University of Strasbourg

Department of Mathematics and Computer Science

Internship Report

ExaMA WP3
Dashboard Performances

Tanguy PIERRE

Master's student in *Scientific Computing and Mathematics of Innovation*

Supervisor: V. CHABANNES

August 22, 2024

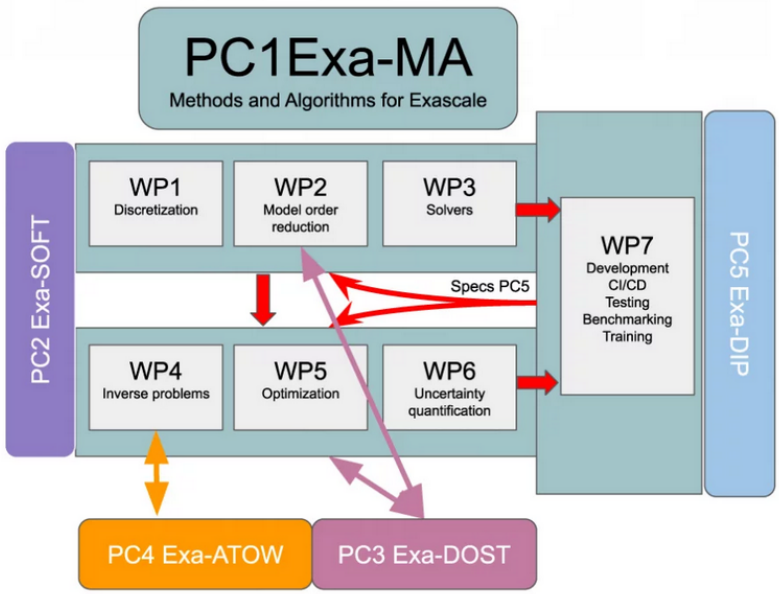
Contents

1 Introduction

This work is part of the *ExaMA* project, which falls under the French program *NumPEx*[6]. NumPEx is highly active across various research domains of HPC, but the *ExaMa* project focuses primarily on methods and algorithms and their adaptation to exascale computing. *Exascale* machines are capable of performing 1 exaflop, which is equivalent to or 10^{18} floating-point operations per second. This immense computational power highlights the need for algorithms and methods that are both robust and efficient, as these will be crucial for taking advantage of these capabilities.

Great power doesn't necessary mean great performances. Algorithms may experience terrible performances on or near exascale machines, often due to the complex communication between the numerous cores and nodes. In order to control the methods and their performances, they must be tested under different conditions. Their behaviour with increasing task numbers or problem complexity will provide clear insights about their efficiency.

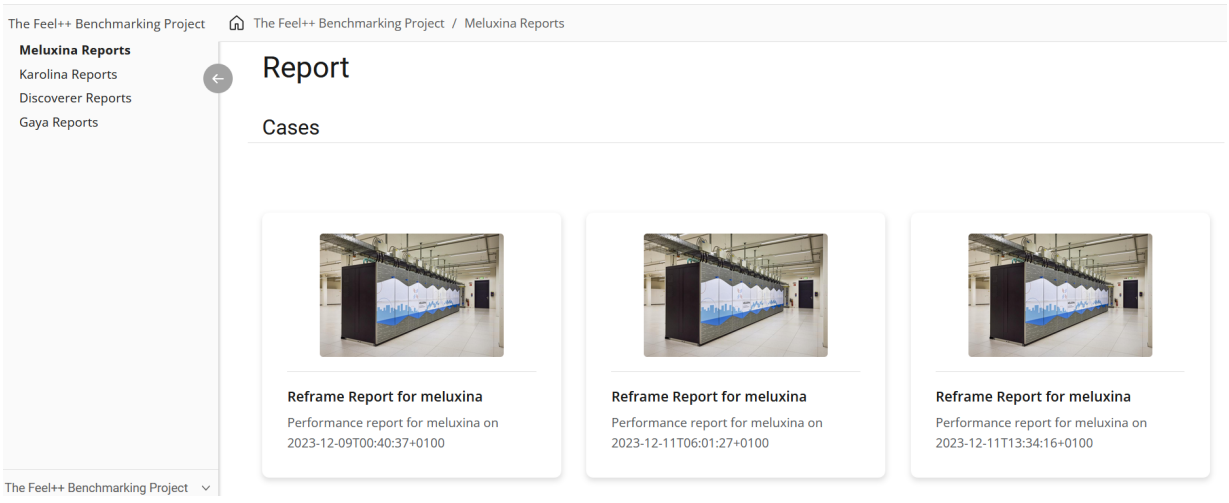
The following graphic illustrates the organization of the ExaMA project The *Dashboard Performance* project is positioned between WP3 and WP7, as its objective is to develop a benchmarking platform for the solvers from WP3.



Benchmarking is an essential step for the ExaMA project, as it needs to analyze and compare performance across systems with different architectures. This process is crucial for several reasons: it guides the development team in optimizing the scalability of the implemented algorithms, helps identify performance bottlenecks. Additionally, benchmarking is also providing complete transparency about the evaluation process by demonstrating the whole approach.

