Exascale **Q**uantification of **U**ncertainties for **Te**chnology and Science Simulation

D4.3 Benchmarking report as tested on the available infrastructure

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	IT4I
Deliverable Type:	Other
Dissemination Level:	Public
Related WP & Task:	WP 4, Task 4.4
Status:	Final

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
Tomas Karasek	IT4I		V0.1	
Stanislav Böhm	IT4I		V0.1	
Brendan Keith	TUM		V0.1	
Ramon Amela	BSC		V0.2	
Rosa M Badia	BSC		V0.2	
Tomas Karasek	IT4I		V0.3	
Stanislav Böhm	IT4I		V0.3	

Change Log

Versions	Modified Page/Sections	Comments
V0.1	First document version	
V0.2	Results with PyCOMPSs and corrections	
V0.2	Results with HyperLoom and corrections	

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Jan Martinovic	IT4I	30.07.19	OK
WP leader	Rosa M. Badia	BSC	30.07.19	OK
Coordinator	Riccardo Rossi	CIMNE	30.07.19	OK

Executive summary

The main focus of this deliverable is testing and benchmarking the available infrastructure using the execution frameworks PyCOMPSs and HyperLoom. A selected benchmark employing the Multi Level Monte Carlo (MLMC) algorithm was run on two systems: TIER-0 (MareNostrum4) and TIER-1 (Salomon) supercomputers. In both systems, good performance scalability was achieved.

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
FILE_IN	Path to a file passed to a function that is not modified
FILE_INOUT	Path to a file passed to a function that is modified during the call
FILE_OUT	Path to a file passed to a function that is created during the call
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
MPI	Message Passing Interface
PBS	Portable Batch System
PyCOMPSs	Python binding for COMPS Superscalar
SLURM	Simple Linux Utility for Resource Management

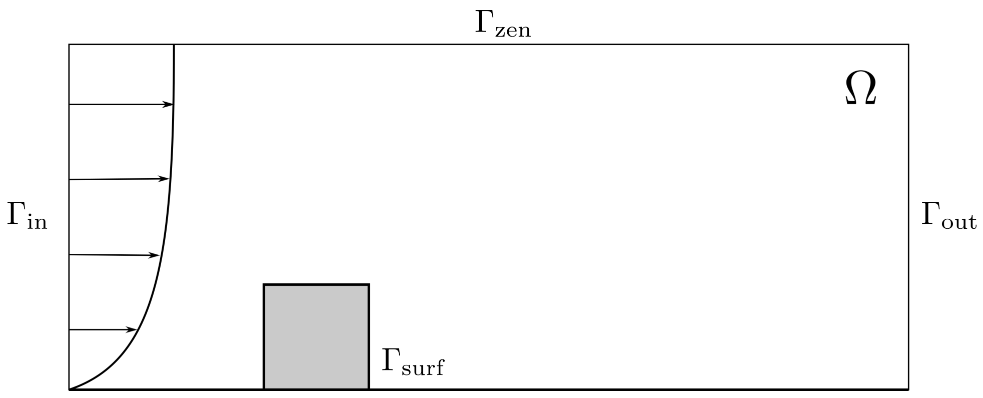

Figure 1: Benchmark domain Ω and boundaries.

1 Introduction

In Task 4.4, infrastructure benchmarking takes place. This deliverable summarizes the results of benchmarking on two supercomputers: MareNostrum 4 operated by BSC, and Salomon operated by IT4Innovations. The deliverable is structured as follows: In Section 2, the benchmark run on all supercomputers is described followed by Section 3, where results for each supercomputer are presented. Section 3 is divided into two parts, one part for each supercomputer, where results for each execution framework, PyCOMPSs and HyperLoom, are presented, together with the conclusions for each individual supercomputer. In Section 4, the overall conclusion, summarizing all results, is presented.

2 Experiments description

In the ExaQUte project, the PDEs of principle concern are the incompressible Navier–Stokes equations, given as follows:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} &= -\nabla p + \nu\nabla^2\mathbf{u}\,, \\ \nabla\cdot\mathbf{u} &= 0\,. \end{aligned} \tag{1}$$

In this section, we will use these equations to define a benchmark problem in order to develop the current deliverable.

In the ExaQUte project, we will be simulating wind flow past large complicated structures under calibrated physical conditions. Figure 1 is an extremely simplified 2D representation of this scenario. Here, the impeding object is simply a square, which leaves behind the computational domain Ω.

