Feilhauer and Sobotka *Journal of Cloud Computing: Advances, Systems and Applications* (2016) 5:20
DOI 10.1186/s13677-016-0070-z

Journal of Cloud Computing: Advances, Systems and Applications

RESEARCH Open Access

DEF - a programming language agnostic framework and execution environment for the parallel execution of library routines

Thomas Feilhauer* and Martin Sobotka

Abstract

There is high demand for library routines that can be included into arbitrary programs and executed in parallel in the Cloud. So our approach is to provide a framework that supports the parallelized execution of library routines, written in different programming languages, from any platform. Our Distributed Execution Framework (DEF) allows to (1) deploy arbitrary routines into a central library and (2) integrate these library routines at runtime into user programs in a way that allows the routines to be executed in parallel in the Cloud. The programming and runtime environment of the library routine is completely transparent to the user and the chosen programming and runtime environment. DEF provides client and library APIs with primitives like search_lib(), create_task(), submit_job() which are integrated into the user's program to access the DEF runtime. DEF allows the user to configure clusters in a public/private Cloud and automatically distributes the tasks for executing the library routines on the workers of the cluster.

Keywords: Parallel execution of library Routines, Cloud computing, Programming language independence

Introduction

Parallel execution and pushing large and time consuming compute-tasks into the Cloud becomes more and more important. Several research projects and providers of execution environments work on tools and frameworks to extend a specific programming language or Problem Solving Environment (PSE) with features to enable their product for parallel execution, e.g. parallel fortran [1], Unified-Parallel C (UPC) [2], MATLAB [3]. Cloud providers offer program language specific APIs for their Cloud environment to enable developers to integrate applications into the Cloud, e.g. Amazon Web Services (AWS)[1].

In our EnFiLo[2] project we were confronted with the situation that several analysts were working on different problems in the domains of energy, finance, and logistics, applying algorithms developed by themselves to complex optimization and simulation problems, each of them using their preferred programming and runtime environment. The analysts found out that a large number of algorithms developed by their colleagues could well be used for the problems they were working on, probably in just a slightly adapted, more abstract form, so that they could be applied to a different domain. And the analysts now wanted to find a way to reuse at least parts of the algorithms of their colleagues to make their lives easier and not always have to re-implement already existing code. So it is a central objective of the EnFiLo project to define and extract reusable parts of computationally complex algorithms in the form of library routines which can be executed in parallel and which can be used across domains. Though the resulting library routines were reusable and domain independent, the next problem was that the analysts were using different programming and runtime environments, which means that the reusable library routines could not directly be invoked from the runtime environments of the colleagues. Therefore a solution must be found to enable the parallel execution of library routines which can be written in an arbitrary programming language and which can be invoked from applications also written in an another programming language, without the application programmer needing to know the implementation details or programming language of the library routine used in his

*Correspondence: thomas.feilhauer@fhv.at
Josef Ressel Center for Applied Scientific Computing in Energy, Finance and Logistics, Fachhochschule Vorarlberg, University of Applied Sciences, Hochschulstr. 1, 6850 Dornbirn, Austria

application. This was the idea to develop the Distributed Execution Framework (DEF).

Therefore the central goal of the DEF is to develop a system that allows for parallelized execution of library routines, independent of programming languages or platforms. We found that, before the DEF, there were no environments that support the parallelized invocation of tasks across arbitrary execution environments. In this paper we describe the design of a flexible, easy-to-use framework and execution environment that allows the parallel execution of library routines across different runtime environments invoked from some arbitrary client.

The DEF allows to (1) deploy arbitrary routines into a central algorithm library and (2) integrate these library routines at runtime into user programs in a way that enables the routines to be executed in parallel in the Cloud. Figure 1 illustrates that users can integrate implementations of their algorithms into an algorithm library. These library routines typically represent reusable modules which are invoked several times to operate independently on different data sets within larger programs. The developer can choose an arbitrary programming language or PSE to implement the routine. In this sense, the library routines follow the Single Program Multiple Data (SPMD) model [4, 5]. Every library routine needs to have a signature description document in which the interface of the library routine is defined.

Based on the signature description document, any user can call the routine from any client program by applying it to the correct parameters and data. Typically, these library routines are called multiple times within a loop which is referred to as "loop parallelization" ([6], p. 122). Therefore these routines represent the computationally intensive and time consuming parts of the program, which predestines them to be executed in parallel on the worker nodes of a compute cluster.

An a priori analysis of the problem sets in the EnFiLo project revealed that the applications to be developed for our project can be reduced to calls of embarrassingly parallel computations, which means that the same routine is invoked in parallel on different independent input parameter sets, i.e. there is no communication required between the parallel computations ([6], p. 60) ([7], p. 121). The parameters (esp. large collections) for the library routines can be uploaded as shared resources before the library routines are invoked (call by reference) or they (esp. small base type parameters) can be directly attached to the routine call (call by value). The DEF then distributes the library calls among the worker nodes of a compute cluster. The results of the calls can synchronously or asynchronously be retrieved from the DEF and can be used for further computations.

To be able to provide flexible cluster structures we have set up the DEF on Cloud technologies. In this sense, the DEF could be viewed as a PaaS (Platform as a Service)[3] that enables the deployment and concurrent execution of library routines. To be installable on multiple platforms, the DEF runtime environment is implemented in Java. With the DEF we want to achieve the following benefits:

- High performance through parallelization
- High efficiency by utilizing potentially all available (local) resources
- High scalability by using elastic Cloud technology
- High user acceptance by providing a simple API for integrating the parallel invocations of library routines into user applications

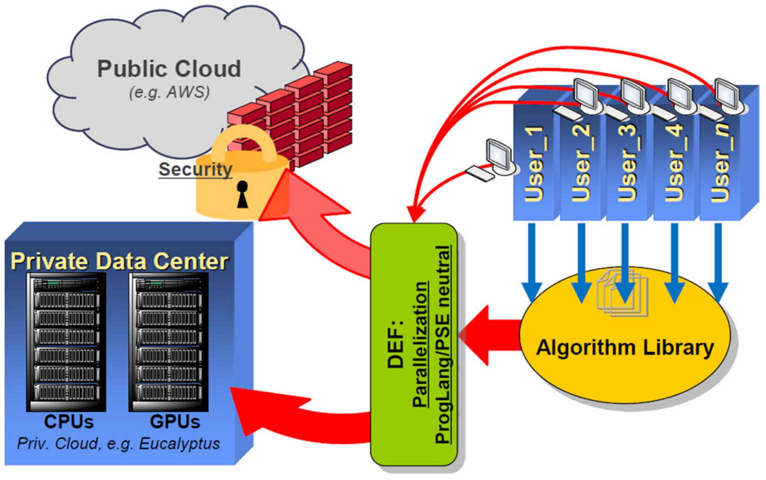

Fig. 1 DEF - General Overview

- High flexibility by offering a programming language independent library of efficient algorithm implementations that can be integrated into user applications
- Openness to different data formats regarding input and output parameters by using JSON as data exchange format
- The DEF components are deployable on all common desktop-server hardware/OS platforms
- Support users on searching for appropriate algorithms in the library
- Security by supporting private and public Cloud infrastructures on demand

The novelty in this approach lies in enabling parallelized invocations of library routines, completely independent of programming languages or platforms. This is achieved by decoupling the client program from the implementation of the library routine and by dynamically distributing the library routine invocations to available worker instances in the cloud. This provides for programming language and platform independence as well as elasticity for large scale computations. To give an example: With the presented approach, analysts will able to execute MATLAB code in parallel without using MATLAB Parallel Computing Toolbox, invoked from a client implemented in Python. In this example the DEF shows an approximately linear speedup by adding more worker nodes.

State of the art

A central issue of this project is to bring together techniques for (*A*) Parallel Execution of Procedures, (*B*) Platform and Programming Language Independent Invocation of Remote Procedures, and (*C*) Distributed Execution Environments. A superordinate topic for all remote/distributed execution techniques is security which is crucial for running real world applications in distributed environments, especially in the Cloud.

Parallel execution of procedures

The most common techniques applied for parallel execution of procedures are *shared memory* and *message passing* ([6], pp. 10) ([8], pp. 27).

Message passing

The message passing model focuses on exchanging messages between interacting parallel processes, while each participating process has its own address space ([6], pp. 10). Its most important representatives are the Message Passing Interface (MPI) [9] and Parallel Virtual Machine (PVM) ([8], pp. 29) of which MPI turned out to be more successful. MPI is based on a bidirectional interprocess communication (IPC). There are implementations of MPI for a large number of programming languages. MPI is still widely used for parallelizing programs which makes it something like a de facto standard for message passing applications. On the other side, all parts of the parallel application must be implemented in a specific programming language for which MPI is available and the MPI constructs with their dependencies tend to complicate the code which makes MPI a complex framework, especially for users with a non-programming background, like many data analysts.

