Universität Hamburg
Faculty: Mathematik, Informatik und Naturwissenschaften
Department: Informatik
Group: Wissenschaftliches Rechnen

Ophidia Big Data Analytics Framework: Performance Benchmark

Report

Project 'Big Data' WS 2017/18

Dominik Scherer, Nele Lips
Matr.Nr. 6540167, 6824157

Supervisors: Dr. Julian Kunkel, Jakob Lüttgau

Hamburg, 31.03.2018

Abstract

The aim of this report is to give the reader an idea of what to expect in terms of performance of the Ophidia platform. We will explain our approach, give a brief overview of the Ophidia framework and present the results of the benchmark in tabular form as well as in diagrams. Overall, we executed 5 different types of queries in various conditions including compute, I/O and network intensive scenarious with varying amounts of data and an increasing number of CPU cores, resulting in a total of 18 different test cases.

Contents

1 Introduction

Today's scientific research produces immensely large amounts of data which cannot be adequately processed via traditional tools. Often such tools will not scale well for very large sets (tera- to petabyte scale) of multidimensional data [1]. Ophidia aims to tackle these problems by providing a framework for big data access, analysis and processing with a hierarchical storage model and parallel implementations of primitives and operators for a variety of purposes such as data reduction, subsetting and statistical analysis. Ophidia aims to process data locally and thus limits internode connection as much as possible. The framework can be used in a variety of scientific scenarious, including earth sciences, engineering, astrophysics and life sciences [1].
Our project entailed the evaluation of the performance of the Ophidia framework across different tasks. The goal was to investigate how well Ophidia works in terms of speed and scalability. We modelled a series of test queries which we then executed on the Ophidia cluster of the CMCC (Euro-Mediterranean Center on Climate Change) on sets of randomly generated data. We became acquainted with the Ophidia system and how it works over the past few months.

2 Project Plan and Execution

Our initial project plan, which we could only partially realize as outlined, was as follows:
The first step of our project was to become familiar with the way Ophidia and its storage model works. We also read up on the different operators Ophidia offers. As a next step we wanted to install the system with all of its components and learn how to use it. After that we planned to design different benchmarks to test Ophidia thoroughly. We wanted to use different sizes of datacubes ranging from a few gigabyte to a few hundered gigabyte to see how well Ophidia scales. To test how well the parallelism of the framework is implemented we also planned to use different amounts of cores varying from one core to about 64.
Then we wanted to execute our benchmarks and display the results in graphs to illustrate the different execution times for the different cube sizes and core numbers. In the final stages of our project we then wanted to analyze the results of our benchmarks and draw conclusions about the performance of Ophidia. Our overall aim was to find out, given the system and hardware used, whether Ophidia is a fast and efficient solution for the problems it was designed to handle. We also thought we might want to find out whether Ophidia was as good as or better than other comparable systems and if it could potentially be used at the DKRZ (German Climate Computing Center).
Over the course of the project we encountered a few problems which forced us to adjust some aspects of our intial project plan, as described in the following paragraph.
As planned, after we familiarised ourselves with the Ophidia framework and its

inner workings, we tried to install the complete system but ultimately failed to achieve this. Our first mistake was to try to install Ophidia on Ubuntu 16 but that didn't work since this Ubuntu version was not supported, so we encountered many problems with compatibility and program versions during installation. The big challenge with installing Ophidia was for one that there are many preliminaries required (such as MySQL, Munge, Slurm, etc.) of which some had their own problems during installation, and also that there are several different components to the Ophidia system itself that need to be installed and properly configured such as the Ophidia I/O Server, the Ophidia Server, the Ophidia Terminal and the Ophidia Analytics Framework. So after our first failed attempt, we set up Ubuntu 14 and redid the whole installation. This time most things went much more smoothly. We still encountered some problems, mostly in regard to configuration files, but we were able to solve all problems except for one. We always got the the following error message in the Ophidia Terminal when submitting a request: 'Internal server error: no response has been received from analytics framework'.
We could not resolve this issue on our end even with help from the developers and deemed it to time consuming to track the problem any further at that time, so we resorted to using a virtual machine with a functional Ophidia system to learn how to use the program. Since a virtual machine is of course not suitable for benchmarking, the developers provided accounts for us for using the Ophidia cluster at the CMCC to execute our test series on. To access the Ophidia system installed on the cluster we only had to have the Ophidia Terminal installed on our local machines. The Ophidia Terminal alone is quite easy to install in comparison to the whole system so that wasn't a problem. However, the accounts were limited to using a maximum of 40 cores and there also seemed to be limits to computation time and datacube sizes as we encountered problems when trying to execute queries that took more than one hour to compute or produce datacubes that were larger than 50GB.
Now that we finally had access to a functional instance of the Ophidia framework on a system suitable for benchmarking we then created a series of tests queries which we executed on the cluster. The first iteration of our benchmark was not very accurate however. We mostly had to rely on our own understanding of Ophidia which of course was not perfect and the developers were too busy at that time to give feedback on a regular basis, thus leading to results that were not accurately representing the performance of the Ophidia framework. Eventually the developers had the time to assist us in revising some aspects of the benchmark and we redid all of the tests. This time we obtained better results reflecting the performance of the system much more accurately. As a side note, we hope it has become clear that the suggestions made by the developers of Ophidia did have an impact on the design and consequently the results of the benchmark which should be kept in mind by the reader.

