A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems: literate and reproducible environment

Emmanuel Agullo, Marek Felšöci, Guillaume Sylvand

A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems: literate and reproducible environment

Emmanuel Agullo, Marek Felšöci, Guillaume Sylvand

TECHNICAL REPORT
N° 0513
June 2021
Project-Team HiePACS

ISSN 0249-0803 ISRN INRIA/RT--0513--FR+ENG

A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems: literate and reproducible environment

Emmanuel Agullo*, Marek Felšöci†, Guillaume Sylvand‡

Project-Team HiePACS

Technical Report n° 0513 — June 2021 — 100 pages

Abstract: This is an accompanying technical report for the *A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems* Inria research report №9412. Based on the principles of literate programming, this technical report aims at providing detailed guidelines for reproducing the experiments of that research report. We use Org mode for literate programming and GNU Guix for software environment reproducibility. Note that part of the software involved is proprietary.

Key-words: reproducible, literal programming, Guix, Org mode

* Inria Bordeaux Sud-Ouest (emmanuel.agullo@inria.fr)
† Inria Bordeaux Sud-Ouest (marek.felsoci@inria.fr)
‡ Airbus Central R&T / Inria Bordeaux Sud-Ouest (guillaume.sylvand@airbus.com)

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Une comparaison de solveurs choisis pour la résolution de systèmes linéaires couplés FEM/BEM résultant de la discrétisation de problèmes aéroacoustiques: un environnement expérimental reproductible par programmation lettrée

Résumé : Ce document représente un rapport technique complémentaire au rapport de recherche Inria *Une comparaison de solveurs choisis pour la résolution de systèmes linéaires couplés FEM/BEM résultant de la discrétisation de problèmes aéroacoustiques* portant le numéro 9412. Basé sur les principes de la programmation lettrée, ce rapport technique vise à fournir des indications détaillées pour reproduire les expérimentations du rapport de recherche. Nous utilisons Org mode pour faire de la programmation lettrée et GNU Guix pour assurer la reproducibilité de l'environement logiciel expérimental. Notons que certains logiciels sont propriétaires.

Mots-clés : reproductible, programmation lettrée, Guix, Org mode

Contents

1 Introduction

Reproducibility of numerical experiments has always been a complex matter. Building exactly the same software environment on multiple computing platforms may be long, tedious and sometimes virtually impossible to be done manually.

Within our research work, we put a strong emphasis on ensuring the reproducibility of the experimental environment. On one hand, we make use of the GNU Guix transactional package manager [5] allowing us to define and manage reproducible software environments. On the other hand, we rely on the principles of literate programming [22] in an effort to maintain an exhaustive, clear and accessible documentation of our experiments and the associated environment in Org mode [19, 4].

The goal of this research report is to provide such a documentation of our study entitled 'A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems' [17]. Especially, we aim to describe and explain here all of the source code and procedures involved in the construction of the experimental software environment, the execution of benchmarks as well as the gathering and the post-processing of the results.

The present document is organized as follows. In Section 2, we provide a set of guidelines on how to assemble the elements of this report to reproduce the software environment and the numerical experiments on the target high-performance computing platform. In Section 3, we introduce the concept of literate programming and the associated tools. In Section 4, we present GNU Guix and how it can be used to build a reproducible software environment. In Section 5 we detail the design of our numerical experiments and in Section 6, we describe how we collect and process the results. We conclude in Section 7.

2 How to use this document

We include here a set of guidelines for reproducing the experimental software environment we describe in this technical report as well as the experiments presented in [17] on the PlaFRIM platform [12]. We assume that the user has access to the platform as well as to the source code of the proprietary Airbus packages being part of the software environment (see sections 4 and 4.2). All the shell commands below shall be run from `$HOME/environment` unless otherwise stated.

We begin by acquiring the Org source of the present document using the `wget` tool and saving it as `environment.org` in `$HOME/environment`. If this output path does not exist yet, we create it and navigate there before downloading. Note that we execute the download command in an appropriate Guix environment.

```
mkdir -p $HOME/environment
cd $HOME/environment
guix environment --pure --ad-hoc wget -- wget \
     https://mfelsoci.gitlabpages.inria.fr/thesis/attachments/RT-0513.org
mv RT-0513.org environment.org
```

To tangle (see Section 3) the source code from this Org document, we enter a Guix environment containing the necessary packages and use the command line interface of the Emacs text editor.

```
guix environment --pure --ad-hoc emacs emacs-org -- emacs --batch \
     --no-init-file -l org --eval \
'(progn (setq org-src-preserve-indentation t) (org-babel-tangle-file
↪  "environment.org")'
```

The tangled source files should appear in `$HOME/environment/tangle`. Next, we have to setup the Guix software channels (see Section 4.2).

```
mkdir -p $HOME/.config/guix
cp $HOME/environment/tangle/channels.scm $HOME/.config/guix/channels.scm
guix pull
```

Before running the benchmarks, we have to set up the `gcvb` filesystem (see Section 5.3.1). It should appear at `$HOME/benchmarks/rr-2020`. The `mkgcvbfs.sh` script has to be run directly from the `$HOME/environment/tangle` directory. Notice that we use the `setenv.sh` script (see Section 4.3) to enter to the appropriate Guix environment.

```
cd $HOME/environment/tangle
$(./setenv.sh -ge 'fs') -- ./mkgcvbfs.sh -f ./rr-2020.fstab \
                           -o $HOME/benchmarks/rr-2020
cd $HOME/environment
```

We also need to explicitly acquire the source tarballs of the Airbus packages `hmat`, `mpf` and `scab`. They should be placed under `$HOME/src`. One must ensure that there is an SSH agent running on the system and that the SSH key necessary to acces the Airbus Git repositories has been imported.

```
./tangle/setenv.sh -p
```

At this point, we can navigate to the root of the `gcvb` filesystem, generate a new benchmark session (see Section 5.1)

```
cd $HOME/benchmarks/rr-2020
$(./scripts/setenv.sh -ge 'benchmark') -- python3 -m gcvb generate \
                                       --yaml-file rr-2020.yaml
```

and submit (see Section 5.9) all the benchmark jobs to the Slurm scheduler. Using the packages `starpu` and `starpu-fxt` in the same environment would be conflicting. Therefore, we launch the benchmarks depending on `starpu-fxt` in a separate environment.

```
$(./scripts/setenv.sh -ge 'benchmark') -- ./scripts/submit.sh -s results/1 \
                                       -E '^fxt'
$(./scripts/setenv.sh -gxe 'benchmark') -- ./scripts/submit.sh -s results/1 \
                                        -F '^fxt'
```

Once all the benchmark jobs finish, we can go back to the `$HOME/environment` directory, gather the global results (see Section 6.1.1)

```
cd $HOME/environment
$(./tangle/setenv.sh -ge 'gather') -- ./tangle/gather.R \
                                   $HOME/benchmarks/rr-2020/results/1 \
                                   rr-2020.csv
```

and extract some additional data (see Section 6.1.2). The corresponding files shall appear in `$HOME/environment`.

```
$(./tangle/setenv.sh -ge 'extract') -- ./tangle/extract.sh \
  -B multi-solve-1-mumps-spido-256-250000 \
  -B multi-factorization-1-mumps-spido-250000-7418 \
  -s $HOME/benchmarks/rr-2020/results/1 -d $HOME -rt
```

Eventually, we can generate one or more figures and visualize the results. To do so, we ensure the output folder for the figures `$HOME/environment/figures/rr-2020` and open the Org source of this report in Emacs.

```
mkdir -p $HOME/environment/figures/rr-2020
$(./tangle/setenv.sh -ge 'post_process') -- \
                                        emacs $HOME/environment/environment.org
```

Then, we navigate to Section 6.2.10, choose a figure to plot, place the cursor on the corresponding R source code block and issue the key sequence `Ctrl+C`, `Ctrl+V` and `Ctrl+E` to evaluate the latter and produce the output `*.svg` vector image file.

3 Literate programming

We choose to write the source code related to the design and the automatization of our numerical experiments in respect of the paradigm known as literate programming [22]. The idea of this approach is to associate source code with an explanation of its purpose written in a natural language.

There are numerous software tools for literate programming. We rely on Org mode for the Emacs text editor [19, 4] which defines the Org markup language. The latter combines formatted text, images and figures with traditional source code. Files containing Org documents typically end with the `.org` extension. Furthermore, an Org document can be exported to various output formats [9] for better visualization and presentation such as LaTeX, Beamer, HTML and so on. See an example Org document in Figure 1.

Extracting a compilable or interpretable source code from an Org document is referred to as tangling [10]. It is also possible to evaluate a particular source code block directly from the Emacs editor [8] while editing. This is particularly useful for a quick visualization of experimental results. See an example in Figure 2. Notice the usage of the `noweb` syntax [11] in the example, i. e. `«dataframe-sparse-scalability»`. This expression is a reference to a code block defined elsewhere in the document (for instance, see Section 6.2.6) and allow us to include and reuse it here without having to duplicate the source code.

The Org source of the hereby document features the source code blocks of multiple scripts and configuration files that needs to be tangled into separate source files before constructing the software environment and realizing the experiments yielding the results presented in [17].

```
Memory usage statistics of a particular
process are stored in ~/proc/<pid>/statm~
where ~<pid>~ is the process identifier
(PID). In this file, the field =VmRSS=
holds the amount of residual memory
used by the process at instant $t$.
See the associated function below.

#+BEGIN_SRC python
def rss(pid):
    with open("/proc/%d/statm" % pid, "r")
        line = f.readline().split();
        VmRSS = int(line[1])
        return VmRSS
#+END_SRC
```

(a) Org source

Memory usage statistics of a particular process are stored in `/proc/<pid>/statm` where `<pid>` is the process identifier (PID). In this file, the field `VmRSS` holds the amount of residual memory used by the process at instant t. See the associated function below.

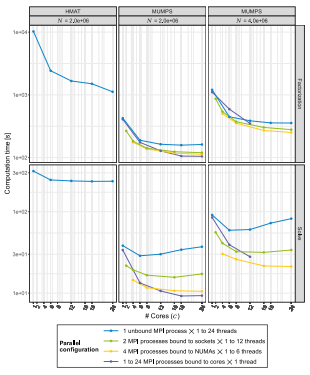
```python
def rss(pid):
    with open("/proc/%d/statm" % pid,
    ↪  "r") as f:
        line = f.readline().split();
        VmRSS = int(line[1])
        return VmRSS
```

(b) LaTeX PDF output

FIGURE 1: Excerpt of an Org document describing a Python script.

```
#+NAME: code:scalability-fem
#+HEADER: :results output graphics file :exports both
#+HEADER: :width 10 :height 12
#+HEADER: :file ../figures/rr-2020/scalability-fem.svg
#+BEGIN_SRC R :noweb no-export
<<dataframe-sparse-scalability>>
scalability(dataframe_sparse_scalability)
#+END_SRC
```

(a) R source code block



(b) Result of evaluation

FIGURE 2: Example of an R source code block yielding a figure as a result.

4 Building reproducible software environments

We are likely to work on various computing platforms and need to manage multiple software packages in multiple versions and their dependencies across different machines and computation platforms. To keep our software environment as much reproducible as possible, we use the GNU Guix [5] package manager. However, we are confronted to two major limitations in our effort. On one hand, we rely on some non-free software packages the access to the source code of which is restricted, e. g. Intel(R) Math Kernel Library. On the other hand, part of the source of the Airbus software we work on is proprietary and can not be disclosed to the public due to industrial constraints.

4.1 GNU Guix

Guix is a transactional package manager and a stand-alone GNU Linux distribution where each user can install his or her own packages without any impact on the others and with the possibility to switch between multiple system generations and package versions. Moreover, a software environment created using Guix is generally bit-for-bit reproducible.

4.2 Channels

Software packages in Guix are available through dedicated Git repositories containing package definitions. These repositories are called channels.

The first and default channel is the system channel providing Guix itself as well as the definitions of some commonly used packages such as system libraries, compilers, text editors and so on. Afterwards, we need to include additional channels using a custom channel file `channels.scm` written in the Scheme language.

For each channel, we specify the commit of the associated repository to acquire. This way, we make sure to always build the environment using the exact same versions of every single package in the system and guarantee the reproducibility of the environment.

```
(list
  (channel
    (name 'guix)
    (url "https://git.savannah.gnu.org/git/guix.git")
    (commit "1ac4959c6a94a89fc8d3a73239d107cfb1d96240"))
```

Following the system channel, we include `guix-hpc` and `guix-hpc-non-free` providing various open-source and non-free scientific software and libraries.

```
  (channel
    (name 'guix-hpc)
    (url "https://gitlab.inria.fr/guix-hpc/guix-hpc.git")
    (commit "9cc4593aaaaeaf17a602b620e9ab1974b5b82984"))
  (channel
    (name 'guix-hpc-non-free)
    (url "https://gitlab.inria.fr/guix-hpc/guix-hpc-non-free.git")
    (commit "ddc9a7af42203a0135da67f4435b622ce4706528"))
```

The implementation of the Airbus solver framework is not open-source. Therefore, some of the Airbus packages we use, for instance `hmat`, `mpf` and `scab`, are defined in the private channel

`guix-hpc-airbus`. To access the package definitions in the latter, one must have access to the corresponding Git repository and duly configurated SSH credentials on the target machine.

```
  (channel
    (name 'guix-hpc-airbus)
    (url "git@gitlab.inria.fr:mfelsoci/guix-hpc-airbus.git")
    (commit "bff0aa3f20140dbfda53d6882e25d3e9770e2911")))
```

4.3 Environments

To enter a particular software environment using Guix, we use the command `guix environment`. The latter allows us to specify the packages to include together with the desired version, commit or branch.

To avoid typing lot of command line options anytime we want to enter the environment, we define the `setenv.sh` shell script to automatize the process. It also allows us to adjust the environment using specific options.

We use the benchmark suite `test_FEMBEM` from Airbus provided in one of their proprietary software packages, namely `scab`. An independent open-source version of `test_FEMBEM` is available too [15]. `test_FEMBEM` allows us to easily evaluate various solver configurations.

`scab` as well as its direct dependencies `mpf` and `hmat` are proprietary packages with closed source residing in private Git repositories. Before setting up the environment, one has to acquire a local copy of their respecitve sources so that Guix can build the environment.

The `setenv.sh` script begins with a help message function accessible through the `-h` option.

```
function help() {
  echo "Set up the experimental environment for the Ph.D. thesis of Marek" \
       "Felšöci." >&2
  echo "Usage: $0 [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -A                Instead of the mainstream version of 'hmat', use" \
       "its derived version containing the developments made by Aurélien" \
       "Falco during his Ph.D. thesis." >&2
  echo "  -e ENVIRONMENT    Switch the software environment to ENVIRONMENT." \
       "Available environments are: 'benchmark' (include packages for" \
       "performing benchmarks, default choice), 'fs' (include packages for" \
       "intializing gcvb filesystem using the mkgcvbfs.sh script), 'gather'" \
       "(include packages for gathering results from multiple '*.csv' files" \
       "into a single dataframe) and 'post_process' (include packages for" \
       "post-processing)" >&2
  echo "  -g                Get the final 'guix environment' command line" \
       "that can be followed by an arbitrary command to be executed inside" \
       "the environment. The trailing '--' should be added manually!" >&2
  echo "  -h                Show this help message." >&2
  echo "  -O                Use OpenBLAS instead of Intel(R) MKL." >&2
  echo "  -p                Prepare the source tarballs of the Airbus" \
       "packages. If combined with '-r', the tarballs shall be placed into" \
       "the directory specified by the latter. If combined with '-A', the" \
       "tarballs corresponding to the package versions of the Aurélien" \
       "Falco's fork of 'hmat' shall be prepared." >&2
  echo "  -r ROOT           Search for Airbus sources at ROOT in lieu of the" \
       "default location at '$HOME/src'." >&2
  echo "  -x                Use StarPU with the FXT support." >&2
}
```

Follows a generic error message function. The error message to print is expected to be the first argument to the function. If not present, a generic message is displayed.

```
function error() {
  if test $# -lt 1;
  then
    echo "An unknown error occurred!" >&2
  else
    echo "Error: $1" >&2
  fi
}
```

The variable `WITH_INPUT_STARPU_FXT` contains the `--with-input` option for the `guix environment` command. This replaces the usage of the ordinary StarPU [18] package with the one providing FXT trace (see Section 6.2.3) generation support whenever the `-x` option is passed on the command line.

```
WITH_INPUT_STARPU_FXT=""
```

Similarly, the `WITH_INPUT_MKL` variable contains the `--with-input` options necessary to replace the usage of OpenBLAS [7] by Intel(R) Math kernel library [6] in concerned packages.

```
WITH_INPUT_MKL="--with-input=pastix-5=pastix-5-mkl \
--with-input=mumps-scotch-openmpi=mumps-mkl-scotch-openmpi \
--with-input=openblas=mkl
"
```

`AIRBUS_ROOT` specifies the location where to search for the source tarballs of the Airbus packages. This can be modified using the `-r` option. The default value is `$HOME/src`.

```
AIRBUS_ROOT="$HOME/src"
```

By default, the script suppose the tarballs already exist and tries to set up the environment directly. `PREPARE_TARBALLS` is a boolean switch indicating whether the source tarballs of the Airbus packages should be generated first before setting up the environment. For machines without access to the associated private Airbus repositories, the tarballs can be generated on an another machine using the `-p` option.

```
PREPARE_TARBALLS=0
```

