
dataset_cmd(ds, cmd, input, output)
dataset_close(ds)
```

The `dataset_open` function opens a dataset given by an URI address `uri` and returns a handle `ds` to the dataset. An URI address has the following format: `<scheme>://<authority>/<path>`. The data access method is specified by `<scheme>://<authority>` and the dataset is selected by `<path>`. A first part of `<path>` represents the base path that selects a specific data provider and the remaining second part of `<path>` identifies a specific dataset of this data provider. Currently, the following types for `<scheme>` are supported to choose between the data access methods described in Sect. 3.2.1:

scdc Access datasets within the same software component through direct function calls. The `<authority>` has to be empty.

scdc+uds Access datasets of software components executed on the same compute node through inter-process communication with Unix Domain sockets. The socket within the local file system is identified by `<authority>`.

scdc+tcp Access datasets of software components executed on different compute nodes through network communication with TCP sockets. The hostname of the compute node and (optionally) the port is specified by `<authority>`.

scdc+mpi Access datasets within a distributed memory parallel program with message-passing based on MPI. The `<authority>` specifies either an address of an existing communicator or a port name to establish a connection with the MPI operation `MPI_Comm_connect`.

The `dataset_cmd` function is used to execute an command string `cmd` with a given dataset `ds`. A command string always consists of a command name with optional parameters, whereas it depends on the data provider of the dataset which commands are supported. The predefined data providers described in the previous subsection implement their functionalities with the commands "put", "get", "cd", "rm", and "ls". In contrast, the hook data provider supports arbitrary command names and passes them to the given hook function. After finishing the work with a dataset, the `dataset_close` function is used to close it.

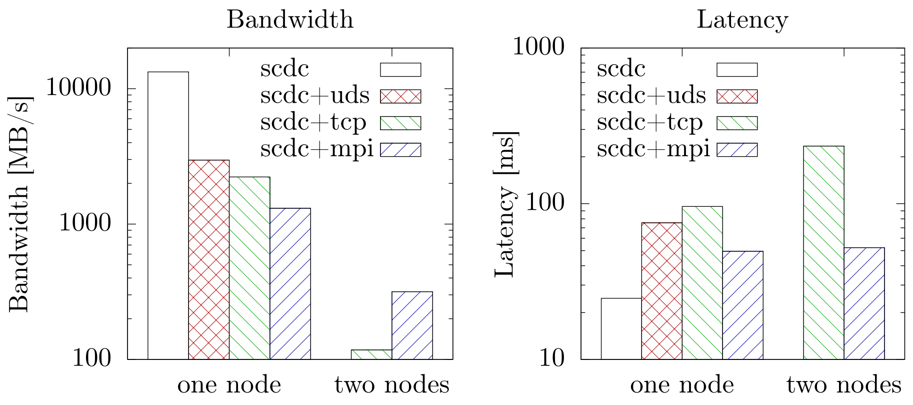

Figure 4: Bandwidth and latency results achieved with direct function calls (scdc), Unix Domain Sockets (scdc+uds), TCP sockets (scdc+tcp), and message-passing with MPI (scdc+mpi).

The input data for a command is represented by a `dataset_input` object which contains a memory buffer and a format identifier that can be freely chosen. Additionally, the object contains a reference to a function that can be specified by the programmer to supply a continuous stream of input data. This function is called inside the `dataset_cmd` function when new input data can be processed. The same functionality is provided for the output data of a command represented by a `dataset_output` object. In this case, the `dataset_cmd` function returns a reference to a function that can be used by the programmer to process a continuous stream of output data. This stream-oriented data handling supports input and output data without limitations of their size.

3.3. Performance results for data exchanges

A benchmark program consisting of two components implemented in Python was used to compare the performance of the different data access methods supported by the SCDC library. The service component uses a predefined benchmark data provider that supports the generation of arbitrarily large zero output data with the "get" command. The client component accesses this service to retrieve 10^9 bytes output for bandwidth measurements and 1000 times 1 byte output for latency measurements. Two compute nodes connect with an 1 Gigabit Ethernet and a 10 Gigabit InfiniBand connection are used. Data exchanges within a node through direct function calls are achieved by integrating the benchmark data provider into the client component. TCP and MPI communication within a node are performed through shared memory by the operating system and the MPI implementation. Communication between two nodes uses the Ethernet connection for TCP and the InfiniBand connection for MPI. Switching the data access methods is achieved through the URI address used by the client component without additional programming efforts.