As launching tests on supercomputers isn't always easy because of their availability and costs, there is a real need to have fast-deployable test programs and to store the results efficiently. Additionally, this serves as a manner to verify that new features haven't caused any pullback in performance.
Storing the results will provide a possibility to perform analyses depending on specific context, such as machine or application. Therefore, it is also crucial for the project to create a well-structured database in order to get a clear overview.

It is exactly what the *Dashboard Performances* project aims to achieve: providing a clear and easy-to-use interface between tests and results. This interface is already available at `https://feelpp.github.io/benchmarking/benchmarking/index.html`.



2 Objectives

- Establish a **Continuous Integration/Continuous Deployment** workflow. This means that every time a new test is executed and integrated into the repository main branch, a task will automatically be launched for updating the documentation site.

- Improve the **Dashboard Presentation**. Enhancement of the visualizing tools, such as incorporating interactive graphs. This will not only make the data more accessible, but will be also helpfué for comprehension.

- Provide a **Database** for easy access and retrieval of test results. Currently, it is very difficult to perform clear data analysis based on different aggregations, as it requires manual searching. Therefore, we want to develop an automated solution to select relevant fields.

- Conduct **Representative Tests**. Identifying appropriate values for performance analysis in specific contexts can be challenging, but is essential for obtaining reliable results about the algorithm's behavior.

3 Tools

For testing the implemented algorithms and predicting how they will behave, we will use ***ReFrame HPC***[3]. ReFrame is a robust framework based on Python classes that allows us to focus only on the algorithm by isolating the running code from the system's complexity. The interesting aspect of this framework is that it allows launching multiple tests at once, ensuring their sanity, and extracting performance metrics. Furthermore, its pipeline allows to create tests for specific stages of the program, such as the compilation or execution phase. For our purpose, we will only use the execution testing part of it.

The algorithms we want to test will be from the ***Feel++***[2] open-source library, more specifically, its mono- and multiphysics *toolboxes*. This library is in active development by the *Cemosis* group, the *"Center of Modeling and Simulation in Strasbourg for mathematics, industry and other disciplines,"* This is a "**F**inite **E**lement **E**mbedded **Library** in C++". It is used for solving physical problems, up to dimension 3, through the resolution of partial differential equations employing continuous or discontinuous *Galerkin* methods.

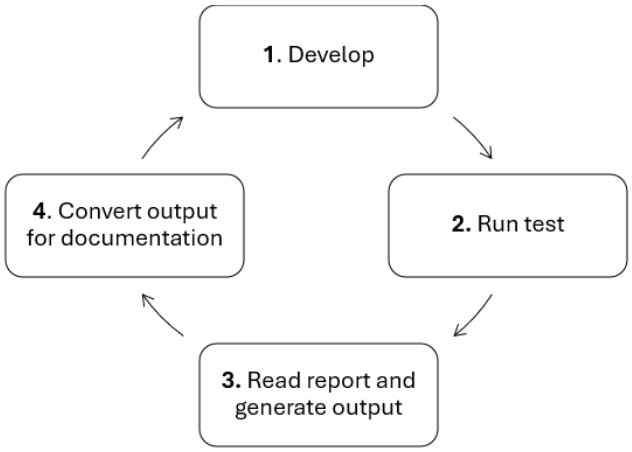
For reporting the results, we will use ***Antora***[1]. *Antora* gives the opportunity to publish documentation on the web. The documentation needs to be written in *AsciiDoc*. Once it's done, *AsciiDoctor* will handle the conversion to *html* for responsiveness and browser compatibility. *AsciiDoc* also lets us include Python functions directly in the documentation, enabling us to add dynamic content, such as graphs.

As we will work with *.adoc* files, we first need to create such files based on the collected data. For this task, we will use ***Jinja2***[4], a template engine for Python code. This tool requires a template to generate files designed according to the desired specifications and layout. With *Jinja2*, you can easily create dynamic content by combining templates with data from your Python code. We will need it for plotting the results into the documentation pages.

The ***Plotly***[7] library will be used for data visualization. This graphing library offers various methods for plotting data. It provides interactive features, like buttons for example, that will enhance user-experience for visualizations. Its flexibility makes it an ideal choice for our project, as it supports responsive graphs that can adapt to different data.

4 Process

Before diving into more technical details, let's first outline the sequence of each step. The diagram below illustrates the different stages of the process, from retrieving reports to generating the final documentation.



1. The first step of this cycle is, of course, development. This is the only way to move forward regarding performance. Pushing or merging new test reports should trigger a new process.

2. Use of ReFrame for launching specified test. The results will be reported in a *.json* file. If a code doesn't pass all the tests, a warning will be emitted to inform the development team of a performance decline regarding the new code.

3. The relevant run report data will be extracted with a Python script. Then, the Python library *Jinja2*[4] generates an *.adoc* file containing plotting methods.