Normally, we want to set up the environment using the mainstream release of `hmat`. The `-A` option and the associated boolean `AIRBUS_AF` allows to switch to the version of HMAT containing the developments made by Aurélien Falco [20] that have not been merged into the mainstream version of the package yet and are thus not assessed in our study.

```
AIRBUS_AF=0
```

At this stage, we are ready to parse the options and check the validity of option arguments where applicable.

```
GET_COMMAND=0

ENVIRONMENT="benchmark"

while getopts ":Ae:gOhpr:x" option;
do
  case $option in
    A)
      AIRBUS_AF=1
      ;;
```

The `-e` option allows to choose among multiple software environments.

```
    e)
      ENVIRONMENT=$OPTARG
      ;;
```

The `-g` option allows to print out the final `guix environment` command instead of directly entering the environment. This is useful for writing one-line commands, for example, in the continuous integration configuration.

```
    g)
      GET_COMMAND=1
      ;;
    O)
      WITH_INPUT_MKL=""
      ;;
    p)
      PREPARE_TARBALLS=1
      ;;
    r)
      AIRBUS_ROOT=$OPTARG

      if test ! -d "$AIRBUS_ROOT";
      then
        error "'$AIRBUS_ROOT' is not a valid directory!"
        exit 1
      fi
      ;;
    x)
      WITH_INPUT_STARPU_FXT="--with-input=starpu=starpu-fxt"
      ;;
```

We must also take into account unknown options, missing option arguments, syntax mismatches as well as the case when the `-h` option is specified.

```
    \?)
      error "Arguments mismatch! Invalid option '-$OPTARG'."
      echo
      help
      exit 1
      ;;
    :)
      error "Arguments mismatch! Option '-$OPTARG' expects an argument!"
      echo
      help
      exit 1
      ;;
```

```
    h | *)
      help
      exit 0
      ;;
  esac
done
```

The following variables indicate the commit numbers, branch names and archive locations to use by default for the generation of the Airbus source tarballs.

```
HMAT_BASENAME="hmat-git.81db556"
HMAT_TARBALL="$AIRBUS_ROOT/$HMAT_BASENAME.tar.gz"
HMAT_COMMIT="81db556434c3c5b0b30418392d352827ac16ed82"
HMAT_BRANCH="master"
MPF_BASENAME="mpf-git.fec66d4"
MPF_TARBALL="$AIRBUS_ROOT/$MPF_BASENAME.tar.gz"
MPF_COMMIT="fec66d43b4aba7fc18988f05106259d4c17a0bd2"
MPF_BRANCH="master"
SCAB_BASENAME="scab-git.297fe52"
SCAB_TARBALL="$AIRBUS_ROOT/$SCAB_BASENAME.tar.gz"
SCAB_COMMIT="297fe52cab22ab4e2f1277975c683fce89215440"
SCAB_BRANCH="master"
```

However, when the `-A` option is used, we need to update these specifications accordingly.

```
if test $AIRBUS_AF -ne 0;
then
  HMAT_BASENAME="hmat-af-git.60743b1"
  HMAT_TARBALL="$AIRBUS_ROOT/$HMAT_BASENAME.tar.gz"
  HMAT_COMMIT="60743b119afb966b6987c4421b949482dfbbf04f"
  HMAT_BRANCH="mf/af/bcsf"
  MPF_BASENAME="mpf-af-git.6fbcc2e"
  MPF_TARBALL="$AIRBUS_ROOT/$MPF_BASENAME.tar.gz"
  MPF_COMMIT="6fbcc2e67713205b1c3bbebdda1f6377d9698070"
  MPF_BRANCH="af/devel"
  SCAB_BASENAME="scab-af-git.2971244"
  SCAB_TARBALL="$AIRBUS_ROOT/$SCAB_BASENAME.tar.gz"
  SCAB_COMMIT="29712447d805d957d715a1f5ab7d8e6903997652"
  SCAB_BRANCH="af/ND"
fi
```

If the `-p` option is specified, we get a clone of the Airbus repositories and create the source tarballs of `hmat`, `mpf` and `scab` using the specified commit numbers and branch names before trying to setup up the environment.

```
if test $PREPARE_TARBALLS -ne 0;
then
```

We begin by removing any previous clones of the Airbus repositories in `AIRBUS_ROOT`.

```
  rm -rf $AIRBUS_ROOT/$HMAT_BASENAME $AIRBUS_ROOT/$MPF_BASENAME \
     $AIRBUS_ROOT/$SCAB_BASENAME $HMAT_TARBALL $MPF_TARBALL $SCAB_TARBALL
```

Then, we make fresh clones, checkout the required revisions

```
  git clone --recurse-submodules --single-branch --branch $HMAT_BRANCH \
      https://<some-private-server>/airbus/hmat $AIRBUS_ROOT/$HMAT_BASENAME
  cd $AIRBUS_ROOT/$HMAT_BASENAME
  git checkout $HMAT_COMMIT
  git submodule update

  git clone --single-branch --branch $MPF_BRANCH \
      https://<some-private-server>/airbus/mpf $AIRBUS_ROOT/$MPF_BASENAME
  cd $AIRBUS_ROOT/$MPF_BASENAME
  git checkout $MPF_COMMIT

  git clone --single-branch --branch $SCAB_BRANCH \
      https://<some-private-server>/airbus/scab $AIRBUS_ROOT/$SCAB_BASENAME
  cd $AIRBUS_ROOT/$SCAB_BASENAME
  git checkout $SCAB_COMMIT
```

and verify that the cloned repositories are valid directories.

```
  if test ! -d $AIRBUS_ROOT/$HMAT_BASENAME || \
      test ! -d $AIRBUS_ROOT/$MPF_BASENAME || \
      test ! -d $AIRBUS_ROOT/$SCAB_BASENAME;
  then
    error "Failed to clone the Airbus reporitories!"
    exit 1
  fi
```

We remove the `.git` folders from inside the clones to shrink the size of the final tarball created using the `tar` utility.

```
  rm -rf $AIRBUS_ROOT/$HMAT_BASENAME/.git \
     $AIRBUS_ROOT/$HMAT_BASENAME/hmat-oss/.git $AIRBUS_ROOT/$MPF_BASENAME/.git \
     $AIRBUS_ROOT/$SCAB_BASENAME/.git

  tar -czf $HMAT_TARBALL -C $AIRBUS_ROOT $HMAT_BASENAME
  tar -czf $MPF_TARBALL -C $AIRBUS_ROOT $MPF_BASENAME
  tar -czf $SCAB_TARBALL -C $AIRBUS_ROOT $SCAB_BASENAME
```

At the end of the procedure, we check if the tarballs were created and remove the clones.

```
  if test ! -f $HMAT_TARBALL || test ! -f $MPF_TARBALL || \
      test ! -f $SCAB_TARBALL;
  then
    error "Failed to create tarballs!"
    exit 1
  fi

  rm -rf $AIRBUS_ROOT/$HMAT_BASENAME $AIRBUS_ROOT/$MPF_BASENAME \
     $AIRBUS_ROOT/$SCAB_BASENAME
fi
```

Eventually comes the `guix environment` command itself. We use a variable to hold the name of the `scab` package as it changes to `scab-af` when the `-A` option is used.

Also note that, by the default, we need to use a different version of the `chameleon` package than the one available through the commit `9cc4593` of the `guix-hpc` channel.

```
SCAB="scab"
WITH_COMMIT_CHAMELEON="
--with-commit=chameleon=b1d809179e2a8de168996483858ec5438672e082
"
```

Nevertheless, it is not the case when building from the Aurélien Falco's fork of `hmat`.

```
if test $AIRBUS_AF -ne 0;
then
  SCAB="scab-af"
  WITH_COMMIT_CHAMELEON=""
fi
```

In order to access the additional features we implemented into the `gcvb` package (see Section 5), we switch to our fork of the package's repository. Sometimes, a local clone of the latter is necessary. Being hosted on GitHub, it can not be acquired online by Guix on some computing platforms having too restrictive proxy settings.

```
if test ! -d $AIRBUS_ROOT/gcvb;
then
  git clone https://github.com/felsocim/gcvb.git $AIRBUS_ROOT/gcvb
fi
```

The list of packages to include into the resulting environment as well as the options to pass to the `guix environment` command are based on the environment switch `-e`. Available environments are listed below. Note that, the `--preserve` option allows us to inherit selected environment variables from the parent environment.

- `benchmark`: environment for performing benchmakrs,

```
OPTIONS_BENCHMARK="$WITH_COMMIT_CHAMELEON $WITH_INPUT_MKL $WITH_INPUT_STARPU_FXT
--with-git-url=gcvb=$AIRBUS_ROOT/gcvb
--with-commit=gcvb=40d88ba241db4c71ac3e1fe8024fba4d906f45b1 --preserve=^SLURM"
PACKAGES_BENCHMARK="bash coreutils inetutils findutils grep sed bc slurm openmpi
openssh python python-psutil gcvb r r-starvz $SCAB"
```

- `fs`: environment for initializing benchmark filesystem using the `mkgcvbfs.sh` script (see Section 5.3),

```
PACKAGES_FS="bash coreutils"
```

- `gather`: environment for gathering benchmark results from multiple `*.csv` files into a single data frame,

```
OPTIONS_GATHER="--preserve=TZDIR"
PACKAGES_GATHER="r r-plyr r-dplyr r-readr"
```

- `extract`: environment for extracting additional benchmark results from a selected set of benchmarks using the script `extract.sh` (see Section 6.1.2)

```
PACKAGES_EXTRACT="bash coreutils sed python2"
```

- `post_process`: environment for post-processing benchmark results and publishing HTML and LaTeX documents.

```
OPTIONS_POST_PROCESS="--preserve=TZDIR"
PACKAGES_POST_PROCESS="bash sed which emacs emacs-org2web emacs-org
emacs-htmlize emacs-biblio emacs-org-ref emacs-ess python-pygments texlive r
r-plyr r-dplyr r-readr r-tidyr r-ggplot2 r-scales r-cowplot r-stringr
r-gridextra r-starvz inkscape@0.92"
```

Based on the value of `$ENVIRONMENT`, we select the environment to set up.

```
OPTIONS=""
PACKAGES=""

case $ENVIRONMENT in
  benchmark)
    OPTIONS="$OPTIONS_BENCHMARK"
    PACKAGES="$PACKAGES_BENCHMARK"
    ;;
  fs)
    PACKAGES="$PACKAGES_FS"
    ;;
  gather)
    OPTIONS="$OPTIONS_GATHER"
    PACKAGES="$PACKAGES_GATHER"
    ;;
  extract)
    PACKAGES="$PACKAGES_EXTRACT"
    ;;
  post_process)
    OPTIONS="$OPTIONS_POST_PROCESS"
    PACKAGES="$PACKAGES_POST_PROCESS"
    ;;
  *)
    error "'$ENVIRONMENT' is not a valid software environment switch!"
    exit 1
    ;;
esac
```

Now it is possible to assemble the `guix environment` command and its options. To unset any existing environment variables of the current environment, we use the `--pure` option. Then, the `--ad-hoc-` option includes all the packages, the list of which follows the option, in the resulting environment.

```
ENVIRONMENT_COMMAND="guix environment --pure $OPTIONS --ad-hoc $PACKAGES"
```

If the `-g` option is set, we only print the command on the standard output. Otherwise, we directly enter the new environment and launch a shell interpreter. The `--norc` option of bash prevents the sourcing of the current user's `.bashrc` file.

```
if test $GET_COMMAND -ne 0;
then
  echo $ENVIRONMENT_COMMAND
  exit 0
fi

$ENVIRONMENT_COMMAND -- bash --norc
```

5 Performing benchmarks

To automatize the generation and the computation of benchmarks, we use `gcvb` [2], an open-source tool developed at Airbus. `gcvb` allows us to define series of benchmarks, generate corresponding shell job scripts for every benchmark or a selected group of benchmarks, submit these job scripts for execution, then gather and optionally validate or post-process the results. To generate multiple variants of the same benchmark `gcvb` provides templates.

5.1 `gcvb`

`gcvb` uses a specific file and directory structure. There are two main Yaml files to configure and define a series of benchmarks. Both files must be placed in the same folder. Furthermore, the name of the configuration file must be `config.yaml`. On the other hand, the benchmark definition file may have an arbitrary name. The folder we place this couple of files in, for instance `$HOME/benchmarks/`, represents the root of the filesystem of our benchmark series where:

- `$HOME/benchmarks/rr-2020/`
  - `data/` contains data necessary to generate and perform benchmarks.
    - `all/` represents one of possibly more folders containing benchmark data. For the sake of simplicity, we use one single folder for all benchmarks.
      - `input` holds any input file necessary to generate benchmarks.
      - `references` holds any reference file needed for result validation.
      - `templates` provides file templates for template-based benchmarks. We use templates to produce specific batch job script header directives for the workload manager on the target computing platform (see Section 5.2). Each of the subfolders contains a script header template. Headers generated based on these templates are prepended to the final job scripts produced by `gcvb`.
        1. `monobatch/sbatch`
        2. `polybatch/sbatch`
        3. `coupled/sbatch`
        4. `scalability/sbatch`
  - `results/` contains benchmark results. Here, one subfolder is produced every time a new session of benchmarks is generated based on the definition file. It contains job scripts and one folder per generated benchmark. These may hold any templated-based input file as well as the result of the corresponding benchmark execution.
    - `1/`
    - `...`
  - `config.yaml` represents the configuration file.

- `gcvb.db` represents an auto-generated NoSQL database that can be used to store benchmark results.
- `rr-2020.yaml` represents the benchmark definition file.

5.2 `sbatch` template files

We use Slurm [14] to schedule and execute our experiments on the target high-performance computing platforms. `gcvb` produces a job script for each benchmark described in the definition file. This script is then passed to Slurm for to be scheduled on a computation node or nodes.

Each job script produced by `gcvb` is prepended with a header contaning the configuration statements for the `sbatch` command of Slurm [13] used to submit jobs for computation. We take advantage of the template feature in order to be able to dynamically generate `#SBATCH` headers specific to a given set of benchmarks.

An `sbatch` template begins as a standard shell script.

```
#! /usr/bin/env bash
#
# Slurm batch job script
#
```

We use multiple template files but most of the `#SBATCH` directives are common to all of them such as:

- the count of computational nodes to reserve,

```
#SBATCH -N {slurm[count]}
```

- the count of task slots per node to reserve,

```
#SBATCH -n {slurm[tasks]}
```

- the restriction on the node type depending on the node family name,

```
#SBATCH --constraint={slurm[node]}
```

- the exclusion of the other users from the usage of the reserved resources,

```
#SBATCH --exclusive
```

- the reservation time,

```
#SBATCH --time={slurm[time]}
```

- the location to place the slurm log files in where `%x` is the corresponding job name and `%j` the identifier of the job.

```
#SBATCH -o slurm/%x-%j.log
```

Note that, `{slurm[count]}` and so on represent placeholders for values replaced by actual values based on the benchmark definition file during template expansion (see Section 5.5).

The only `#SBATCH` directive specific to each template file is the job name. Based on the latter, we distinguish different sets of benchmarks. Grouping individual benchmarks into a single job script allows us to submit fewer jobs. This way, they are more balanced in terms of the computation time we need to allocate for them on the target computing platform. For example, instead of submitting 12 jobs having each the time limit of 10 minutes, we submit two jobs with the time limit of 1 hour each. Benchmarks to be placed into a common job script are identified by matching their job name against a regular expression.

In the `sbatch` header template `monobatch` used for benchmarks with all the jobs running on single computational node, the job name is simply composed of a prefix which typically corresponds to the constant part of a benchmark name (see Section 5.5).

```
<<sbatch-beginning>>
#SBATCH --job-name={slurm[prefix]}
<<sbatch-end>>
```

When a template-based benchmark definition yields a large amount of benchmarks, we prefer to group them into multiple job scripts and launch the latter in parallel. The value of `{job[batch]}` in the `polybatch` header determines which benchmark belongs to which job script.

```
<<sbatch-beginning>>
#SBATCH --job-name={slurm[prefix]}-{job[batch]}
<<sbatch-end>>
```

For coupled solver benchmarks, we need a more fine grained distribution of the latter among Slurm jobs. So, in the `coupled sbatch` header, we add to the job name also the names of sparse and dense solvers involved in the benchmark.

```
<<sbatch-beginning>>
#SBATCH --job-name={slurm[prefix]}-{job[batch]}-{sparse[name]}-{dense[name]}
<<sbatch-end>>
```

Same for scalability benchmarks. In `scalability`, we add to the job name the name of the solver being used.

```
<<sbatch-beginning>>
#SBATCH --job-name={slurm[prefix]}-{solver[name]}-{job[map]}
<<sbatch-end>>
```

At the end of the header, we add a couple of commands to get the time and date when the job was scheduled and on which node.

```
echo "Job scheduled on $(hostname), on $(date)"
echo
```

Also, we disable the creation of memory dump files in case of memory error. Even if they can be particulary useful, in some cases they consume too much disk space and prevent other jobs from running.

```
ulimit -c 0
```

5.3 Initializing filesystem

5.3.1 Initialization script

We wrote the shell script `mkgcvbfs.sh` to automatize the initialization of a `gcvb` filesystem or to check if a specific `gcvb` filesystem is valid.

Traditionally, the script begins with a help message function that can be triggered using the `-h` option.

```
function help() {
  echo "Initialize a gcvb file system described in FSTAB at FSPATH." >&2
  echo "Usage: ./$(basename $0) [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -h            Show this help message." >&2
  echo "  -c            Check if a valid gcvb filesystem is present in PATH." >&2
  echo "  -f FSTAB      Initialize the gcvb filesystem specified in FSTAB." >&2
  echo "  -o FSPATH     Set the output path for the filesystem to create." >&2
}
```

Then, we include a generic error function.

```
<<shell-error-function>>
```

The script requires an `.fstab` file describing the filesystem to create (see Section 5.3.2), e. g. the entries to initialize the filesystem with and the destination path of the latter.