On the inflow boundary, Γ_{in}, the following temporal average velocity is prescribed:

$$\mathbf{u}(\mathbf{t},\mathbf{\Gamma}_{\text{in}})\cdot\mathbf{n} = \overline{\mathrm{u}}\Big(\frac{\mathbf{z}}{\mathbf{z_0}}\Big)^{\alpha}, \qquad \mathbf{u}(\mathbf{t},\mathbf{\Gamma}_{\text{in}})\cdot\mathbf{n}^{\perp} = \mathbf{0}\,, \tag{2}$$

where $\boldsymbol{n}$ is the unit normal vector field on $\partial\Omega$ and $\boldsymbol{n}^{\perp}$ is any unit length vector field orthogonal to $\boldsymbol{n}$. The novelty here is that this boundary condition is stochastic, but time-invariant, with uncertain parameters $\overline{\mathrm{u}} \sim N(10, 1.0)$ and $\alpha \sim N(0.12, 0.12)$, where . The remaining boundaries, Γ_{surf}, Γ_{zen}, and Γ_{out} have wall, free slip, and zero flux boundary conditions, respectively; i.e.,

$$\begin{aligned} \mathbf{u}(\mathbf{t}, \mathbf{\Gamma}_{\mathrm{surf}}) &= \mathbf{0}\,, & \boldsymbol{\sigma}(t, \Gamma_{\mathrm{out}})\boldsymbol{n} &= \mathbf{0}\,, \\ \mathbf{u}(t, \Gamma_{\mathrm{zen}}) \cdot \boldsymbol{n} &= 0\,, & \boldsymbol{\sigma}(t, \Gamma_{\mathrm{zen}})\boldsymbol{n} \cdot \boldsymbol{n}^{\perp} &= 0\,, \end{aligned} \tag{3}$$

where $\boldsymbol{\sigma} = -p\mathrm{I} + \nu\big(\nabla\mathbf{u} + \nabla\mathbf{u}^{\top}\big)$ is the Cauchy stress tensor.

The above-described problem is stochastic, and the chosen algorithm to study uncertainty propagation for this deliverable is Multilevel Monte Carlo (MLMC). We refer to the work package 5 for details about the algorithm. The Quantity of Interest of study is the drag coefficient, computed on the square building.

2.1 API modification

For purpose of benchmarking of execution frameworks the common API defined in deliverable 4.1 is updated in the following way:

- To prevent memory leaks, intermediate temporary results were separated from objects that are requested by the user application. This has been implemented by creating a keep flag
- Implementing checks which ensures that creating a successful run of an application with the ExaQUte API on one node (local backend) also leads to a correct run in a distributed environment.
- To avoid problems with cloudpickle and for easier testing, explicit init() function has been introduced.
- Update of the programming style to make the API compliant with Python coding style.

2.1.1 Keep flags

A first improvement of the common API was the introduction of "keep" flag. If the "keep" flag is set to "true" while task is created, then it indicates that the user wants to hold the resulting value(s) of the task and is responsible for an explicit clean-up (by calling the `delete_object` function). Otherwise, the task result is considered temporary. In the case of chains of tasks, the results is kept until the first "synchronization point", that is, by calling "barrier" or "get_value_from_remote". At this synchronization point, the object is fully cleaned by the system when necessary and the user does not have to care about its deletion. In this case, using the object after the synchronization point returns an exception.

Example:

```
@exaqute.task()
def task1()
   return 10

@exaqute.task()
def task2(x, y)
   return x + y


a = task1()
b = task2(a, 10, keep=True)
```

```
result = get_value_from_remote(b)

# "a" is automatically cleaned as soon as possible, "a" is not available upto this
    point
# "b" has to be cleaned manually


```

By introducing the keep flag, the invocation to the "compute" method is not required as the necessary information can be derived from the keep flags.

2.1.2 Checks

To ensure that creating a successful run of an application with the ExaQUte API on one node (local backend) also leads to a correct run in a distributed environment, a check procedure has been implemented.

In the previous version, the "empty" implementation was used, e.g.:

```
def get_value_from_remove(obj):
    return obj  # just return first parameter

```

This solution worked and leads to functional program; however, it does not prevent an invalid usage of the API in some cases. It can also easily lead to an application that uses the API wrongly without any error. However, when a different backend is used, it may end to an error state.

For example:

```
x = 10
get_value_from_remote(x)
```

Do not indicate any error in the original implementation, but obviously, as "x" is not a result of task; it is an invalid usage of the API.

2.1.3 Init()

The original API used an implicit initialization by the module import. This makes some simplification for users but brings some non-trivial problems.