4. The last step is to convert the created *.adoc* file into *.html* for generating the website. As mentioned before, this task is performed by *Antora* together with *AsciiDoctor*.

It's apparent that this task involves significant repetition. Therefore, we want our process to work on the most autonomous way possible. For this point, we use the previously mentioned tools in a specific order. Each script has its role to play at a particular moment in our process timeline. Let's see when they occur, and how they all interact together.

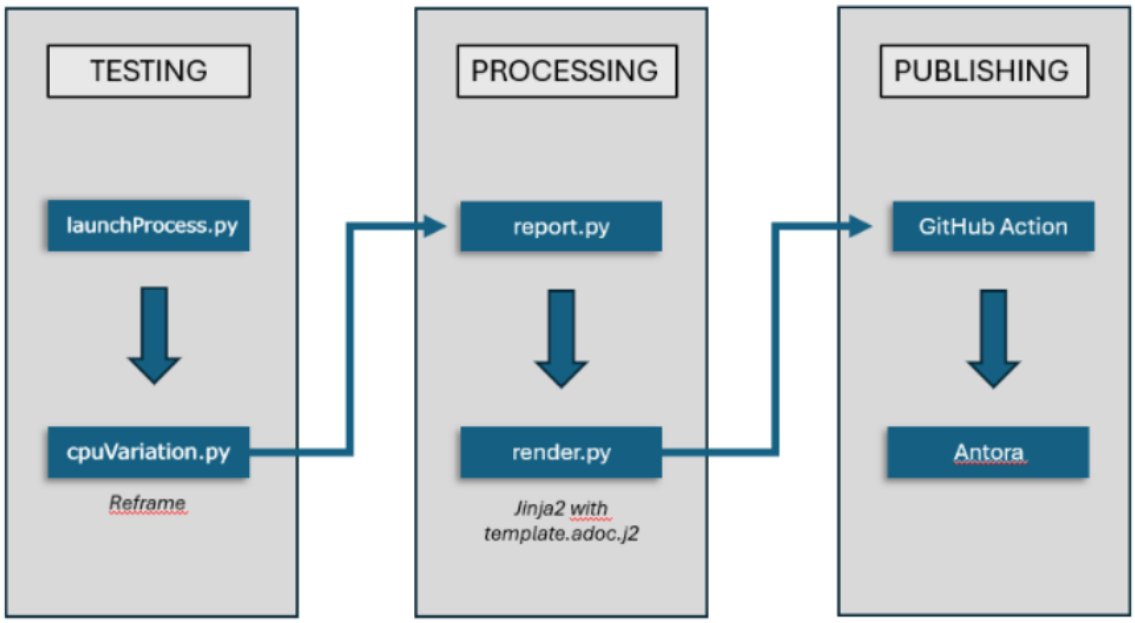


Figure 1: Workflow interaction

4.1 Launching

To launch the process, we first had to determine the best approach to do it. In the following section, we will briefly present the different methods we considered and explain why they were not optimal, before introducing the final method.

1. The first method was a simple *bash* script. It was dedicated to launching every *.cfg* file inside the `BENCH_CASES_CFG` directory with the corresponding *Feel++* toolbox. There was no possibility for parametrization at all, except by redeclaring the needed inside the file.

2. We then continued to develop this script by adding command-line options. We aimed to create a script that builds a list of *.cfg* files in a specified directory. These files would then be executed by *ReFrame*. We encountered two problems with this approach: Each configuration file for the toolboxes would be launched with the same parameterization (CPU number, mesh size, etc.), and the arguments were no longer manageable for building the list. Here follows the main loop of this script:

```
for tb in "${toolboxes[@]}"; do
    yes '=' | head -n "$columns" | tr -d '\n'
    echo "[PROCESS␣LAUNCHED␣ON␣${tb^^}␣TOOLBOX]"

    extended_path="${FEELPP_TOOLBOXES_CASES}/${tb}/cases"
    while read -r cfgPath; do
        relative_path=${cfgPath#"$FEELPP_TOOLBOXES_CASES"}
        relative_dir=$(dirname "${relative_path#*/cases/}")
        base_name=$(basename "${relative_path%.cfg}")

        matched=true

        if [ ${#cases[@]} -gt 0 ]; then
            matched=false
            for case_name in "${cases[@]}"; do
                if [[ "$relative_path" == *"$case_name"* ]]; then
                    matched=true
                    break
                fi
            done
        fi
        if [ ${#directories[@]} -gt 0 ]; then
            matched=false
            for dir_name in "${directories[@]}"; do
                if [[ "$relative_path" == *"/$dir_name/"* ]]; then
                    matched=true
                    break
                fi
            done
        fi
        if $matched; then
            config1="/home/tanguy/Projet/benchmarking/src/feelpp/benchmarking/␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reframe/config-files/reframeConfig.py"
            config2="/home/tanguy/Projet/benchmarking/src/feelpp/benchmarking/␣\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣reframe/config-files/${hostname}.py"
```

```
            if $listing; then
                echo "$relative_path"
            else
                yes '-' | head -n "$columns" | tr -d '\n'
                echo "[Starting␣$relative_path]"
                report_path=$(pwd)/docs/modules/${hostname}/pages/reports/${tb}/ \
                                        ${relative_dir}/${current_date}-${base_name}.json
                reframe -C "$config1" -C "$config2" -c "$RFM_TEST_DIR/toolboxTest.py" \
                                                -S case="$cfgPath" --system=$hostname \
                                        --report-file="$report_path" --exec-policy=serial -r
            fi
        fi
    done < <(find "$extended_path" -type f -name "*.cfg")
```

