Université de Strasbourg

ExaMA WP3 – Dashboard Performances

Tanguy Pierre

Supervisors: V. Chabannes

August 22, 2024

Introduction

- Part of the *ExaMA* project from NumPEx
- **M**ethods and **A**lgortihms for exascale
- 1 exaflop $= 10^{18}$ floating-point operation per second
- Need for robust algorithms
- In between WP3 and WP7 *(Benchmarking for solvers)*

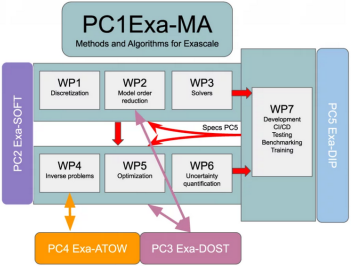


Figure: ExaMA organisation

Benchmarking

Great power doesn't necessary mean great performances...

Algorithms may experience terrible performances due to complex communication.
⇒ need to control their behaviour with increasing task number or problem complexity increasment

Benchmarking is for:

- performances comparison
- transparency about the evaluation process
- data analysis depending on context
- storing results as reference
- identifying performance decline and guiding the dev-team

Objectives

- Establish a **Continuous Integration/Continuous Deployment** workflow
- Enhance the **Dashboard Presentation** for comprehensive and interactive data visualization
- Conduct **Representative Tests** for obtaining reliable results about the application's behaviour
- Provide a **Database** for easy access and retrieval of test results

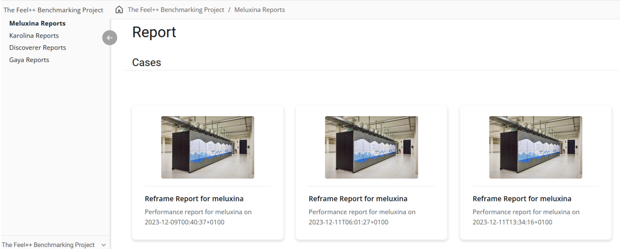


Figure: https://feelpp.github.io/benchmarking/benchmarking/index.html

Tools

- *ReFrame HPC*, a framework that allows to abstract from system's complexity in order to focus only on the algorithm's performance
- *Feel++*, a C++ library for Galerkin methods, especially its mono- and multi-physics toolboxes
- *Jinja2*, a template engine used for generating *.adoc* files
- *Antora*, a documentation site generator that handles code blocks and their execution(*.adoc* to *.html*)
- *Plotly*, the well-known data visualization library will be used for its responsiveness and flexibility

Process

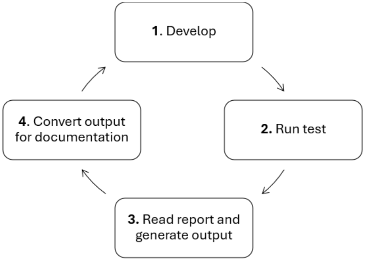


Figure: Process cycle

1. Develop new features or enhance *Feel++* applications
2. Use *ReFrame* for launching multiple tests at once
3. *Jinja2* will generate *.adoc* files containing code blocks with *ReFrame's* data
4. *AsciiDoctor* will convert the *.adoc* files into *.html* and Antora will update the documentation site

Process

It's evident that this task involves significant repetition.
$\Rightarrow$ we want the process to work on the most autonomous way.

Here follows the guideline of interactions between the different scripts:

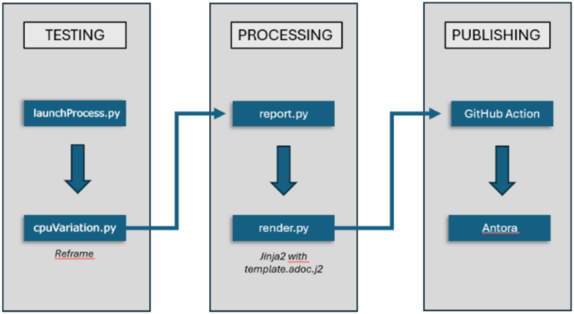


Figure: Workflow interaction

Launching

Regarding the launching, we first had to determine the most efficient approach to do it.