Fig. 4 shows bandwidth and latency results for data exchanges within a node and between two nodes using the different data access methods described in the previous subsection. Average results of 10 measurements are shown, whereas the relative standard deviation was less than 4 %. The bandwidth achieved with direct function calls corresponds to about 86 % of the performance achieved for generating zeros with the `memset` operation. The UDS method is faster than TCP and MPI, thus showing the advantages of supporting a method dedicated to inter-process communication within a single node. The bandwidth between two nodes with TCP corresponds to about 94 % of the maximum bandwidth of 1 Gigabit. In contrast, MPI achieves only about 315 MB/s while a dedicated MPI benchmark program could achieved about 845MB/s. However, since the implementation of the SCDC library has not yet been optimized for performance, there might be still opportunities for performance improvements left.

Using direct function calls leads to the lowest latency results and represents the minimum overhead introduced by the SCDC library. The latency with MPI is lower than with UDS and TCP, thus benefiting from its HPC optimization. While the latency with MPI is about the same within a node and between two nodes, the latency with TCP increases by a factor of about 2.4. In general, the results demonstrate the individual advantages of the different methods currently supported. Even though supporting only TCP as a general method for all data accesses would be functionally sufficient, dedicated methods such as direct function calls and MPI might be strongly required for HPC applications.

3.4. Discussion of the properties of the SCDC library

The SCDC library follows an application-independent approach without any domain-specific data types or tools. The underlying generic data model consists of data providers, datasets, and plain input and output data with a free format identifier. By defining the data objects and functionalities represented by datasets, the application programmer can flexibly adapt this model to its application area. Existing tools and libraries for domain-specific tasks, such as data interpolation or domain decomposition, have to be integrated individually with hook data providers. Nevertheless, it is also possible to extend the C++-based implementation of the SCDC library with new predefined data providers.

The SCDC library integrates a diverse set of data access methods, from direct function calls and memory accesses to network-based communication. However, switching between different access methods does not require additional programming efforts, thus supporting a flexibly distributed execution of the software components. The results shown in Sect. 3.3 demonstrate the individual performance of the different access methods with the SCDC library. Furthermore, the predefined relay data provider (see Sect. 3.2.2) can be used as a bridge between any of the supported access methods.

The API of the SCDC library uses configuration strings and functions with variable arguments to integrate a diverse set of methods and techniques behind a common interface. Only a small number of functions is used while at the same time, the library can be flexibly extended with new data access methods and data providers without breaking the existing interface. Neither a static nor any other kind of global configuration is required to set up and perform data exchanges between different software components. Instead, each software component can set up and modify its data access methods and data providers dynamically at runtime. Furthermore, by accessing the configuration parameters of data providers as special datasets, each component can also be modified remotely by other components. This allows, for example, to switch the execution of simulation jobs with a jobrun data provider transparently from a local compute node to multiple worker nodes behind a jobrun_relay data provider and to let these worker nodes register themselves autonomously at the relay.

The proposed method for building complex simulations with the SCDC library aims to improve the sustainability of the application development process. Usually, this is achieved by introducing a fixed programming interface to increase the reusability of autonomously developed software components. Our approach focuses instead on the development of applications that are better maintainable and modifiable, for example, by flexibly replacing single software components or executing them on distributed computing resources. These goals are further supported by the lightweight library approach that is less invasive to existing application codes and does not involve extensive runtime environments or mandatory dependencies to other software packages. Each data access method and each predefined data provider can be individually disabled if prerequisites (e. g., support for MPI or MySQL) are missing. An optional tracing of output about the behavior of each software component is provided for debugging.