3 Ophidia Overview

We now want to give a brief introduction into the functionality and the architecture of the Ophidia framework. The technical information in the following paragraphs is based on the paper *Ophidia: toward big data analytics for eScience* by S. Fiore et al. [1].
In Ophidia the multidimensional data is stored in datacubes which are made up of dimensions and facts. In a climate science context, explicit dimensions could be latitude and longitude. The facts are measure values which are stored in arrays and could have time as implicit dimension in this example. Each of these arrays is identified by a key which is the combination of the explicit dimensions. The implicit dimension gives the positions of the measure values in the array. The datacubes are stored in containers which can hold several, one or no datacubes at all. All containers are part of a session. One can have multiple separate sessions with different data to separate experiments. Sessions, containers, and datacubes are all part of the virtual file system of Ophidia which is different from the real file system. The real file system is external to Ophidia and is used to import and export files from or to the Ophidia platform.
There are different kinds of commands in the Ophidia framework, operators and primitives. The operators are divided into different catagories. Operators for data analysis work mostly with datacubes. They take one or more datacubes as input and output another datacube with the results (aggregations, duplication, merging, reductions, subsetting, etc.). Further operators include commands for data import and export (NetCDF, FITS, HTML), metadata (size, dimensions, etc.), the virtual file system (creation/deletion of containers, folders and so on), workflow management, administration and some other miscellaneous operators (commands dealing with the real file system for example).
The Ophidia primitives are array-based. These primitves can be divided into the following categories: operations on the core array (concatenations, permutations, sorting, shifting, etc.), primitives for selection (extracting subsets, masking), arithmetic operations, statistical operations, transformations, numerical analysis and mining.
Ophidia manages data hierarchically. On the highest level we have multiple I/O nodes (multi-host). Each host has multiple DBMS instances. Then there are multiple instances of physical databases on the same DBMS and multiple fragments (smaller tables) on the same physical database. This storage model allows the Ophidia framework to exploit data locality and limit internode-communication which allows high scalability. Chosing the right fragment distribution for specific purposes is a key aspect of working with the Ophidia system.

4 Benchmark Outline

Our benchmark is divided into two categories, 'compute intensive' and 'I/O and network intensive'. The former consists of a series of test in which primitives were used that apply computationally expensive formulas, the latter contains operators that challenge the network and require more I/O but are not particularly taxing on the CPU. We had to combine I/O and network into one category since they are not easily separable due to the architecture of the Ophidia framework. The tests of each category are further divided into two sets: constant workload with an increasing number of cores, also known as strong scaling, and a constant number of cores with increasing workload. In the first set, we applied the formulas $f(x) = e^x$, $f(x) = (x^3 - x^2)/x$ and $f(x) = \sqrt{x^{(sin(x^3)/cos(x^2))}}$ on the elements of datacubes of sizes 10 Gigabyte and 50 Gigabyte for the 'compute intensive' test series, and exported and imported datacubes of sizes 10 Gigabyte, 20 Gigabyte and 50 Gigabyte from Ophidia to NetCDF files and back again for the 'I/O and network intensive' test series. The second set (increasing workload, fixed number of cores) consists of the same operators as the first. We chose datacubes of sizes 1, 2, 5, 10, 20 and 50 Gigabytes for both categories. In the 'compute intensive' tests cores were fixed to 32 and 16, in the 'I/O and network intensive' tests we used 32, 16 and 4 cores.