1. Simple bash script for discovering *ReFrame's* environment by launching one parametrized test at a time
2. Extension of this script by adding command line options:
   - difficult options handling
   - only one `.cfg` file for the *Feel++* executable
   - same parametrization *(CPU number, mesh size, ...)*
3. *JSON* configuration file for case specific parametrization to be read in *Reframe's* test class
4. Configuration reader and *Reframe* in Python
   $\Rightarrow$ transcript the launcher from bash to Python for simplicity
   $\Rightarrow$ *argparse* Python library for easier arguments handling

Launcher comparison


```bash
-tb|--toolbox)
if [[ -n "$2" ]]; then
  toolboxes=($(split_arguments "$2"))
  for tb in "${toolboxes[@]}"; do
    if \
    ![["␣${valid_toolboxes[@]}␣" =~ "␣$tb␣"]];
    then
      echo "Error:␣$tb␣toolbox␣is␣unknown"
      echo "Valids:␣${valid_toolboxes[@]}"
      exit 0
    fi
  done
  shift 2
else
    echo "Error:␣--toolbox␣requires␣an␣argument"
    exit 0
fi
```

```python
if args.dir:
for dir in args.dir:
    path = os.path.join(dir, '**/*.json')
    jsonFiles = glob.glob(path, recursive=True

    for file in jsonFiles:
        basename = os.path.basename(file)
        if args.exclude and basename in args.e
            continue
        if args.config and basename not in arg
            continue
        configLst.append(file)
```

- Only for checking if a name is in a list
- Complex path operators

- Builds the complete list of files that will be launched
- Simple paths comparison/building

Listing: Launcher script comparison, bash and Python

Benchmarking with ReFrame

Context

- Manages to launch multiple tests at once
- Needs a system configuration file for: modules, CPU number, launcher, scheduler, ...
- Exact timeline
  $\Rightarrow$ decorators for specifying when to launch a particular function
- "Sanity functions" for verifying a test's integrity, but also for extracting performances with *regex patterns*

As we will launch tests with the `-bind-to core` option from *mpiexec* for maximal efficiency, we are particularly interested in the *number of physical CPUs* on each node and the *number of nodes*.

Benchmarking with ReFrame

CPU Parametrization

```
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: Task number parametrization, adapted from[1]

- `reframe.core.runtime` module to access the host's topology
- Parameterization starts with `minCPU`-task and increases by power of 2 up to the number of physical CPUs
- Then, tasks increase by increments of number of physical CPUs up to `maxCPU * maxNodes`.

[1] Vasileios Karakasis. *ReFrame Webinar 2022*.

Benchmarking with ReFrame

Values extraction

Tanguy PIERRE

```python
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: Performance values extraction

- `findScaleFiles()` searchs for file containing `scalibility` and `.data`
- `valPattern` is enclosed by () for specifying extraction
- The ˆ and * characters guarantee that the expression starts on a new line and ends with any escape character for avoiding any unintended interactions

Benchmarking with ReFrame

Process extracted data

The `Report` class is used for extracting the performances.
It will load the ReFrame report and build following dataframes:
`df_perf, df_partialPerf, df_speedup, df_partialSpeedup`

- The *perf* dataframes are immediately built while reading as they don't need any calculation, they are provided in the report
- The *speedup* dataframes are based on a reference value, which is projected to a higher number of tasks.
- Reference value = performance from the test with the lowest number of tasks.

Some reports do not include values for partial performances. Therefore, we had to construct two different frames for organizing the data efficiently This organization facilitates comparisons between partial components of a stage, especially regarding plotting.

Results

Single node

The following study has been done on case3 from *Thermal Bridges ENISO10122*. This is as 3D case representing temperature distribution and heat flows through a wall-balcony junction.

