ExaQUte

Exascale Quantification of Uncertainties for Technology and Science Simulation

D4.4 API and runtime (complete with documentation and basic unit testing) for IO employing fast local storage

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	BSC
Deliverable Type:	Other
Dissemination Level:	Public
Related WP & Task:	WP 4, Task 4.5
Status:	Final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **800898**

Authoring

Prepared by:				
Authors	Partner	Modified Page/Sections	Version	Comments
Rosa M Badia	BSC	Sections 1, 4 & Exec. Summary	V0.1,V0.4	
Jorge Ejarque	BSC	Sections 2.1-2.3.1, 2.4, 3.1 & 3.2.2	V0.2	
Stanislav Böhm	IT4I	Sections 2.3.2 & 3.2.2	V0.3	
Rosa M Badia	BSC	Sections 1, 4 & Exec. Summary	V1.0	

Change Log

Versions	Modified Page/Sections	Comments
V0.1	First document version	
V0.2	Storage and MPI API and PyCOMPSs contribution	
V0.3	HyperLoom and Quake contribution	
V0.4	Introduction, conclusion and Executive summary	
V1.0	Final Review	

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Jorge Ejarque	BSC	27 November 2019	OK
WP leader	Rosa M. Badia	BSC	28 November 2019	OK
Coordinator	Riccardo Rossi	CIMNE	29 November 2019	OK

Executive summary

This deliverable presents the activities performed on the ExaQUte project task 4.5 Development of interface to fast local storage. The activities have been focused in two aspects: reduction of the storage space used by applications and design and implementation of an interface that optimizes the use of fast local storage by MPI simulations involved in the project applications.

In the first case, for one of the environments involved in the project (PyCOMPSs) the default behavior is to keep all intermediate files until the end of the execution, in case these files are reused later by any additional task. In the case of the other environment (HyperLoom), all files are deleted by default. To unify these two behaviours, the calls "delete_object" and "detele_file"have been added to the API and a flag "keep" that can be set to true to keep the files and objects that maybe needed later on. We are reporting results on the optimization of the storage needed by a small case of the project application that reduces the storage needed from 25GB to 350MB.

The second focus has been on the definition of an interface that enables the optimization of the use of local storage disk. This optimization focuses on MPI simulations that may be executed across multiple nodes. The added annotation enables to define access patters of the processes in the MPI simulations, with the objective of giving hints to the runtime of where to allocate the different MPI processes and reduce the data transfers, as well as the storage usage.

Table of contents

List of Figures

Nomenclature / Acronym list

Acronym	Meaning
API	Application Programming Interface
ExaQUte	EXAscale Quantification of Uncertainties for Technology and Science Simulation
DAG	Directed Acyclic Graph
GPFS	General Parallel File System
HPC	High Performance Computing
IN	Parameter of a function that is not modified
INOUT	Parameter of a function that is modified during the call
OpenMP	Open Multi Processing
MLMC	Multi-Level MonteCarlo
MPI	Message Passing Interface
PyCOMPSs	Python binding for COMPS Superscalar
SLURM	Simple Linux Utility for Resource Management
SSD	Solid-State Drive
OUU	Optimization Under Uncertainties
UQ	Uncertainty Quantification

1 Introduction

The goal of Task 4.5 in ExaQUte is to provide a simple C and Python API that allows to take advantage of the fast local storage available in the current HPC system nodes for IO operations. While most of these systems come with a global file system, many vendors are nowadays providing new local persistent storage devices. For example, while MareNostrum 4 is equipped with a 14PB GPFS shared file system, each node also comes with with a 240GB SSD local disk. However, most of the current applications are not getting benefit of these local devices. Our goal is to provide a useful interface that enables the runtime to benefit from these local persistent storages. Another aspect considered in this deliverable is how the task-based runtimes can optimize the applications by means of taking into account the data used in the different MPI simulations that compose the project applications. We take into account that while a given MPI simulations may be accessing a large amount of data, this data is not globally needed by each of the processes that compose the MPI simulation, which at the same time can be running in different HPC system nodes. For this reason, an extension to the ExaQUte API is proposed that enables to describe the access pattern of the different processes. This access pattern is then used by the runtime to reduce the data transfers and storage needed by each MPI simulation.