`FSTAB` holds the path to an `.fstab` description file provided using the `-f` option.

```
FSTAB=""
```

`FSPATH` holds the destination path to create the filesystem in (see the `-o` option).

```
FSPATH=""
```

The `-c` option, corresponding to the `CHECK_ONLY` boolean variable, allows to check an existing `gcvb` filesystem against an `.fstab` description instead of creating it.

```
CHECK_ONLY=0
```

At this stage, we are ready to parse the options and check the validity of option arguments where applicable.

```
while getopts ":hcf:o:" option;
do
  case $option in
    c)
      CHECK_ONLY=1
      ;;
    f)
      FSTAB=$OPTARG

      if test ! -f $FSTAB;
      then
        error "'$FSTAB' is not a valid file!"
        exit 1
      fi
      ;;
    o)
      FSPATH=$OPTARG
      ;;
```

We must also take into account unknown options, missing option arguments, syntax mismatches as well as the case when the `-h` option is specified.

```
    \?) # Unknown option
      error "Arguments mismatch! Invalid option '-$OPTARG'."
      echo
      help
      exit 1
      ;;
    :) # Missing option argument
      error "Arguments mismatch! Option '-$OPTARG' expects an argument!"
      echo
      help
      exit 1
      ;;
    h | *)
      help
      exit 0
      ;;
  esac
done
```

Next, we have to check if the user has provided the path to the `.fstab` file

```
if test "$FSTAB" == "";
then
  error "No filesystem description file was specified!"
  exit 1
fi
```

as well as the destination path of the `gcvb` filesystem to create.

```
if test "$FSPATH" == "";
then
  error "No output location for the filesystem was specified!"
  exit 1
fi
```

Eventually, we process all of the entries in the `.fstab` description file. Each line represents a specification of an entry in the `gcvb` filesystem to initialize (see Section 5.3.2). Notice that to separate information in an entry specification we use colons.

```
for entry in $(cat $FSTAB);
do
```

The first information tells us whether a file or a directory should be initialized.

```
  ACTION=$(echo $entry | cut -d':' -f 1)
  case $ACTION in
```

If it is a file, follows its source path and its destination in the target filesystem.

```
    F|f)
      SOURCE=$(echo $entry | cut -d':' -f 2)
      DESTINATION=$(echo $entry | cut -d':' -f 3)
```

If the `-c` option is passed (see variable `CHECK_ONLY`), we only check that the target filesystem contains the file.

```
      if test $CHECK_ONLY -ne 0;
      then
        if test ! -f $FSPATH/$DESTINATION;
        then
          error "Filesystem is incomplete! Missing '$FSPATH/$DESTINATION'."
          exit 1
        fi

        continue
      fi
```

Otherwise, we need to check if the source file exists

```
      if test ! -f $SOURCE;
      then
        error "Failed to initialize file '$SOURCE'!"
        exit 1
      fi
```

before creating it at the desired path in the destination filesystem.

```
      mkdir -p $FSPATH/$(dirname $DESTINATION) && \
        cp $SOURCE $FSPATH/$DESTINATION
      if test $? -ne 0;
      then
        error "Failed to initialize file '$FSPATH/$DESTINATION'!"
        exit 1
      fi
      ;;
```

If the entry specifies a directory, follows its destination path in the filesystem being initialized.

```
    D|d)
      DESTINATION=$(echo $entry | cut -d':' -f 2)
```

If the `-c` option is passed (see variable `CHECK_ONLY`), we only check that the target filesystem contains the directory.

```
      if test $CHECK_ONLY -ne 0;
      then
        if test ! -d $FSPATH/$DESTINATION;
        then
          error "Filesystem is uncomplete! Missing '$FSPATH/$DESTINATION'."
          exit 1
        fi

        continue
      fi
```

Otherwise, we create the directory at the specified path.

```
      mkdir -p $FSPATH/$DESTINATION
      if test $? -ne 0;
      then
        error "Failed to initialize directory '$FSPATH/$DESTINATION'!"
        exit 1
      fi
      ;;
```

We also need to take care of the case where the action specified in the description file is not known.

```
    *)
      error "Failed to initialize filesystem! '$ACTION' is not a valid action."
      exit 1
      ;;
  esac
done
```

We finish by printing an information about successful filesystem initialization or verification.

```
if test $CHECK_ONLY -ne 0;
then
  echo "Successfully checked the filesystem '$FSPATH'."
else
  echo "Successfully initialized a fresh gcvb filesystem at '$FSPATH'."
fi
```

5.3.2 Description file

The format of an `.fstab` description file is very straightforward. Each line must begin with either a `D` or an `F` (case insensitive) indicating whether a directory or a file should be initialized. In case of a directory, this is followed by a colon and the destination path of the directory. In

case of a file, follows a colon, the source path of the file, a colon and the destination path in the target filesystem.

Listing 1 features the file `rr-2020.fstab` describing the `gcvb` filesystem of our benchmarks series.

```
D:data/all/input
D:data/all/references
D:data/all/templates
D:results
D:slurm
F:monobatch:data/all/templates/monobatch/sbatch
F:polybatch:data/all/templates/polybatch/sbatch
F:coupled:data/all/templates/coupled/sbatch
F:scalability:data/all/templates/scalability/sbatch
F:setenv.sh:scripts/setenv.sh
F:submit.sh:scripts/submit.sh
F:rss.py:scripts/rss.py
F:inject.py:scripts/inject.py
F:parse-test_FEMBEM.sh:scripts/parse-test_FEMBEM.sh
F:config.yaml:config.yaml
F:rr-2020.yaml:rr-2020.yaml
```

Listing 1: `gcvb` filesystem description file for our benchmark series.

5.4 Configuration file

The configuration file is designed to provide a machine-specific information for a `gcvb` benchmark collection such as the submit command for job scripts, etc. Nevertheless, our configuration does not vary from machine to machine, so we use the same `config.yaml` everywhere.

The configuration of a `gcvb` benchmark collection is simple. It usually holds in a few lines of code beginning by a machine identifier.

```
machine_id: generic
```

The most important is to define the path to the executable used to submit job scripts produced by `gcvb`.

As we rely on the Slurm workload manager, we use its `sbatch` command to submit job scripts. We also want to keep the identifier of the last submitted job for later use (see Section 5.9). Note that, the `%f` placeholder is replaced with the path to the job script before execution.

```
submit_command: "sbatch %f | sed \"s#Submitted batch job ##\" > .lastjob"
```

Eventually, an associative list of executables can be defined for a handy access from definition file. Although, as the executables are not available in the validation phase (see Section 5.5), we can not make use of the mechanism and initialize `executables` as an empty list.

```
executables: []
```

5.5 Definition file

The definition file lists all the benchmarks to generate. For instance, the file `rr-2020.yaml` defines our benchmark series. It begins by a set of default values automatically set for each benchmark defined in the file.

At first, we make all the benchmarks use the same data folder (see Section 5.1). Defining the benchmarks as of type template allows us to make `gcvb` automatically generate benchmarks for different set of parameters (see Section 5). We address this functionality further in this section too.

```
default_values:
  test:
    description: "A test_FEMBEM benchmark run."
    data: "all"
    type: "template"
```

For each task, we want to use by default one MPI process mapped and ranked by node without any binding.

```
  task:
    nprocs: "-np 1 -map-by ppr:1:node -rank-by node -bind-to none"
```

Benchmark tasks are launched using `mpirun`.

```
    executable: mpirun
```

We can define a generic launch command for the future tasks. Note that `{@job_creation}` is a special tag recognized by the `gcvb` parser which allows us to access task attributes such as the `executable` and `options` keys from within `launch_command` for example.

The launch command begins by the creation of a dedicated temporary folder for the benchmark. This is useful to measure the disk space used during the execution of the latter using a storage resource monitoring Python script (see Section 5.6). `full_id` represents a unique task identifier.

```
    launch_command: "mkdir -p
    /tmp/vive-pain-au-chocolat/{@job_creation[full_id]} &&
```

Also, we echo current parallel configuration to the standard output log in order to save it for later processing (see Section 5.7).

```
    echo \"{@job_creation[nprocs]}\" \"{@job_creation[nthreads]}\"
    > stdout.log &&
```

Follows setup of the environment variable defining the path to the temporary folder to use during the benchmark execution.

```
    TMPDIR=/tmp/vive-pain-au-chocolat/{@job_creation[full_id]}
```

We also include the set of environment variables specifying the count of OpenMP and MKL threads and StarPU workers to put in action. This information is accessible through the key

`nthreads` which is a task attribute like `nprocs`. As these values vary from one solver to another, we set them specifically for each set of benchmarks.

Moreover, by default, we do not want SPIDO and HMAT solvers to use the out-of-core computation functionality.

```
    {@job_creation[nthreads]} MPF_SPIDO_INCORE=1 HMAT_DISABLE_OOC=1
```

Then, we specify the primary executable `mpirun` as defined in `executable` together with MPI process configuration parameters stored in `nprocs`.

```
    {@job_creation[executable]} {@job_creation[nprocs]}
```

We launch `test_FEMBEM` through the storage resource monitoring script (see Section 5.6) with options common to all the benchmarks followed by the placeholder `{@job_creation[options]}` for options specific to each set of benchmarks.

Finally, we redirect standard and error outputs into associated log files and clean any files produced during benchmark execution except logs, results and FXT traces, e. g. files with extensions `.log`, `.csv`, `.yaml` or beginning with `prof_file`. The output of the parsing, e. g. the file `data.csv` is deleted only if execution stops prematurely, e. g. if no `traceCall.log` is produced.

Note that commands are run from within benchmark-specific folders under the `results/<session>` directories (see Section 5.1).

```
    ../../../scripts/rss.py test_FEMBEM -nbrhs 50 -z -radius 2
    -height 4 {@job_creation[options]} >> stdout.log 2> stderr.log &&
    rm -f $(find . -type f ! -name \"*.log\" ! -name \"*.csv\" ! -name
    \"*.yaml\" ! -name \"prof_file*\") && test ! -f ongoing_traceCall.log ||
    rm -f data.csv"
```

In our case, we do not perform any result validation in terms of value check. Although, we take advantage of the validation phase in `gcvb` to gather data from log files into a separate `data.csv` file per benchmark and inject them into the `gcvb` NoSQL database (see Section 5.1) using a Python script (see Section 5.8) which calls our parsing script (see Section 5.7) to extract data from the output logs.

We begin by defining the type of each validation. The most adapted type for our needs is the generic one, i. e. `configuration_independent`. For further details, we invite the reader to consult [2].

```
  validation:
    type: "configuration_independent"
```

There is no `option` key available under `validation`. Therefore, we include default options and the path to the injecting Python script in `executable`.

Follow the parameters to be passed to the inner call of the parsing script. In this case, we specify here the output log file to be `stdout.log` as defined in `launch_command`.

The `-r .` parameter tells the parsing script to look for the resource monitoring logs produced by the corresponding Python script (see Section 5.6) in the current working directory and the `-o data.csv` parameter defines the output file for the parsing result.

```
    executable: "../../../scripts/inject.py data.csv
    ../../../scripts/parse-test_FEMBEM.sh -s stdout.log -r . -o data.csv"
```

At this stage, we can define actual benchmarks. They are structured in packs. For now, we have only one pack but this document shall evolve and include all our future benchmark definitions.

Each pack contains a list of benchmarks and each benchmark may be composed of one or more tasks having one or more validation tasks each.

```
Packs:
  -
    pack_id: "test_FEMBEM"
    description: "Analyze the impact of solver paramaters."
    Tests:
```

Firstly, we want to benchmark the SPIDO solver on purely dense linear systems arising from BEM discretization for various unknown counts. Under `template_instantiation` there are two array constructs later expanded by `gcvb` to generate multiple variants of the benchmark, e. g. for various problem sizes.

`slurm` holds the common job name prefix and the scheduling information used for the generation of the associated `sbatch` header file `monobatch` (see Section 5.2).

The `nbpts` array defines the problem sizes to generate benchmarks for. Note that `{slurm[prefix]}`, `{slurm[platform]}`, `{nbpts}` and so on are the placeholders for the values defined in `template_instantiation`.

Given the current `template_instantiation` configuration, we generate here $1 \times 3 = 3$ SPIDO benchmarks grouped into a single job script with a time limit of 2 hours.

```
      -
        id: "spido-{nbpts}"
        template_files: "monobatch"
        template_instantiation:
          slurm:
            - { prefix: "spido", platform: "plafrim", node: "miriel", count: 1,
                tasks: 24, time: "0-02:00:00" }
          nbpts: [ 25000, 50000, 100000 ]
```

Follows the task corresponding to this benchmark. The launch command is read from the list of default values defined at the beginning of the file. We only override here the `nthreads` key to set the proper count of OpenMP and MKL threads to use for the computation. The values are propagated to the launch command through the `{@job_creation[options]}` placeholder.

```
        Tasks:
          -
            nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24"
            options: "--bem -withmpf -nbpts {nbpts}"
```

For the corresponding validation phase we need to specify an identifier as well as a launch command composed of the validation `executable` obtained through the `{@job_creation[va_executable]}` placeholder. Then, we define some options specific to this benchmark such as the information on the solver used, the target platform as well as the variation of the benchmark to make a difference between regular benchmarks based on parameter variation and scalability benchmarks.

```
        Validations:
          -
            id: "validation-spido-{nbpts}"
            launch_command: "{@job_creation[va_executable]} -K solver=spido
            -K variation=parameters,platform={slurm[platform]}
            -K node={slurm[node]}"
```

In the second place, we benchmark the HMAT solver on purely dense BEM systems under similar conditions as SPIDO. Although, here we also vary the low-rank compression threshold ϵ (see the `epsilon` array) in `template_instantiation`.

HMAT is parallelized using StarPU. This means that we need to define the count of StarPU workers through the `HMAT_NCPU` environment variable in `nthreads`. Nevertheless, there is a portion of the computation at the beginning of the benchmark, a BEM matrix product, that is parallelized using OpenMP and MKL. Therefore, we also set OpenMP and MKL thread counts to 24.

Here, we generate $1 \times 2 \times 7 = 14$ sequentially run benchmarks with an execution time limit of 3 hours.

```
    -
      id: "hmat-bem-{epsilon}-{nbpts}"
      template_files: "monobatch"
      template_instantiation:
        slurm:
          - { prefix: "hmat-bem", platform: "plafrim", node: "miriel",
          count: 1, tasks: 24, time: "0-03:00:00" }
        epsilon: [ "1e-3", "1e-6" ]
        nbpts: [ 25000, 50000, 100000, 200000, 400000, 800000, 1000000 ]
      Tasks:
        -
          nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 HMAT_NCPU=24"
          options: "--bem -withmpf --hmat --hmat-eps-assemb {epsilon}
          --hmat-eps-recompr {epsilon} -nbpts {nbpts}"
          Validations:
            -
              id: "validation-hmat-bem-{epsilon}-{nbpts}"
              launch_command: "{@job_creation[va_executable]}
              -K solver=hmat-bem,variation=parameters
              -K platform={slurm[platform]},node={slurm[node]}"
```

In the category of sparse solvers, we begin by benchmarking MUMPS on purely sparse linear systems arising from FEM discretization. Besides unknown count (see the `nbpts` array), we evaluate the performance for various low-rank compression thresholds when the low-rank compression mechanism is enabled (see the `epsilon` key under the `BLR` map). The `options` key in the `BLR` map defines the options for `test_FEMBEM` to set the accuracy threshold and choose the low-rank compression implementation variant to use.

In total, we generate $1 \times 3 \times 7 = 21$ sequentially run benchmarks limited to 4 hours of computation.

```
    -
      id: "mumps-basic-{BLR[epsilon]}-{nbpts}"
      template_files: "monobatch"
      template_instantiation:
        slurm:
          - { prefix: "mumps-basic", platform: "plafrim", node: "miriel",
          count: 1, tasks: 24, time: "0-04:00:00" }
```

```
      BLR:
        - { epsilon: "1e-3", options: "--mumps-blr --mumps-blr-variant 1
        --mumps-blr-accuracy" }
        - { epsilon: "1e-6", options: "--mumps-blr --mumps-blr-variant 1
        --mumps-blr-accuracy" }
        - { epsilon:     "", options: "--no-mumps-blr" }
      nbpts: [ 250000, 500000, 1000000, 2000000, 4000000, 8000000,
      10000000 ]
    Tasks:
      -
        nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24"
        options: "--fem -withmpf {BLR[options]} {BLR[epsilon]}
        --mumps-verbose -nbpts {nbpts}"
        Validations:
          -
            id: "validation-mumps-basic-{BLR[epsilon]}-{nbpts}"
            launch_command: "{@job_creation[va_executable]} -K solver=mumps
            -K variation=parameters,platform={slurm[platform]}
            -K node={slurm[node]}"
```

For illustration purposes, we define several additional benchmarks of MUMPS on the same kind of linear systems. In this case, the problem sizes (see the `nbpts` array) match the counts of unknowns related to the FEM-discretized domain in case of coupled FEM/BEM systems having 250,000, 500,000 and 1,000,000 unknowns respectively. Also, we consider both symmetric and non-symmetric systems (see the `symmetry` map). The low-rank compression threshold is kept constant at 1×10^{-3}.