The presented approach is suitable for scientific simulations in many aspects: It provides the flexibility of a service-oriented approach, but omits the usage of complex development frameworks or middlewares that introduce performance overheads or rely on programming languages and software platforms that are not appropriate for scientific simulations. Instead, a programming library is provided which easily coexists with other programming libraries common in scientific HPC applications. The efficient transfer of large data sets is supported based on an inherent stream-oriented processing of unlimited-size data that is not restricted by intermediate buffers. The SCDC library can be used for various kinds of scientific simulation components, e. g. sequential and parallel codes, interactive user programs and non-interactive compute programs, for the wrapping of closed-source applications and directly in application codes. Finally, the Python interface provides programming support for a scripting language that is widely used in scientific computing and the jobrun data provider is specifically designed to ease the distributed execution of compute-intensive simulations.

4. Mechanical engineering application example

The simulation and optimization of lightweight structures is used as an application example for a complex simulation program consisting of different application codes. Furthermore, the variety of the applications codes and the high computational demands of the simulations require an efficient utilization of distributed computing platforms. In the following, the optimization process and its software components are described. The necessary development steps for building the overall simulation program are demonstrated and performance results for a distributed execution with different computing resources are shown.

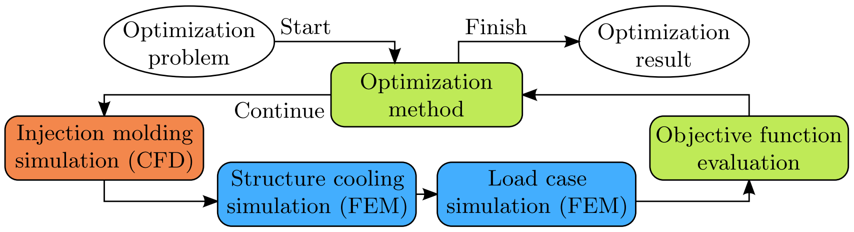
Optimization problem
Start
Finish
Optimization result
Optimization method
Continue
Injection molding simulation (CFD)
Objective function evaluation
Structure cooling simulation (FEM)
Load case simulation (FEM)

Figure 5: Overview of the optimization process.

4.1. Simulation and optimization of lightweight structures

The goal of the considered simulation program is the optimization of manufacturing parameters for lightweight structures. Figure 5 gives an overview of the optimization process. Each optimization run starts with an optimization problem that specifies the geometry of the structure, the objective function and constraints for the optimization, and parameters, for example, about the materials, the manufacturing process, and the operating load cases. The optimization method is implemented within a separate software component that executes an optimization loop. In each iteration, the optimization method selects specific values for the parameters to be optimized, starts the simulations, and evaluates the objective function to decide whether the optimization is finished or not.

The manufacturing by injection molding is simulated with a computational fluid dynamics (CFD) application. The application simulates the injection of molten plastic into a mold and determines the distribution of fillers, such as glass and carbon fibers, that are mixed in to improve the mechanical properties. We employ a customized open-source CFD application based on OpenFOAM, a C++ library implementing the finite volume method [1]. However, our general optimization process will also be capable of employing alternative simulations, for example, with specialized closed-source applications [2]. Input data of the simulation are the structure geometry, the material properties, and the parameters of the injection molding process. Simulation results are the fiber orientation and the temperature distribution within the structure.

The cooling of the structure and the operating load cases are simulated with a finite element method (FEM) application. The cooling leads to residual stresses within the material and to a shrinking of the structure. The load case simulation results in displacements and stresses of the structure. Both simulations are performed with an in-house adaptive 3D FEM application [3]. However, our general optimization process will also be capable of employing alternative simulations with commercial closed-source FEM applications, such as ANSYS or Abaqus. Input data of the simulation are the structure geometry, the material properties, and the load case to simulate. Additionally, the fiber orientation and the temperature distribution calculated by the CFD simulation are used. Simulation results are the displacement field and the resulting stresses, which will be used for evaluating the objective function of the optimization process.