We are now going to present and explain every command that was used in the benchmark:

oph_apply cube=<PID>;query=oph_math('OPH_DOUBLE', 'OPH_DOUBLE', measure, 'OPH_MATH_EXP');ncores=<cores>

The command oph_apply takes the datacube with the address <PID> and executes the query on it, generating an output datacube in the process. The query in this case is the command *oph_math*. It applies the e-function (*OPH_MATH_EXP*) on the measurement arrays of type double and outputs them again as double value. The *ncores* parameter specifies the number of cores used for execution.

oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', '(x^3-x^2)/x', '0');ncores=<cores>

In this case, the query consist of the *oph_predicate* command. Again, it takes in and outputs double values on the measurement arrays. The fourth argument can be any expression. If it satisfies the expression in the fifth argument, the sixth argument will be executed, otherwise the last argument will be executed. In this specific command the expression '1' naturally always satifies '>0', so the formula $(x^3 - x^2)/x$ is applied every time and the last argument is never executed. There is currently no other way in Ophidia to apply a formula directly (without nesting commands) to each element of the datacube arrays.

oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', 'sqrt(x^(sin(x^3)/cos(x^2)))', '0'); ncores=<cores>

This query is the same as in the aforementioned command, only differing in the formula that is being applied. In this case, the formula is $\sqrt{x^{(sin(x^3)/cos(x^2))}}$.

oph_exportnc2 cube=<PID>;output_path=<path>;export_metadata=no; ncores=<cores>

The command *oph_exportnc2* exports the datacube with the address <PID> to the path specified in the *output_path* parameter as a single NetCDF file. Metadata is not exported in this specific case (*export_metadata=no*).

oph_importnc measure=measure;src_path=<path>;import_metadata=no; ncores=<cores>

The command *oph_importnc* imports the specified measurement of a NetCDF file with the location <path> into Ophidia as a datacube. Metadata is not imported in this specific case (*import_metadata=no*).

The data with which we worked with was randomly generated. Conveniently, Ophidia already offers a command to generate datacubes filled with random values called *oph_randcube*. Every datacube that was used in this benchmark was created with the following command with only the <size> parameter changed as needed:

oph_randcube container=benchmark;nhost=2;ndbms=1;ndb=1;nfrag=64; ntuple=1000;measure=measure;measure_type=double;exp_ndim=2; dim=lat|lon|time;concept_level=c|c|d;dim_size=128|1000|<size>; compressed=no;host_partition=test_3;filesystem=local; ioserver=ophidiaio_memory

The parameter <size> corresponds roughly to the size of the datacube produced in MB. For example, if a datacube of size 1GB=1000MB was used in the benchmark, the datacube was created with the above command with <size>=1000, a 10GB datacube with <size>=10000, 50GB with <size>=50000 and so on. The exact size of each datacube can be obtained by using the command *oph_cubesize* which is provided as well for every test. The specific *oph_randcube* command shown above creates a random datacube with 2 hosts, 1 DBMS per host, 1 database per host, 64 fragments per database and 1000 tuples per fragment. Each tuple then contains <size> elements of type double. We just called these measurements 'measure' for our tests. The dimensions are 'lat', 'lon' and 'time' and the datacubes are not compressed. The last arguments of the command specifiy the host partition, filesystem and the ioserver, for which we used the natively supported inmemory server that works in the RAM.

The infrastructure of the Ophidia cluster on which the benchmark was executed consists of 5 IBM x3650 BD dual-processor nodes with Intel Xeon E5-2660v2 CPUs (10 cores @ 2.2GHz), 256GB RAM and 12TB of raw disk space each, for a total of 100 cores and 1280GB of memory. As described above, two of these nodes where used in our test series (*nhost=2*). The nodes are interconnected through a dedicated high-speed 10-Gigabit network. Input and output files are stored on a GlusterFS shared file system installed over the node hard drives which uses the same communication network.

All time measurements seen in the results of the benchmark were copied as displayed in the Ophidia Terminal 1.2.0 after each execution of a command.