In total, we generate here $1 \times 2 \; times3 = 6$ sequentially run benchmarks limited to 30 minutes of execution time.

```
  -
    id: "mumps-additional-{symmetry[label]}-{nbpts}"
    template_files: "monobatch"
    template_instantiation:
      slurm:
        - { prefix: "mumps-additional", platform: "plafrim", node: "miriel",
        count: 1, tasks: 24, time: "0-00:30:00" }
      symmetry:
        - { label: "symmetric", options: "" }
        - { label: "non-symmetric", options: "--nosym" }
      nbpts: [ 235165, 476423, 962831 ]
    Tasks:
      -
        nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24"
        options: "--fem -withmpf --mumps-blr --mumps-blr-variant 1
        --mumps-blr-accuracy 1e-3 --mumps-verbose -nbpts {nbpts}
        {symmetry[options]}"
        Validations:
          -
            id: "validation-mumps-additional-{symmetry[label]}-{nbpts}"
            launch_command: "{@job_creation[va_executable]} -K solver=mumps
            -K variation=parameters,symmetry={symmetry[label]}
            -K platform={slurm[platform]},node={slurm[node]}"
```

For HMAT on FEM systems, we evaluate the impact of the same parameters as in the case of MUMPS. Here, we generate $1 \times 2 \times 4 = 8$ sequentially run benchmarks limited to 2 hours of execution.

In the benchmark prefix, we specify that we are working with symmetric matrices as we define

below further HMAT benchmarks involving non-symmetric matrices.

```
-
  id: "hmat-fem-symmetric-{epsilon}-{nbpts}"
  template_files: "monobatch"
  template_instantiation:
    slurm:
      - { prefix: "hmat-fem-symmetric", platform: "plafrim",
          node: "miriel", count: 1, tasks: 24, time: "0-02:00:00" }
    epsilon: [ "1e-3", "1e-6" ]
    nbpts: [ 250000, 500000, 1000000, 2000000 ]
  Tasks:
    -
      nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 HMAT_NCPU=24"
      options: "--fem -withmpf --hmat --hmat-eps-assemb {epsilon}
      --hmat-eps-recompr {epsilon} -nbpts {nbpts}"
      Validations:
        -
          id: "validation-hmat-fem-{epsilon}-{nbpts}"
          launch_command: "{@job_creation[va_executable]}
          -K solver=hmat-fem,variation=parameters,symmetry=symmetric
          -K platform={slurm[platform]},node={slurm[node]}"
```

We also evaluate the performance of the prototype implementation of the Nested Dissection re-ordering technique in HMAT for the solution of sparse systems (see the `variant` map). The current implementation limits the application of the algorithm to non-symmetric matrices. To be able to compare it to runs without using ND, we must redo the latter on non-symmetric matrices as well (see the `variant` map).

When ND is enabled, the HMAT solver uses a significantly higher amount of memory and cases counting more than 250,000 unknowns cause memory overflow. Therefore, we benchmark the algorithm on smaller systems. Eventually, we generate $1 \times 2 \times 2 \times 6 = 24$ sequentially run benchmarks limited to 1 hour of execution.

```
-
  id: "hmat-fem-non-symmetric{variant[ND]}-{epsilon}-{nbpts}"
  template_files: "monobatch"
  template_instantiation:
    slurm:
      - { prefix: "hmat-fem-non-symmetric", platform: "plafrim",
          node: "miriel", count: 1, tasks: 24, time: "0-01:00:00" }
    variant:
      - { ND: "", options: "--hmat-lu --nosym" }
      - { ND: "-nd", options: "--hmat-nd" }
    epsilon: [ "1e-3", "1e-6" ]
    nbpts: [ 25000, 50000, 100000, 200000, 250000 ]
  Tasks:
    -
      nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 HMAT_NCPU=24"
      options: "--fem {variant[options]} -withmpf --hmat --hmat-eps-assemb
      {epsilon} --hmat-eps-recompr {epsilon} -nbpts {nbpts}"
      Validations:
        -
          id: "validation-hmat-fem-non-symmetric{variant[ND]}-{epsilon}-\
          {nbpts}"
          launch_command: "{@job_creation[va_executable]}
          -K solver=hmat-fem{variant[ND]},symmetry=non-symmetric
          -K variation=parameters,platform={slurm[platform]}
          -K node={slurm[node]}"
```

Once we have evaluated the performance of the solvers on homogeneous, either BEM or FEM systems, we follow up with benchmarks on coupled FEM/BEM systems. For solvers allowing data compression, we always set the ϵ to 10^{-3}.

In the following, we benchmark the two-stage implementation scheme multi-solve using MUMPS as sparse solver and SPIDO as dense solver. We vary problem's unknown count (see the `nbpts` key in the `job` map) as well as the count of right-hand sides to be processed at once by MUMPS during the Schur complement computation (see the `nbrhs` key in the `job` map). The maps `sparse` and `dense` defines the options for the sparse and the dense solver respectively. Moreover, the `tracing` key in the `job` map allows us to enable execution timeline tracing for selected runs.

$1 \times 20 \times 1 \times 1 = 20$ benchmark runs are generated here and split into 4 parallel batch jobs labelled from 1 to 4 using the `job` map. The jobs are limited to 20 hours of execution each.

```
-
  id: "multi-solve-{job[batch]}-{sparse[name]}-{dense[name]}-\
  {job[nbrhs]}-{job[nbpts]}"
  template_files: "coupled"
  template_instantiation:
    slurm:
      - { prefix: "multi-solve", platform: "plafrim", node: "miriel",
      count: 1, tasks: 24, time: "0-20:00:00" }
    job:
      # N = 250k
      - { nbpts:  250000, nbrhs:  32, batch:  1, tracing: "" }
      - { nbpts:  250000, nbrhs:  64, batch:  1, tracing: "" }
      - { nbpts:  250000, nbrhs: 128, batch:  1, tracing: "" }
      - { nbpts:  250000, nbrhs: 256, batch:  1,
      tracing: "--timeline-trace-calls" }
      # N = 500k
      - { nbpts:  500000, nbrhs:  32, batch:  1, tracing: "" }
      - { nbpts:  500000, nbrhs:  64, batch:  1, tracing: "" }
      - { nbpts:  500000, nbrhs: 128, batch:  1, tracing: "" }
      - { nbpts:  500000, nbrhs: 256, batch:  1,
      tracing: "--timeline-trace-calls" }
      # N = 1M
      - { nbpts: 1000000, nbrhs:  32, batch:  1, tracing: "" }
      - { nbpts: 1000000, nbrhs:  64, batch:  1, tracing: "" }
      - { nbpts: 1000000, nbrhs: 128, batch:  1, tracing: "" }
      - { nbpts: 1000000, nbrhs: 256, batch:  1,
      tracing: "--timeline-trace-calls" }
      # N = 2M
      - { nbpts: 2000000, nbrhs:  32, batch:  1, tracing: "" }
      - { nbpts: 2000000, nbrhs:  64, batch:  1, tracing: "" }
      - { nbpts: 2000000, nbrhs: 128, batch:  1, tracing: "" }
      - { nbpts: 2000000, nbrhs: 256, batch:  1, tracing: "" }
      # N = 4M
      - { nbpts: 4000000, nbrhs:  32, batch:  2, tracing: "" }
      - { nbpts: 4000000, nbrhs:  64, batch:  3, tracing: "" }
      - { nbpts: 4000000, nbrhs: 128, batch:  4, tracing: "" }
      - { nbpts: 4000000, nbrhs: 256, batch:  4, tracing: "" }
    sparse:
      - { name: "mumps", options: "--mumps-verbose --mumps-blr
      --mumps-blr-variant 1 --mumps-blr-accuracy 1e-3 --mumps-multi-solve
      -nbrhsmumps" }
    dense:
      - { name: "spido", options: "" }
  Tasks:
    -
      nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24"
      options: "--fembem -withmpf --coupled {sparse[options]} {job[nbrhs]}
      {dense[options]} -nbpts {job[nbpts]} {job[tracing]}"
```

```
      Validations:
        -
          id: "validation-multi-solve-{sparse[name]}-{dense[name]}-\
          {job[nbrhs]}-{job[nbpts]}"
          launch_command: "{@job_creation[va_executable]}
          -K solver={sparse[name]}/{dense[name]},variation=parameters
          -K platform={slurm[platform]},node={slurm[node]}"
```

Afterwards, we benchmark the two-stage implementation scheme multi-factorization using MUMPS as sparse solver and either SPIDO or HMAT as dense solver. Like in the previous case, we vary problem's unknown count (see the `nbpts` key in the `job` map) as well as the size of blocks of the Schur complement matrix during the computation (see the `schur` key in the `job` map). The `n_blocks` key under `job` allows us to log the information about the number of blocks per row and column the Schur complement matrix is split into. The value `auto` means that the count is determined automatically by the solver. Finally, through the `tracing` key in the `job` map we can turn on execution timeline tracing for selected runs.

In case of SPIDO, the largest size of the Schur complement block, when considering double complex arithmetics, is 11,585 due to implementation limitations. On the other hand, this is not the case of HMAT. Therefore, we split the definition of these benchmarks in two in order to avoid failing SPIDO benchmarks on larger values of `schur`.

For the first coupling of solvers, we generate $1 \times 16 \times 1 \times 1 = 16$ benchmarks split into 4 parallel jobs with a time limit of 14 hours each.

```
    -
      id: "multi-factorization-{job[batch]}-{sparse[name]}-{dense[name]}-\
      {job[nbpts]}-{job[schur]}"
      template_files: "coupled"
      template_instantiation:
        slurm:
          - { prefix: "multi-factorization", platform: "plafrim",
          node: "miriel", count: 1, tasks: 24, time: "0-14:00:00" }
        job:
          # N = 250k
          - { nbpts:  250000, schur:  3856, n_blocks: "auto", batch:  1,
          tracing: "" }
          - { nbpts:  250000, schur:  4945, n_blocks:    "3", batch:  1,
          tracing: "" }
          - { nbpts:  250000, schur:  7418, n_blocks:    "2", batch:  1,
          tracing: "--timeline-trace-calls" }
          # N = 500k
          - { nbpts:  500000, schur:  3906, n_blocks: "auto", batch:  1,
          tracing: "" }
          - { nbpts:  500000, schur:  5895, n_blocks:    "4", batch:  1,
          tracing: "" }
          - { nbpts:  500000, schur:  7859, n_blocks:    "3", batch:  1,
          tracing: "--timeline-trace-calls" }
          # N = 1M
          - { nbpts: 1000000, schur:  3914, n_blocks: "auto", batch:  1,
          tracing: "" }
          - { nbpts: 1000000, schur:  7434, n_blocks:    "5", batch:  1,
          tracing: "" }
          - { nbpts: 1000000, schur:  9293, n_blocks:    "4", batch:  1,
          tracing: "--timeline-trace-calls" }
          # N = 1.2M
          - { nbpts: 1200000, schur:  3886, n_blocks: "auto", batch:  2,
          tracing: "" }
          - { nbpts: 1200000, schur:  6999, n_blocks:    "6", batch:  3,
```

```
        tracing: "" }
        - { nbpts: 1200000, schur:  8399, n_blocks:    "5", batch:  3,
        tracing: "" }
        - { nbpts: 1200000, schur: 10498, n_blocks:    "4", batch:  2,
        tracing: "" }
        # N = 1.4
        - { nbpts: 1400000, schur:  3912, n_blocks: "auto", batch:  4,
        tracing: "" }
        - { nbpts: 1400000, schur:  7747, n_blocks:    "6", batch:  3,
        tracing: "" }
        - { nbpts: 1400000, schur:  9295, n_blocks:    "5", batch:  3,
        tracing: "" }
      sparse:
        - { name: "mumps", options: "--mumps-verbose --mumps-blr
        --mumps-blr-variant 1 --mumps-blr-accuracy 1e-3
        --mumps-multi-facto" }
      dense:
        - { name: "spido", options: "-diskblock" }
    Tasks:
      -
        nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24"
        options: "--fembem -withmpf --coupled {sparse[options]}
        {dense[options]} {job[schur]} -nbpts {job[nbpts]} {job[tracing]}"
        Validations:
          -
            id: "validation-multi-factorization-{sparse[name]}-\
            {dense[name]}-{job[nbpts]}-{job[schur]}"
            launch_command: "{@job_creation[va_executable]}
            -K solver={sparse[name]}/{dense[name]},n_blocks={job[n_blocks]}
            -K variation=parameters,platform={slurm[platform]}
            -K node={slurm[node]} -l traceCall.log=CreateSchurComplement"
```

For the coupling involving HMAT as dense solver, we generate $1 \times 32 \times 1 \times 1 = 32$ benchmarks split into 6 parallel jobs with a time limit of 17.5 hours each.

```
  -
    id: "multi-factorization-{job[batch]}-{sparse[name]}-{dense[name]}-\
    {job[nbpts]}-{job[schur]}"
    template_files: "coupled"
    template_instantiation:
      slurm:
        - { prefix: "multi-factorization", platform: "plafrim",
        node: "miriel", count: 1, tasks: 24, time: "0-17:30:00" }
      job:
        # N = 250k
        - { nbpts:  250000, schur:  3856, n_blocks: "auto", batch:  1 }
        - { nbpts:  250000, schur:  4945, n_blocks:    "3", batch:  1 }
        - { nbpts:  250000, schur:  7418, n_blocks:    "2", batch:  1 }
        - { nbpts:  250000, schur: 14835, n_blocks:    "1", batch:  1 }
        # N = 500k
        - { nbpts:  500000, schur:  3906, n_blocks: "auto", batch:  1 }
        - { nbpts:  500000, schur:  5895, n_blocks:    "4", batch:  1 }
        - { nbpts:  500000, schur:  7859, n_blocks:    "3", batch:  1 }
        - { nbpts:  500000, schur: 11789, n_blocks:    "2", batch:  1 }
        - { nbpts:  500000, schur: 23577, n_blocks:    "1", batch:  1 }
        # N = 1M
        - { nbpts: 1000000, schur:  3914, n_blocks: "auto", batch:  1 }
        - { nbpts: 1000000, schur:  7434, n_blocks:    "5", batch:  1 }
        - { nbpts: 1000000, schur:  9293, n_blocks:    "4", batch:  1 }
        - { nbpts: 1000000, schur: 12390, n_blocks:    "3", batch:  1 }
        - { nbpts: 1000000, schur: 18585, n_blocks:    "2", batch:  1 }
```

```
        # N = 1.2M
        - { nbpts: 1200000, schur:  3886, n_blocks: "auto", batch:  3 }
        - { nbpts: 1200000, schur:  6999, n_blocks:    "6", batch:  3 }
        - { nbpts: 1200000, schur:  8399, n_blocks:    "5", batch:  2 }
        - { nbpts: 1200000, schur: 10498, n_blocks:    "4", batch:  2 }
        - { nbpts: 1200000, schur: 13998, n_blocks:    "3", batch:  2 }
        - { nbpts: 1200000, schur: 20996, n_blocks:    "2", batch:  2 }
        # N = 1.4
        - { nbpts: 1400000, schur:  3912, n_blocks: "auto", batch:  4 }
        - { nbpts: 1400000, schur:  7747, n_blocks:    "6", batch:  4 }
        - { nbpts: 1400000, schur:  9295, n_blocks:    "5", batch:  4 }
        - { nbpts: 1400000, schur: 11621, n_blocks:    "4", batch:  3 }
        - { nbpts: 1400000, schur: 15494, n_blocks:    "3", batch:  3 }
        - { nbpts: 1400000, schur: 23241, n_blocks:    "2", batch:  3 }
        # N = 1.5 M
        - { nbpts: 1500000, schur:  3870, n_blocks: "auto", batch:  5 }
        - { nbpts: 1500000, schur:  6094, n_blocks:    "8", batch:  6 }
        - { nbpts: 1500000, schur:  8125, n_blocks:    "6", batch:  6 }
        - { nbpts: 1500000, schur:  9750, n_blocks:    "5", batch:  6 }
        - { nbpts: 1500000, schur: 12188, n_blocks:    "4", batch:  6 }
        - { nbpts: 1500000, schur: 16250, n_blocks:    "3", batch:  6 }
      sparse:
        - { name: "mumps", options: "--mumps-verbose --mumps-blr
        --mumps-blr-variant 1 --mumps-blr-accuracy 1e-3
        --mumps-multi-facto" }
      dense:
        - { name: "hmat", options: "--hmat --hmat-eps-assemb 1e-3
        --hmat-eps-recompr 1e-3 --coupled-mumps-hmat --mumps-sizeschur" }
    Tasks:
      -
        nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 HMAT_NCPU=24"
        options: "--fembem -withmpf --coupled {sparse[options]}
        {dense[options]} {job[schur]} -nbpts {job[nbpts]}"
        Validations:
          -
            id: "validation-multi-factorization-{sparse[name]}-\
            {dense[name]}-{job[nbpts]}-{job[schur]}"
            launch_command: "{@job_creation[va_executable]}
            -K solver={sparse[name]}/{dense[name]},n_blocks={job[n_blocks]}
            -K variation=parameters,platform={slurm[platform]}
            -K node={slurm[node]} -l traceCall.log=CreateSchurComplement"
```

Finally, we benchmark the resolution of coupled systems using the single-stage partially sparse-aware implementation scheme using HMAT for both sparse and dense operations. We generate $1 \times 4 = 4$ benchmarks grouped into a single batch job with a time limit of 3 hours.

```
  -
    id: "full-hmat-{nbpts}"
    template_files: "monobatch"
    template_instantiation:
      slurm:
        - { prefix: "full-hmat", platform: "plafrim", node: "miriel",
        count: 1, tasks: 24, time: "0-03:00:00" }
      nbpts: [ 250000, 500000, 1000000, 2000000 ]
    Tasks:
      -
        nthreads: "OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 HMAT_NCPU=24"
        options: "--fembem -withmpf --coupled --hmat --hmat-eps-assemb 1e-3
        --hmat-eps-recompr 1e-3 -nbpts {nbpts}"
        Validations:
          -
```

```
            id: "validation-full-hmat-{nbpts}"
            launch_command: "{@job_creation[va_executable]}
            -K solver=hmat/hmat,variation=parameters
            -K platform={slurm[platform]},node={slurm[node]}"
```

To evaluate the scalability of the solvers, we make them solve a large dense BEM or sparse FEM linear system (see Table 1) while varying the parallel configuration.