Shared memory

The shared memory model is based on a Parallel RAM (PRAM) [10]. The processes in a shared memory system share a single address space. They communicate by reading and writing shared variables ([6], pp. 10). The most widely used implementation for shared memory systems are threads (POSIX, NT) and directives like OpenMP [11] ([8], pp. 29).

For a shared-address-space, the memory must be accessible to all processors. In a distributed system we have autonomous hosts each having its own local memory. But a shared memory system can be emulated on a distributed system using distributed shared memory (DSM) [12].

Summary

According to Mattson ([6], pp. 13) neither MPI nor OpenMP provides an optimal solution for hybrid architectures that aggregate multiprocessor worker instances, each with multiple processors/cores and shared memory, to larger systems with separate address spaces for each worker. He also states that programs that make use of MPI instructions can be difficult to implement, because the programmer must take care for distributing the data and is responsible for configuring the messages required for IPC ([6], p. 13).

An important issue for our project is that the application developers implementing the programs that make use of the DEF and the offered library routines do not want to get involved with complicating IPC and the corresponding message configuration. For our problem sets of embarrassingly parallel computations with different independent input parameter sets, we have routines that do not communicate with concurrently running processes and they write the results to separate result entries that do not conflict with the result entries of the other concurrent processes. Therefore, for this scenario, we can expect a minimum of conflicts between the concurrent processes, which means that we do not need to consider any process communication during the execution of a library routine. For the same reason, we did not build our DEF on models like Partitioned Global Address Space (PGAS) [13].

Platform and programming language independent invocation of remote procedures

If we want our compute cluster to be independent of a specific underlying platform, it is the simplest way to make use of virtualization for being able to run a specific platform on top of all available worker hosts, independent of their underlying hardware configurations and installed operating systems. A common technology that combines virtualization with distributed systems is Cloud computing.

Cloud computing

Cloud computing allows flexible access to a seemingly unlimited number of compute resources which can be flexibly configured with little effort and which can be measured so that the usage could be charged "pay-per-use" [14]. Because of these features, the Cloud is a good source for compute resources to be used for HPC [15, 16].

There are different approaches of how to apply High Performance Computing (HPC) on the Cloud, depending on the parallel computation model [17, 18]. This means that there are examples for Cloud based MPI implementations, e.g. [19, 20]. And there are also few publications that deal with distributed shared memory solutions in a Cloud environment. One is restricted to an implementation for UPC [21] and as an alternative there is an implementation for Cloud task parallelization in .NET [22]. We have discarded the process communication approaches, hence these examples will not be taken into further consideration for our solution. Still Cloud computing in general is a strong technology that enables an elastic acquisition of compute resources which can be flexibly configured, e.g. with different platforms and runtime environments.

Programming language independent remote invocation

There had been a few approaches for programming language independent execution environments of which Common Object Request Broker Architecture (CORBA) and Web services are the most popular ones which are still in use.

CORBA [23] enables the invocation of remote methods in an object oriented fashion across platforms and programming languages [24, 25]. CORBA proofed to be very stable [26], but there are also a number of drawbacks that limit the applicability for our purpose [27]. One issue is security; CORBA messages are sent unencrypted between the participating hosts and to grant remote access to CORBA objects, the firewall must open up TCP-ports for each service. This cannot be tolerated for public Cloud environments. Another issue is that CORBA does not support MATLAB and Octave, which is essential for our project. Because of these shortcomings, CORBA cannot be an adequate environment for our purposes, even though there are extensions of CORBA that would allow the parallel execution of tasks [28, 29].

Web services is an underlying technology of Cloud computing that enables the programming language independent call of procedures on compute nodes in the Cloud. They provide synchronous and asynchronous calls of remote procedures. There are two types of Web services: RESTful [30] and SOAP Web services [31]. In Pautasso et al. [32] give a deep insight into the differences of both technologies. While SOAP Web services with their mandatory WSDL[4] interface definition support both static calls and dynamic invocation, RESTful Web services only allow dynamic invocation without the possibility for generating client side stubs. Web services are not object oriented, but if the service is implemented stateless and side effect free, then it would have no negative effect on calling independent routines from a library.

Both CORBA and Web services allow static calls using client stubs and dynamic invocations. In our case, dynamic invocations provide more flexibility when calling routines that can be added to the library on the fly. Additionally, no stubs have to be generated and compiled before the client can invoke a library routine. Therefore Web services provide a good foundation for invoking routines in a distributed Cloud environment, but both Web services and CORBA offer no support for the parallel execution of routines. With the strong integration of Web services into Cloud technologies, we decided to setup the communication our DEF environment on RESTful Web service techniques.

Distributed execution environments

Distributed execution environments ease the development of distributed applications that can be executed on typically heterogeneous computing resources connected via network. A central characteristic of a distributed execution environment is distribution transparency [33].

There are distributed execution environments that allow to implement distributed applications for specific programming languages, e.g. DJO [34]. Others concentrate on the distribution of workflows, for which specific workflow modeling languages are defined, e.g. [35, 36]. These solutions are programming language specific and provide no solution for the parallel execution of tasks.

Apache Hadoop[5] is a framework for the parallel distributed processing of data. With its MapReduce implementation it is focusing on "Big Data" applications and supports different programming languages only through its streaming API. Apache Spark[6] is only available for Java, Scala, Python, and R. The Big Data frameworks typically execute the analysis routines where the data is, because of the huge amount of data that is too expensive to move. This is not an issue for the type of problems that we are dealing with in our project. Established cluster frameworks, like Globus[7] and SGE[8], require knowledge about the runtime environment and specially prepared programs along with the data for execution. Our framework should be able to utilize any available cluster resources by simply using the framework's client APIs from an arbitrary programming language.

Another interesting approach is Berkeley Open Infrastructure for Network Computing (BOINC)[9]. BOINC is an open source framework for distributed applications. Originally implemented for the SETI@home project, it is designed for volunteer computing (scientists), creating a Virtual Campus Supercomputing Center (universities), and desktop Grid computing (companies). It supports a large number of operating systems and is running on several hundred-thousand hosts. BOINC provides a C++ API, but there are also wrappers for Fortran, Java, and Python, while PSEs like MATLAB and Octave are not supported. Parallel invocations of procedures are possible using MPI, but only on a single host. So a synchronization of concurrent tasks is not possible across multiple hosts.

Ludescher et al. [37] provide a code execution framework that allows to upload code for a set of programming languages and PSEs into the Cloud and invoke this code from within some client program written in some other programming language. Their framework uses Web services for invoking remote procedures in the Cloud and the underlying concept is independent of programming languages or PSEs. This approach does not support library routines in our sense and it lacks the potential to execute code in parallel, but the idea behind it could be a base on which we can build up our DEF system.

Security

Some of the research performed in the EnFiLo project is highly sensitive: banks, insurance companies and utilities who handle highly confidential customer data are partners in this project. Clearly none of the partners will accept an insecure environment, which is especially relevant for computations in the Public Cloud. Hence, security is an eminent factor. We therefore need to provide solutions for securely transferring confidential data across public networks and trustworthy executing calculations based on these data in off-premise environments.

First of all, data privacy has to be ensured, i.e., only authorized personnel is granted access to the data. The communication channels can be encrypted by proven techniques like Transport Layer Security (TLS) using HTTPS. But the data also has to be secured while being stored in the DEF and in the cluster. We therefore need an authentication mechanism for accessing this data. Authentication is also required to invoke library routines on the cluster. It is a central functionality of the DEF to distribute the tasks of a user's program across the available worker nodes. All these tasks must be executed on behalf of the authenticated user that started the client-side program. Therefore a single sign-on (SSO) system with ticket delegation can be used to manage authentication and authorization in the DEF. Kerberos[10] provides a promising solution for these issues. Kerberos can be adapted to work within a Cloud infrastructure[11] and it is able to work in mixed environments, support dynamically generated DNS names, and work with tasks running for several days[12]. To secure the infrastructure it is necessary to implement a two-way authentication mechanism, which means that both client and server must be able to verify the counterpart's identity. We can handle this issue by injecting the security credentials into the dynamically created virtual machine instances during its boot sequence. The whole security concept is described in detail in Ludescher et al. [38].