The deliverable is structured as follows: In section 2 we describe the storage abstractions used to optimize the storage space used. A simple API is proposed, and the runtime functionality implemented is described. The section concludes with the description obtained with an experimentation using this API against the default case. In section 3 we present the extensions to the API and runtime to optimize the use of local storage and to reduce the data transfers. Finally section 4 concludes the deliverable.

2 Storage API and runtime implementation

This section presents how the ExaQUte API and runtime environments manages the application data storage and how they work to optimize the required storage and the usage of the different storage technologies available in current supercomputer such as shared parallel file systems (such as GPFS or Lustre), SSD local disk, and Non-Volatile memories.

2.1 Storage abstraction

ExaQUte proposes the usage of a task-based programming model for implementing applications to enable Uncertainty Quantification (UQ) and Optimization Under Uncertainties (OUU) in complex engineering problems.

The programming model consist of annotating the Python methods as tasks. A Python method is candidate to be a task if it has a certain amount of computation in order to benefit from a remote execution, it is executed several times and can run concurrent other tasks. To define tasks, developers have to indicate the directionality of the method parameters that indicate how the task accesses the data. (i.e. direction IN when the task only reads the data, INOUT if the task read and writes the data or OUT if it generates or completely overwrites the data).

This model has the benefit of providing a level of abstraction that most of developers are familiar with. The complexity of implementing an application is quite similar to implementing its sequential version since the programming model provides the view of a single address space memory and a shared file system where objects and files are created using standard clauses, and hides the complexity of the parallelism and data management since it does not define an specific API for defining explicit parallelism or data transfers.

The application parallelism and the required data management is inferred at runtime based on the task definition and the available storage and computing infrastucture. For instance, data directionality enable the runtime to detect data dependencies between tasks, while depending on the infrastructure where the data is stored and computed, the runtime will decide when to copy, move or remove data. The runtime mechanism to achieve these functionalities will be explained in the next subsections.

2.2 API calls for optimizing the storage

As introduced in previous section, data management actions is inferred by the runtime system based on the task definition. However there are some situations where task annotations are not sufficient to infer an optimal data management decision. To overcome these situations, ExaQUte provides a set of mechanisms to optimize the use of the storage space.

2.2.1 Deleting object

The first situation to optimize is the deletion of objects. In Python, objects are automatically managed by the Python interpreter, so the interpreter will decide when an object is not used anymore by the application based on the references. However, depending on the object management done at runtime, the object can be referenced in some runtime data structures which prevent the Python's garbage collection to free the resources. For that reason, we introduced the API call *delete_object*. This API method enables the user to indicate to the runtime that this object is not going to be used in a task or synchronized anymore. So, the runtime can take the appropriate data management decisions to release storage space without generating any side-effect.

2.2.2 Deleting Files

Files also require an special treatment when they are removed inside the application code. When a file is used or generated by a task, this file can be replicated (to enable further parallelism) or generated in the locations where tasks was executed (to exploit locality). In this situation performing an standard file deletion is not safe. The delete call can fail because the file is not in the final destination path or it can only remove the replica located in the master node keeping the replicas in the worker nodes. So, to perform a proper file deletion and an optimal use of storage resource, developers have to use the *delete_file* when deleting a file that will go through the COMPSs runtime.

2.2.3 Keeping used task data

Finally, the runtime system may have a policy to minimise the storage usage by considering that input parameters are by default temporal data and the runtime can remove them when it considers than this is necessary. This can produce some data races and the data required by several tasks have been removed from the system before all the tasks have been executed. For that reason, the user may set the flag *keep* to *True* to prevent removing this data once the task finishes.

2.3 Runtime management

The ExaQUte API to define tasks and perform data optimizations has been implemented for two runtime environments PyCOMPSs and Hyperloom. In this subsection, we describe how each runtime manages the storage during the application execution.