4.2. Building the complex simulation program

Based on the programming model presented in Sect. 2 and on the SCDC library presented in Sect. 3, the complex simulation program is built in several development steps. The development starts with a given simulation process describing the basic tasks that have to be performed consecutively or concurrently. The granularity of the tasks has to be chosen such that each task will finally represent a separate software component. For the application example, the given simulation process is shown in Fig. 5.

Definition of the data flow between the components:. Each potential data exchange between two components has to be represented by an edge within the data flow graph. For the application example, the resulting data flow corresponds to the graph shown in Fig. 2 with the components L, A, B, and E representing the optimization method, the CFD application, the FEM application, and the objective function evaluation, respectively.

Specification of control structures for each component:. Within the application example, the optimization method is an active client component while all other components will be mainly passive server components that react only to requests. This corresponds to the pseudocode shown in Fig. 3, but with the while loop of the component L replaced by the loop of the optimization method.

Assignment of client-server roles for each data flow edge:. This step defines the control flow within the simulation program and corresponds to the selection of push- or pull-based implementations as described in Sect. 2.4. For the application example, the roles are chosen in the same way as for the example in Sect. 2.4. The optimization method pushes the input data to the CFD and FEM applications to start their computations as asynchronous jobs. The FEM application, the objective function evaluation, and the optimization method pull the results from their predecessors and thus are blocked until this data is received.

Implementation with the SCDC library:. The behavior derived for each component is implemented in Python using the SCDC library. The optimization method uses only client functions for accessing the datasets provided by other components. Each of the other three components is implemented as a service with a data provider of type "jobrun". This implementation requires less programming effort, because the simulation applications still use the same file-based data input and output as if they were executed manually. However, transferring the input and output data as well as executing the applications is performed automatically by the SCDC library. The library provides auxiliary functions for the efficient reading and writing of files and directories during a transfer. As the Python interface of the SCDC library is only a wrapper around its C interface, there are no significant performance differences expected. In future works, it is planned to redirect the file I/O of commercial closed-source applications by intercepting system calls or with a dedicated (e. g., FUSE-based) file system.

Figure 6 shows the Python code for implementing the component A using the SCDC library. In line 2, a jobrun data provider is set up for the base

```
1  import * from scdc
2  dp = dataprov_open("A", "jobrun", "do_cfd.sh", "<workdir>")
3  dataset_cmd(DATASET_NULL,
4    "scdc:///A/CONFIG put max_parallel_jobs 4", None, None)
5  np = nodeport_open("tcp")
6  nodeport_start(np, NODEPORT_START_ASYNC_UNTIL_CANCEL)
7  raw_input("Press <ENTER> to quit\n")
8  nodeport_stop(np)
9  nodeport_close(np)
10 dataprov_close(dp)
```

Figure 6: Python code for setting up and running the component A.

path "A". For each submitted job, the `do_cfd.sh` script is used to execute the CFD application within a subdirectory of `<workdir>`. Separate processes for executing the script are created to run several jobs in parallel. The maximum number of parallel jobs is set in line 3 by accessing the special dataset of the configuration parameters. The dataset `DATASET_NULL` represents a shortcut that omits the opening of a dataset by using the URI address from the start of the command string. In lines 5–6, access through network communication with TCP sockets is set up and the corresponding TCP server is started in a non-blocking way. Thus, it is necessary to keep the component running (line 7). Finally, the TCP server and the data provider are disabled (lines 8–10). The Python code will be running during the whole execution together with a separate thread created by the jobrun data provider inside the SCDC library. This thread dynamically launches the CFD application with the `do_cfd.sh` script whenever a new simulation job can be executed. Alternatively, it is also possible to implement the `do_cfd.sh` script as a Python function or to implement the component in C to omit the use of scripts.

Distributed execution:. Executing the complex simulation program on a distributed computing system requires to start the components once on their respective computing resources interactively by the user and to chose the corresponding URI addresses for accessing the datasets of the distributed components. The software components have to be deployed manually and all involved applications codes, such as the CFD application started by the `do_cfd.sh` script, have to be installed. To distribute the execution of the simulation applications among different computing resources, the components A and B are set up and started on several compute nodes. In this case, the component L can select the compute node for the execution of each simulation job through the URI address.