There is a range of security risks that apply especially to Cloud solutions [39–43][13]. A central issue with regard to processing sensitive data in the Cloud is trust with respect to the Cloud provider: The data and the software owned by the customer must be transferred to a Cloud resource in a Cloud environment, which is completely under the control of the Cloud provider and its administrators, as described in Wan et al. [44]. Therefore inspection or modification of data or software by the Cloud provider cannot be ruled out. In Descher et al. [45] a Secure Virtual Machine (SVM) for the Cloud is described that defines a complete chain of trust for a virtual machine image that can be installed on a Cloud resource. The SVM consists of an encrypted partition and a special boot system and is based on the Trusted Platform Module (TPM), ensuring a trusted boot sequence and system startup, and includes a Xen aware implementation of SELinux[14] by Barham et al. [46]. This guarantees that only authorized virtual machines with verified images can be used as workers. Using this SVM, the Cloud provider could still inhibit the execution of the Cloud image, but it can no longer view, inspect, or manipulate data and software without being detected by the customer who owns the data and software. Another solution for a trusted IaaS Cloud environment is introduced by J. Seol et al. in [47].

Architecture

The DEF features a modular architecture. All functionalities are encapsulated and provided via a facade. This guarantees a high level of abstraction and allows the exchange of tools, frameworks and techniques used in the prototype. Figure 2 gives a schematic overview on the top-level architectural components of the DEF.

DEF Module: Central module that manages all clusters, the library routines, handles the client requests, and

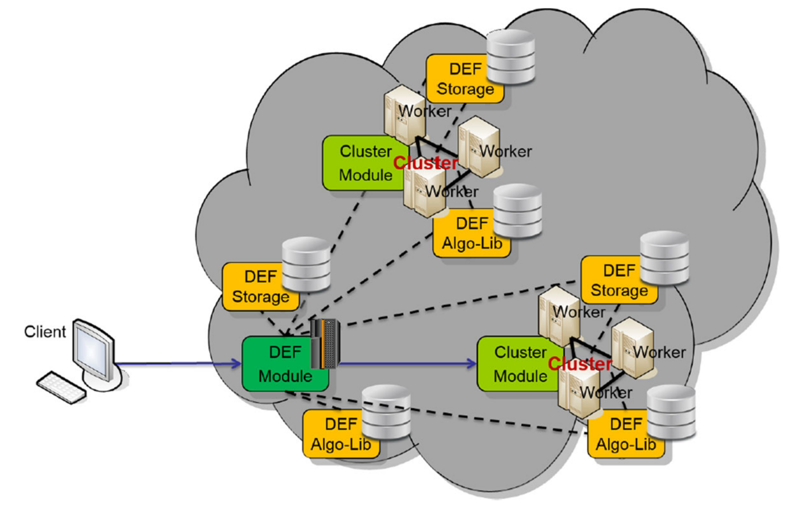

Fig. 2 DEF - Top-Level Architectural Components

redirects the *job*-related client requests to the corresponding cluster modules; it provides its own DEF Storage for holding persistent *job* results; it offers a (RESTful) Web service interface for all DEF functionalities; the DEF module is typically installed within the private Cloud.

Client: Runs on basically any platform that allows HTTP-calls; communicates via Client-API with the DEF Module through RESTful Web services.

DEF Algo-Lib: Network storage that provides executables, dependencies, and signature documents of the library routines; all worker instances of a cluster have read-only access.

DEF Storage: Network data store, provides storage space for library routine calls, including shared resources for parameters, results, and logs.

Worker: Worker instances are used to execute the single *tasks*, representing a library routine call, invoked by the client; every worker is assigned to exactly one cluster; a worker is based on a virtual machine image providing the different runtime environments for executing the library routines; the worker functionality is implemented in a Worker Module which allows the invocation of the library routines and, via a Worker-API, allows the library routine to interact with the DEF.

Cluster: A Cluster comprises a Cluster Module and a set of worker instances that operate on a coherent problem (*program*) and therefore share the same DEF Storage. All workers of a cluster are located in the same region of a (public) Cloud; a cluster is managed by a Cluster Module and has its own instance of a DEF Algo-Lib.

Cluster Module: Central DEF component of the cluster; it manages the worker instances and the *jobs* that are to be executed within the cluster; the Cluster Module provides a scheduler and load balancing facilities to distribute the *tasks* among the cluster's worker instances; it is started as a separate process within the cluster.

For the first prototype we decided to use NFS as a distributed storage technique for DEF Algo-Lib and DEF Storage, because it is integrated into the Linux operating systems (used by the worker instances) and well supported by all programming languages and PSEs used for the algorithm library.

To allow an easy and elastic configuration of the cluster, our workers run within a Cloud environment [48]. The following Cloud infrastructure environments are used for the first prototype:

- Eucalyptus for private Cloud [15]
- Amazon Web Services (AWS) for public Cloud [16]

StarCluster[17] helps to configure and maintain the cluster in our Cloud environment.

Overview of the DEF components

DEF Module and Cluster Module encapsulate smaller components, which have their corresponding counterparts on the DEF Module and Cluster Module side, respectively, see Fig. 3. The DEF Module components have a superordinate function and usually delegate to the corresponding Cluster Module components:

DEF WS: Web service interface, single point of access for all clients using the DEF; distributes the incoming requests to the corresponding components.

Computation Manager/Computation Controller: Initialize the DEF Storage and its internal structure;

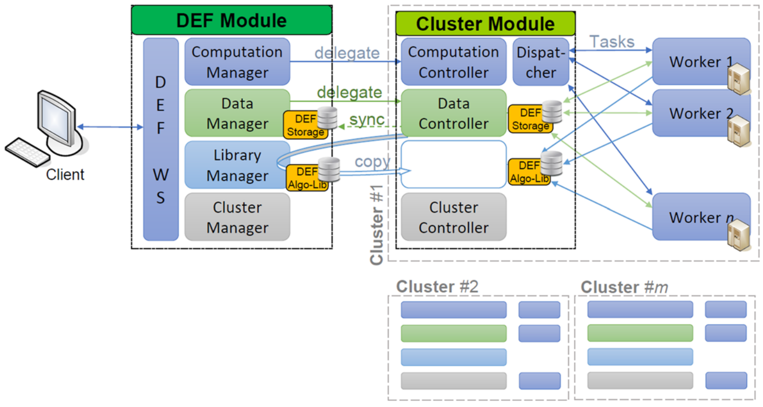

Fig. 3 DEF - Components

manage the compute *jobs*, their shared resources, results, and status changes.

Dispatcher: Manages a scheduling queue for distributing the *tasks* among the available worker instances; in the current prototype implemented by JPPF[18].

Data Manager / Data Controller: Manage access to (shared) resources and results from DEF Storage and control access to it; synchronize data between the DEF Storage of the DEF Module and the DEF Storage of the Cluster Module.

Library Manager: Manages the DEF Algo-Lib; provides a unique reference ID for each library routine; enables the deployment of new routines; allows a search for library routines; provides a version of the DEF Algo-Lib to each cluster.

Cluster Manager: Manages clusters; creates clusters according to the preferences of the user; configures the cluster with a DEF Algo-Lib and DEF Storage; aggregates resource consumption of the managed clusters; enables the deletion of clusters.

Cluster Controller: Creates and manages worker instances; monitors running worker instances; aggregates resource consumption of the managed workers; enables the deletion of workers.

The workers run as separate (virtual) machines in the Cloud. To provide a maximum of flexibility and to allow all available library routines to be executable within the cluster, all workers are homogeneous with regard to the installed software. This means that the runtime environments for all library routines need to be installed at all workers. The workers also provide a Worker Module with an API that allows the DEF to invoke the library routines, which is currently performed by a sys-call. One requirement in the project was to keep the usage of the workers simple and cheap, even for a large number of workers. This means that the workers must be free of license costs. We therefore chose Linux as operating system and all installed runtime environments are also free of charge. Even the MATLAB runtime[19] can be installed free of license costs on each worker, which allows to also execute (compiled) library routines written in MATLAB to be executed in parallel on the DEF workers.

API description and invocation of library functions

After a first analysis of the problem sets to be solved by the different project partners, we found out that they typically follow a common pattern. The problem is formalized by some programmer in an arbitrary programming language/ PSE within a *program*. The *program* usually consists of segments with dependencies amongst each other, which means that these segments have to be executed in a certain sequence. These segments are denominated as *jobs* and it must be ensured that a predecessor *job* has to be completed before the successor *job* can be started. Within the *jobs*, we often find sections, embraced by loops, that repeat a sequence of operations independently of the other loop iterations, following the “loop parallelism pattern” ([6], p. 122). These loop iterations can be executed in parallel and are called *tasks*. The client application developer has to identify the *tasks* following the “task parallelism pattern” ([6], p. 64) ([8], p. 86). A *job* in our client applications may consist of a set of independent *tasks* that can potentially be executed in parallel on the worker instances of the DEF. The *tasks* are represented by library routine calls from DEF Algo-Lib. After all *tasks* have finished, the results of the *tasks* can either be completely downloaded to the client or a reduce step can be appended at the cluster, in which a simple cross-*task* operation can be applied to the *tasks*’ results for consolidation. Figure 4 depicts the relationships between the central operational entities of the DEF described above.