2.3.1 PyCOMPSs

This section describes how the PyCOMPSs runtime transparently manages data using the different available storage options. In the following paragraphs, we will discuss how users can specify the storage devices to use as well as how the runtime manages the different data versions and replicas as well as deciding when to copy or remove versions and replicas.

Defining the storage infrastructure

When executing an ExaQUte application with PyCOMPSs, the user has to describe which infrastructure wants to use. This is described in the resources.xml and project.xml. In this resources.xml, users describe the available infrastructure, indicating the available computing units, the memory and storage devices, while in the project.xml indicates which of these resources are used for the actual execution and other configuration options such as applications' and libraries' paths as well as the working directories. When using a supercomputer, these files are automatically generated when submitting the application to the system job scheduler with the *enqueue_compss* command. When the submitted application job starts the execution in the system a prolog script gets the information from the allocated resources an generates the PyCOMPSs configuration files according to the allocation and other flags provided in the *enqueue_compss* command.

As introduced above, users can describe the available storage infrastructures in the projects and resources file. It is mainly done according to the following:

1. **Defining the available local storage device**
   In each compute node defined in the resources.xml file, we describe the available local storage describing the type, size and bandwidth. It is useful to describe local SSD disk as well as Non-volatile memories mounted as a local disk.

2. **Defining a shared parallel file systems**
   If the computing system has a file system which is shared between nodes, it can be described also in the resources.xml file. In this case, we will have a global definition of the shared disk to describe these properties (size, etc), and, inside the computing nodes which have access to this shared disk, we will set the path where the disk is mounted in the computing node.

3. **Define master and worker working directories**
   Finally, user can define where the runtime is going to deploy the working directories for the master and worker in the project.xml. It can be a path inside a shared disk or in a local storage device.

Depending on the specified path either in a application file path, the working directories paths and the described storage interface, the runtime will automatically detect if a data is located in a shared disk or in local storage which provides a valuable information to make a proper data management.

Data versioning and replication Each data which is accessed by tasks is tracked by the PyCOMPSs runtime. According to the access done by tasks to data, it detects the data dependencies between tasks. The PyCOMPSs runtime uses data versioning and replication to solve some of these dependencies and exploiting the maximum inherent parallelism of the application. On the one hand, the data versioning can be applied when a data is accessed as INOUT or OUT. In this situation, the runtime creates a version for the original value and another one for the modified. So, the tasks which are modifying the data can run in parallel with any previous task which is reading this data. On the other hand, replication can be used when we have several tasks reading a data version and they can not be executed in the resource where the data is already located, in this case the runtime replicates the data to another resource to allow more parallelism in the application.

To sum up, the runtime keeps tracks of all the versions generated for an accessed data knowing what is the latest version and which task is generating this version. For each version, it stores the location of all its replicas and the number of tasks which are currently reading (or waiting for reading) a data version. This information will determine how the runtime can do an optimal use of the storage resource without degrading the application performance

Obsolete data clean-up There are some situations that a data version can be declared as obsolete and the runtime can proceed to remove all its replicas in order to save space. This situation is detected when a data version, which is not the latest, does not have task readers for this value. Then all the locations of this data version will be marked as obsolete. Every time that a task is submitted to a node, it includes the obsolete data which is pending to delete in this node.

Besides to the in-task obsolete clean, the runtime periodically forces to clean up all obsolete data in all the nodes in order to avoid long delays from different clean-up processes due to the long duration of the tasks.

Out of order data remove The autonomous obsolete data clean-up process described above ensures that we are not storing old versions of the data. However, the runtime can not delete the final versions by itself because it can not differentiate if a data is temporal or it will be used after a synchronisation point. For this reason the user has to use the delete API calls. When calling the delete method the runtime has to ensure that this data is removed once all the tasks using its data versions have finished. The PyCOMPSs implementation of this call is performing the deletion in an asynchronous way in most of the cases, and it only perform the deletion in a synchronous way when a data file is going to be reused later in the application. This second case is indicated in the call to the delete with a parameter (True).