5 Results

<u>Test Series 1</u>: strong scaling with compute intensive operators

Test 1.1:

Datacube size:
10GB (OPH_CUBESIZE=9773.438477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_math('OPH_DOUBLE', 'OPH_DOUBLE', measure, 'OPH_MATH_EXP');ncores=<cores>

Test 1.2:

Datacube size:
10GB (OPH_CUBESIZE=9773.438477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', '(x^3-x^2)/x', '0');ncores=<cores>

Test 1.3:

Datacube size:
10GB (OPH_CUBESIZE=9773.438477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', 'sqrt(x^(sin(x^3)/cos(x^2)))', '0');ncores=<cores>

number of cores	time in seconds
1	49.11
2	25.75
4	14.02
8	7.60
16	4.99
32	3.95

Table 1: Test 1.1, 10GB, e^x

number of cores	time in seconds
1	210.27
2	107.77
4	56.59
8	31.05
16	16.46
32	10.07

Table 2: Test 1.2, 10GB, $(x^3 - x^2)/x$

number of cores	time in seconds
1	510.71
2	258.25
4	133.71
8	70.08
16	38.16
32	20.53

Table 3: Test 1.3, 10GB, $\sqrt{x^{sin(x^3)/cos(x^2)}}$

Test 1.4:

Datacube size:
50GB (OPH_CUBESIZE=48835.938477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_math('OPH_DOUBLE', 'OPH_DOUBLE', measure, 'OPH_MATH_EXP');ncores=<cores>

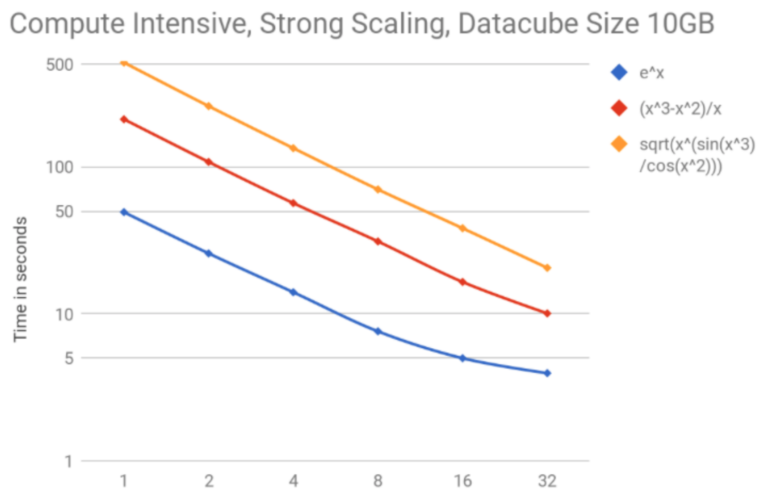

Figure 1: Tests 1.1-1.3

Test 1.5:

Datacube size:
50GB (OPH_CUBESIZE=48835.938477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', '(x^3-x^2)/x', '0');ncores=<cores>

Test 1.6:

Datacube size:
50GB (OPH_CUBESIZE=48835.938477 MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', 'sqrt(x^(sin(x^3)/cos(x^2)))', '0');ncores=<cores>

number of cores	time in seconds
1	241.10
2	122.90
4	66.05
8	35.36
16	17.53
32	12.75

Table 4: Test 1.4, 50GB, e^x

number of cores	time in seconds
1	1060.24
2	532.32
4	275.22
8	143.50
16	75.51
32	42.71

Table 5: Test 1.5, 50GB, $(x^3 - x^2)/x$

number of cores	time in seconds
1	missing value (error)
2	1281.59
4	660.59
8	339.55
16	180.91
32	93.16

Table 6: Test 1.6, 50GB, $\sqrt{x^{sin(x^3)/cos(x^2)}}$

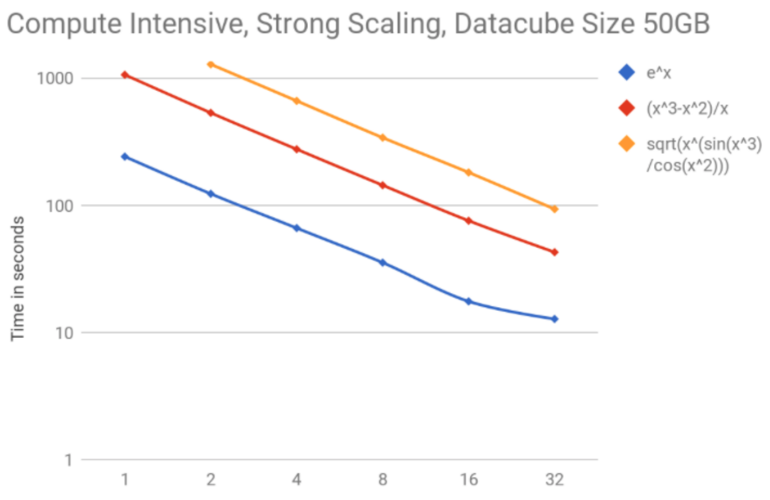

Figure 2: Tests 1.4-1.6

Test Series 2: workload scaling with compute intensive operators

Test 2.1:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_math('OPH_DOUBLE', 'OPH_DOUBLE', measure, 'OPH_MATH_EXP');ncores=32