This outline of *program*, *job*, *task* determines the structure of the client application and strongly influences

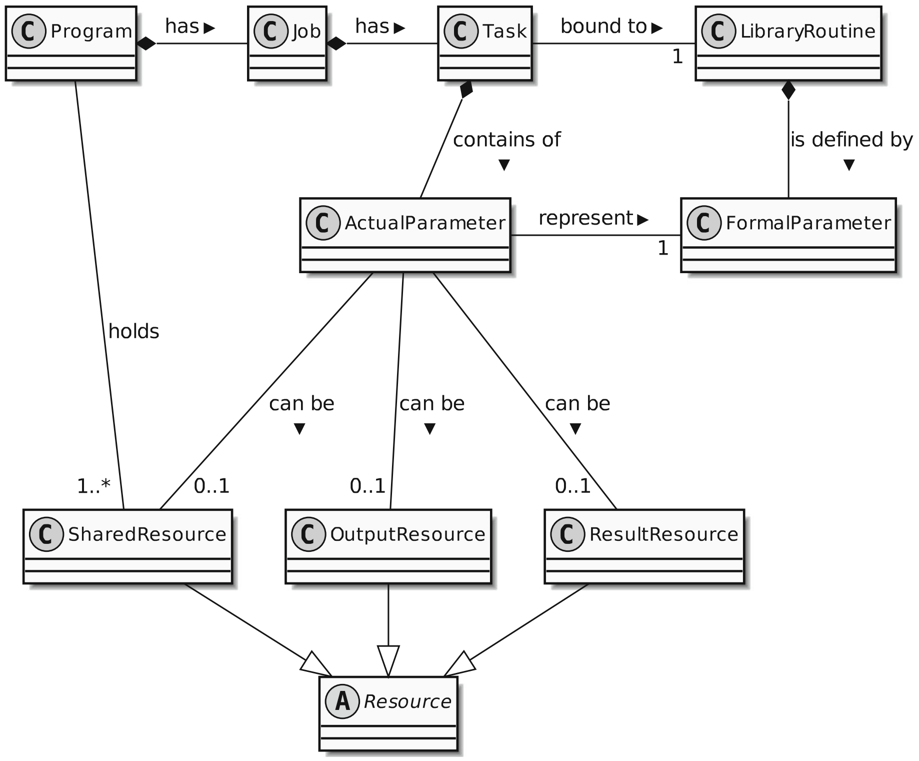

Fig. 4 Relationships between the central operational entities

the internal structures provided by the DEF Storage. Each *program*, *job*, and *task* has a unique ID which is implemented as a UUID. The DEF Storage builds a hierarchy on these UUIDs in which the *program*-UUID contains all resources of the *program*, including its *jobs*, the *job*-UUID contains all resources of the corresponding *job*, including its *tasks*, and the *task*-UUID contains all resources of the corresponding *task*, including its results and log data. The DEF Storage is currently implemented as a file system and therefore results in a directory structure like the following:

```
./ (DEF Storage Root)
|
+- <PUUID1>/ (Program)
| |
| +- log.json (Program Log)
| |
| +- <UUID>.json (Shared Resource)
| |
| +- <UUID>.json (Shared Resource)
| ...
| +- <JUUID1>/ (Job)
| | |
| | +- log.json (Job Log)
| | |
| | +- result.json (Job Result)
| | |
| | +- <TUUID1>/ (Task)
| | | |
| | | +- log.json (Task Log)
| | | |
| | | '- result.json (Task Result)
| | |
| | +- <TUUID2>/ (Task)
| | | |
| | | +- log.json (Task Log)
| | | |
| | | '- result.json (Task Result)
| ...
```

The DEF enables a client to execute a *program* consisting of a sequence of *jobs* that are executed sequentially within a compute cluster, while the *jobs* may consist of *tasks* being executed in parallel at the workers of the cluster. The functionality offered by the DEF is provided through Web services and can be invoked through a Client-API with the following (pseudo code) calls:

init_client(): Initialize client with a reference to the DEF

in: URL reference to the DEF Module
out: Reference to the DEF

init_cluster(): Initialize cluster

in: Cluster configuration parameters, e.g. number of workers, Cloud specification
out: Cluster-ID

search_lib(): Search for specific library routine

in: Name of the library routine to be searched for
out: List of library routine-IDs for which the corresponding routines match the input name together with their signatures

create_shared_resource(): Create a reference to a (empty) shared resource parameter in the cluster

in: *Program*-ID
out: Reference to a shared resource in the cluster

upload_data(): Upload shared resource parameter

in: Reference to a shared resource, data to be uploaded into the shared resource
out: Boolean

create_program(): Setup *program*

in: Cluster-ID on which the *tasks* of the client's *program* are to be executed
out: *Program*-ID

create_job(): Setup *job*

in: *Program*-ID
out: *Job*-ID

create_task(): Setup *task* - prepares the dynamic invocation of the specified routine

in: *Job*-ID, library routine-ID, its corresponding parameters (references to shared resources or explicit values)
out: *task*-ID

submit_job(): Submit *job* (asynchronous)

in: *Job*-ID
out: *Job* state

sync_submit_job(): Submit *job* (synchronous)

in: *Job*-ID
out: *Job* state

get_job_state(): Request *job* state

in: *Job*-ID
out: *Job* state

get_job_results(): Download result

in: *Job*-ID
out: JSON data structure (collection of all task results)

cleanup_program(): Cleanup resources

in: *Program*-ID
out: Boolean

delete_cluster(): Terminate cluster resources

in: Cluster-ID
out: Boolean

The structure of the client application could exemplarily look like the following Java-like pseudo code in which choose_best_result() represents an arbitrary client side procedure that reduces the set of results returned by the *tasks* to a single result (the real client can be implemented in any programming language/PSE that allows RESTful Web service calls):

```
DEFclient dc = init_client();
Cluster c = dc.init_cluster();
Program p = dc.create_program(c);
Routine[] routines = dc.search_lib("mylib");
...
Resource r1 = p.create_shared_resource();
Boolean b1 = r1.upload_data(input1);
Resource result = p.create_shared_resource();
...
while (there are any sequential jobs) do
    ...
    Job j = p.create_job();
    for any task to be executed in parallel do
        Task t = j.create_task(routines[0], ..., r1, result);
    end for
    j.submit_job();
    ...
    String js = j.get_job_state();
    if (js == "success") then
        result = choose_best_result(j.get_job_results());
    end if
    ...
end while
...
Boolean b2 = p.cleanup_program();
...
Boolean b3 = dc.delete_cluster(c);
...
```

While the DEF Module is typically running within the private network of an institute or a company, the Cluster Module and the workers are arranged in a compute cluster that is set up in either a private or a public Cloud. The first prototype of the DEF does not provide a fully implemented Cluster Module and Cluster Manager. It assumes that the cluster is already configured - we are using StarCluster for that which supports the setup of clusters in AWS and Eucalyptus environments. JPPF then automatically distributes the *tasks* among the workers of the cluster. Currently every DEF Module only supports a single cluster which means that only a single program can be executed at any point in time.

The next version of the DEF will allow to flexibly define and manage clusters meeting the demands of the *program* that is executed on the cluster. This means that the init_cluster() function has to instruct the Cluster Manager to configure a cluster of a specific size in one of the available Cloud environments on demand. The client side programmer can specify in which Cloud environment its *program* will be executed. The different Cloud environments may have specific, proprietary APIs and VM image formats. Hence, for setting up the cluster, the DEF will need to support the Cloud APIs and VM image formats for the Cloud environments used by the project partners. This means that the VM images for the Cloud Controller and for the worker must be adapted for the different Cloud infrastructure environments (e.g. AWS, Azure[20], OpenStack[21], Eucalyptus). The initialization of the cluster, implemented in init_cluster(), will be based on the following scenario:

1. The Cluster Manager starts a virtual machine (based on a specific VM image) for the Cluster Module (with DEF Storage and DEF Algo-Lib), using the specific Cloud API.
2. The Cluster Manager takes control over the Cluster Module and registers the new cluster (establish a contract).
3. The Cluster Controller starts virtual machines (based on specific VM images) for the workers and determines their IP addresses, using the specific Cloud API.
4. The Cluster Module takes control of the workers, using the DEF Worker-API which allows the Dispatcher to invoke library routines and control their execution. This enables a trusted communication between the DEF and the workers, while the parameter exchange at the workers is handled by the Library Routine-API.
5. The construction of the cluster is completed, the contracts between the participants are established.