To perform the data deletion in an asynchronous way, the runtime marks all its data versions with a *to_be_deleted* flag. Then, when a version marked with this flag has no more reader tasks, it is marked as obsolete and removed by the autonomous clean-up process.

2.3.2 HyperLoom

When executing ExaQUte API with HyperLoom, no shared filesystem is need or used. HyperLoom uses only local storage and all data communication is realized as peer-to-peer communication between workers over TCP/IP without any intermediate layer. Server manages global references of data objects, worker local references by running tasks. When both drops to zero, data is removed.

The same architecture is used in Quake (described below) where data service realizes communication between workers, again without using shared file system.

2.4 Experimentation results

To validate the optimization of the storage resources, we have compared two executions of the Problem0 of the Multi-Level Montecarlo (MLMC) application. It is the same application as the experiments performed in Deliverable 4.3 with a different configuration (less nodes and concurrent batches). In the first execution, we have executed the application without appliying any storage optimization. In the second, execution we have introduced the delete call to indicate to the runtime what data is temporal in each iteration.

Both executions have been run in the MareNostrum supercomputer usign 12 worker nodes with 48 CPU cores each. The MLMC has been configured with 3 levels of tasks where each level is using the following resources: first level is using a single core, level two is using two cores, and level 3 is using four cores. In every iteration, the MLMC is evaluating a bunch of 120 tasks of the first level, 60 tasks of the second level and 8 for the third level. We started with the tasks of 10 iterations running in parallel and, to make both executions comparable, we have limited the execution to 20 iterations.

To measure the storage usage per execution, we have run an script which was concurrently running with the application execution that periodically monitors the space used in the master working directory, where the master stores the temporal data, and the workers' working directory, where the tasks' data is generated.

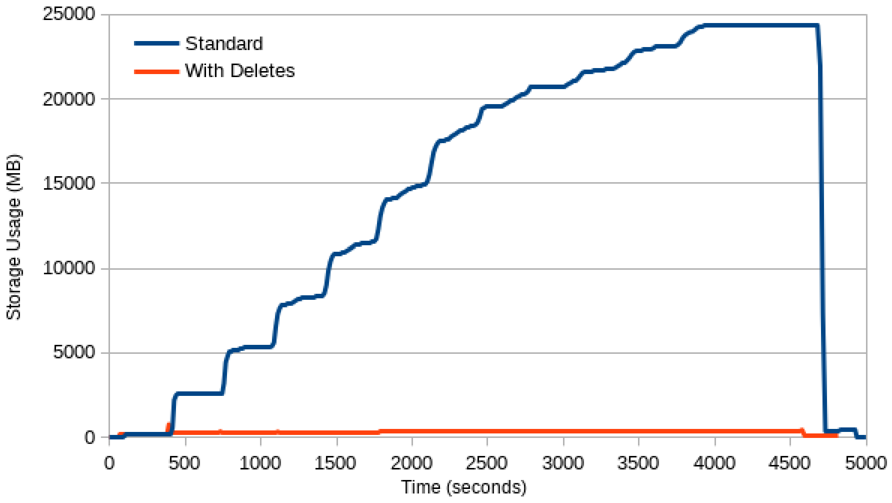

Figure 1: Storage used during the application execution.

Figure 1 shows the values measured by the script when running both executions. The blue line shows the total storage used by the execution without optimizations and the orange line shows the total storage used by the execution with the delete calls. We can observe that the used storage is drastically reduced from almost 25 GB used in the first run to 350 MB in the optimised run. Moreover, we can also see that introducing the delete calls has not introduced any performance degradation because the duration of the optimal execution is even a less than the normal one. This is because in the first case, all the data was transferred back to the expected location at the master's directory, while in the optimized case we only do this for the final data.

3 Extensions for supporting MPI to optimize local storage usage

In the first versions of the ExaQUte applications, the parallelism inside tasks was provided by OpenMP which allow to parallelize codes running in a single node. This option was quite simple to support from the programming model and runtime point of view, by mapping the computing units of the task constraints to the number of OpenMP threads. However, it limits the scalability of the whole application because the maximum parallelism in a task will be limited to available CPU cores in a node. One alternative to improve the task scalability is implementing some of the application tasks with MPI which will be able to use resources from different computing nodes. Supporting these tasks will require some changes in the programming model API in order to describe an MPI task as well as in the runtime to manage the scheduling and execution of these tasks, not only in a infrastructure with shared file systems which are the systems where MPI is normally used, but also systems with fast local storage where the data management will be more complex.