Test 2.2:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', '(x^3-x^2)/x', '0');ncores=32

Test 2.3:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', 'sqrt(x^(sin(x^3)/cos(x^2)))', '0');ncores=32

datacubesize in GB	time in seconds
1	1.73
2	1.94
5	2.64
10	3.84
20	6.59
50	12.75

Table 7: Test 2.1, 32 cores, e^x

datacubesize in GB	time in seconds
1	2.99
2	3.94
5	6.78
10	11.32
20	18.94
50	42.71

Table 8: Test 2.2, 32 cores, $(x^3 - x^2)/x$

datacube size in GB	time in seconds
1	4.32
2	6.84
5	11.37
10	20.52
20	39.44
50	93.16

Table 9: Test 2.3, 32 cores, $\sqrt{x^{sin(x^3)/cos(x^2)}}$

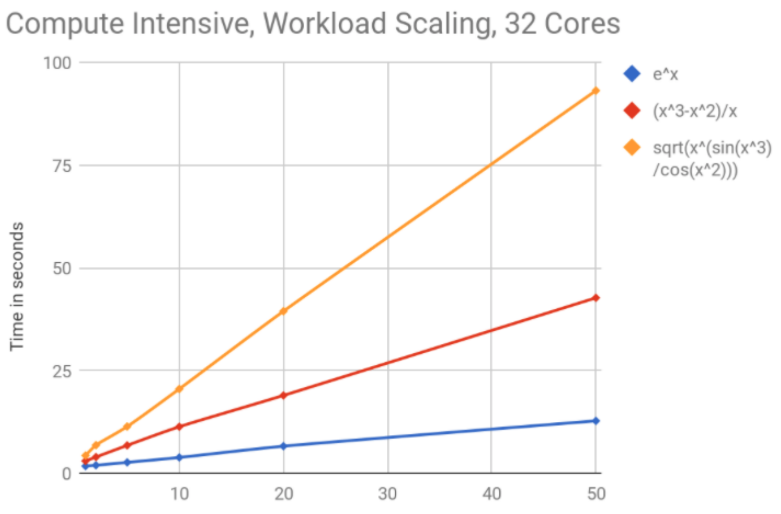

Figure 3: Test Series 2.1-2.3

Test 2.4:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_math('OPH_DOUBLE', 'OPH_DOUBLE', measure, 'OPH_MATH_EXP');ncores=16

Test 2.5:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', '(x^3-x^2)/x', '0');ncores=16

Test 2.6:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Command executed:
oph_apply cube=<PID>;query=oph_predicate('OPH_DOUBLE', 'OPH_DOUBLE', measure, '1', '>0', 'sqrt(x^(sin(x^3)/cos(x^2)))', '0');ncores=16

datacubesize in GB	time in seconds
1	2.15
2	2.48
5	3.68
10	5.78
20	9.21
50	17.53

Table 10: Test 2.4, 16 cores, e^x

datacubesize in GB	time in seconds
1	3.32
2	4.82
5	9.41
10	17.07
20	31.91
50	75.51

Table 11: Test 2.5, 16 cores, $(x^3 - x^2)/x$

datacube size in GB	time in seconds
1	5.29
2	9.33
5	20.01
10	38.73
20	65.37
50	180.91