Listing 1: Bash script extract

3. As previously mentioned, there wasn't any room for parametrization between launched tests. Therefore, we decided to add *JSON* configuration files for the benchmarking process, which would contain, for example, all the necessary paths and the number of CPUs with which the test should be parametrizeAs a result, we had to add a *configuration reader* for these files. This was done in Python. The goal was to pass the configuration path to *ReFrame*, so it could retrieve all the necessary data inside the test class.

4. Some information needed in the launcher script was provided by the *configReader* class, so we had to find an efficient way to split the data. Since switching between *bash* and *python*, then back to *bash*, and finally into *ReFrame's* Python was quite complicated, we decided to rewrite the launcher script in Python. This way, manipulating the different objects during the process and parsing command line options with an appropriate library *(argparse in our case)* is much easier.

The usage of the script is as follows:

```
    Usage: launchProcess.py HOSTNAME --feelppdb PATH [--config CONFIG ...] [--dir DIR ...] \
                                                [--exclude EXCLUDE ...] [--policy {async,serial}] \
                                                [--list] [--list-files] [--verbose] [--help]
    Positional arguments:
      hostname            Name of the machine
                          Valid choices: {gaya, local, discoverer, karolina, meluxina}

    Options:
      --feelppdb, -f      Path to feelppdb folder (required)
      --config, -c        Paths to JSON configuration files
                          In combination with --dir, specify only basenames for selecting JSON files
      --dir, -d           Name of the directory containing JSON configuration files
      --exclude, -e       To use in combination with --dir, mentioned files will not be launched
      --policy, -p        Reframe's␣execution␣policy:␣{async,␣serial}␣(default:␣serial)
␣␣␣␣␣␣␣--list,␣-l␣␣␣␣␣␣␣␣␣List␣all␣parametrized␣tests␣that␣will␣be␣run␣by␣Reframe
␣␣␣␣␣␣␣--list-files,␣-lf␣␣List␣all␣benchmarking␣configuration␣file␣found
␣␣␣␣␣␣␣--verbose,␣-v␣␣␣␣␣␣Select␣Reframe's verbose level by specifying multiple v's
␣␣␣␣␣␣␣--help,␣-h␣␣␣␣␣␣␣␣␣Display␣help␣and␣quit␣program
```

Listing 2: Bash script extract

4.2 Benchmarking with ReFrame

4.2.1 Context

As previously mentioned, ReFrame has the capacity to focus only on the algorithm's performance, operates with Python classes that are customizable through parameterization. Within these classes, one can easily describe and define the environment, but also parametrize the test. General attributes like test description or file-paths can be immediately set up.

ReFrame works with an exact pipeline through different test stages, therefore the necessity to use the decorators provided by the framework. With these decorators, ReFrame schedules the execution of each method. For each 6 different stages, going from "Initialization Phase" to "Cleanup Phase", it is possible to specify to ReFrame when to execute the method. For instance, if you want to set up the test environment, the method should probably be decorated by `@run_before('run')`.

The site on which the test will run has to be described in a Python dictionary (or in a .json file). It's divided into two main categories: systems and environments.
In the first one you can describe the specific system on which the test will be launched. In the environment category, you can define various execution environments, for example, by specifying different compilers.

Here is the site configuration script for Gaya, a machine located at the University of Strasbourg:

```
site_configuration = {
    'systems': [
        {
            'name': 'gaya',
            'descr': 'Gaya',
            'hostnames': ['gaya'],
            'modules_system': 'tmod4',
            'partitions': [
                {
                    'name': 'public',
                    'scheduler': 'squeue',
                    'launcher': 'mpiexec',
                    'max_jobs': 8,
                    'access': ['--partition=public'],
                    'environs': ['env_gaya'],
                    'prepare_cmds': ['source␣/etc/profile.d/modules.sh'],
                    'processor': {
                        'num_cpus': 128
                    },
                    'devices': [
                        {
                            'type': 'cpu',
                            'num_devices': 6
                        }
                    ]
                },
            ]
        }
    ],
    'environments': [
        {
            'name': 'env_gaya',
            'modules': ['hpcx'],
            'cc': 'clang',
            'cxx': 'clang++',
            'target_systems': ['gaya:public']
        }
    ]
}
```

Listing 3: Gaya configuration

- *For more specific clusters of tests, it is perfectly possible to split the system by partition, and to have multiple environments.*
- *Specific node-lists can also be set for the test.*
- *The* `'source/etc/profile.d/modules.sh'` *is needed for loading the hpcx module correctly.*

The tests can be parametrized by the system's topology, which includes the number of CPUs, tasks per node, . . . , or even by various test files.