Sparsity	Solver	Unknown count
dense	SPIDO	100,000
dense	HMAT	100,000 and 1,000,000
sparse	MUMPS	2,000,000 and 4,000,000
sparse	HMAT	2,000,000

TABLE 1: Linear system unknown counts used for scalability benchmarks.

Note that for solvers allowing data compression, we set ϵ to 10^{-3}.

```
    -
      id: "scalability-{solver[name]}-{job[map]}-{job[np]}x{job[nt]}-\
           {solver[nbpts]}"
      template_files: "scalability"
      template_instantiation:
        slurm:
          - { prefix: "scalability", platform: "plafrim", node: "miriel",
          count: 1, tasks: 24, time: "0-20:00:00" }
        solver:
          - { name:    "spido", nbpts:  100000, options: "--bem" }
          - { name:    "mumps", nbpts: 2000000, options: "--fem
              --mumps-verbose --mumps-blr --mumps-blr-variant 1
              --mumps-blr-accuracy 1e-3" }
          - { name:    "mumps", nbpts: 4000000, options: "--fem
              --mumps-verbose --mumps-blr --mumps-blr-variant 1
              --mumps-blr-accuracy 1e-3" }
```

We consider four kinds of parallel configuration:

- 1 MPI process mapped and ranked by node and 1 to 24 OpenMP and MKL threads;
  - `mpirun` configuration example: `OMP_NUM_THREADS=24 MKL_NUM_THREADS=24 mpirun -np 1 -map-by ppr:1:node -rank-by node -bind-to none test_FEMBEM...`

```
        job:
          - { np:  1, nt:  1, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt:  6, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 12, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 18, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 24, map:   "node", rank:   "node", bind:   "none" }
```

- 2 MPI processes mapped by, ranked by and bound to sockets and 1 to 12 OpenMP and MKL threads;
  - `mpirun` configuration example: `OMP_NUM_THREADS=12 MKL_NUM_THREADS=12 mpirun -np 2 -map-by ppr:1:socket -rank-by socket -bind-to socket test_FEMBEM...`

```
            - { np:  2, nt:  1, map: "socket", rank: "socket", bind: "socket" }
            - { np:  2, nt:  2, map: "socket", rank: "socket", bind: "socket" }
            - { np:  2, nt:  4, map: "socket", rank: "socket", bind: "socket" }
            - { np:  2, nt:  8, map: "socket", rank: "socket", bind: "socket" }
            - { np:  2, nt: 12, map: "socket", rank: "socket", bind: "socket" }
```

- 4 MPI processes mapped by, ranked by and bound to numa sub-nodes and 1 to 6 OpenMP and MKL threads;
  - `mpirun` configuration example: `OMP_NUM_THREADS=6 MKL_NUM_THREADS=6 mpirun -np 4 -map-by ppr:1:numa -rank-by numa -bind-to numa test_FEMBEM...`

```
            - { np:  4, nt:  1, map:   "numa", rank:   "numa", bind:   "numa" }
            - { np:  4, nt:  2, map:   "numa", rank:   "numa", bind:   "numa" }
            - { np:  4, nt:  4, map:   "numa", rank:   "numa", bind:   "numa" }
            - { np:  4, nt:  6, map:   "numa", rank:   "numa", bind:   "numa" }
```

- 1 to 24 MPI processes mapped by, ranked by and bound to cores and 1 OpenMP and MKL thread.
  - `mpirun` configuration example: `OMP_NUM_THREADS=1 MKL_NUM_THREADS=1 mpirun -np 24 -map-by ppr:1:core -rank-by core -bind-to core test_FEMBEM...`

```
            - { np:  1, nt:  1, map:   "core", rank:   "core", bind:   "core" }
            - { np:  6, nt:  1, map:   "core", rank:   "core", bind:   "core" }
            - { np: 12, nt:  1, map:   "core", rank:   "core", bind:   "core" }
            - { np: 18, nt:  1, map:   "core", rank:   "core", bind:   "core" }
            - { np: 24, nt:  1, map:   "core", rank:   "core", bind:   "core" }
```

Finally, we have to override the default values of `nprocs` and `nthreads` keys, set solver options (see the `options` value) and configure the validation phase.

```
        Tasks:
          -
            nprocs: "-np {job[np]} -map-by ppr:1:{job[map]} -rank-by {job[rank]}
                     -bind-to {job[bind]}"
            nthreads: "OMP_NUM_THREADS={job[nt]} MKL_NUM_THREADS={job[nt]}
            HMAT_NCPU={job[nt]}"
            options: "-withmpf {solver[options]} -nbpts {solver[nbpts]}"
            Validations:
              -
                id: "parse-scalability-{solver[name]}-{job[map]}-\
                     {job[np]}x{job[nt]}-{solver[nbpts]}"
                launch_command: "{@job_creation[va_executable]}
                -K solver={solver[name]},variation=scalability
                -K platform={slurm[platform]},node={slurm[node]}"
```

For this first set, we generate $1 \times 3 \times 19 = 57$ benchmarks grouped into 12 batch jobs with a time limit of 20 hours each.

We define another two separate sets of scalability benchmarks for HMAT as the MPI parallelization of the latter is not adapted for execution on one single node and consequently not studied.

```
    -
      id: "scalability-{solver[name]}-{job[map]}-{job[np]}x{job[nt]}-\
           {solver[nbpts]}"
      template_files: "scalability"
      template_instantiation:
        slurm:
          - { prefix: "scalability", platform: "plafrim", node: "miriel",
          count: 1, tasks: 24, time: "0-14:00:00" }
        solver:
          - { name: "hmat-bem", nbpts:  100000, options: "--bem --hmat
              --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3" }
          - { name: "hmat-bem", nbpts: 1000000, options: "--bem --hmat
              --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3" }
          - { name: "hmat-fem", nbpts: 2000000, options: "--fem --hmat
              --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3 --no-hmat-nd" }
```

We consider only one kind of parallel configuration here: 1 MPI process mapped and ranked by node and 1 to 24 OpenMP and MKL threads and StarPU workers.

```
        job:
          - { np:  1, nt:  1, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt:  6, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 12, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 18, map:   "node", rank:   "node", bind:   "none" }
          - { np:  1, nt: 24, map:   "node", rank:   "node", bind:   "none" }
      Tasks:
        -
          nprocs: "-np {job[np]} -map-by ppr:1:{job[map]} -rank-by {job[rank]}
                   -bind-to {job[bind]}"
          nthreads: "OMP_NUM_THREADS={job[nt]} MKL_NUM_THREADS={job[nt]}
          HMAT_NCPU={job[nt]}"
          options: "-withmpf {solver[options]} -nbpts {solver[nbpts]}"
          Validations:
            -
              id: "parse-scalability-{solver[name]}-{job[map]}-\
                   {job[np]}x{job[nt]}-{solver[nbpts]}"
              launch_command: "{@job_creation[va_executable]}
              -K solver={solver[name]},variation=scalability
              -K platform={slurm[platform]},node={slurm[node]}"
```

In the case of this second set, we generate $1 \times 3 \times 5 = 15$ benchmarks grouped into 2 batch jobs with a time limit of 14 hours each.

By the means of a third set of HMAT scalability benchmarks, we run the solver on one BEM and one FEM system in three different parallel configurations and generate an FXT execution trace for each one of them. Here, we consider smaller systems ($N = 25000$ for BEM and $N = 50000$ for FEM) on a lower count of cores (4 cores maximum) than in the previous set. The goal is to show the difference in performance between an exclusively StarPU-parallelized execution and an execution involving multiple MPI processes on one single node.

```
    -
      id: "fxt-scalability-{solver[name]}-{job[map]}-{job[np]}x{job[nt]}-\
           {solver[nbpts]}"
      template_files: "scalability"
      template_instantiation:
        slurm:
          - { prefix: "fxt-scalability", platform: "plafrim", node: "miriel",
          count: 1, tasks: 4, time: "0-00:30:00" }
        solver:
```

```
          - { name: "hmat-bem", nbpts:  25000, options: "--bem --hmat
              --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3" }
          - { name: "hmat-fem", nbpts:  50000, options: "--fem --hmat
              --hmat-eps-assemb 1e-3 --hmat-eps-recompr 1e-3 --no-hmat-nd" }
```

We consider three parallel configurations here:

- 1 MPI process mapped and ranked by node and 4 OpenMP and MKL threads and StarPU workers;
- 2 MPI processes mapped, ranked and bound to sockets and 2 OpenMP and MKL threads and StarPU workers;
- 4 MPI processes mapped, ranked and bound to cores.

```
        job:
          - { np:  1, nt:  4, map:   "node", rank:   "node", bind:   "none" }
          - { np:  2, nt:  2, map: "socket", rank: "socket", bind: "socket" }
          - { np:  4, nt:  1, map:   "core", rank:   "core", bind:   "core" }
```

We must also enable StarPU profiling by setting the associated environment variables `STARPU_PROFILING` and `STARPU_WORKER_STATS` to 1. `STARPU_FXT_PREFIX` allows us to set where the FXT traces should be placed. Notice that for the profiling to work, the regular StarPU package must be replaced by `starpu-fxt` in the target software environment (see Section 4.3).

```
      Tasks:
        -
          nprocs: "-np {job[np]} -map-by ppr:1:{job[map]} -rank-by {job[rank]}
          -bind-to {job[bind]}"
          nthreads: "OMP_NUM_THREADS={job[nt]} MKL_NUM_THREADS={job[nt]}
          HMAT_NCPU={job[nt]} STARPU_FXT_PREFIX=$(pwd)/ STARPU_PROFILING=1
          STARPU_WORKER_STATS=1"
          options: "-withmpf {solver[options]} -nbpts {solver[nbpts]}"
          Validations:
            -
              id: "parse-scalability-{solver[name]}-{job[map]}-\
              {job[np]}x{job[nt]}-{solver[nbpts]}"
              launch_command: "{@job_creation[va_executable]}
              -K solver={solver[name]},variation=fxt-scalability
              -K platform={slurm[platform]},node={slurm[node]}"
            -
              id: "visualization-phase-1-scalability-{solver[name]}-\
              {job[map]}-{job[np]}x{job[nt]}-{solver[nbpts]}"
              launch_command: "
              export PATH=$GUIX_ENVIRONMENT/site-library/starvz/tools/:$PATH
              && phase1-workflow.sh $(pwd)/ && phase2-workflow.R $(pwd)/
              $GUIX_ENVIRONMENT/site-library/starvz/etc/default.yaml &&
              rm prof_file*"
```

Finally, we generate another $1 \times 2 \times 3 = 6$ benchmarks to be run in parallel and having each a time limit of 30 minutes.

5.6 Resource monitoring

To measure the amount of memory resources consumed during the execution of a benchmark, we launch `test_FEMBEM` through the Python script `rss.py`. It produces one global disk space

usage log and as much memory usage logs as there are MPI processes. In addition, the last lines in these log files correspond to peak consumption. Note that measures are taken regularly each second and the storing units are kibibytes [kiB].

```
import subprocess
import sys
import threading
import time
import os
import psutil
```

After importing necessary Python modules, we begin by defining a help message function that can be triggered with the `-h` option.

```
def help():
    print("Monitor memory and disk usage of PROGRAM.\n",
          "Usage: ", sys.argv[0], " [options] [PROGRAM]\n\n",
          "Options:\n  -h    Show this help message.",
          file = sys.stderr, sep = "")
```

We must check if at least one argument has been passed to the script. It takes as arguments the executable to measure the resource consumption of and the arguments of the executable, if any. Otherwise, one can specify the `-h` option to display a help message on how to use the tool.

```
if len(sys.argv) < 2:
    print("Error: Arguments mismatch!\n", file = sys.stderr)
    help()
    sys.exit(1)

if sys.argv[1] == "-h":
    help()
    sys.exit(0)
```

Follow some function definitions. At first, a function to determine the MPI rank of the currently monitored process.

```
def mpi_rank():
    rank = 0
    for env_name in ['MPI_RANKID', 'OMPI_COMM_WORLD_RANK']:
        try:
            rank = int(os.environ[env_name])
            break
        except:
            continue
    return rank
```

The script gather resident memory, e.g. the amout of data stored in the random access memory (RAM). On Linux, the values can be obtained from the `/proc` filesystem.

Memory usage statistics of a particular process are stored in `/proc/<pid>/statm` where `<pid>` is the process identifier (PID). In this file, the field `VmRSS` holds the amount of resident memory used by the process at instant t.

```
def rss(pid):
    with open("/proc/%d/statm" % pid, "r") as f:
        line = f.readline().split();
        VmRSS = int(line[1])
        return VmRSS
```

To be able to monitor disk space usage during benchmarks, we assign to each run its specific temporary folder where all the files used by `test_FEMEBM` are put. Finally, we monitor the evolution of the size of `TMPDIR` in our Python script using the `du` Linux command.

```
def hdd(tmpdir):
    du = subprocess.Popen(['du', tmpdir], stdout = subprocess.PIPE)
    tail = subprocess.Popen(['tail', '-n', '1'], stdin = du.stdout,
                            stdout = subprocess.PIPE)
    cut = subprocess.Popen(['cut', '-f1'], stdin = tail.stdout,
                           stdout = subprocess.PIPE)

    return int(cut.stdout.read().decode("utf-8"))
```

Before entering into the monitoring loop, we need to:

- initialize variables to store peak values in;

```
VmHWM = int(0)
HddPeak = int(0)
```

- get the path of the temporary folder from the `TMPDIR` environment variable. If the variable is not set, `/tmp` is returned to allow the script to detect that no separate temporary folder has been created and exit with failure to prevent taking potentially incorrect measures;

```
tmpdir = os.getenv('TMPDIR', '/tmp')

if tmpdir == '/tmp':
    print('Error: Temporary files were not redirected!')
    sys.exit(1)
```

- launch the program to monitor and get the rank of the currently monitored process.

```
myargs = sys.argv[1:]
p = subprocess.Popen(myargs)
rank = mpi_rank()
```

At this point, we open the output log files and begin the monitoring loop. The latter breaks when the monitored process exits.

```
with open("rss-%d.log" % rank, "w") as m, open("hdd.log", "w") as d:
    while p.returncode == None:
```

The following tasks are performed to gather the results:

- get current memory usage, update the peak memory usage if necessary and write the corresponding value converted from pages to kibibytes to the log;

```
        VmRSS = rss(p.pid)

        if VmRSS > VmHWM:
            VmHWM = VmRSS

        m.write("%g\n" % (VmRSS * 4. / 1024.))
        m.flush()
```

- collect disk usage statistics for the main process (rank 0), update the peak disk space usage if necessary and write the corresponding value converted from bytes to kibibytes to the log;

```
        if rank == 0:
            HddNow = hdd(tmpdir)

            if HddNow > HddPeak:
                HddPeak = HddNow

            d.write("%g\n" % (HddNow / 1024.))
            d.flush()
```

- sleep for one second and repeat.

```
        time.sleep(1)
        p.poll()
```

Eventually, before exiting, we append usage peaks to the associated logs.

```
    m.write("%g\n" % (VmHWM * 4. / 1024.))
    d.write("%g\n" % (HddPeak / 1024.))

sys.exit(p.returncode)
```

5.7 Result parsing

The Shell script `parse-test_FEMBEM.sh` parses data from a `test_FEMBEM` benchmark and store the result to a comma-separated values `.csv` file. It begins with a help message function that can be triggered using the `-h` option.

```
function help() {
  echo -n "Parse data from a test_FEMBEM run and store the result to a " >&2
  echo "comma-separated (*.csv) file." >&2
  echo "Usage: ./parse.sh [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -h                              Show this help message." >&2
  echo -n "  -s FILE                         Read the standard output of " >&2
```

```
  echo "test_FEMBEM from FILE rather than from the standard input." >&2
  echo -n "  -r PATH                          Read memory and disk usage " >&2
  echo "logs provided by rss.py from the directory at PATH." >&2
  echo -n "  -l FILE=FUNCTION[,FUNCTION]    Read the trace log of the " >&2
  echo -n "test_FEMBEM run from FILE and parse execution time and " >&2
  echo "iteration count of one or more FUNCTION(s)."
  echo -n "  -K KEY=VALUE[,KEY=VALUE]       Parse one or more additional " >&2
  echo "KEY=VALUE pairs to be added to the output (*.csv) file." >&2
  echo "  -H                               Do not print header on output." >&2
  echo -n "  -o FILE                          Specify the file to store the " >&2
  echo "output of the parsing to." >&2
}
```

Follows a generic error message function.

```
<<shell-error-function>>
```

There should be at least one argument provided on the command line.

```
if test $# -lt 1;
then
  help
  exit 1
fi
```

`STDOUT` holds the path to the input file containing `test_FEMBEM` standard output to parse.

```
STDOUT=""
```

`RSS` contains path and name pattern of input files containing the log of the memory resource monitoring script (see Section 5.6).

```
RSS=""
```