Table 12: Test 2.6, 16 cores, $\sqrt{x^{sin(x^3)/cos(x^2)}}$

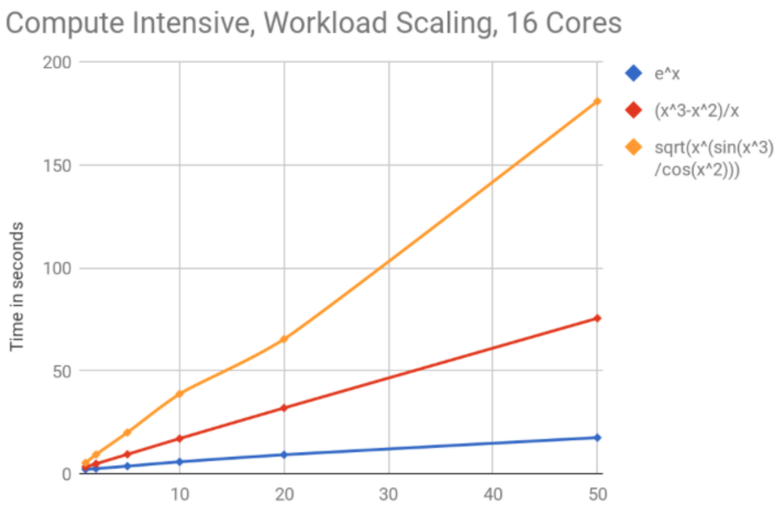

Figure 4: Test Series 2.4-2.6

Test Series 3: strong scaling with I/O and network intensive operators

Test 3.1:

Datacube size:
10GB (OPH_CUBESIZE=9773.438477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=<cores>
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=<cores>

Test 3.2:

Datacube size:
20GB (OPH_CUBESIZE=19539.063477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=<cores>
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=<cores>

Test 3.3:

Datacube size:
50GB (OPH_CUBESIZE=48835.938477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=<cores>
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=<cores>

number of cores	time in seconds EXPORT	time in second IMPORT
1	73.79	64.99
2	61.80	47.34
4	54.30	44.70
8	50.89	73.53
16	50.43	118.42
32	47.28	71.80

Table 13: Test 3.1, 10GB, Ophidia to NetCDF to Ophidia

number of cores	time in seconds EXPORT	time in second IMPORT
1	141.69	119.32
2	115.50	91.63
4	100.28	81.00
8	98.14	128.24
16	88.77	203.22
32	83.04	123.16

Table 14: Test 3.2, 20GB, Ophidia to NetCDF to Ophidia

number of cores	time in seconds EXPORT	time in second IMPORT
1	275.46	259.64
2	217.02	184.08
4	197.60	167.36
8	195.28	195.03
16	192.51	262.79
32	179.48	169.86

Table 15: Test 3.3, 50GB, Ophidia to NetCDF to Ophidia

Test Series 4: workload scaling with I/O and network intensive operators

Test 4.1:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=32
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=32

Test 4.2:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=16
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=16