For the scalability tests, the most important caracteristics are the number of nodes, the number of task per core and the number of task.

Our tests will run with the `-bind-to core` option from *mpiexec* for maximum efficiency. Therefore, we need to know the number of physical CPUs per node. This number is given by multiplying the number of sockets by the number of CPUs per socket. For Gaya, we can see that we have 6 nodes, with 128 physical CPUs each.

Here follows the Python function that allows to launch the testcases with different combination of nodes and number of task per core. Numerous test cases are launched with just one command, this is precisely where ReFrame's strength lies.

```
def parametrizeTaskNumber(minCPU, maxCPU, minNodes, maxNodes):
    for part in rt.runtime().system.partitions:
        nbTask = minCPU
        yield nbTask
        while (nbTask < part.processor.num_cpus) and (nbTask < maxCPU):
            nbTask <<= 1
            yield nbTask

        if not (minNodes == 1 and maxNodes == 1):
            if maxNodes < part.devices[0].num_devices:
                nbNodes = maxNodes
            else:
                nbNodes = part.devices[0].num_devices
            for i in range(minNodes+1, nbNodes+1):
                nbTask = i * part.processor.num_cpus
                yield nbTask
```

Listing 4: Task number parametrization

This script was adapted from the 2022 CSCS Webinar [5] about ReFrame.
We use the `ReFrame.core.runtime` *module to access the current host's topology and the yield feature to parametrize each test by calling this function.*

This parameterization will allow the number of tasks to start at `minCPU` and increase by powers of 2 up to `maxCPU`, (normally corresponding to the physical CPUs number on a single node). Then, the number of tasks will increase by increments of 128, corresponding to the physical CPUs on up to 6 identical nodes

For verifying the sanity of a test, ReFrame provides *"sanity functions"*. These functions run once the test is complete and can, for example, perform checks on the output data to ensure correctness. This feature enhances the reliability of our testing process, helping to catch any anomalies or errors in our results.
In order to avoid any regression in performance, we also have the possibility to provide references for each system configuration.

4.2.2 Scale performances extraction

With the command-line option `--heat.scalability-save=1`, or more generally `--<tbprefix>.scalability-save=1`, it is possible to save scale performances from the different *Feel++* toolboxes.
In order to extract values from files or directly from the terminal output, we use ReFrame's sanity module, particularly `sanity.extractsingle()`. This method will extract values using *regex patterns*. Methods for performance measures needs the decorator `@performance_function`.

The process of performance extraction is following:

1. Find files containing `'scalability'` and `'.data'`
2. Open files and extract each column name (except for the first column, which is the number of task), which are performance variable names
3. Build pattern for line to extract. The pattern depends on the number of names extracted in the previous stage.
4. Extract line and get all performance values.
5. Update *ReFrame's* performance dict.

```
def pattern_generator(self, valuesNumber):
    valPattern = '([0-9e\-\+\.]+)'
    linePattern = r'^\d+[\s]+' + rf'{valPattern}[\s]+' * valuesNumber
    linePattern = linePattern[:-1] + '*'
    return linePattern

@run_before('performance')
def set_perf_vars(self):
        self.perf_variables = {}
        make_perf = sn.make_performance_function

        scaleFiles = self.findScaleFiles()
        for filePath in scaleFiles:
            names = self.get_column_names(filePath)
            perfNumber = len(names)
            line = self.extractLine(self.pattern_generator(perfNumber), filePath, perfNumber)

            perfStage = filePath.split('scalibility.')[-1].split('.data')[0]
            for i in range(perfNumber):
                unit = 's'
                if i == 0 and 'Solve' in perfStage:
                    unit = 'iter'
                self.perf_variables.update( {f'{perfStage}_{names[i]}' : make_perf(line[i], unit=unit)} )
```

Listing 5: Performance value extraction

- The `valPatt` pattern is enclosed by parentheses, indicating that values corresponding to the pattern will be saved
- `valPatt` matches numbers, but also numbers in scientific notation
- The ^ character guarantees that the pattern matches an expression starting on a new line for avoiding any unintended interaction
- The * character at the end guarantees that the pattern matches an expression with any end of line escape character
- The `make_performance_function` from the *ReFrame* sanity module is needed because `sn.extractsingle()` returns a *deferrable expression*. Such expressions are typically evaluated at a later stage.

4.3 Process extracted data

ReFrame's results are stored in a *.json* file. However, before we can properly analyze this data, we need to extract and process the information contained in the report. This task will be performed by the `Report` class. Using the *json* Python library, the class will load the report file and construct the following dataframes : `df_perf, df_partialPerf, df_speedup, df_partialSpeedup`.