This extension allows the DEF to manage several *programs* executed in different clusters at any point in time and in different Cloud environments. The Cloud APIs of the Cloud infrastructure environments allow to dynamically adapt the number of machine instances. Therefore the DEF could be extended by additional Client-API functions to dynamically increase or decrease the number of worker nodes in the cluster at runtime. The user can then accelerate or slow down the execution of a *job* within a *program* if that makes sense for him.

Figure 5 shows the processing of a *job* (initiated by a client) which consists of the following steps:

1. Client submits a *job*
2. *Tasks* of the *job* get scheduled: *Job* status set to "scheduled"
3. *Tasks* of the *job* are executed on the workers: *job* status set to "run"
4. Execution of the *tasks* is completed: *Job* status set to "success", results & logs are available
5. Client requests the results

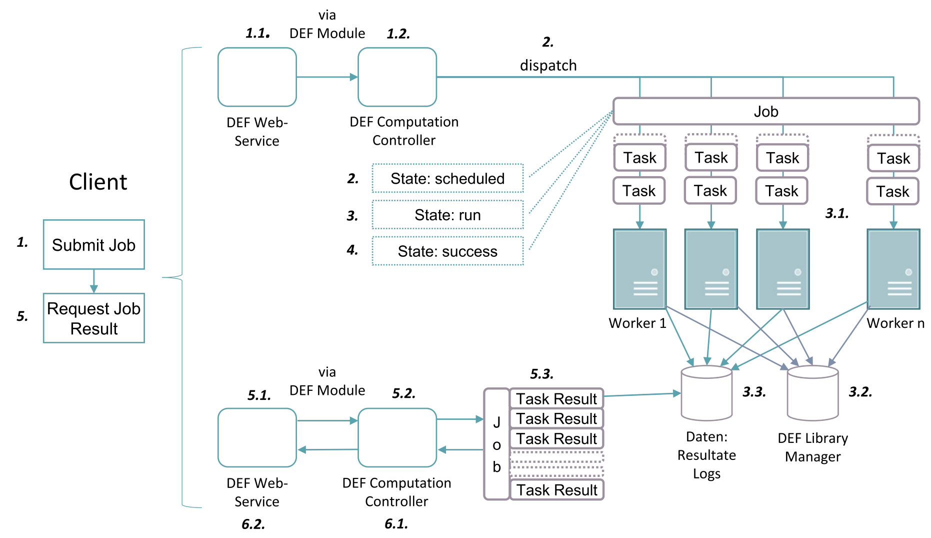

Fig. 5 DEF - Workflow for submitting a *job*

6. The *job* result (collection of *task* results) is downloaded to the client

The *job* results can afterwards be evaluated by the client and used for further processing in succeeding *jobs*.

The worker module maps a dynamic DEF library routine invocation to a call of a specific library routine deployed in one of the supported runtime environments. The library routine needs to interact with the DEF to access the resources managed by the DEF which the library routine needs to execute. These resources could be shared resources, results, logs, or nested calls of other library routines. The access to these DEF functionalities is handled by a Library Routine-API which is available in different programming languages/PSEs and offers the following procedures:

get_in_parameter(): Access a specific input parameter from the DEF Storage, allows call-by-reference (call-by-value parameters are accessed directly)

in: Index (according to signature document) that refers to one of the input parameters
out: Data

set_result(): Saves the result of the routine call to the *task* result section of the DEF Storage

in: Result data
out: Boolean

log(): write a message to the log resource of the *task* at a specific log level

in: Log level, message test
out: Boolean

The procedures get_library() and exec_library_routine() support additional functionalities for library routines to invoke library routines.

Figure 6 gives a complete overview of the central functionalities for executing a client *program* on the DEF and it documents which actions are initiated at the DEF Module/DEF Cluster Module and the DEF Workers by which Client-API call.

To deploy a library routine to the DEF, the library routine must be converted to a package which includes the specific routine (in an executable or parseable format, according to the programming language or PSE), its dependencies and a signature document. The signature document provides information about the execution environment, the input and output parameters, a unique name of the library routine, a version number, and a natural language description of the routine.

Results

The current prototype supports the complete workflows for both library routine developers and library routine users: New library routines can be deployed into the DEF Algo-Lib using a tool that generates the signature document and uploads the routine; the DEF Algo-Lib can be searched for library routines (see Client-API: search_lib()), the library routine's signature description can be downloaded via Web service API and the library routines can be invoked.

In the currently implemented first prototype, the DEF system only spans a single cluster. This means, a Cluster Manager is not required and the *tasks* of the Computation Manager and the Data Manager are taken by the Computation Controller and the Data Controller, respectively. This does not restrict the significance of the prototype in terms of proof of concept, as the current DEF prototype can serve several clients and multiple *jobs* at the same time. In a next step, we will implement the missing components and provide a fully functional DEF, which supports the management of multiple clusters.

The Client-API and the Library Routine-API currently support the following programming languages/PSEs:

- Java 8,
- Python 3,
- C# (Mono),
- Octave,
- MATLAB.

The client can invoke library routines in a synchronous and asynchronous version if this is supported by the programming language / PSE in which it is implemented. DEF Library routines can call other DEF library routines or can invoke themselves recursively; due to limitations of the JPPF scheduling, this repetitive call of library routines is restricted to only one level. The library routines currently included in the DEF Algo-Lib are mainly optimization and simulation algorithms for problems from the energy, finance, and logistics domains, as they are required for the project.

First tests for stability and load distribution were performed with multiple worker instances on a local blade center and on Amazon AWS. The DEF and its worker instances are running stably and the communication between the components is reliable.

We found out that the load balancing based on JPPF is static and therefore does not always work as desired [49]. The available cluster resources represented by the workers are not always completely used by the scheduler and it turns out that sometimes free workers are not served with *tasks* that still need to be executed. This is an important performance issue and another reason for us to replace JPPF and its scheduler in future versions of the DEF.

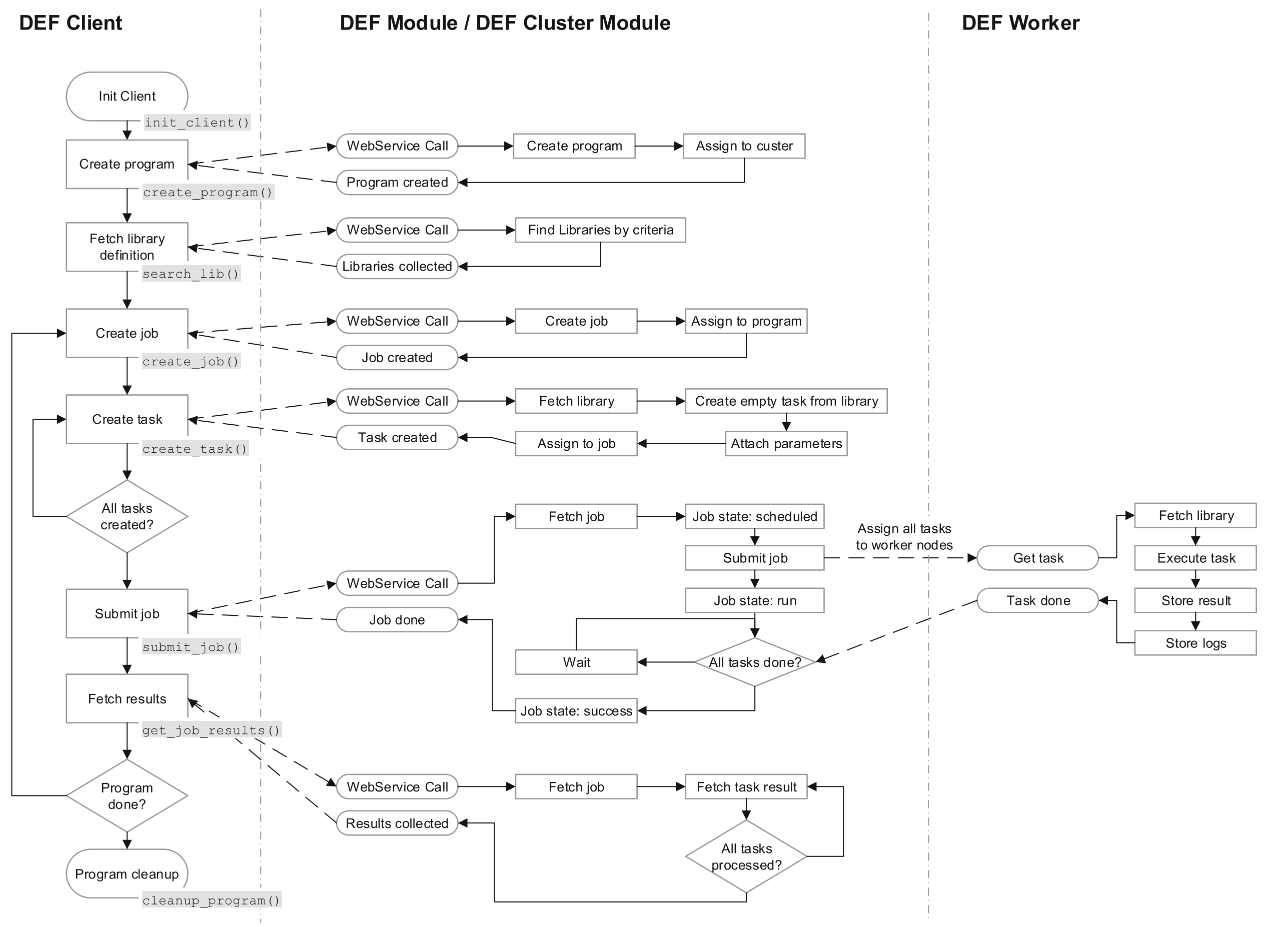

Fig. 6 Complete *program* workflow documenting the steps for submitting a *job*, together with the corresponding functionalities of the modules