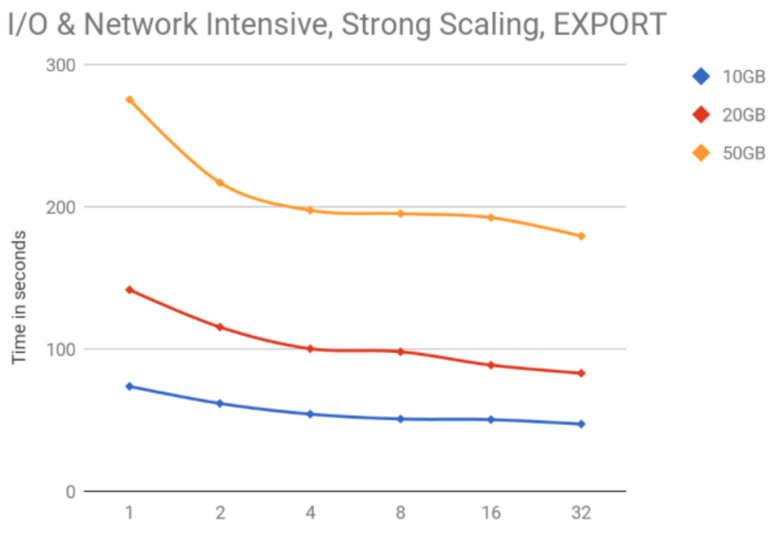

Figure 5: Tests 3.1-3.3 Export

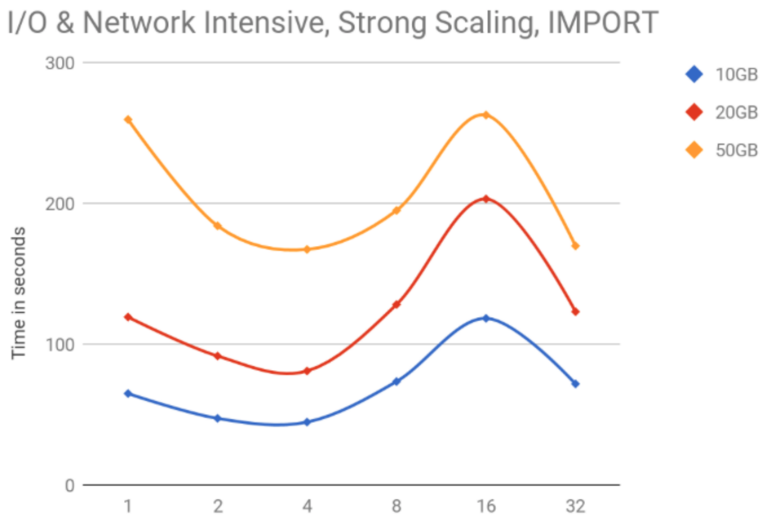

Figure 6: Tests 3.1-3.3 Import

Test 4.3:

Datacube sizes:
1GB (OPH_CUBESIZE=984.375977MB)
2GB (OPH_CUBESIZE=1960.938477MB)
5GB (OPH_CUBESIZE=4890.625977MB)
10GB (OPH_CUBESIZE=9773.438477MB)
20GB (OPH_CUBESIZE=19539.063477MB)
50GB (OPH_CUBESIZE=48835.938477MB)

Commands executed
EXPORT:
oph_exportnc2 cube=<PID>;output_path=<path>; export_metadata=no;ncores=4
IMPORT:
oph_importnc measure=measure;src_path=<path>; import_metadata=no;ncores=4

datacube size in GB	time in seconds EXPORT	time in second IMPORT
1	11.99	10.30
2	12.47	19.20
5	25.62	40.26
10	47.28	71.80
20	83.04	123.16
50	179.48	169.86

Table 16: Test 4.1, 32 cores, Ophidia to NetCDF to Ophidia

datacube size in GB	time in seconds EXPORT	time in second IMPORT
1	7.11	13.99
2	11.31	26.84
5	28.92	65.91
10	50.43	118.42
20	88.77	203.22
50	192.51	262.79

Table 17: Test 4.2, 16 cores, Ophidia to NetCDF to Ophidia

datacube size in GB	time in seconds EXPORT	time in second IMPORT
1	8.61	6.97
2	12.78	11.00
5	28.64	24.29
10	54.30	44.70
20	100.28	81.00
50	197.60	167.36