- The first two dataframes will be built while reading ReFrame's report as their values are immediately reported by ReFrame.
- The second dataframe will be built by expanding a reference value to a higher number of tasks. The reference value is chosen as the performance from the test with the lowest number of tasks.
- For each performance, linear regression values were calculated using `sklearn.linear_model.LinearRegression` and added to the frame.

Some reports do not include values for partial performance variables, but only the total score of a particular execution stage. Therefore, we had to construct two different frames to organize the data more efficiently. This organization facilitates comparisons between each stage component performance, especially for plotting the results.

5 Results

5.1 Single node test

The following study has been performed on case3 from Thermal Bridges ENISO10122 with the Feel++ heat toolbox. This is as 3-dimensional case with temperature distribution and heat flows through the wall-balcony junction. In order to have more calculation, the mesh 'hsize' has been passed from 0.02 to 0.01, and the discretization from P1 to P2. These changes are done using the provided `json.patch` *commands thanks to configReader.*

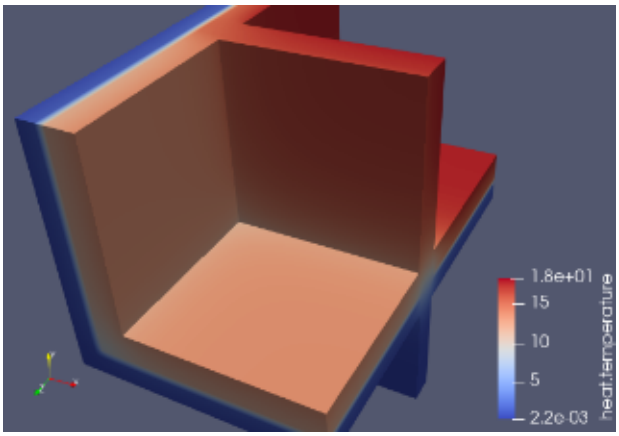
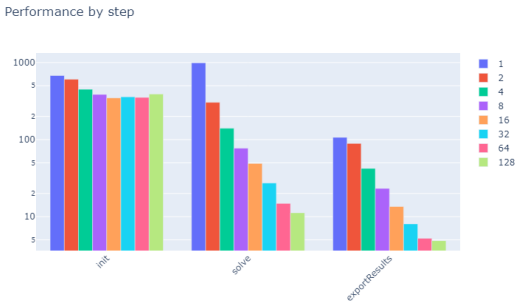


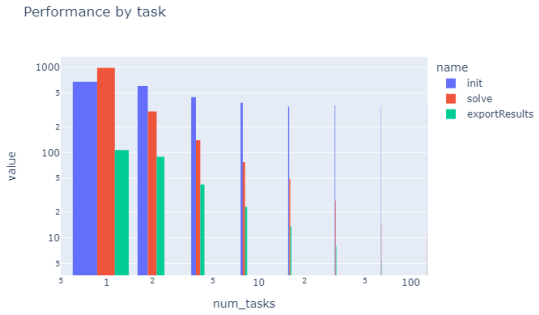
Figure 2: Thermal Bridges ENISO10122, Wall-Balcony[2]

In order to have maximum efficiency regarding the parallelism, we use the *Feelpp Mesh Partitioner*. This is configured in ReFrame's prebuild commands.

For single node tests, we let vary the number of task by powers of 2 up to the number of CPUs on the node.
As Gaya has 128 physical CPUs per node, our test were launched with 1, 2, 4, 8, 16, 32, 64, 128 tasks.



(a) Gaya, Performances by step



(b) Gaya, Performances by task

Both graphics represents the 3 main performance metrics, namely *init, solve, exportResults* for any number of launched tasks. We can notice that the *init* stage of the process doesn't scale well, because some *partial init variables* simply don't run in parallel. The solving and postprocessing stages scale well to higher number of tasks.

Let's see now how the *Feelpp Heat Toolbox* scales performances behaves regarding the speedup.

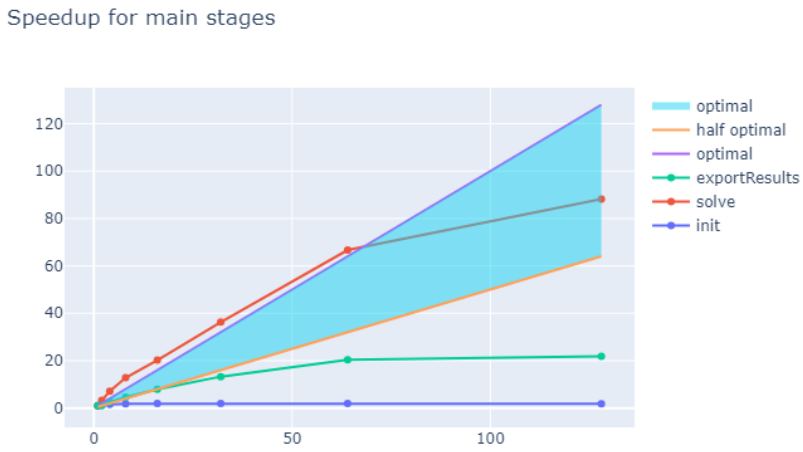


Figure 4: Gaya, speedup for main references

The speedup results correspond to the previous graphs. The *init* stage doesn't scale at all because of its sequential parts, while the *solve* has really good performances.