3.1 API definition for supporting new extensions to MPI tasks

A new Python decorator is used to indicate that a task is implemented with MPI. In this decorator, a developer can specify the following properties:

- ***runner*** This specifies the command to spawn the MPI processes. By default an MPI application is executed with the *mpirun* command, but there are other runners to spawn MPI processes such as the *srun* command in systems managed by Slurm
- ***processes*** Indicates the number of MPI processes used by this task
- ***data layout*** Indicates the mapping between the task parameters and MPI processes. It provides hints to the runtime about how to optimize the scheduling, data transfers and objects serialization/deserialization in order to reduce the MPI execution overhead. A task parameter data layout can be set to *ALL* (default value), to an specific rank number, or defining a block inside a set of data. For instance, if an MPI task has a collection of objects as argument, and each MPI process is using a subset of this collection, the data layout will indicate how the data subsets are mapped to each MPI process. Data subsets are defined following the same syntax used by MPI to define a strided vector. The developer has to indicate the number of blocks, the block length and the stride.

In addition to the *@mpi* decorator, a new type of data is introduced to indicate that a task parameter is a set of parameters. The new type is called *COLLECTION* and it will allow developers to pass a set of parameters without needing to indicate all the elements as arguments of a function. The combination of the *COLLECTION* data type with the *@mpi* decorator allow developers to create complex applications where the output generated by MPI processes of a task can be used by the MPI processes of another task.

```
nprocs = 4

@mpi(runner="mpirun", processes=nprocs)
@task(returns=nprocs)
def init(seed):
    from mpi4py import MPI
    rank = MPI.COMM_WORLD.rank
    return rank+seed

@mpi(runner="mpirun", processes=nprocs)
@task(input_data=COLLECTION_IN, returns=nprocs)
def scale(input_data, i):
    from mpi4py import MPI
    rank = MPI.COMM_WORLD.rank
    a = input_data[rank]*i
    return a

@mpi(runner="mpirun", processes=nprocs)
@task(input_data=COLLECTION_IN, returns=nprocs)
def increment(input_data):
    from mpi4py import MPI
    rank = MPI.COMM_WORLD.rank
    a = input_data[rank]+1
    return a

@mpi(runner="mpirun", processes=nprocs, idata_layout={block_count=nprocs, block_length=nprocs, stride=nprocs})
@task(idata={Type:COLLECTION_IN, Depth:2}, returns=nprocs)
def merge(idata):
    from mpi4py import MPI
    rank = MPI.COMM_WORLD.rank
    a=0
    for data in idata[rank]:
        a=a+data
    return a

if __name__ == '__main__':
    input_data = init(0)
    partial_res=[]
    for i in [1,10,20,30]:
        p_data = scale(input_data, i)
        for j in range(2):
            p_data = increment(p_data)
        partial_res.append(p_data)
    results= merge(partial_res)
```

Figure 2: Application example with collections and MPI tasks

Figure 2 shows an example of this complex applications where the *@mpi* decorator and *COLLECTION* data type are combined. This sample application is performing several simple operations (*scale*, *increment*) over a set of data generated in the *init* task. All these operations are parallelized with MPI an defined as MPI tasks. In the code snippet, you can see that the init task implements an MPI task which returns a list of four values, each one generated by a different MPI process. Then the *scale* and *increment* tasks get a collection of four values and generate a new collection, where each value of the collection is computed by a MPI process. Finally, the *merge* task is getting a collection of 16 values (four lists of four elements) and we are indicating with the layout property, that indicates data is treated in four disjoint blocks of four elements.