Table 18: Test 4.3, 4 cores, Ophidia to NetCDF to Ophidia

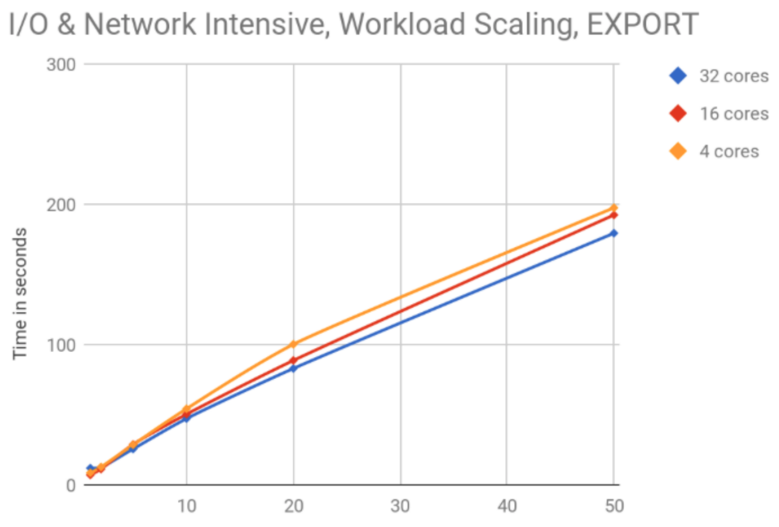

Figure 7: Tests 4.1-4.3 Export

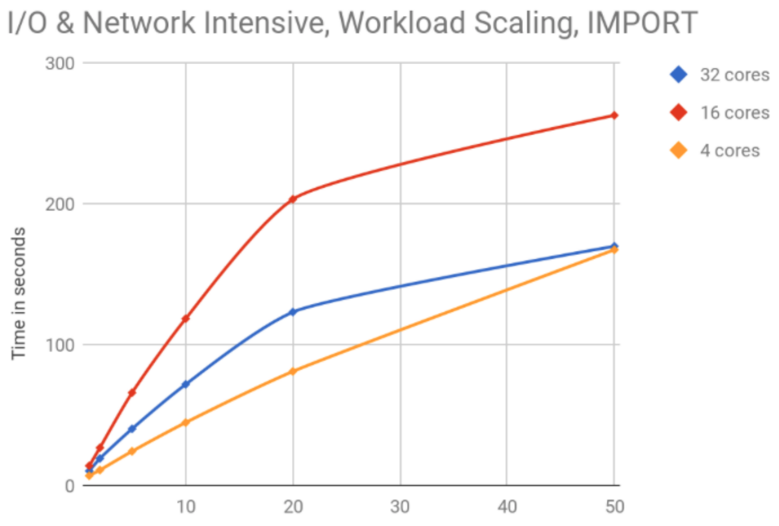

Figure 8: Tests 4.1-4.3 Import

6 Analysis

The results of test series 1 are consistently good. In these types of tests, the optimal result theoretically would be a speedup that is linear. The Ophidia framework comes close to that in almost all instances if we take into account that time measurements are subject to inaccuracies and that most tests did not run for very long absolutely speaking (seconds to minutes), so slight deviations affect the speedup values more drastically. While these results are good, they were also to be expected considering the architecture of the framework. Due to the fact that Ophidia tries to exploit data locally and thus limits internode communication, very little communication overhead is created which makes queries scale very well. Considering this, we would expect the speedup values to stay this good even on a much larger scale (more nodes, more data volume).
However, as previously mentioned, a suitable fragment distribution has to be chosen for specific purposes to use the system optimally, not only for performance reasons but also for practical reasons. For example, if one wants to calculate the averages in a datacube by using the operator *oph_aggregate*, the group size of which the average is to be calculated can at most be set to a value equal to how many tuples are in the table. Depending on what averages of which values need to be computed, the fragmentation scheme of the datacube might has to be changed.
The results of test series 2 are in line with those of test series 1. When workload is increased, resources can usually be used more efficiently, meaning that when data size is doubled we ideally need less than double the amount of time. This is the case with all the results obtained in this test series.
As seen in test series 3 and 4, I/O and network intensive tasks suffer from a significantly worse speedup which is to be expected. All data has to be sent and collected in one NetCDF file for the export tests and distributed again into an Ophidia dat-

acube for the import tests. Taking a look at test series 3 regarding export, we can see that for datacubes of smaller sizes (Test 3.1 for example, Table 13) increasing the number of cores does not make a huge difference in terms of speed. Even with a datacube size of 50GB (Test 3.3, Table 15) going above 2 cores does not yield any significant performance increase. Regarding import, increasing the number of cores beyond 4 even decreases performance quite significantly, indicating that a large amount of overhead is being created and that there is probably not enough parallelism intrinsic to the operation.
Scaling behaviour is a lot better again with fixed numbers of cores and increasing workloads as seen in test series 4 for export as well as import tasks. Taking both I/O and network test series into consideration, the best overall performance for the datacube sizes tested here can be achieved when using 4 cores for these types of operations. Looking at Figure 5 and Figure 7 and taking time measure inaccuracies into account, one could even assume that not more than 4 cores are being used for export tasks even if *ncores* is set to a higher value than 4. However, this does not seem to be the case for import tasks since performance is clearly decreasing when using more than 4 cores.

7 Conclusion

We have shown and discussed the results of our benchmark aimed at analysing the performance of the Ophidia framework. It has become clear that the system is particularly suited for handling compute intensive tasks on large amounts of data due to its good scaling behaviour which is mainly achieved by employing a hierarchical storage model. The I/O and network test results have shown that the export of data does not profit from using more than 4 cores and that the import oeration even suffers performance loss when using more than 4 cores.
Moreover we have revealed how we carried out the project and also discussed the problems we had to overcome. We mostly struggeled in the beginning with the installation of the framework which ultimately was unsuccessful despite our best efforts. Once we had access to an operational Ophidia system things became easier and the project went on much more smoothly. In the end we were able to design and execute an extensive benchmark with the help of the developers of the Ophidia platform.

Bibliography

[1] S. Fiore, A. D'Anca, C. Palazzo, I. Foster, D. N. Williams, G. Aloisio. *Ophidia: toward big data analytics for eScience.* Elsevier B.V., 2013.

Appendices

List of Figures

List of Tables