`TRACE` is the input file containing the trace call log produced by `test_FEMBEM`.

```
TRACE=""
```

`OUTPUT` is the path to the output file to store the extracted results to.

```
OUTPUT=""
```

`FUNCTIONS` contains names of functions to parse information about from the trace call log produced by `test_FEMBEM`.

```
FUNCTIONS=""
```

`CUSTOM_KV` holds custom key-value pairs to be included in the output `*.csv` file.

```
CUSTOM_KV=""
```

`DISABLE_HEADING` toggles heading in the output `*.csv` file.

```
DISABLE_HEADING=0
```

At this point, we can parse provided arguments and check the validity of option values.

```
while getopts ":hs:r:l:K:o:H" option;
do
  case $option in
```

Standard output of the `test_FEMBEM` executable run can be passed to the script either through its standard input or in a file specified using the `-s` option.

```
    s)
      STDOUT=$OPTARG

      if test ! -f $STDOUT;
      then
        error "'$STDOUT' is not a valid file!"
        exit 1
      fi
      ;;
```

When the `-r` option is provided, the script shall also parse storage resource monitoring logs.

```
    r)
      RSS=$OPTARG

      if test ! -d $RSS;
      then
        error "'$RSS' is not a valid directory!"
        exit 1
      fi
      ;;
```

When the `-l` option is provided, the script shall read the trace call log of `test_FEMBEM` from the file passed as argument and parse execution time, iteration count and floating-point operation (Flops) count of one or more functions.

```
    l)
      TRACE=$(echo $OPTARG | cut -d '=' -f 1)
      FUNCTIONS=$(echo $OPTARG | cut -d '=' -f 2 | sed 's/,/\t/g')

      if test ! -f $TRACE;
      then
        error "'$TRACE' is not a valid file!"
        exit 1
      fi
      ;;
```

The `-K` option allows to add one or more `KEY=VALUE` pairs into the output `*.csv` file.

```
    K)
      if test "$CUSTOM_KV" == "";
      then
        CUSTOM_KV="$OPTARG"
      else
        CUSTOM_KV="$CUSTOM_KV,$OPTARG"
      fi
      ;;
```

The `-H` option toggles printing of the header in the output `*.csv` file.

```
    H)
      DISABLE_HEADING=1
      ;;
```

Using the `-o` option, one can specify the name of the output `*.csv` file.

```
    o)
      OUTPUT=$OPTARG
      ;;
```

Eventually, we take care of unknown options or missing arguments and raise an error, if any.

```
    \?) # Unknown option
      error "Arguments mismatch! Invalid option '-$OPTARG'."
      echo
      help
      exit 1
      ;;
    :) # Missing option argument
      error "Arguments mismatch! Option '-$OPTARG' expects an argument!"
      echo
      help
      exit 1
      ;;
    h | *)
      help
      exit 0
      ;;
  esac
done
```

If the standard output of `test_FEMBEM` is not provided in a file, we try to read its content from the standard input of the script.

```
if test "$STDOUT" == "";
then
  rm -rf .input

  while read line;
  do
    echo $line >> .input
  done

  STDOUT=.input
fi
```

If no standard output from `test_FEMBEM` is found, we abort the script and raise an error.

```
if test $(wc --bytes $STDOUT | cut -d' ' -f 1) -lt 1;
then
  error "'$STDOUT' contains no 'test_FEMBEM' standard output to parse!"
  exit 1
fi
```

The treatment starts by separating custom key-value pairs, if any, by a tabulation character, so they can be looped over using a `for` loop.

```
CUSTOM_KV=$(echo $CUSTOM_KV | sed 's/,/\t/g')
```

We print the header line to the output file if it was not explicitly disabled using the `-H` option.

```
if test $DISABLE_HEADING -ne 1;
then
  echo -n "processes,by,mapping,ranking,binding,omp_thread_num," > $OUTPUT
  echo -n "mkl_thread_num,hmat_ncpu,mpf_max_memory,nbpts,radius," >> $OUTPUT
  echo -n "height,nbrhs,step_mesh,nbem,nbpts_lambda,lambda," >> $OUTPUT
  echo -n "thread_block_size,proc_block_size,disk_block_size," >> $OUTPUT
  echo -n "tps_cpu_facto_mpf,tps_cpu_solve_mpf,error," >> $OUTPUT
  echo -n "h_assembly_accuracy,h_recompression_accuracy," >> $OUTPUT
  echo -n "assembled_size_mb,tps_cpu_facto,factorized_size_mb," >> $OUTPUT
  echo -n "tps_cpu_solve,mumps_blr,mumps_blr_accuracy," >> $OUTPUT
  echo -n "mumps_blr_variant,coupled_method,coupled_nbrhs," >> $OUTPUT
  echo -n "size_schur,rm_peak,hdd_peak" >> $OUTPUT
```

The common header entries are followed by custom key-value pairs and additional details about selected functions optionally acquired from the trace call logs of `test_FEMBEM`.

```
  for kv in $CUSTOM_KV;
  do
    KEY=$(echo $kv | cut -d '=' -f 1)
    echo -n ",$KEY" >> $OUTPUT
  done

  for f in $FUNCTIONS;
  do
    echo -n ","$f"_exec_time,"$f"_nb_execs,"$f"_flops" >> $OUTPUT
  done

  echo >> $OUTPUT
fi
```

Next, we parse all the interesting information from the provided standard output of `test_FEMBEM`. The regular expressions and parameters of `grep` and `sed` utilities used in the script are chosen based on the output format of `test_FEMBEM` messages (see Listing 23 in Appendix).

```
processes=$(cat $STDOUT | head -n 1 | cut -d ' ' -f 2)
by=$(cat $STDOUT | head -n 1 | cut -d ' ' -f 4 | cut -d ':' -f 2)
mapping=$(cat $STDOUT | head -n 1 | cut -d ' ' -f 4 | cut -d ':' -f 3)
ranking=$(cat $STDOUT | head -n 1 | cut -d ' ' -f 6)
binding=$(cat $STDOUT | head -n 1 | cut -d ' ' -f 8)
omp_thread_num=$(cat $STDOUT | grep "OpenMP thread number" | cut -d '=' -f 2 | \
                 sed 's/[^0-9]//g')
```

```
mkl_thread_num=$(cat $STDOUT | grep "MKL thread number" | cut -d '=' -f 2 | \
                    sed 's/[^0-9]//g')
hmat_ncpu=$(cat $STDOUT | grep "HMAT_NCPU" | cut -d '=' -f 4 | \
               sed 's/[^0-9]//g')
mpf_max_memory=$(cat $STDOUT | grep "MPF_MAX_MEMORY" | cut -d '=' -f 2 | \
                   sed 's/[^0-9]//g')
nbpts=$(cat $STDOUT | grep "<PERFTESTS> NbPts =" | cut -d '=' -f 2 | \
          sed 's/[^0-9]//g')
radius=$(cat $STDOUT | grep "Reading radius" | cut -d '=' -f 2 | \
          sed 's/[^0-9.]//g')
height=$(cat $STDOUT | grep "Reading height" | cut -d '=' -f 2 | \
          sed 's/[^0-9.]//g')
nbrhs=$(cat $STDOUT | grep "<PERFTESTS> NbRhs" | cut -d '=' -f 2 | \
          sed 's/[^0-9]//g')
step_mesh=$(cat $STDOUT | grep "<PERFTESTS> StepMesh" | cut -d '=' -f 2 | \
               sed 's/[^-0-9e.+]//g')
nbem=$(cat $STDOUT | grep "<PERFTESTS> NbPtsBEM" | cut -d '=' -f 2 | \
        sed 's/[^0-9]//g')
nbpts_lambda=$(cat $STDOUT | grep "<PERFTESTS> nbPtLambda" | \
                  cut -d '=' -f 2 | sed 's/[^-0-9e.+]//g')
lambda=$(cat $STDOUT | grep "<PERFTESTS> Lambda" | cut -d '=' -f 2 | \
          sed 's/[^-0-9e.+]//g')
thread_block_size=$(cat $STDOUT | grep "thread block" | cut -d ':' -f 2 | \
                       cut -d 'x' -f 1 | sed 's/[^0-9]//g')
proc_block_size=$(cat $STDOUT | grep "proc block" | cut -d ':' -f 2 | \
                     cut -d 'x' -f 1 | sed 's/[^0-9]//g')
disk_block_size=$(cat $STDOUT | grep "disk block" | cut -d ':' -f 2 | \
                     cut -d 'x' -f 1 | sed 's/[^0-9]//g')
tps_cpu_facto_mpf=$(cat $STDOUT | grep "<PERFTESTS> TpsCpuFactoMPF =" | \
                       cut -d '=' -f 2 | sed 's/[^0-9.]//g')
tps_cpu_solve_mpf=$(cat $STDOUT | grep "<PERFTESTS> TpsCpuSolveMPF =" | \
                       cut -d '=' -f 2 | sed 's/[^0-9.]//g')
error=$(cat $STDOUT | grep "<PERFTESTS> Error" | cut -d '=' -f 2 | \
          sed 's/[^-0-9e.+]//g')
h_assembly_accuracy=$(cat $STDOUT | grep "\[HMat\] Compression epsilon" | \
                         cut -d ':' -f 2 | sed 's/[^-0-9e.+]//g')
h_recompression_accuracy=$(cat $STDOUT | \
                              grep "\[HMat\] Recompression epsilon" | \
                              cut -d ':' -f 2 | sed 's/[^-0-9e.+]//g')
assembled_size_mb=$(cat $STDOUT | grep "<PERFTESTS> AssembledSizeMb" | \
                       cut -d '=' -f 2 | sed 's/[^0-9.]//g')
tps_cpu_facto=$(cat $STDOUT | grep "<PERFTESTS> TpsCpuFacto =" | \
                   cut -d '=' -f 2 | sed 's/[^0-9.]//g')
factorized_size_mb=$(cat $STDOUT | grep "<PERFTESTS> FactorizedSizeMb" | \
                        cut -d '=' -f 2 | sed 's/[^0-9.]//g')
tps_cpu_solve=$(cat $STDOUT | grep "<PERFTESTS> TpsCpuSolve =" | \
                   cut -d '=' -f 2 | sed 's/[^0-9.]//g')
mumps_blr=$(cat $STDOUT | grep "Using Block-Low-Rank compression")

if test "$mumps_blr" != "";
then
  if test "$(echo $mumps_blr | grep \"NOT\")" != "";
  then
    mumps_blr=0
  else
    mumps_blr=1
  fi
fi

mumps_blr_accuracy=$(cat $STDOUT | \
                        grep "Accuracy parameter for Block-Low-Rank" | \
                        cut -d ':' -f 2 | sed 's/[^-0-9e.+]//g')
mumps_blr_variant=$(cat $STDOUT | grep "BLR Factorization variant" | \
```

```
                    cut -d ':' -f 2 | sed 's/[^-0-9e.+]//g')

coupled_method=""
if test "$(cat $STDOUT | grep 'Multi solve method.')" != "";
then
  coupled_method="multi-solve"
elif test "$(cat $STDOUT | grep 'Multi facto method.')" != "";
then
  coupled_method="multi-facto"
fi

coupled_nbrhs=$(cat $STDOUT | grep "Number of simultaneous RHS" | \
                  cut -d ':' -f 2 | sed 's/[^0-9.]//g')

if test "$coupled_nbrhs" == "";
then
  coupled_nbrhs=32 # Default value
fi

size_schur=$(cat $STDOUT | grep "Size of the block Schur" | \
                  cut -d ':' -f 2 | sed 's/[^0-9.]//g')
```

If the `-r` option was provided, we parse also storage resource monitoring logs. Memory usage log files are named `rss-x.log` where x is MPI process rank. Disk usage log files are named `hdd.log`.

```
if test "$RSS" != "";
then
  rm_peak="0.0"
  for rss in $(ls $RSS | grep -e "^rss");
  do
    rm_peak="$rm_peak + $(cat $RSS/$rss | tail -n 1)"
  done

  rm_peak=$(echo $rm_peak | bc -l)

  hdd_peak=$(cat $RSS/hdd.log | tail -n 1)
fi
```

Eventually, we print parsed values and optionally values from custom key-value pairs as well as additional function information from trace call logs of `test_FEMBEM`.

```
echo -n "$processes,$by,$mapping,$ranking,$binding,$omp_thread_num," >> $OUTPUT
echo -n "$mkl_thread_num,$hmat_ncpu,$mpf_max_memory,$nbpts," >> $OUTPUT
echo -n "$radius,$height,$nbrhs,$step_mesh,$nbem,$nbpts_lambda," >> $OUTPUT
echo -n "$lambda,$thread_block_size,$proc_block_size," >> $OUTPUT
echo -n "$disk_block_size,$tps_cpu_facto_mpf,$tps_cpu_solve_mpf," >> $OUTPUT
echo -n "$error,$h_assembly_accuracy,$h_recompression_accuracy," >> $OUTPUT
echo -n "$assembled_size_mb,$tps_cpu_facto,$factorized_size_mb," >> $OUTPUT
echo -n "$tps_cpu_solve,$mumps_blr,$mumps_blr_accuracy," >> $OUTPUT
echo -n "$mumps_blr_variant,$coupled_method,$coupled_nbrhs," >> $OUTPUT
echo -n "$size_schur,$rm_peak,$hdd_peak" >> $OUTPUT

for kv in $CUSTOM_KV;
do
  VALUE=$(echo $kv | cut -d '=' -f 2)
  echo -n ",$VALUE" >> $OUTPUT
done

for f in $FUNCTIONS;
```

```
do
  exec_time=$(cat $TRACE | grep $f | cut -d '|' -f 2 | sed 's/[^0-9.]//g')
  nb_execs=$(cat $TRACE | grep $f | cut -d '|' -f 3 | sed 's/[^0-9]//g')
  flops=$(cat $TRACE | grep $f | cut -d '|' -f 4 | sed 's/[^0-9]//g')
  echo -n ",$exec_time,$nb_execs,$flops" >> $OUTPUT
done
```

At the very end, we print the trailing new line character to the output file, perform cleaning and terminate.

```
echo >> $OUTPUT
rm -rf .input
exit 0
```

5.8 Database injecting

The Python script `inject.py` allows to call a custom parsing script (see Section 5.7), producing a `.csv` file on output and gather the results exported in the latter, then inject the values into a `gcvb` NoSQL database (see Section 5.1) for a possible result visualization using the `gcvb` dashbord feature (experimental). See a concrete usage of this script in Section 5.5.

At the beginning, we import necessary Python modules as well as the `gcvb` module.

```
import gcvb
import subprocess
import csv
import sys
```

The script expect the `-h` option or at least two arguments:

- `DATASET` which stands for the `.csv` file to gather the data from,
- `PARSER` which represents the path to the parsing script to use.

`DATASET` should be a `.csv.` file containing two lines, a heading providing the captions and the associated values.

May follow the arguments to call `PARSER` with.

The script continues with a help message function, that can be triggered with the `-h` option, and argument check.

```
def help():
    print("Launch PARSER (with ARGUMENTS if needed) instrumented to produce a ",
          "comma-separated values .csv output DATASET, deserialize it and ",
          "inject to the current gcvb database provided by the gcvb module.\n",
          "Usage: ", sys.argv[0], " DATASET PARSER [ARGUMENTS]\n\n",
          "Options:\n  -h    Show this help message.",
          file = sys.stderr, sep = "")

if len(sys.argv) < 2:
    print("Error: Arguments mismatch!\n", file = sys.stderr)
    help()

if sys.argv[1] == "-h":
```

```
    help()
    sys.exit(0)
```

In the main function, we parse the arguments

```
def main():
    args = sys.argv
    args.pop(0) # Drop the script name.
    dataset = args.pop(0)
    parser = args.pop(0)
```

and launch the parsing script.

```
    subprocess.run([parser] + args)
```

Once it's finished, we open the data set produced, gather data into a dictionary and inject key-value pairs one by one into the currently used `gcvb` database provided by the `gcvb` module.

```
    with open(dataset, mode = 'r') as data:
        reader = csv.DictReader(data)

        for row in reader:
            for item in row:
                gcvb.add_metric(item, row[item])

if __name__ == '__main__':
    main()
```

5.9 Job submission

To simplify the scheduling of the benchmark jobs specified in a definition file (see Section 5.5) based on their job names (see Section 5.2), we use the shell script `submit.sh`. It determines the list of benchmark sets from a given benchmark session, finds the corresponding Slurm batch script configuration headers and schedule the associated jobs.