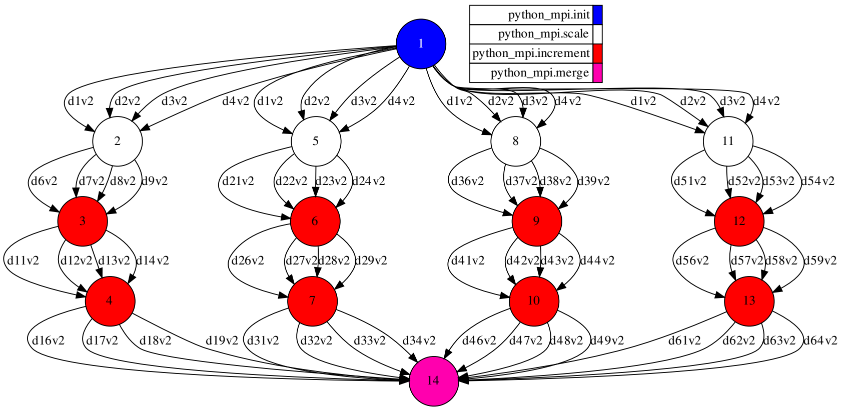
python_mpi.init 🟦
python_mpi.scale ⬜
python_mpi.increment 🟥
python_mpi.merge 🟪
1
d1v2 d2v2 d3v2 d4v2 d1v2 d2v2 d3v2 d4v2 d1v2 d2v2 d3v2 d4v2 d1v2 d2v2 d3v2 d4v2
2 5 8 11
d6v2 d7v2 d8v2 d9v2 d21v2 d22v2 d23v2 d24v2 d36v2 d37v2 d38v2 d39v2 d51v2 d52v2 d53v2 d54v2
3 6 9 12
d11v2 d12v2 d13v2 d14v2 d26v2 d27v2 d28v2 d29v2 d41v2 d42v2 d43v2 d44v2 d56v2 d57v2 d58v2 d59v2
4 7 10 13
d16v2 d17v2 d18v2 d19v2 d31v2 d32v2 d33v2 d34v2 d46v2 d47v2 d48v2 d49v2 d61v2 d62v2 d63v2 d64v2
14

Figure 3: Example application execution graph.

When the code of the figure is executed, the runtime system is able to detect the dependencies between tasks generating the DAG depicted in Figure 3. Next section provides the details about how the runtime manages the execution of this type of graphs composed by MPI tasks.

3.2 Runtime design

Beside the specific API defined to support MPI, the runtime system has to be extended with new features in order to perform an efficient MPI execution. Next paragraphs describe how these features have been implemented.

3.2.1 PyCOMPSs

Multi-node task scheduling The first required extension in the runtime is the support for scheduling and executing tasks which are using several nodes. To implement this feature, the MPI tasks is defined as a set of single tasks where the number of single tasks is the number of MPI processes defined by the user in the MPI annotation. All this single task set is scheduled as other normal tasks. To avoid deadlock in the scheduling of tasks belonging to different sets, once a task of a set is scheduled the rest of tasks of this set have priority to other tasks (from other sets). This ensures that all tasks of a set are scheduled together.

Data staging

Additionally to the changes in the scheduling, another extension required focuses on data staging. Designing a efficient data staging mechanism in MPI tasks is complex. In standard tasks, all input data which is not in the node where a task is allocated must be transferred. In the case of MPI tasks, input data should be located in all the nodes by default because, all MPI processes could access input data. This implies that several replicas must be produced with the corresponding network contention. However, not all the data will be used by all the MPI processes, so we are performing unnecessary copies.

Developers know how the MPI processes are using the data, so they can provide hints with data layout properties defined in MPI task. It allows the runtime to perform an efficient data management because it tells the runtime which data is required by the different processes. With this information, the runtime can try to allocate the resources where the data is located or just make the required data transfers to the nodes where the data is going to be used.

Task Execution

Once all the tasks of an MPI task set are scheduled and data staging performed, the first single task of the subset will act as master managing the execution of the MPI task. It will create the MPI configuration files and flags (number of processes, hostfile, rank to host mapping , etc.) based on the MPI task set allocations and data layout. Then, it will execute an executor Python script by means of the *mpirun* command with the generated configuration. Each MPI process will execute the executor script which is in charge of deserializing the input data which corresponds to this process according to the data layout and executing the method defined as MPI task and serializing the output data.