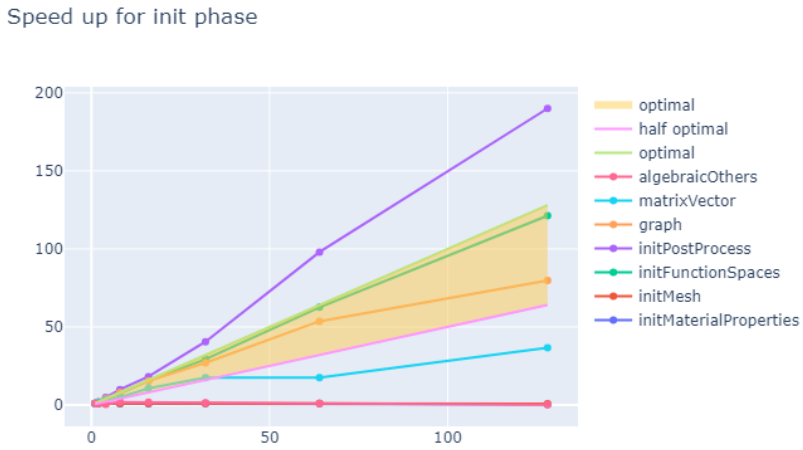


Figure 5: Gaya, speedup for init-phase partial references

On this illustration, we can figure out why, in the first graph, the *init* phase doesn't scale well. The *initMesh* is running in sequential, as the other curves fit in a good way to larger models. This has of course a huge impact on the whole *init* performance.

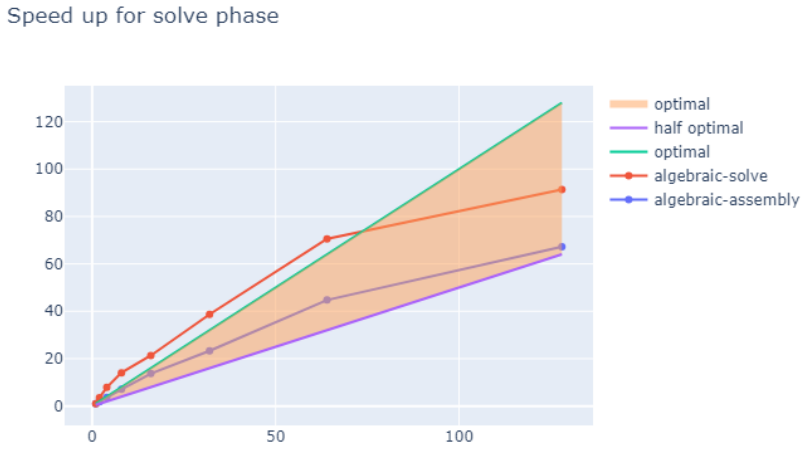


Figure 6: Gaya, speedup for solving phase partial references

Even if we could think there was a problem for the values of the *algebraic-solve*, because it was upon the optimal values. The fact that it decreased with augmenting tasks number is a good sign for its aptitude to scale well.

5.2 Multi-node test

For multi-node, we set the number of task per core to the maximum of available physical CPUs. The total number of tasks is the product from this value with the number of used nodes.

Our benchmarking process pointed out a recurrent issue regarding the use of the `MPI` library in the toolboxes, which was resolved by Mr. Chabannes.

```
   terminate called after throwing an instance of 'boost::wrapexcept<boost::mpi::exception>'
  what():  MPI_Test: MPI_ERR_TRUNCATE: message truncated
*** Aborted at 1716774707 (unix time) try "date -d @1716774707" if you are using GNU date ***
terminate called after throwing an instance of 'boost::wrapexcept<boost::mpi::exception>'
*** SIGABRT (@0x10fa00158edd) received by PID 1412829 (TID 0x7fc557cab000) from PID 1412829; stack trace: ***
```

For our study case, we will take a closer look at the proneEye-M2.json configuration file. This case has been run with the help of the `heatfluid` toolbox . Note that for heatfluid, which is a multi-physics toolbox, there are scalability reports regarding each mono-physic toolbox beside the classic ones.

This graph represents the main performances values of each execution stage. Here, we can see the most interesting part for us, the `HeatFluidSolve` stage, scales well until 128 CPUs. Over this value, performances decrease according to Amdahl's Law, as communication overhead and non-parallel tasks begin to reduce the benefits of adding more processors, causing a drop in efficiency. It is also possible to figure out that some other stage performances decrease with higher task number, like the `FluidMechanicsConstructor`.