4.3. Performance results for local and distributed executions

We have implemented the application example as described in the previous subsection based on the SCDC library. The implementation is simplified in such a way, that both the optimization method and the objective function evaluation

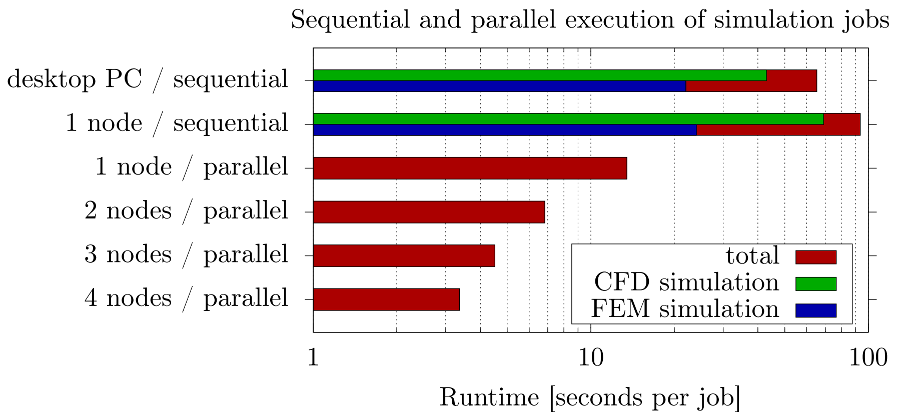

Figure 7: Runtime per simulation job for the local and distributed execution of CFD and FEM simulations on the desktop PC and up to four compute nodes.

are integrated in a single loop component L. This component creates a number of simulation jobs, executes these jobs on the available compute resources, and gathers the simulation results. Each job consists of an OpenFOAM CFD simulation performed by a component A and an adaptive 3D FEM simulation performed by a component B. Since the FEM simulation uses the results of the CFD simulation, the component B retrieves the data from the component A before it starts the CFD simulation. All simulation jobs are independent from each other and thus can be executed in parallel. This represents use cases where large numbers of independent simulations are performed with different parameters, for example, for parameter studies or complex optimization problems.

The loop component L represents an SCDC client that is executed on a desktop PC with a 4-core Intel Core i7-3770 processor with 3.40 GHz and 8 GiB main memory. The components A and B represent SCDC services that are executed either on the desktop PC or on up to four dedicated compute nodes. Each compute node has two 6-core Intel Xeon X5650 processors with 2.67 GHz and 12 GiB main memory. Executing the CFD simulation and the FEM simulation is provided by data providers of type jobrun that are available on the desktop PC as well as on all compute nodes. The desktop PC executes all simulations sequentially while each compute node executes up to six CFD simulations and six FEM simulations at the same time. Furthermore, the simulations were performed with their sequential program variants, thus using only one core per program execution. If only the desktop PC is used, then all components are implemented within a single program and all accesses between clients and services within the SCDC library are mapped to direct function calls. Otherwise, the accesses are performed through network communication with TCP sockets. Switching between a local or distributed execution does not involve additional programming efforts and is solely achieved by specifying different URI addresses for the target services executing the simulation jobs.

Figure 7 shows runtimes per job for the local and distributed execution of the

CFD and FEM simulations on the desktop PC and up to four compute nodes. A total number of four simulation jobs was used for the sequential executions and four jobs per available core were used for the parallel execution (i. e., 48 jobs for one compute node with 12 cores). For the sequential execution, the desktop PC achieves a smaller runtime than the compute node due to its higher processor speed. The major differences in runtime (i. e., about 26 seconds) are caused by CFD simulation while the differences for the FEM simulation are significantly smaller (i. e., about 2 seconds). Performing the simulations in parallel with one compute node decreases the runtime only by about a factor of seven. This can attributed to the fairly low number of simulation jobs per core and to the sharing of limited resources, such as memory and I/O bandwidths, by all cores. Furthermore, the dependencies between the CFD and FEM simulation jobs lead to significant waiting times until the first FEM simulation can be performed. Reducing these waiting times could be achieved with an improved job scheduling and will be considered within the next development steps towards a performance optimization. The usage of additional compute nodes leads to an almost perfect scaling in comparison to the parallel execution with one compute node, i. e. using four compute nodes decreases the runtime by about a factor of four. Potential waiting times due to the sequential starting of a large number of simulation jobs by the loop component L are reduced by using a round-robin scheme for the distribution of jobs to compute nodes.