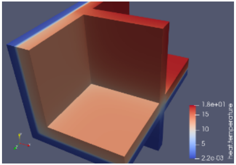


Figure: Case3, Thermal Bridges ENISO10122 Consortium, *Documentation*

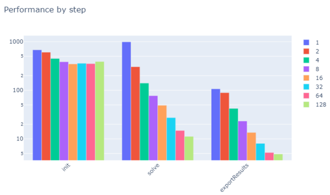
The case is using the 'benchConfigs/heat/ThermalBridgesCase3.json' configuration file. Calculation increasment by modifying some values located in the installed Feel++ Heat Toolbox testcases:

- *hsize* : 0.02 → 0.01
- *discretization* : $P1 \rightarrow P2$

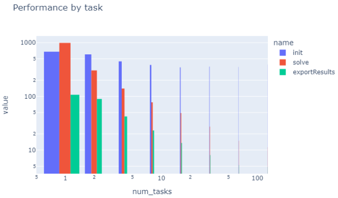
Results

Single node tests

As Gaya has 128 physical cpus per node, our test has been launched with 1, 2, 4, 8, 16, 32, 64 and 128 tasks.



(a) Gaya, Performances by step



(b) Gaya, Performances by task

Both graphics represents the 3 main performances metrics: init, solve and exportResults.
We can easily identify that both *solve*- and *exportResults*-stages do scale well, while it isn't the case for the *init*-phase.

Results

Single node tests

Let's look how the previous 3 main references behave regarding the speedup.

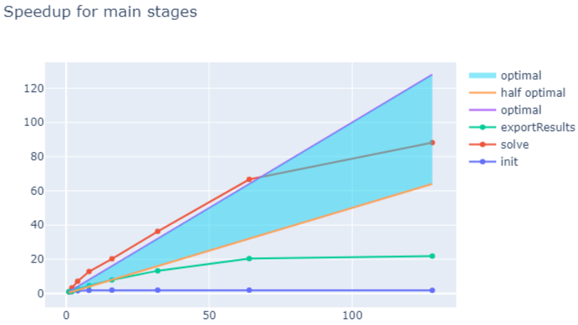
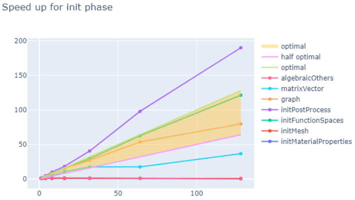


Figure: Gaya, speedup for main references

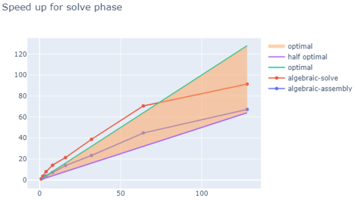
This graph correspond to the previous ones.
As we can see, the *init*-stage doesn't scale at all with increasing number of tasks.

Results

Single node tests



(a) Gaya, Performances by step



(b) Gaya, Performances by task

- Figure **(a)** shows the behaviour of every partial reference obtained through Feel++.
  $\Rightarrow$ *initMesh, initFunctionSpaces and algebraicOthers* jobs run in sequential
- In contrast, we see in figure **(b)** that every part of the *solve*-stage scale well. This will of course conduct to great performance for the whole stage.

Results

Multiple nodes tests

During the internship, we could point out that following bug was occurring when tests were launched on more than 1 node:

```
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: ***
```

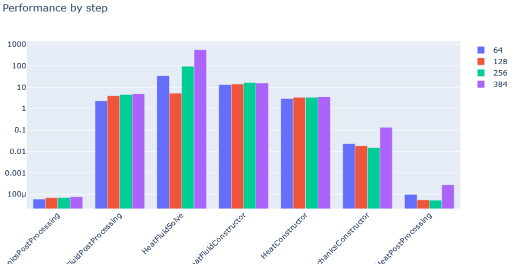
This bug was caused by troubles during MPI communication between the different cores.
By reporting it with *Reframe's* report, Mr. Chabannes could solve it.
This shows again the importance of doing regular test.