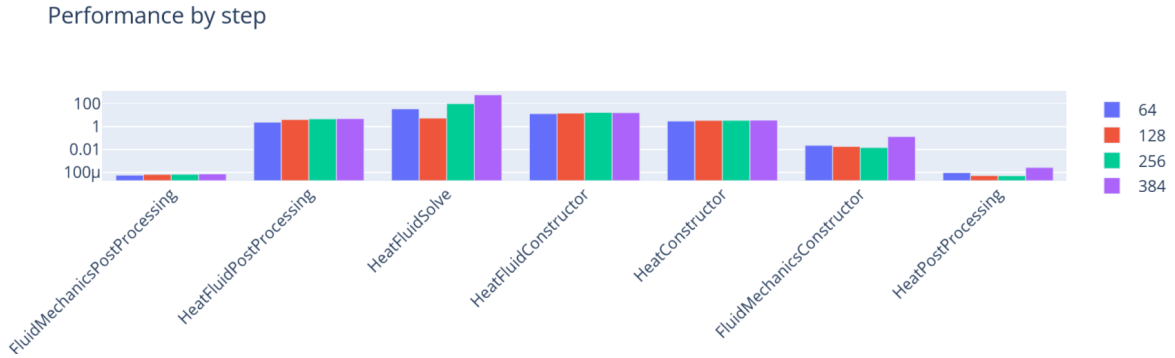


Figure 7: Gaya, Prone Eye by step

Let's now have a closer look at the `fluidMechanicsConstructor` phase.
On this picture of the graph, we have taken the choice to display the regressed curves (the origin should have been involved the frame for closer results) We also have a second button for displaying optimality curves (linear, half-linear and the area between them).

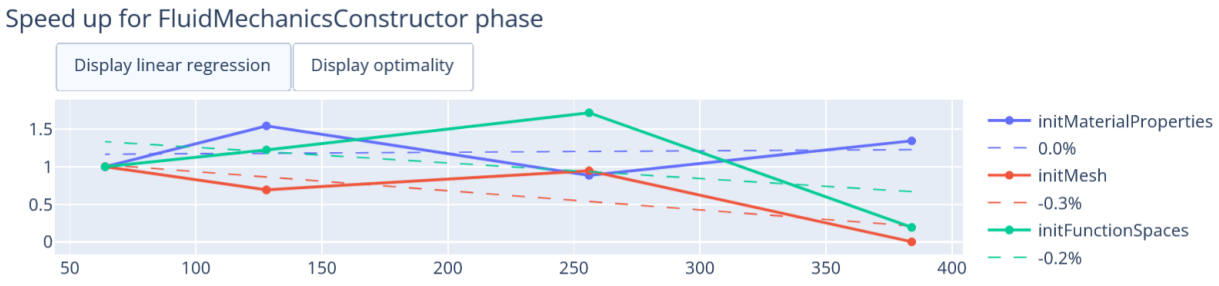


Figure 8: Gaya, fluidMechanicsConstructor parts

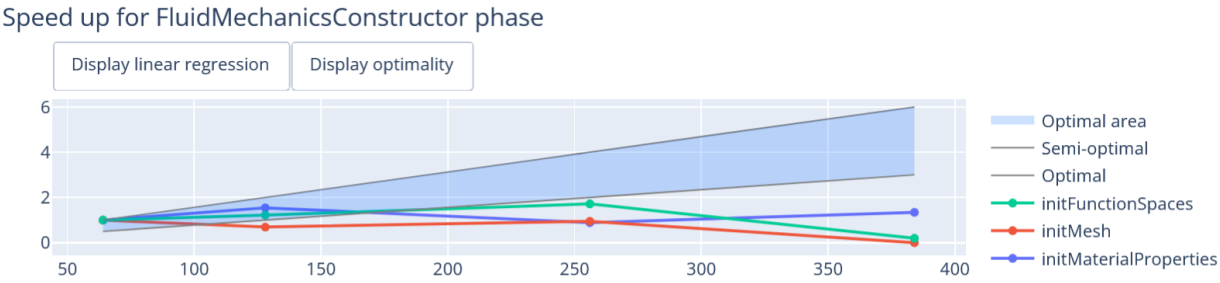


Figure 9: Gaya, fluidMechanicsConstructor parts with optimality

6 Bibliography

[1] Antora. *Documentation*. https://docs.antora.org/antora/latest/.

[2] Feel++ Consortium. *Documentation*. https://docs.feelpp.org/home/index.html.

[3] ReFrame HPC. *Documentation*. https://reframe-hpc.readthedocs.io/en/stable/.

[4] Jinja2. *Documentation*. https://palletsprojects.com/p/jinja/.

[5] Vasileios Karakasis. *ReFrame Webinar 2022*. https://www.cscs.ch/fileadmin/user_upload/contents_publications/tutorials/cscs-webinar-2022-final.pdf.

[6] NumPEx. *Documentation*. https://numpex.org/fr/.

[7] Plotly. *Documentation*. https://plotly.com/python/.