5. Related work

The development of complex simulations that capture multiple physical processes is supported by a vast number of domain-specific and generic frameworks [4]. Environmental research is one of the most prominent areas for the coupling of simulation models, as it involves models from different disciplines, such as atmospheric sciences, hydrology, geology, and ecology. The Earth System Modeling Framework [5] represents a component-based coupling framework for earth system simulations composed of separate atmosphere and surface models. Building complex simulations with these frameworks is achieved by connecting components through fixed model-specific interfaces. These approaches lead to single monolithic applications and usually require substantial modifications to adapt existing application codes. The OASIS framework [6] supports the coupling of individual application codes with a programming interface that requires only minor modifications to existing application codes. A separate communication library is used to encapsulate the data exchange operations. The SCDC library shares the support for individual application codes and the encapsulation of data exchanges, but is independent from specific simulation models.

The MUSIC framework [7] introduces a standard API for data exchanges of neuronal network simulators to increase their interoperability and reusability. The C++-based programming framework distinguishes between a setup phase that specifies the mapping and exchange of data and a runtime phase that advances the simulation. Glue code is provided to perform the data exchanges with MPI. Thus, all MUSIC applications are based on MPI, have to be started at the same time, and are restricted to static data exchange pattern. In contrast, the SCDC library supports data exchanges through different communication methods, autonomously started applications, and dynamic data exchanges.

The Model Coupling Toolkit (MCT) [8] generalizes the concept of model coupling for applications from different areas. The Fortran-based programming framework provides application-specific data structures and operations implemented with MPI and, thus, has similar restrictions as the MUSIC framework. MCT and other frameworks described so far provide a large set of application-specific tools for common tasks, such as mesh or grid data management, data interpolation and transformation, domain decomposition, data storage, and I/O. However, the usage of these tools is often very invasive to existing application codes. In contrast, the SCDC library is less invasive to existing application codes as it is limited to their data exchanges and to wrapping them as services executed in their preferred software and hardware environment. Furthermore, the tools provided by the SCDC library target at application-independent tasks, such as job execution, data storage, and plain data transfers.

The Multiscale Coupling Library and Environment (MUSCLE 2) [9] is a software infrastructure for multiscale simulations. Within the Multiscale Modeling and Simulation Framework [10], MUSCLE 2 is used for the distributed execution of applications from areas, such as biomedical physics, nano materials, and hydrology. MUSCLE 2 consists of a java-based runtime environment with one global Simulation Manager and one or more Local Managers for starting instances of submodels and for performing data exchanges between submodels. The SCDC library also targets at using distributed resources, but omits the usage of a statically defined coupling model as well as the introduction of an entire runtime environment. Instead, it is designed to be a lightweight library approach that provides dynamic data exchanges between flexibly distributed components while maintaining their autonomous execution.

Coupling via an I/O infrastructure represents a low-level alternative to model coupling. Specialized communication libraries, such as PSMILe from the OASIS framework [6], the parallel coupler PALM [11], or the Typed Data Transfer (TDT) library [12] provide operations for data exchanges between parallel software components. Besides the differences described so far, our approach differs in several ways from the existing frameworks and libraries:

- Data exchanges within the presented approaches use a two-sided communication model, where operations, such as put/get or send/receive, have to match each other. Our approach uses a one-sided service-oriented model, where data exchanges are the input and output of operations invoked on a target process.

- None of the existing approaches supports the coupling through direct function calls. Data exchanges with PSMILe, MUSIC, and MCT are performed solely with MPI. PALM uses MPI and has support for IP-based communication, but only through a separate API and for the main purpose of coupling applications that use different MPI implementations. The TDT library supports MPI and TCP/IP, but uses different APIs to establish the connections and does not support relaying the communication between different connections.