We begin by defining a help message function that can be triggered if the script is run with the `-h` option.

```
function help() {
  echo "Submit benchmark runs at SESSION and make them run in parallel." >&2
  echo "Usage: $(basename $0) [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -h            Print this help message." >&2
  echo "  -E REGEX      Submit all benchmark runs but those verifying" \
       "REGEX (mutually exclusive with '-F')." >&2
  echo "  -F REGEX      Submit only the benchmark runs verifying REGEX." >&2
  echo "  -i ID[,ID]    Submit only the benchmark runs with ID." >&2
  echo -n "  -l            Only list the benchmark runs to run instead of " >&2
  echo "directly submitting them." >&2
  echo "  -s SESSION    Benchmark session to submit the runs of." >&2
  echo -n "  -w            Make the script wait until all of the submitted " >&2
  echo  "jobs complete." >&2
}
```

Then, we include a generic error function

```
<<shell-error-function>>
```

and parse options.

```
REGEX=""
EXCLUDE=0
INCLUDE=0
IDs=""
SESSION=""
LIST_ONLY=0
WAIT=0

while getopts ":hE:F:i:ls:w" option;
do
  case $option in
```

Using the `-E` option, one can decide to exclude from scheduling the benchmarks the names of which match a given regular expression.

```
    E)
      REGEX=$OPTARG
      EXCLUDE=1

      if test "$REGEX" == "";
      then
        error "Bad usage of the '-E' option (empty regular expression)!"
        exit 1
      fi
      ;;
```

Using the `-F` option, one can decide to schedule only the benchmarks the names of which match a given regular expression.

```
    F)
      REGEX=$OPTARG
      INCLUDE=1

      if test "$REGEX" == "";
      then
        error "Bad usage of the '-F' option (empty regular expression)!"
        exit 1
      fi
      ;;
```

The `-i` option allows to select concrete benchmarks to schedule by providing their identifiers (separated by commas).

```
    i)
      IDs="$IDs $(echo $OPTARG | sed 's/,/ /g')"

      if test "$IDs" == "";
      then
        error "Bad usage of the '-i' option (empty list of identifiers)!"
        exit 1
```

```
      fi
      ;;
```

The `-l` option enables to list the jobs to schedule without actually scheduling them. It may be particularly useful in combination with the `-F` option to verify whether the provided regular expression matches the intended benchmarks.

```
    l)
      LIST_ONLY=1
      ;;
```

The `-s` option is used to specify the benchmark session in the `results` directory (see Section 5.1) to schedule the jobs from.

```
    s)
      SESSION=$OPTARG

      if test ! -d $SESSION;
      then
        error "'$SESSION' is not a valid directory!"
        exit 1
      fi
      ;;
```

To make the script wait until all the scheduled jobs finish, the `-w` option can be used.

```
    w)
      WAIT=1
      ;;
```

Eventually, we have to take care of unknown options or missing arguments, if any, raise an error and terminate the script in that case.

```
    \?) # Unknown option
      error "Arguments mismatch! Invalid option '-$OPTARG'."
      echo
      help
      exit 1
      ;;
    :) # Missing option argument
      error "Arguments mismatch! Option '-$OPTARG' expects an argument!"
      echo
      help
      exit 1
      ;;
    h | *)
      help
      exit 0
      ;;
  esac
done
```

The `-E` and `-F`, `-F` and the `-i` options as well as the `-l` and `-w` options are mutually exclusive.

```
if test $EXCLUDE -ne 0 && test $INCLUDE -ne 0;
then
  error "Options '-E' and '-F' must not be used together!"
  exit 1
fi

if test "$REGEX" != "" && test "$IDs" != "";
then
  error "Options '-E' and '-F' must not be used together with the '-i' option!"
  exit 1
fi

if test $WAIT -ne 0 && test $LIST_ONLY -ne 0;
then
  error "Options '-l' and '-w' must not be used together!"
  exit 1
fi
```

After argument parse and check, we naviagte to the root of the `gcvb` filesystem of the given benchmark session folder.

```
cd $SESSION/../../
```

We also check whether we are in a valid `gcvb` filesystem contaning a `config.yaml` and a `results` folder (see Section 5.1).

```
if test ! -f config.yaml || test ! -d results;
then
  error "'$SESSION' is not a correct gcvb session directory!"
  exit 1
fi
```

We ensure to clean all the temporary folders we may have used before. To isolate our temporary files from other users, we use a dedicated temporary folder root `/tmp/vive-pain-au-chocolat`.

```
rm -rf /tmp/vive-pain-au-chocolat
if test $? -ne 0;
then
  error "Unable to clean any temporary folder(s) previously used!"
  exit 1
fi
```

Before submitting, we look for all the `sbatch` configuration headers. If there is none, there is no valid benchmark job to submit.

```
RUNS=$(find $SESSION -maxdepth 2 -type f -name "sbatch")

if test "$RUNS" == "";
then
  error "No sbatch file was found! No valid job for submission."
  exit 1
fi
```

The script loops over all the headers found and extracts the job name from each of them. We use an associative array to store the path to the associated header file for each benchmark job

name, e. g. `multi-solve-1-mumps-spido` (see Section 5.5).

```
declare -A BATCH_JOBS
```

In the session folder, there is one folder for each benchmark, not only per benchmark job possibly containing multiple benchmarks. So, the same header file may be present multiple times in different folders. Using an associative array, we manage to keep the path to only one copy of each header and prevent redundant job submissions.

```
for run in $RUNS
do
  JOB_NAME=$(grep "\-\-job-name" $run | cut -d '=' -f 2)
```

If the `-i` options is used to specify concrete benchmark identifiers, we need to filter the detected benchmarks.

```
  if test "$IDs" != "";
  then
    for id in $IDs;
    do
      if test $(echo $run | grep "$id" | wc -l) -gt 0;
      then
        BATCH_JOBS[$JOB_NAME]=$run
      fi
    done
```

Also, in the case the `-F` option followed by a regular expression is specified, we pick only the jobs the name of which matches the expression.

```
  elif test "$REGEX" == "" || \
      (test $INCLUDE -ne 0 && [[ "$JOB_NAME" =~ $REGEX ]]) || \
      (test $EXCLUDE -ne 0 && ! [[ "$JOB_NAME" =~ $REGEX ]]);
  then
    BATCH_JOBS[$JOB_NAME]=$run
  fi
done
```

If the user instruments the script to wait until the submitted jobs complete using the `-w` option, before actually submitting the jobs, we must reinitialize the list of jobs to wait for and clear any previous `.lastjob` file containing the identifier of the last submitted job (see below).

```
if test $WAIT -ne 0;
then
  JOB_LIST=""
  rm -f .lastjob
fi
```

Finally, we call the `gcvb` submission command for each job name with the `--filter-by-test-id` and `--header` options allowing us to specify a sub-set of jobs to submit and prepend associated shell scripts with corresponding Slurm batch script configuration headers.

```
SET_COUNT=${#BATCH_JOBS[@]}
i=1
```

```
for unique in "${!BATCH_JOBS[@]}";
do
  PREFIX="[ $i / $SET_COUNT ]"
```

If the `-l` option is specified, the jobs selected for scheduling are not submitted but only displayed on the screen for informative purposes.

```
  if test $LIST_ONLY -ne 0;
  then
    echo "$PREFIX Listing job '$unique'."
  else
    python3 -m gcvb --filter-by-test-id "$unique" compute \
            --header ${BATCH_JOBS[$unique]}
    if test $? -ne 0;
    then
      error "$PREFIX Failed to submit batch job '$unique'!"
      exit 1
    else
      echo "$PREFIX Submitted batch job '$unique'."
    fi
```

Also, if the `-w` option is set, we collect job identifiers of the launched jobs to make the script wait for the latter to complete before exiting itself. Here, we use the `.lastjob` file (see Section 5.4) produced everytime we submit a job with `gcvb` to gather the identifiers of scheduled jobs.

```
    if test $WAIT -ne 0;
    then
      while test ! -f .lastjob || \
          test $(wc -c .lastjob | cut -d ' ' -f 1) -lt 1;
      do
        sleep 1
      done

      JOB_LIST="$JOB_LIST $(cat .lastjob)"
      rm .lastjob
    fi
  fi

  i=$(expr $i + 1)
done
```

Once jobs are submitted through the `sbatch` command and if the `-w` option is set, we make the script wait until all the submitted jobs complete. We use the Slurm's `squeue` command to retrieve the list of running jobs.

```
if test $WAIT -ne 0;
then
  echo -n "Waiting for submitted jobs to complete... "

  while test "$JOB_LIST" != "";
  do
    STILL_RUNNING=""
    for id in $JOB_LIST;
    do
      if test $(squeue | grep "$id" | wc -l) -gt 0;
      then
```

```
        STILL_RUNNING="$STILL_RUNNING $id"
      fi
    done

    JOB_LIST=$STILL_RUNNING
    sleep 1m
  done
  echo "Done"
fi

exit 0
```

6 Post-processing results

6.1 Gathering data

6.1.1 Common benchmark results

Each `gcvb` benchmark produces a `.csv` file containing one line of results. To gather all these results into a single data frame, we use the R [16] script `gather.R`.

The script expects at least one argument which should be either the `-h` option to trigger a help message or the path to a `gcvb` benchmark session in a `results` folder (see Section 5.1) and optionally the path to the destination `.csv` file. If the latter is omitted, the file shall be placed into the current working directory under the name `dataframe.csv`. If no `.csv` file is found, no output is produced.

We begin by retrieving the command line arguments and prepare the usage message text.

```
argv = commandArgs(trailingOnly = TRUE)
usage = paste("Usage:", argv[0], "SESSION|-h [PATH]\n")
```

Then, we check the number of arguments and show an error message if it is not correct.

```
if(length(argv) < 1 | length(argv) > 2) {
  stop(paste("Error: Arguments mismatch!", usage, sep = "\n"))
```

If the first argument is the `-h` option, we show a help message.

```
} else if(argv[1] == "-h") {
  cat(
    paste(
      usage,
      paste(
        "Find and merge all the separate result files from the gcvb benchmark",
        "session at SESSION into a single (*.csv) file that shall be placed at",
        "PATH or into the current working directory of the script if PATH is",
        "not specified.\n"
      ),
      sep = "\n"
    )
  )
```

Otherwise, we begin the gathering of results by listing all the `.csv` files in the given `gcvb` benchmark session directory.

```
} else {
  library(plyr)
  library(dplyr)
  library(readr)

  files = list.files(
    path = argv[1],
    pattern = "*.csv",
    recursive = TRUE,
    full.names = TRUE
  )
```

Eventually, we read all the files and merge the contents into a single data frame that we store to a unique `.csv` file.

```
  if(length(files) > 0) {
    data = files %>% lapply(read_csv) %>% rbind.fill

    if(length(argv) == 2) {
      write.csv(data, argv[2])
    } else {
      write.csv(data, "dataframe.csv")
    }
  }
}
```

6.1.2 Individual benchmark results

In some cases, we need to extract a more detailed information from the log files produced by one or more benchmarks. For example, regarding residual memory (RAM) consumption, we gather only the peak usage values by default. Although, we may also need to know how the consumption evolves during the execution time. Furthermore, some benchmarks may produce additional data which is not taken into account by the common data gathering script (see Section 6.1.1).

Therefore, we define here the `extract.sh` shell script allowing to extract additional benchmark results for one or more user-specified benchmarks.

The script begins traditionnaly with a help message function that can be triggered using the `-h` option.

```
function help() {
  echo "Extract additional results for selected benchmark or benchmarks." >&2
  echo "Usage: ./extract.sh [options]" >&2
  echo >&2
  echo "Options:" >&2
  echo "  -h            Show this help message." >&2
  echo -n "  -B ID[,ID]    Select the benchmark matching ID. Multiple " >&2
  echo "identifiers may be specified to select multiple benchmarks." >&2
  echo "  -d PATH       Place the extracted files to PATH." >&2
  echo "  -r            Extract detailed real memory (RAM) consumption." >&2
  echo "  -s PATH       Search for benchmark results in PATH:" >&2
  echo "  -t            Extract the execution timeline. " >&2
}
```

Then, we include a generic error function

```
<<shell-error-function>>
```

and parse options.

```
BENCHMARKS=""
DESTINATION=$(pwd)
RSS=0
SOURCE=""
TIMELINE=0

while getopts ":hB:d:rs:t" option;
do
  case $option in
```

The `-B` option allows to specify one or more benchmarks to extract additional results for. If more than one identifier is present, they must be separated by commas. The latter are then converted to tabulations for easier post-processing.

```
    B)
      BENCHMARKS="$BENCHMARKS $(echo $OPTARG | sed 's/,/\t/g')"

      if test "$BENCHMARKS" == "";
      then
        error "Bad usage of the '-B' option (no identifiers specified)!"
        exit 1
      fi
      ;;
```

By default, the extracted files shall be placed to the current working directory of the script. One can use the `-d` option to specify another destination directory.

```
    d)
      DESTINATION=$OPTARG

      if test ! -d "$DESTINATION";
      then
        error "'$DESTINATION' is not a valid destination directory!"
        exit 1
      fi
      ;;
```

The `-r` option tells the script to extract detailed RAM consumption data.

```
    r)
      RSS=1
      ;;
```

The `-s` option allows one to specify the path to the `gcvb` benchmark session directory (see Section 5.1) to look for the results in.

```
    s)
      SOURCE=$OPTARG
```

```
      if test ! -d "$SOURCE";
      then
        error "'$SOURCE' is not a valid source directory!"
        exit 1
      fi
      ;;
```

Using the `-t` option, one can extract the execution timelines of the selected benchmarks. We also have to check that the external proprietary extraction tool from Airbus we use is available from the current working directory.

```
    t)
      TIMELINE=1

      if test ! -f ./timeline.py;
      then
        error "Timeline extraction tool was not found!"
        exit 1
      fi
      ;;
```

Eventually, we have to take care of unknown options or missing arguments, if any, raise an error and terminate the script in that case.

```
    \?) # Unknown option
      error "Arguments mismatch! Invalid option '-$OPTARG'."
      echo
      help
      exit 1
      ;;
    :) # Missing option argument
      error "Arguments mismatch! Option '-$OPTARG' expects an argument!"
      echo
      help
      exit 1
      ;;
    h | *)
      help
      exit 0
      ;;
  esac
done
```

The `-B` and the `-s` options are mandatory and at least one of the extraction options, `-r` or `-t`, must be specified as well.

```
if test "$SOURCE" == "";
then
  error "No source directory was specified! Nothing to do."
  exit 1
fi

if test "$BENCHMARKS" == "";
then
  error "No benchmark identifier was specified!"
  exit 1
```

```
fi

if test $RSS -eq 0 && test $TIMELINE -eq 0;
then
  error "Use at least one of the extraction options! Nothing to do."
  exit 1
fi
```

For each benchmark identifier specified using the `-B` option we:

- extract all of the real memory consumption logs if the `-r` option is present,

```
for b in $BENCHMARKS;
do
  if test $RSS -ne 0;
  then
    RSS_LOGS=$(ls $SOURCE/$b/rss-*.log)

    if test "$RSS_LOGS" == "";
    then
      error "There are no real memory (RAM) consumption logs at '$RSS_LOGS'!"
      exit 1
    fi

    COUNTER=0
    for l in $RSS_LOGS;
    do
      cp $l $DESTINATION/rss-$COUNTER-$b.log
      if test $? -ne 0;
      then
        error "Failed to extract the real memory (RAM) consumption log '$l'!"
        exit 1
      fi

      COUNTER=$(expr $COUNTER + 1)
    done
  fi
```

- extract the execution timelines from the standard output log of the selected benchamrks if the `-t` option is present.

```
  if test $TIMELINE -ne 0;
  then
    ./timeline.py $SOURCE/$b/stdout.log > $DESTINATION/timeline-$b.log

    if test $? -ne 0 || test ! -f $DESTINATION/timeline-$b.log;
    then
      error "Failed to extract the execution timeline of the benchmark '$b'!"
      exit 1
    fi
  fi
done
```

6.2 Data visualization

To visualize benchmark results we rely on the R language and a couple of additional R graphics libraries.

6.2.1 Prerequisites

At the beginning we need to import several R libraries for:

- manipulating data frames,

```
library(plyr)
library(dplyr)
library(readr)
```

- converting data frames from wide to long format (see below),

```
library(tidyr)
```

- defining and exporting plots.

```
library(ggplot2)
library(scales)
library(grid)
require(cowplot)
library(stringr)
```

6.2.2 Style presets

In this section, we define a set of common data visualization elements to ensure a unified graphical output when producing plots.

6.2.2.1 Label functions

Label function allows us to format and prettify column names or values when used in plot legends.