- Modules that tracks imports usually do not assume behaviour with such a strong side effect of the import; therefore, usage of cloudpickle was broken.
- It is hard to test such libraries. It is hard to setup a test environment before import, import separated functions for unit testing without calling init, or create a fixture that initializes and deinitializes environment for each test.

So solve these problems we have introduced method init() in the ExaQUte API that contains the initialization. The user is obligated to call it before any ExaQUte function (except usage of decorators).

2.1.4 Programming style update

We have renamed two entities in ExaQUte API:

- ExacuteTask() to task(): It already resides in ExaQUte package, so it is not necessary to repeat this. Also, decorators are usually named by snake_case convention.

- processing Units to processing_units: According PEP8 (standard coding style for Python), argument of a function should be named by snake case, not CamelCase convention.

3 Infrastructures benchmark

The following two supercomputers were used to execution the numerical experimet described in section 2:

- The TIER-0 system MareNostrum 4 operated by BSC with 11.15 Petaflops of peak performance, which consists of 3,456 compute nodes equipped by two Intel®Xeon Platinum 8160 (24 cores at 2,1 GHz each) processors.
- The TIER-1 system Salomon operated by IT4Innovations, with 2 Petaflops of peak performance consists of 1,009 compute nodes equipped by two Intel®Xeon Haswell (12 cores at 2,5 GHz each) processors.

Those supercomputers were selected to measure the performance using the two execution frameworks (PyCOMPSs and HyperLoom) on broad range of machines because nowadays not all companies and researchers have access to the most powerful TIER-0 systems and such benchmarks performed on smaller systems would be very valuable source of information.

The access to the different supercomputers was granted through different processes. In the case of MareNostrum 4, the project partners UPC and BSC applied to a call of the Spanish Supercomputing Network (RES) in January 2019. The application requested 300.000 CPU/hours, 2000 GB of disck and 2000 GB of scracth disk space. The application was accepted with priority. The application has been renewed in April, obtaining 320.000 CPU hours. In the case of Salomon, the project partners IT4I and BSC submitted proposal for multiyear project through Open Call scheme securing 1.2M core hours for duration of 18 months. For the next period the partners will apply for a larger allocation since the current one has been consumed very fast.

To measure the performance of the mentioned systems using the two execution frameworks, a strong scalability experiment was designed.

To equalize the differences in number of cores/node of the different systems, different number of compute nodes is defined for each system. The total number of cores in each configuration is the same. To obtain a reasonable scalability graph, four different allocations on each system has been used. The table below shows the number of compute nodes on each system.

System	No Nodes
MareNostrum	4, 8, 16, 32
Salomon	8, 16, 32, 64

Table 1: Number of compute nodes used on different systems

This will result in using of 192, 384, 768, and 1,536 cores in total.

The size of the batches used in each of the Levels of the Multi Level Monte Carlo algorithm was also agreed a priori, to compare equivalent results through the different systems and different execution environments:

Level	Batch Size	Number of OpenMP threads
0	1500	1
1	125	2
2	10	4 or optionally, the number of cores available in the node

Table 2: Batch size for different levels of MLMC

3.1 Results obtained on MareNostrum 4

Since the MLMC algorithm is asynchronous and the possibility of exploiting this fact is very important, we have chosen to run 9 full iterations with 4 active batches at each time. We also decided to run second benchmark with more batches in each level of MLMC on MareNostrum 4 (see table 3).

Level	Batch Size	Number of OpenMP threads
0	2000	1
1	150	2
2	10	4 or optionally, the number of cores available in the node

Table 3: Batch size for different levels of MLMC on MareNostrum 4

3.1.1 PyCOMPSs

Table 4 shows the results with the batch size [1500, 125, 10] described at the beginning of this chapter. Table 5 shows the results for second benchmark with batch size [2000, 150, 10].