To measure the performance and scalability of the DEF System, a simple library routine to calculate π according to the following function ([6], p. 129)

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

was implemented. This integral can be solved in a numerical way. The advantage of using the above function within a test routine for the DEF is that the computation can easily be expanded for a higher accuracy of π and it can also be split up into independent portions of nearly any desired size. The portions can then be easily distributed among any number of nodes for which the DEF should be tested. Another advantage is that the execution time doesn't get too long, even for a large number of *tasks*.

The serial calculation with 10^{11} iterations and a step size of $1/10^{11}$ serves as reference value. To calculate π in parallel, the geometric decomposition pattern is used to split the iterations up to 100 identical parts ([6], p. 78) ([7], p. 100). Figure 7 shows how DEF scales with 1, 2, 4, 6, and 8 workers. Every worker uses a single

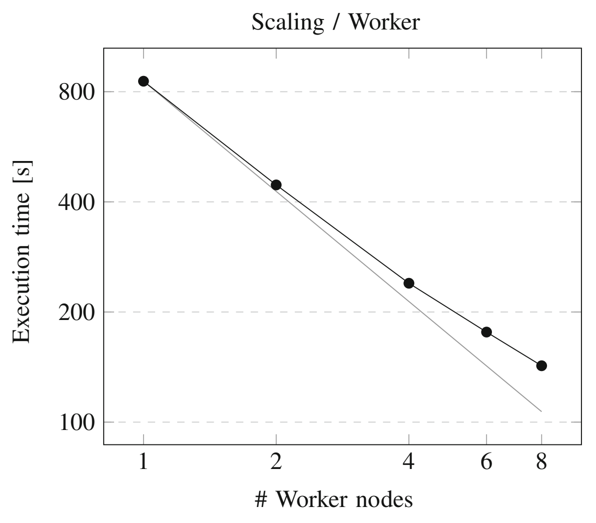

Fig. 7 Scaling of DEF *tasks*

CPU and one working thread. As a reference, the computation is also carried out sequentially on a local host (without DEF) resulting in a mean execution time of 813 s.

The gap between sequential execution on a local host and parallel execution with one worker instance on the DEF is about 40 seconds. This gap is caused by

- the communication between client and DEF,
- internal communication and distribution within the DEF,
- initiating a new process for each *task*, and
- the reduce step (calculating the arithmetic mean) on the client side

and it is not really relevant for long running *tasks*. Figure 7 also illustrates that the DEF system scales well with an increasing number of workers: 2 workers solves the problem 1.9 faster than 1 worker, 4 workers act 1.8 times faster than 2 workers and 8 worker act 1.7 times faster than 4 workers.

The parameters for the test routine are small scalar values and therefore provided on a "call by value" base. Large parameters that are provided on a "call by reference" base are loaded from / stored to the DEF Storage, which takes significantly longer to access than "call by value". On the other hand, each parameter is accessed only once during *task* execution and therefore this overhead can be neglected for *tasks* with a long execution time. This means, the results gained from the execution of the test routine can be transferred to long running *tasks* executing routines from our algorithm library.

We finally want to optimize the overall execution time of the problem on a given number of workers. Every *task* causes some fixed overhead for initialization and invocation. So if the execution time for a *task* is reduced, the ratio between the overall execution time and the number of *tasks* deteriorates. This means that we would have an optimal ratio of execution time and number of *tasks* if we could provide every worker with a single (equivalent or uniform) *task*. On the other hand, the client application developer cannot always split up the problem into a fixed number of *tasks* so that the number of *tasks* is identical to the number of workers or a small multiple of it. In many other cases a DEF user may not know how many worker instances are available when his *job* will be executed. Another issue is that the worker instances are not necessarily homogeneous and therefore require different execution times for identical *tasks*. In these cases, the number of *tasks* cannot be evenly distributed among the workers and this means that some workers still need to work on a *task* while other workers have already completed their *task* and run idle. We analyzed the π-calculation problem on an 8-worker cluster to be split up into 10, 50, 100, 500, and 1000 uniform *tasks* respectively. For our example scenario it turned out that the setup with 100 *tasks* performed best, as shown in Fig. 8. This relationship of course varies with the number of worker instances and we found that the scheduling algorithms at JPPF are optimized for large numbers of *tasks*. But this result confirms Ian Foster's statement that there should be at least an order of magnitude more *tasks* than worker instances [50] and it also shows that the overhead for using the DEF is not very high compared to the performance gain that can be achieved through the parallelization enabled by it.

For the sake of completeness we have compared the DEF prototype with an implementation of MPI on the π-calculation problem above. In both cases the calculation is running 10^{11} iterations on 4 physical nodes with each 8 CPU cores. While we can see from Fig. 7 that the DEF's performance shows a significant deviation from the ideal inversely proportional behavior for an increasing number of worker nodes, even with 8 worker nodes. In comparison, Fig. 9 shows that MPI accomplishes a nearly perfect linear behavior with an increasing number of processes, even for 32 processes. It is obvious that MPI scales better than the DEF, because it is optimized on communication and reduced overhead for starting processes. This gets explicit by comparing the overhead for executing a computation on a single host without parallelization compared to starting it within the MPI runtime environment on a remote host. Based on our π-calculation scenario we found that while MPI takes 200ms to start an execution directly on the cluster nodes, DEF needs about 40s to start a remote execution of some task from a remote client over the network. The reason for the delay at the

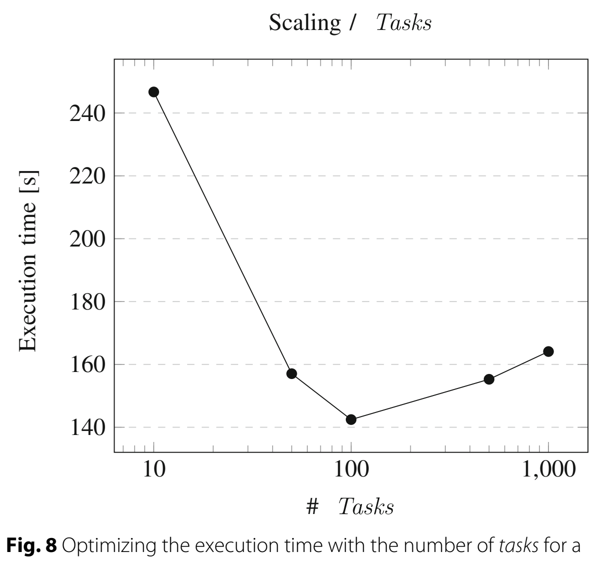

Fig. 8 Optimizing the execution time with the number of *tasks* for a given number of workers. # Worker nodes: 8

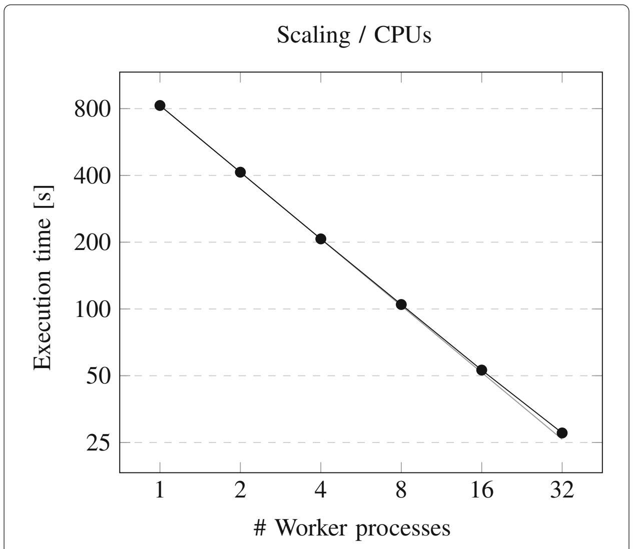

Fig. 9 Scaling of MPI tasks

DEF lies first of all in communicating the Client-API commands between client and DEF via the network, but also in initializing the dynamic directory structures in the DEF Storage and in an inefficient communication between the scheduler (JPPF) and the worker. In the upcoming version of the DEF we will reduce the overhead caused by creating the DEF Storage structures and by the scheduler which will also improve the scaling behavior of the DEF. On the other hand, there will always be some delay between the execution of a DEF program and an MPI program which has to be paid for the conveniences provided by the DEF: The MPI offers no programming language independence for the different parts of the implementation and the number of worker processes must be defined before the execution is started and should match the number of available CPU cores for optimal performance.