Multiple nodes tests

Main performances

The following study case is about the 'heatfluid' toolbox. As this is a multi-physics, there were more values to extract (and to process). This case calculates the temperature from a liquid inside human eyes, when in standing position.

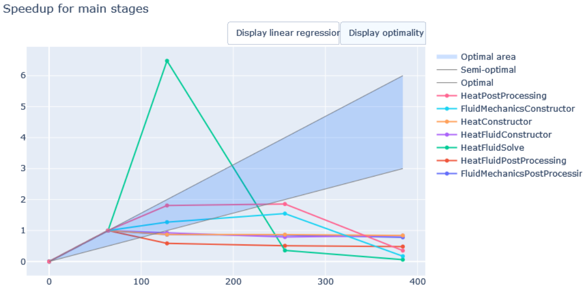
- Configuration-file: `benchConfigs/heatfluid/proneEye-M2-simple.json`
- CPU number: 64, 128, 256, 384
- Solver: lsc
- Mesh index: M2



Multiple nodes tests

Main performances

Speedup graphs have the capacity to show or not regression lines, but also the optimal area for the performance value.
Most values are scaling well until 256 CPUs ⇒ the application has reached a plateau.

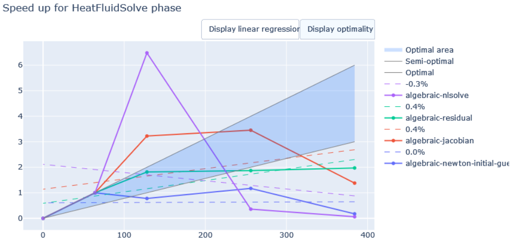


num_tasks	name	value	linearRegression	slope
384	1echanicsPostProce	0.782	0.954	0.001
384	atFluidPostProcess	0.482	0.574	0.000
384	HeatFluidSolve	0.061	0.879	-0.003
384	leatFluidConstructo	0.836	0.974	0.001
384	HeatConstructor	0.842	1.003	0.001
384	dMechanicsConstru	0.172	0.856	0.000
384	HeatPostProcessing	0.353	1.167	0.001
256	1echanicsPostProce	0.853	0.805	0.001

Multiple nodes tests

Partial performances

As we have noticed before, the 'heatFluidSolve' part had a weird behaviour when running on 128 CPUs.
This graph represents every performance of the problematic stage:



num_tasks	name	value	linearRegression	slope
384	raic-newton-initial-	0.174	0.649	0.000
256	raic-newton-initial-	1.171	0.634	0.000
128	raic-newton-initial-	0.778	0.620	0.000
64	raic-newton-initial-	1.000	0.613	0.000
0	raic-newton-initial-	0.000	0.606	0.000
384	algebraic-jacobian	1.379	2.689	0.004
256	algebraic-jacobian	3.453	2.173	0.004
128	algebraic-jacobian	3.225	1.656	0.004
64	algebraic-jacobian	1.000	1.398	0.004

Figure: Prone Eye, Partial performances

Bibliography

Antora. *Documentation*. https://docs.antora.org/antora/latest/.

AsciiDoctor. *Documentation*. https://docs.asciidoctor.org/.

Consortium, Feel++. *Documentation*. https://docs.feelpp.org/home/index.html.

HPC, ReFrame. *Documentation*. https://reframe-hpc.readthedocs.io/en/stable/.

Jinja2. *Documentation*. https://palletsprojects.com/p/jinja/.

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

NumPEx. *Documentation*. https://numpex.org/fr/.

Plotly. *Documentation*. https://plotly.com/python/.

Exemple de code en deux colonnes

```c
void code()
{
    nothing...
}
```

```c
void code()
{
    WHAT HAPPENED
}
```

Listing: Codes d'exemple en C