3.2.2 Quake

HyperLoom is system heavily optimized to running a large number of relatively short single node tasks (hundreds of milliseconds). This is very different use case than running MPI programs. They has different requirements on scheduling, runtime and task specification. The biggest difference came from the fact that tasks are multi node and result of such a task is naturally distributed. On the other hand makespan of MPI applications is more than order of magnitude bigger than what was target makespan for HyperLoom.

We decided not to modify HyperLoom for running MPI programs, because it would not get benefits from most of optimizations and architectural decisions, while these existing optimization would increase code complexity. Therefore, we have decided to create a new solution from scratch that matches ExaQUte needs for MPI applications called *Quake*.

Quake (Quantified Uncertainty - Another tasK Engine) architecture is depicted in Figure 4. It consists of:

- Server – it controls and monitors the computation. It also does the task graph scheduling. Quake client is a simple Python wrapper that communicated with Quake server. The server initiates MPI application via `mpirun` that perform the actual tasks.
- Worker – a wrapper of actual task process that downloads or data necessary fort ask execution. Worker communicates only with its local data service.
- Data service – Data service is responsible for holding objects between MPI tasks and allows to interchange of data peer-to-peer with other computing nodes. It can be seen as a simplified peer-to-peer filesystem.
- Client – A thin python library that communicates with the server. Its main role is to submit a task graph to server and download results to clients application.

The current version of Quake is a working prototype with a simple scheduling algorithm.

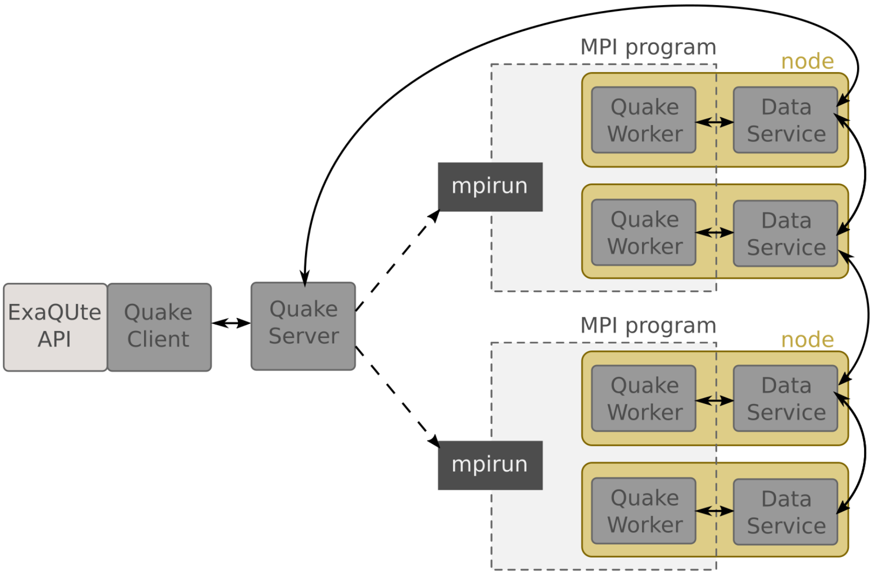

Figure 4: Quake architecture.

4 Conclusions

IO operations can be a bottleneck in HPC application and therefore the use of storage is a critical aspect. Additionally, new storage devices are made available to the HPC communities and it becomes very important to include them in the design of new programming models and runtimes.

In ExaQUte, the applications involve HPC workflows with a large number of simulations and analytics, that generate a large number of files as well. As it has been seen in figure 1, even for a small runs, the use of storage can be very inefficient if not managed appropriately.

Furthermore, MPI simulations are spread along multiple nodes. When involving them in workflows, it is important to offer hints to the runtime that orchestrates the execution to exploit that data locality and reduce data transfers. What is more, the proposed interface supports the coupling of MPI simulations involving different MPI ranks, which is a feature needed in ExaQUte.