Conclusion

The DEF follows a completely new approach which decouples the client side implementation of some program from the implementation of the server side which consists of a set of library routines. At runtime, the DEF distributes the library routine calls within a Cloud cluster. An arbitrary client program can use the DEF to coordinate the parallel execution of the library routines to be invoked. It is especially the programming language independence of the client side code from the invoked routines that distinguishes the DEF from other parallelization frameworks for clusters.

The first prototype of the DEF has proofed that our concept is feasible. All intended features mentioned in Section Introduction could be achieved. We can now easily initiate parallel invocations of library routines in a Cloud cluster, set up by a client program written in a programming language different from the programming language in which the library routine was developed in. This means, for example, that a library routine implemented in MATLAB can be invoked as often as necessary on the worker nodes of a cluster, started by a program implemented in C#. The execution of the MATLAB routine does not even require the installation of the MATLAB Parallel Toolbox - the installation of the freely available MATLAB runtime on every worker is enough to execute the MATLAB routine in parallel on the cluster. The user can therefore save the costs of the licenses for the MATLAB Parallel Toolbox which otherwise would be required for every single worker node.

The overhead for starting a larger number of long running *tasks* in parallel is acceptable and the system has been tested with several different library routines in different Cloud environments. The current version of the DEF was implemented as a proof of concept, based on simple technologies, which means that there is also some potential for improvements in the details of its implementation. However, these upcoming enhancements are restricted to security and optimizing the runtime behavior of the complete system by adding additional features like a worker-side reduce step to the DEF or coming up with a solution for fast data provisioning for which there are established techniques around. So the innovative part of the DEF development has been done in providing the architectural prototype, while the next steps in making it a production ready system are more or less restricted to applying well-known technical solutions.

Future work

For reasons of simplicity, the *tasks* on the workers are currently invoked by sys-calls. The corresponding initialization of the new process and the following process communication with the different runtime environments is relatively inefficient. It is planned to provide a runtime-pool of initialized PSE and programming language runtime environments that are instantiated on the workers at startup and wait for incoming *task* requests. The *tasks* can then be executed as threads. This will bring a high gain of efficiency, especially for short running *tasks*, as the overhead for starting the *tasks* will be significantly reduced.

In the first version of the DEF prototype the resources (parameters, results, logs) used by the library algorithms are stored on a file system mounted to the workers via NFS. For the next version of the DEF we are looking for solutions that allow faster access to the resources. Promising candidates are distributed (in-memory) key-value stores which are currently evolving in multiple implementations.

We also plan to optimize the up-/down-load of large parameters from the clients to the DEF which is currently handled by Web service. For the productive operation of the DEF we need more reliable and faster solutions similar to GridFTP [51].

Another objective for optimization is the scheduling of the *tasks*. The first prototype of the DEF is using the JPPF which provides its own scheduler that can be configured to some certain extend. Nevertheless we soon found out that the JPPF scheduler does not satisfy all our needs; it especially does not support dynamic load balancing. We plan to evaluate flexible scheduling methods that allow a dynamic allocation of *tasks* to workers driven by their current load. This could be done by learning from previous executions of the algorithm or by simulation. The new scheduler must be extendable for load balancing capabilities.

In discussions with our project partners we found that the relevant result of a *job* often is not the complete set of the results of its *tasks*, but probably only the "best" *task* result or a simple transformation over a set of *task* results. Often the result the user is interested in can be determined by a simple operation on the set of *task* outputs resulting from a *job*. Such an operation can be viewed as a reduction step ([7], p. 90). Currently this reduce step needs to be executed as a separate *job* to be executed on some worker or at the client after the last *task* of the current *job* has finished. Therefore we plan to extend the execution of a *job* by a sequence of flexible reduce-steps after its *tasks* have finished. In these reduce-steps the results of the *tasks* can be consolidated by a cross-*task* operation right at the worker locations without having to start a separate "reduce" *job* after the preceding *job*.

Another performance improvement for specific *programs* would be the support of a master/worker paradigm ([6], p. 122) in which an arbitrary *program* can be deployed and executed within a DEF cluster as a "master" which can directly invoke *tasks* on the workers of its cluster and therefore circumvents the DEF Module for executing *tasks*.

To be able to easily monitor performance measurements, we plan to implement an automatic monitoring of execution time, CPU, and memory usage for *tasks* and *jobs* with the upcoming release of the DEF. The next version of the DEF should also fully implement the planned architecture by splitting up the components to a DEF Module and a Cluster Module and developing the Cluster Controller. This will allow the dynamic creation of clusters and the dedicated addressing of specific clusters (in a private or public Cloud) by *programs* executed by the users. Finally we plan to introduce checkpointing to the DEF to make it fault tolerant for long running *tasks*.

According to the requirements of our partners in the EnFiLo project, existing Cloud security solutions will be evaluated and those most suited for our problem setting will be applied to the DEF. As pointed out in section1 Security, the following security techniques will be applied:

- TLS for an encrypted communication between client $\leftrightarrow$ DEF Module & DEF Module $\leftrightarrow$ Cluster Module, ensuring privacy.
- Ticketing for authentication and authorization with single sign-on.
- a Secure Virtual Machine based on the TPM for Trust.
- only workers instantiated and controlled by the Cluster Controller will be used for computations so that the *tasks* will not be compromised.
- the data will be removed from the cluster immediately after the program has terminated.

The details will be worked out in the course of the project.

Endnotes

[1] http://aws.amazon.com/tools/

[2] http://www.enfilo.at/

[3] http://dx.doi.org/10.6028/NIST.SP.800-145

[4] https://www.w3.org/TR/wsdl

[5] http://hadoop.apache.org/

[6] https://spark.apache.org/

[7] https://www.globus.org/

[8] https://arc.liv.ac.uk/trac/SGE

[9] http://boinc.berkeley.edu/

[10] http://web.mit.edu/kerberos/

[11] http://www.oasis-open.org/committees/download.php/38245/Kerberos-Cloud-use-cases-11june2010.pdf

[12] http://www.kerberos.org/software/mixenvkerberos.pdf

[13] https://www.enisa.europa.eu/publications/cloud-computing-risk-assessment/at_download/fullReport

[14] http://selinuxproject.org/

[15] http://open.eucalyptus.com

[16] http://aws.amazon.com

[17] http://star.mit.edu/cluster/

[18] http://www.jppf.org/

[19] http://www.mathworks.com/products/compiler/mcr/

[20] https://azure.microsoft.com

[21] https://www.openstack.org/

Acknowledgements
This work is partially sponsored by the Christian Doppler Gesellschaft under the "Josef Ressel-Zentrum für angewandtes wissenschaftliches Rechnen in Energie, Finanzwirtschaft und Logistik". The entire research team contributed to the discussions that led to this paper and provided the environment in which the ideas could be implemented and evaluated. We thank all reviewers, whose comments and suggestions helped to improve this paper.

Authors' contributions
Thomas Feilhauer (TF) and Martin Sobotka (MS) have all contributed to the DEF concept. They designed the paper structure and gave their feedbacks to all its versions. TF has designed the architecture of the described system and MS was responsible for implementation and testing. TF is responsible for the DEF work package in the EnFiLo project, in the context of which the solution presented in the paper was developed. Both authors approved the final manuscript.

Competing interests
The work that led to this publication has partly been funded by the governmental organization Christian Doppler Gesellschaft. The authors declare that they have no competing interests.