MUSCLE 2 uses message-passing based on shared memory and TCP/IP and requires dedicated proxy daemons to connect distributed resources. In contrast, the SCDC library is specifically designed to utilize varying communication methods through a single interface and supports direct function calls, Unix Domain Sockets, TCP/IP, and MPI. Relaying communication is supported by specific data providers that bridge between any of these methods.

- The existing approaches use a static specification of the coupling through XML files (PSMILe and TDT), a graphical user interface (PALM), script files (MUSCLE 2), or a separate setup phase of the application (MUSIC). With the SCDC library, the setup of components and their coupling is performed at runtime and can be dynamically changed while the applications are executed.

- Several of the existing approaches support the integration of local files as data sinks or sources. In contrast, the SCDC library contains specific data providers for both file-system and database storage that can be integrated not only locally, but through any supported communication method.

- The existing approaches are application-specific in the way that they support the exchange and redistribution of specific data types and structures with a limited size. The SCDC library supports only plain data with a format identifier that can be freely chosen while special emphasis is given to the processing and exchange of data streams with unlimited size.

Only few service-oriented approaches exist for HPC applications. The Common Component Architecture (CCA) [13] is an effort to provide component-based software engineering for high-performance scientific computing. Besides the service-oriented approach, common properties of the SCDC library and CCA-compliant frameworks are the dynamic coupling of components at runtime, the interaction with local components through function calls, and the support for a seamless switch between local and remote components without additional programming efforts. However, a significant difference is how components interact with each other: CCA uses Remote Method Invocation where the method name and its parameters are fixed within the source code. With the SCDC library, a component and functionality to be used is specified through an URL-based address and a command string that can be changed at runtime. Furthermore, it was reported that CCA-compliant frameworks support either parallel or distributed computing, thus using both in one application would require additional efforts to couple different CCA frameworks [13]. In contrast, the SCDC library supports a variety of different communication methods as well as relay functionalities to connect parallel and distributed components.

The Application Hosting Environment (AHE) [14] is a middleware implementing the Software as a Service paradigm of cloud computing for HPC applications. AHE supports the development of complex simulations based on an integrated workflow engine and eases the transfer of input and output files for applications executed on distributed computing resources. Similar to the SCDC library, AHE uses an URI-based addressing of resources and aims at a flexible utilization of different computing resources. However, AHE is a complex java-based software infrastructure that requires to encapsulate the application codes and their file-based data exchanges. In contrast, the SCDC library is a lightweight programming library primarily designed to implement data exchanges through communication from within an application code.

The development of parallel and scientific applications with Python is supported by dedicated programming frameworks, such as Pyro [15] for Remote Method Invocation or Scoop [16] for distributed task-based parallel programming, that could also be used to ease the implementation of our application example from mechanical engineering. However, their usage is restricted to Python programs and data exchanges are performed by passing fixed-size parameters to function calls. In contrast, the SCDC library also connects programs with different programming languages and supports stream-oriented data exchanges with unlimited size. Additionally, there exist Python packages, such as SciPy [17], pyOpt [18], or OpenMDAO [19], dedicated to optimization problems. However, SciPy provides only optimization algorithms and pyOpt is solely based on a parallel execution with MPI. OpenMDOA is specialized for multidisciplinary design analysis and optimization (MDAO) and supports distributed computing only through user-defined remote allocation manager classes. In contrast, the SCDC library supports job executions through a predefined data provider and is neither limited to MDAO problems nor to the distributed execution of jobs.

6. Conclusion

In this article, we have discussed the requirements for building complex simulation programs in science and engineering. To support the development of such programs, a component-based client-server programming model was proposed which leads to a step-wise realization of the simulation components based on their data dependencies. A major goal was the flexibility of the resulting simulation programs, especially with regard to interchanging application codes and the hardware platforms for their execution. The implementation of the client-server components for distributed computing systems is supported by a software library providing the data exchange operations. The approach was demonstrated with a simulation program for the optimization of lightweight structures in mechanical engineering. We presented performance results to demonstrate the large performance differences of the data access methods supported by our library. This shows the advantages of supporting them individually. Further performance results demonstrated the distributed execution of simulation applications on several compute nodes. This justifies our efforts of supporting such distributed executions in an application-independent way. The given API descriptions and code examples provide important insights about the usability of our library.