3.1.2 HyperLoom

To be able to compare the results obtained on Salomon, the same version on HyperLoom was installed on MareNostrum as well. During the testing we encountered problem by

No of nodes	No of cores	Execution time (s)	Speedup	Ideal Speeedup	%
4	192	36 947	1.00	1	1.00
8	384	19 325	1.91	2	0.96
16	768	10 721	3.45	4	0.86
32	1536	5 822	6.35	8	0.79

Table 4: PyCOMPSs results with batch size of [1500, 125, 10] in Mare Nostrum

No of nodes	No of cores	Execution time (s)	Speedup	Ideal Speeedup	%
4	192	48 143	1.00	1	1.00
8	384	24 790	1.94	2	0.97
16	768	13 019	3.70	4	0.93
32	1536	7 273	6.62	8	0.83
64	3072	4 089	11.78	16	0.74
128	6144	2 742	17.56	32	0.55

Table 5: PyCOMPSs results with batch size of [2000, 150, 10] in MareNostrum

executing the Problem_zero use-case. We investigated this problem and we were able to track down the problem to the point of transferring the computation to a worker. In HyperLoom, the commonly used module "cloudpickle" is used to serialize computation. The problem when running the Problem_zero use-case is, that when the computation is serialized and later on de-serialized a function that is using Kratos, the Python interpreter get stucked. Unfortunately we were not able to construct a smaller example to debug it further. We tested Kratos and HyperLoom independently and both were working fine. Also, the de-serialization of any other object different to a Kratos involved function was working and the de-serialization of such object outside of Loom was working as well. It should be noted that we have been running the same configuration on Salomon and we never encountered such a problem. Moreover, the de-serialization in "cloudpickle" is quite lightweight, so we could rule it out as a root cause of the problem.

We were unable to investigate it further because unfortunately, the debugging tool Valgrind is not working on MareNostrum, so we were not able to use Helgrind or DRD to use these tools for detecting lock violations. With a plain GDB tool, we are able to see that is hanging somewhere in the Python interpreter, but without any futher details. We have asked MareNostrum support to install Python debuginfo package that should provide us with insight on what is happening in Python. MareNostrum support installed Python debuginfo package in version 3.6.6. Since Kratos was instaled on MareNostrum using version 3.6.1 of Python, it has to be re-builded using same version of the Python i.e. 3.6.6. Unfortunately new problems arises during installation of Kratos which are being currrently being solved by MareNostrum support.

3.1.3 Conclusion

Results obtained on MareNostrum 4 by PyCOMPSs shows very good scalability of the example with batch sizes [1500, 125, 10].

With the second batch size [2000, 150, 10] we observe that with 128 nodes the efficiency decreases in a larger proportion than before with smaller node-counts. It would be interesting to test a highest batch size since we are at the limit of filling the computer with enough computation.

3.2 Results obtained on Salomon

Since Salomon is much smaller machine than MareNostrum 4 we run a smaller case with a batch size of [1500, 125, 10] and with an alternative number of iterations. With PyCOMPSs we setthe number of iterations to 9 and experiment executed by HyperLoom

No of nodes	No of cores	Execution time (s)	Speedup	Ideal Speeedup	%
4	96	66 705	1.00	1	1.00
8	192	32 780	2.03	2	1.01
16	384	18 289	3.65	4	0.91
32	768	10 164	6.56	8	0.82
64	1536	7 068	9.44	16	0.40

Table 6: PyCOMPSs results with batch size of [1500, 125, 10] in Salomon (9 iterations)

No of nodes	No of cores	Execution time (s)	Speedup	Ideal Speeedup	%
4	96	25 171	1.00	1	1.00
8	192	12 376	2.03	2	1.02
16	384	6 014	4.18	4	1.05
32	768	4 673	5.38	8	0.67
64	1536	3 859	6.52	16	0.41

Table 7: HyperLoom results with batch size of [1500, 125, 10] in Salomon (2 iterations)

used 2 iterations. In both cases we keep number of active batches the same as we run on MareNostrum 4 (4 batches).

3.2.1 PyCOMPSs

Table 6 shows the results with the batch size [1500, 125, 10] and 9 iterations performed by PyCOMPs on Salomon supercomputer.

3.2.2 HyperLoom

Times obtained by running experiment with batch size [1500, 125, 10] and 2 iterations are reported in Table 7.

3.2.3 Conclusion

Both cases run on Salomon supercomputer shows very good scalability, close to ideal one, up to the 16 compute nodes i.e. 384 cores. After this, the scalability drops significantly. From presented results could be concluded that even for TIER-1 system much bigger case could be designed and run.

4 Conclusion

Results obtained on both systems, MareNostrum 4 (TIER-0) and Salomon (TIER-1), exhibit very good scalability up to a certain. However, speedup decrease could be explained by the size of the benchmark which do not have enough computation workload for such large machines.

Results also show that there is no large difference between TIER-0 and TIER-1 systems when it comes the the scalability of presented benchmarks. Both systems shows good scalability up to the 768. In the future benchmarking will continue with bigger batches and bigger examples which will on finest level of MLMC run on multiple compute nodes.