Received: 14 July 2016 Accepted: 24 November 2016
Published online: 22 December 2016

References

1. Karp AH, Babb RG, et al (1988) A comparison of 12 parallel fortran dialects. Softw IEEE 5(5):52–67
2. El-Ghazawi T, Carlson W, Sterling T, Yelick K (2005) UPC: Distributed Shared Memory Programming, Vol. 40. John Wiley & Sons, Hoboken
3. Sharma G, Martin J (2009) Matlab: a language for parallel computing. Int J Parallel Prog 37(1):3–36
4. Darema F, George DA, Norton VA, Pfister GF (1988) A single-program-multiple-data computational model for epex/fortran. Parallel Comput 7(1):11–24. doi:10.1016/0167-8191(88)90094-4
5. Darema F (2001) The SPMD Model: Past, Present and Future. In: Cotronis Y, Dongarra J (eds). Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting Santorini/Thera, Greece, September 23–26, 2001 Proceedings. Springer, Berlin. pp 1–1. doi:10.1007/3-540-45417-9_1
6. Mattson TG, Sanders BA, Massingill BL (2004) Patterns for Parallel Programming. Pearson Education, Boston
7. McCool MD, Robison AD, Reinders J (2012) Structured Parallel Programming: Patterns for Efficient Computation. Elsevier, Waltham
8. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to Parallel Computing. Pearson Education, Boston
9. Dongarra J, Walker D, Lusk E, Knighten B, Snir M, Geist A, Otto S, Hempel R, Lusk E, Gropp W, et al (1994) Special issue-mpi-a message-passing interface standard. Int J Supercomputer Appl High Perform Comput 8(3-4):165
10. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the Tenth Annual ACM Symposium on Theory of Computing. ACM. pp 114–118
11. Dagum L, Enon R (1998) Openmp: an industry standard api for shared-memory programming. Comput Sci Eng IEEE 5(1):46–55
12. Nitzberg B, Lo V (1991) Distributed shared memory: A survey of issues and algorithms. Distributed Shared Memory-Concepts and Systems. IEEE, New York. pp 42–50
13. Yelick K, Bonachea D, Chen WY, Colella P, Datta K, Duell J, Graham SL, Hargrove P, Hilfinger P, Husbands P, et al (2007) Productivity and performance using partitioned global address space languages. In: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation. ACM. pp 24–32
14. Badger L, Grance T, Patt-Corner R, Voas J (2012) Cloud Computing Synopsis and Recommendations. National Institute of Standards and Technology (NIST) Special Publication 800–146. US Department of Commerce. Available online at: doi:10.6028/NIST.SP.800-146
15. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, et al (2010) A view of cloud computing. Commun ACM 53(4):50–58
16. Pellerin D, Ballantyne D, Boeglin A (2015) An introduction to high performance computing on aws. Amazon Whitepaper
17. Vecchiola C, Pandey S, Buyya R (2009) High-performance cloud computing: A view of scientific applications. In: Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium On. IEEE, New York. pp 4–16
18. Ekanayake J, Fox G (2009) High performance parallel computing with clouds and cloud technologies. In: Cloud Computing. Springer, Berlin, Heidelberg. pp 20–38
19. Raveendran A, Bicer T, Agrawal G (2011) A framework for elastic execution of existing mpi programs. In: Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium On. IEEE, New York. pp 940–947
20. Agarwal D, Karamati S, Puri S, Prasad SK (2014) Towards an mpi-like framework for the azure cloud platform. In: Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium On. IEEE, New York. pp 176–185
21. Anbar A, Narayana VK, El-Ghazawi T (2012) Distributed shared memory programming in the cloud. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012). CCGRID '12. IEEE Computer Society, Washington. pp 707–708. doi:10.1109/CCGrid.2012.48
22. Bläser L (2015) Task parallelization as a service, HSR University of Applied Sciences Rapperswil, Switzerland
23. Group OM (2012) Common object request broker architecture (corba) specification, version 3.3, part 1: Corba interfaces. OMG Formal Document Number: formal/2012-11-12. Needham
24. Yang Z, Duddy K (1996) Corba: a platform for distributed object computing. SIGOPS Oper Syst Rev 30(2):4–31
25. Vinoski S (1993) Distributed object computing with corba. C++ Report 5(6):32–38
26. Maffeis S, Schmidt DC (1997) Constructing reliable distributed communication systems with corba. IEEE Commun Mag 35(2):56–60. doi:10.1109/35.565656
27. Henning M (2006) The rise and fall of corba. Queue 4(5):28–34. doi:10.1145/1142031.1142044
28. René C, Priol T Mpi code encapsulating using parallel corba object. Cluster Comput 3(4):255–263. doi: 10.1023/A:1019096607706
29. Wang L (2008) Implementation and performance evaluation of the parallel corba application on computational grids. Adv Eng Softw 39(3):211–218. doi: 10.1016/j.advengsoft.2007.02.001
30. Fielding RT (2000) Architectural styles and the design of network-based software architectures. PhD thesis. University of California, Irvine
31. Seely S (2001) SOAP: Cross Platform Web Service Development Using XML. Prentice Hall PTR, Upper Saddle River
32. Pautasso C, Zimmermann O, Leymann F (2008) Restful web services vs. "big" web services: Making the right architectural decision. In: Proceedings of the 17th International Conference on World Wide Web. WWW '08. ACM, New York. pp 805–814. doi:10.1145/1367497.1367606, http://doi.acm.org/10.1145/1367497.1367606
33. Coulouris GF, Dollimore J, Kindberg T (2005) Distributed systems: concepts and design. Pearson Education, Boston
34. Erdogan N, Selcuk YE, Sahingoz O (2004) A distributed execution environment for shared java objects. Inf Softw Technol 46(7):445–455. doi:10.1016/j.infsof.2003.09.017
35. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, Foster I (2011) Swift: A language for distributed parallel scripting. Parallel Comput 37(9):633–652. doi:10.1016/j.parco.2011.05.005
36. Wang J, Altintas I, Berkley C, Gilbert L, Jones MB (2008) A high-level distributed execution framework for scientific workflows. In: eScience, 2008. eScience'08. IEEE Fourth International Conference On. IEEE, New York. pp 634–639
37. Ludescher T, Feilhauer T, Brezany P (2013) Cloud-Based Code Execution Framework for scientific problem solving environments. J Cloud Comput Adv Syst Appl 2(1):11
38. Ludescher T, Feilhauer T, Brezany P (2012) Security concept and implementation for a cloud based e-Science infrastructure. In: Availability, Reliability and Security (ARES), 2012 Seventh International Conference on. IEEE, New York. pp 280–285
39. Arjun U, Vinay S (2016) A short review on data security and privacy issues in cloud computing. In: Current Trends in Advanced Computing (ICCTAC), IEEE International Conference on. IEEE, New York. pp 1–5

40. Kazim M, Zhu SY (2015) A survey on top security threats in cloud computing. Int J Adv Comput Sci Appl (IJACSA) 6(3):109–113
41. Sen J (2014) Security and privacy issues in cloud computing. In: Martinez AR, Lopez RM, Garcia FP (eds). Architectures and Protocols for Secure Information Technology Infrastructures. pp 1–45. Hershey, PA: IGI Global. doi:10.4018/978-1-4666-4514-1.ch001
42. Saravanakumar C, Arun C (2014) Survey on interoperability, security, trust, privacy standardization of cloud computing. In: Contemporary Computing and Informatics (IC3I), 2014 International Conference on. IEEE, New York. pp 977–982
43. Puthal D, Sahoo BPS, Mishra S, Swain S (2015) Cloud computing features, issues, and challenges: a big picture. In: Computational Intelligence and Networks (CINE), 2015 International Conference on. IEEE, New York. pp 116–123
44. Wan X, Xiao Z, Ren Y (2012) Building Trust into Cloud Computing Using Virtualization of TPM, 2012 Fourth International Conference on Multimedia Information Networking and Security, Nanjing. IEEE, New York. pp 59–63. doi:10.1109/MINES.2012.82
45. Descher M, Masser P, Feilhauer T, Tjoa AM, Huemer D (2009) Retaining data control to the client in infrastructure clouds. In: In Proceedings of the Fourth International Conference on Availability, Reliability and Security (ARES 2009). IEEE, New York. pp 9–16
46. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebar R, Pratt I, Warfield A (2003) Xen and the art of virtualization. In: ACM SIGOPS Operating Systems Review. ACM, New York Vol. 37. pp 164–177
47. Seol J, Jin S, Lee D, Huh J, Maeng S (2016) A trusted iaas environment with hardware security module. IEEE Trans Serv Comput 9(3):343–356. doi:10.1109/TSC.2015.2392099
48. Von Laszewski G, Diaz J, Wang F, Fox GC (2012) Comparison of multiple cloud frameworks. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference On. IEEE, New York. pp 734–741
49. Patel DK, Tripathy D, Tripathy CR (2016) Survey of load balancing techniques for grid. J Netw Comput Appl 65:103–119. doi:10.1016/j.jnca.2016.02.012
50. Foster I (1995) Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley Longman Publishing Co., Inc., Boston
51. Allcock W, Bresnahan J, Kettimuthu R, Link M, Dumitrescu C, Raicu I, Foster I (2005) The globus striped gridftp framework and server. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing. SC '05. IEEE Computer Society, Washington. p 54. doi:10.1109/SC.2005.72