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] H. Jasak, A. Jemcov, 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.

[2] D. Niedziela, J. Tröltzsch, A. Latz, L. Kroll, On the numerical simulation of injection molding processes with integrated textile fiber reinforcements, J. Thermoplastic Composite Materials 26 (1) (2013) 74–90.

[3] S. Beuchler, A. Meyer, M. Pester, SPC-PM3AdH v1.0 - Programmer's manual, Preprint SFB/393 01-08, TU-Chemnitz.

[4] D. Groen, S. Zasada, P. Coveney, Survey of multiscale and multiphysics applications and communities, Computing in Science & Engineering 16 (2) (2014) 34–43.

[5] C. Hill, C. DeLuca, V. Balaji, M. Suarez, A. da Silva, The architecture of the earth system modeling framework, Computing in Science & Engineering 6 (1) (2004) 18–28.

[6] R. Redler, S. Valcke, H. Ritzdorf, OASIS4 – A coupling software for next generation earth system modelling, Geoscientific Model Development 3 (1) (2010) 87–104.

[7] M. Djurfeldt, J. Hjorth, J. Eppler, N. Dudani, M. Helias, T. Potjans, U. Bhalla, M. Diesmann, J. Kotaleski, O. Ekeberg, Run-time interoperability between neuronal network simulators based on the music framework, Neuroinformatics 8 (1) (2010) 43–60.

[8] J. Larson, R. Jacob, E. Ong, The Model Coupling Toolkit: A new Fortran90 toolkit for building multiphysics parallel coupled models, Int. J. High Performance Computing Applications 19 (3) (2005) 277–292.

[9] J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, C. P.V., A. Hoekstra, Distributed multiscale computing with MUSCLE 2, the Multiscale Coupling Library and Environment, Journal of Computational Science 5 (5) (2014) 719–731.

[10] B. Chopard, J. Borgdorff, A. Hoekstra, A framework for multi-scale modelling, Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 372 (2021) (2014) 20130378.

[11] A. Piacentini, T. Morel, A. Thévenin, 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.

[12] C. Linstead, Typed Data Transfer (TDT) user's guide (2004).

[13] D. Bernholdt, B. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. Dahlgren, K. Damevski, W. Elwasif, T. Epperly, M. Govindaraju, D. Katz, J. Kohl, M. Krishnan, G. Kumfert, J. Larson, S. Lefantzi, M. Lewis, A. Malony, L. Mclnnes, J. Nieplocha, B. Norris, S. Parker, J. Ray, S. Shende, T. Windus, S. Zhou, A component architecture for high-performance scientific computing, Int. J. High Performance Computing Applications 20 (2) (2006) 163–202.

[14] S. Zasada, D. Chang, A. Haidar, P. Coveney, Flexible composition and execution of large scale applications on distributed e-infrastructures, Journal of Computational Science 5 (1) (2014) 51–62.

[15] Pyro – python remote objects, `pypi.python.org/pypi/Pyro4`.

[16] Y. Hold-Geoffroy, O. Gagnon, M. Parizeau, Once you SCOOP, no need to fork, in: Proc. of the Annual Conf. of the Extreme Science and Engineering Discovery Environment (XSEDE'14), ACM, 2014, pp. 1–8.

[17] SciPy: Open source scientific tools for Python, `www.scipy.org`.

[18] R. Perez, P. Jansen, J. Martins, pyOpt: a Python-based object-oriented framework for nonlinear constrained optimization, Structural and Multidisciplinary Optimization 45 (1) (2012) 101–118.

[19] C. Heath, J. Gray, OpenMDAO: Framework for flexible multidisciplinary design, analysis and optimization methods, in: Proc. of the 8th AIAA Multidisciplinary Design Optimization Specialist Conf., 2012.