`label_percentage` multiplies `value` by 100 and returns the results as a string with percentage sign at the end.

```
label_percentage = function(value) {
  return(paste0(value * 100, "%"))
}
```

`label_epsilon` converts low-rank compression threshold values from numeric values to strings having the form 1×10^{-n} or to 'unset (no compression)' if the value is not assigned.

```
label_epsilon = function(labels) {
  out = c()

  for (i in 1:length(labels)) {
    if(!is.na(labels[i])) {
      out[i] = formatC(
        as.numeric(labels[i]),
        format = "e",
        digits = 1
      )
    } else {
      out[i] = "unset (no compression)"
    }
  }

  out
}
```

`label_storage` prettifies storage support names.

```
label_storage = function(labels) {
  out = c()
  for(i in 1:length(labels)) {
    out[i] = switch(
      as.character(labels[i]),
      "rm_peak" = "Real memory (RAM)",
      "vm_peak" = "Virtual memory (operating system)",
      "hdd_peak" = "Disk"
    )
  }

  out
}
```

`label_solver` prettifies solver names.

```
label_solver = function(labels) {
  out = c()
  for(i in 1:length(labels)) {
    out[i] = switch(
      as.character(labels[i]),
      "spido" = "SPIDO",
      "hmat-bem" = "HMAT",
      "hmat-fem" = "HMAT",
      "hmat-fem-nd" = "HMAT (proto-ND)",
      "mumps" = "MUMPS",
      "pastix" = "PaStiX",
      "mumps/spido" = "MUMPS/SPIDO",
      "pastix/spido" = "PaStiX/SPIDO",
      "mumps/hmat" = "MUMPS/HMAT",
      "hmat/hmat" = "HMAT"
    )
  }

  out
}
```

`label_mapping` transforms MPI process mapping settings into strings giving a more detailed information about the underlying parallel configuration.

```
label_mapping = function(labels) {
  out = c()
  for(i in 1:length(labels)) {
    out[i] = switch(
      as.character(labels[i]),
      "node" = "1 unbound MPI process \U2A09 1 to 24 threads",
      "socket" = "2 MPI processes bound to sockets \U2A09 1 to 12 threads",
      "numa" = "4 MPI processes bound to NUMAs \U2A09 1 to 6 threads",
      "core" = "1 to 24 MPI processes bound to cores \U2A09 1 thread"
    )
  }

  out
}
```

`label_nbpts` converts problem's unknown counts into the form `N = <count>` where `<count>` uses the scientific notation for the associated numerical values.

```
label_nbpts = function(labels) {
  out = c()

  for (i in 1:length(labels)) {
    out[i] = paste(
      "\U1D441 =",
      formatC(
        as.numeric(labels[i]),
        format = "e",
        digits = 1
      )
    )
  }

  out
}
```

`label_scientific` converts numeric label values to scientific notation.

```
label_scientific = function(labels) {
  out = c()

  for (i in 1:length(labels)) {
    out[i] = formatC(
      as.numeric(labels[i]),
      format = "e",
      digits = 1
    )
  }

  out
}
```

`label_coupling` prettifies names of the implementation schemes for the solution of coupled linear systems.

```
label_coupling = function(labels) {
  out = c()
  for(i in 1:length(labels)) {
    out[i] = switch(
```

```
      as.character(labels[i]),
      "multi-solve" = "Multi-solve (two-stage)",
      "multi-facto" = "Multi-factorization (two-stage)",
      "full-hmat" = "Partially sparse-aware (single-stage)",
    )
  }

  out
}
```

`label_peak_kind` prettifies peak memory usage value kinds.

```
label_peak_kind = function(labels) {
  out = c()
  for(i in 1:length(labels)) {
    out[i] = switch(
      as.character(labels[i]),
      "assembly_estimation" = expression("assembled system"),
      "schur_estimation" = expression("Schur complement matrix " * italic(S)),
      "peak_symmetric_factorization" = expression(
        "symmetric factorization of " * italic(A[vv])),
      "peak_non_symmetric_factorization" = expression(
        "non-symmetric factorization of " * italic(A[vv]))
    )
  }

  out
}
```

6.2.2.2 Label maps

A label map is a variation of a label function. It provides alternative names for given columns of a dataframe.

`phase.labs` provides prettified names for factorization and solve time columns `tps_facto` and `tps_solve`, the corresponding efficiency columns `efficiency_facto` and `efficiency_solve` as well as the column holding the execution time of the Schur block creation phase `CreateSchurComplement_exec_time`.

```
phase.labs = c(
  "Factorization",
  "Solve",
  "Factorization",
  "Solve",
  "Schur complement computation")

names(phase.labs) = c(
  "tps_facto",
  "tps_solve",
  "efficiency_facto",
  "efficiency_solve",
  "CreateSchurComplement_exec_time"
)
```

6.2.2.3 Theme

To preserve a unique color palette as well as sets of point shapes and line types throughout all the plots, we provide each legend entry with the information on its:

- color,

```
colors = c(
  "0.001" = "#F07E26",
  "1e-06" = "#9B004F",
  "node" = "#1488CA",
  "socket" = "#95C11F",
  "numa" = "#FFCD1C",
  "core" = "#6561A9",
  "32" = "#F07E26",
  "48" = "#9B004F",
  "64" = "#1488CA",
  "128" = "#95C11F",
  "256" = "#E63312",
  "250000" = "#384257",
  "5e+05" = "#1488CA",
  "1e+06" = "#E63312",
  "1200000" = "#95C11F",
  "1400000" = "#FFCD1C",
  "1500000" = "#6561A9",
  "multi-facto" = "#6561A9",
  "multi-solve" = "#89CCCA",
  "full-hmat" = "#C7D64F"
)
```

- point shape style,

```
shapes = c(
  "spido" = 16,
  "hmat-bem" = 15,
  "hmat-fem" = 15,
  "hmat-fem-nd" = 23,
  "mumps" = 16,
  "pastix" = 17,
  "mumps/spido" = 16,
  "pastix/spido" = 17,
  "mumps/hmat" = 15,
  "hmat/hmat" = 8,
  "rm_peak" = 16,
  "vm_peak" = 17,
  "hdd_peak" = 15
)
```

- line type.

```
linetypes = c(
  "spido" = "solid",
  "hmat-bem" = "dotted",
  "hmat-fem" = "dotted",
```

```
  "hmat-fem-nd" = "longdash",
  "mumps" = "solid",
  "pastix" = "dashed",
  "mumps/spido" = "solid",
  "pastix/spido" = "dashed",
  "mumps/hmat" = "dotted",
  "hmat/hmat" = "longdash",
  "rm_peak" = "solid",
  "vm_peak" = "dotted",
  "hdd_peak" = "longdash",
  "assembly_estimation" = "solid",
  "peak_symmetric_factorization" = "dotted",
  "peak_non_symmetric_factorization" = "longdash",
  "schur_estimation" = "dashed"
)
```

Furthermore, each plot object uses a set of common theme layers provided by the function `generate_theme`. If needed, the latter allows us to provide custom breaks for color, point shape or line type aesthetics, legend title, the number of lines allowed in the legend as well as the legend box placement (vertical or horizontal).

```
generate_theme = function(
  color_breaks = waiver(),
  color_labels = waiver(),
  shape_breaks = waiver(),
  shape_labels = waiver(),
  linetype_breaks = waiver(),
  linetype_labels = waiver(),
  legend_title = element_text(family = "Arial", size = 14, face = "bold"),
  legend_rows = 1,
  legend_rows_color = NA,
  legend_rows_fill = NA,
  legend_rows_shape = NA,
  legend_rows_linetype = NA,
  legend_box = "vertical",
  legend_position = "bottom"
) {
  return(list(
    scale_color_manual(
      values = colors,
      na.value = "#384257",
      labels = color_labels,
      breaks = color_breaks
    ),
    scale_fill_manual(
      values = colors,
      na.value = "#384257",
      labels = color_labels,
      breaks = color_breaks
    ),
    scale_shape_manual(
      values = shapes,
      labels = shape_labels,
      breaks = shape_breaks
    ),
    scale_linetype_manual(
      values = linetypes,
      labels = linetype_labels,
      breaks = linetype_breaks
    ),
    guides(
```

```
      shape = guide_legend(
        order = 1,
        nrow = ifelse(
          is.na(legend_rows_shape),
          legend_rows,
          legend_rows_shape
        ),
        byrow = TRUE
      ),
      linetype = guide_legend(
        order = 1,
        nrow = ifelse(
          is.na(legend_rows_linetype),
          legend_rows,
          legend_rows_linetype
        ),
        byrow = TRUE
      ),
      color = guide_legend(
        order = 2,
        nrow = ifelse(
          is.na(legend_rows_color),
          legend_rows,
          legend_rows_color
        ),
        byrow = TRUE
      ),
      fill = guide_legend(
        order = 2,
        nrow = ifelse(
          is.na(legend_rows_fill),
          legend_rows,
          legend_rows_fill
        ),
        byrow = TRUE
      )
    ),
```

We set the font family to Arial [1], the font size to 16 points and the legend text size to 16 points.

```
    theme_bw(),
    theme(
      text = element_text(family = "Arial", size = 16),
      legend.text = element_text(family = "Arial", size = 14),
      legend.title = legend_title,
```

We place the legend at the desired position, at the bottom of the plot by default, and sourround it with a rectangle.

```
      legend.position = legend_position,
      legend.background = element_rect(color = "gray40", size = 0.5),
```

We place legend items either side by side or one by line.

[1] The Arial typeface may not be available on your system. In this case, it shall be automatically substituted by the default sans-serif font without any impact on the contents of the figures. For information, possible alternatives are 'FreeSans', 'Nimbus Sans L' or 'Liberation Sans'.

```
      legend.box = legend_box,
```

For better visibility, we make legend symbols longer.

```
      legend.key.width = unit(3, "line"),
```

Finally, we rotate X-axis text to avoid issues with long labels.

```
      axis.text.x = element_text(angle = 60, hjust = 1)
    )
  ))
}
```

6.2.3 FXT trace plotting

To see what happens during the execution of an application based on the StarPU runtime, it is possible to instrument StarPU for it to produce an execution trace using the FXT tool [1].

We use the StarVZ library [23, 21] available for R to visualize FXT execution traces using `ggplot2` [3]. To adjust the appearance of the output plots produced by StarVZ, we use the Yaml configuration file `starvz-rr-2020.yaml`. See an example visualization of an FXT trace in Figure 3.

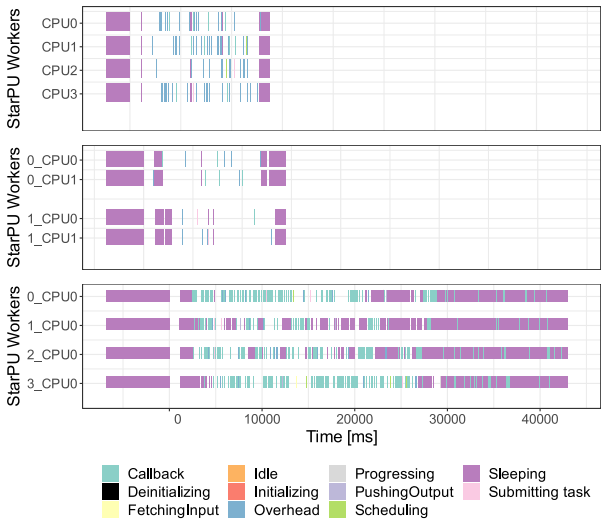


FIGURE 3: Example of a plot produced by `ggplot2` based on an FXT execution trace. Colors represents StarPU runtime actions and blank spaces correspond to actual computation tasks.

We leave cosmetic options take their default values, except the base font size (see `base_size` value), and we override those specifying what kind of information shall be included in resulting plots. In our case, we want to plot only the StarPU worker statistics (items `st` and `starpu`).

```
default:
  base_size: 14
  starpu:
    active: TRUE
    legend: TRUE
```

```
  ready:
    active: FALSE

  submitted:
    active: FALSE

  st:
    active: TRUE
```

Showing critical path time bound is not necessary.

```
    cpb: FALSE
```

On the other hand, we want to display all the labels, the legend and the application time span.

```
    labels: "ALL"
    legend: TRUE
    makespan: TRUE
```

To reduce the size of the resulting plot, we enable the aggregation of visualized tasks in a time step.

```
    aggregation:
      active: TRUE
      method: "static"
      step: 500
```

6.2.4 Generic plotting functions

In performance analysis, the metrics we study are in general the same. So, we prepared a series of generic plotting functions for each type of data visualization. This way, we can, for example, reuse the same function to plot disk usage peaks by problem unknown's count for any combination of solvers. The only parameter expected for these functions is a data frame.

6.2.4.1 `times_by_nbpts`

The function returns a plot of factorization and solve times relatively to linear system's unknown count for a given set of solvers.

We combine both metrics in one plot in which they are distinguished thanks to a facet grid. Although, to be able to use the latter, we need to convert the data frame from wide format where factorization and solve times have their own columns (see an example in Table 2) into long format where values from both columns are in one single column and a secondary column indicates whether in a row we store factorization or solve time (see an example in Table 3).

Solver	Factorization time [s]	Solve time [s]
SPIDO	125.58	30.2
H-MAT	40.21	8.64

TABLE 2: Mock table example in wide format featuring random values.

Solver	Computation phase	Time [s]
SPIDO	factorization	125.58
SPIDO	solve	30.2
H-MAT	factorization	40.21
H-MAT	solve	8.64

TABLE 3: Mock table example in long format featuring random values.

```
times_by_nbpts = function(dataframe) {
  dataframe_long = gather(
    dataframe,
    key = "computation_phase",
    value = "time",
    c("tps_facto", "tps_solve")
  )
```

Now, we can begin to define the plot object. The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `tps_facto` representing factorization time in seconds,
- `tps_solve` representing solve time in seconds,
- `desired_accuracy` representing low-rank compression threshold ϵ,
- `computation_phase` telling whether the row contains factorization or solve time (used in the long formatted data frame `dataframe_long`),
- `time` containing factorization or solve time (used in the long formatted data frame `dataframe_long`).

```
  plot = ggplot(dataframe_long, aes(x = nbpts, y = time))
```

In case the function is used to plot benchmark results of a solver using various data compression levels, we add a color aesthetics in order to distinguish among sets of points corresponding to each value of ϵ.

```
  if(!all(is.na(dataframe_long$desired_accuracy))) {
    plot = ggplot(
      dataframe_long,
      aes(
        x = nbpts,
        y = time,
        color = as.character(desired_accuracy),
        shape = solver,
        linetype = solver
      )
    )
  }
```

As there may be a considerable scale difference between factorization and solve time values, we use $\log_{10}$ scale on the Y-axis to be able to combine both metrics for each solver on the same axis without loosing on clarity.

```
  plot = plot +
    geom_line() +
    geom_point(size = 2.5) +
    scale_x_continuous(
      "# Unknowns (\U1D441)",
      trans = "log10",
      breaks = dataframe_long$nbpts,
      labels = scientific
    ) +
    scale_y_continuous(
      "Computation time [s]",
      trans = "log10",
      labels = scientific
    ) +
```

Finally, using facet grid, we distinguish between factorization and solve time on horizontal and between each solver considered on vertical. The plot theme is provided by the `generate_theme` function (see Section 6.2.1).

```
    facet_grid(
      . ~ computation_phase,
      labeller = labeller(computation_phase = phase.labs)
    ) +
    labs(color = "\U1D700", shape = "Solver", linetype = "Solver") +
    generate_theme(
      color_labels = label_epsilon,
      shape_labels = label_solver,
      linetype_labels = label_solver
    )

  return(plot)
}
```

6.2.4.2 `multisolve_times_by_nbrhs_and_nbpts`

The function returns a plot of factorization time, including the Schur complement computation phase, relatively to coupled linear system's unknown count for different counts of right-hand sides used during the Schur complement computation when relying on the two-stage multi-solve implementation scheme.

We begin by defining the plot object. The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `tps_facto` representing factorization time in seconds,
- `coupled_nbrhs` giving the count of right-hand sides used by the sparse solver during the Schur complement computation,
- `solver` containing the names of the solvers featured in the coupling.

```
multisolve_times_by_nbpts_and_nbrhs = function(dataframe) {
  plot = ggplot(
    dataframe,
    aes(
```

```
      x = nbpts,
      y = tps_facto,
      color = as.character(coupled_nbrhs),
      shape = solver,
      linetype = solver
    )
  ) +
  geom_line() +
  geom_point(size = 2.5) +
```

The X-axis shows the count of unknowns in the linear system and the Y-axis shows the factorization time phase.

```
  scale_x_continuous(
    "# Unknowns (\U1D441)",
    breaks = dataframe$nbpts,
    labels = scientific
  ) +
  scale_y_continuous("Factorization time [s]", labels = scientific) +
```

Finally, we set the legend title, apply the custom theme and return the plot object.

```
  labs(
    shape = "Solver coupling",
    linetype = "Solver coupling",
    color = expression(n[c])
  ) +
  generate_theme(
    color_breaks = c("32", "48", "64", "128", "256"),
    shape_labels = label_solver,
    linetype_labels = label_solver
  )

  return(plot)
}
```

6.2.4.3 `multisolve_rss_by_nbrhs_and_nbpts`

The function returns a plot of residual memory (RAM) usage peaks relatively to coupled linear system's unknown count for different counts of right-hand sides used during the Schur complement computation when relying on the two-stage multi-solve implementation scheme.

The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `rm_peak` giving real memory usage peaks, stored in mibibytes (MiB) but converted to gibibytes (GiB),
- `coupled_nbrhs` giving the count of right-hand sides used by the sparse solver during the Schur complement computation,
- `solver` containing the names of the solvers featured in the coupling.

We begin by defining the plot object directly.

```
multisolve_rss_by_nbpts_and_nbrhs = function(dataframe) {
  plot = ggplot(
    dataframe,
    aes(
      x = nbpts,
      y = rm_peak / 1024.,
      color = as.character(coupled_nbrhs),
      shape = solver,
      linetype = solver
    )
  ) +
  geom_line() +
  geom_point(size = 2.5) +
```

The X-axis shows the count of unknowns in the linear system and the Y-axis shows the RAM usage peaks.

```
  scale_x_continuous(
    "# Unknowns (\U1D441)",
    breaks = dataframe$nbpts,
    labels = scientific
  ) +
  scale_y_continuous(
    "Real memory (RAM) usage peak [GiB]",
    labels = function(label) sprintf("%.0f", label),
    limits = c(NA, 126),
    breaks = c(0, 30, 60, 90, 120)
  ) +
```

Finally, we set the legend title, apply the custom theme and return the plot object.

```
  labs(
    shape = "Solver coupling",
    linetype = "Solver coupling",
    color = expression(n[c])
  ) +
  generate_theme(
    color_breaks = c("32", "48", "64", "128", "256"),
    shape_labels = label_solver,
    linetype_labels = label_solver
  )

  return(plot)
}
```

6.2.4.4 `multifacto_times_by_nbpts_and_schur_size`

The function returns a plot of factorization time, including Schur complement computation phase, relatively to coupled linear system's unknown count and the number of blocks n_b the Schur complement submatrix is split into during the Schur complement computation when relying on the two-stage multi-factorization implementation scheme.

The Schur complement block size column is not the same for all the solver couplings we consider (column `disk_block_size` when SPIDO is used as dense solver and `schur_size` in case of HMAT), so we need to copy the data to the same column, `size_schur`.

```
multifacto_times_by_nbpts_and_schur_size = function(dataframe) {
  local = dataframe
  local$size_schur[local$solver == "mumps/spido"] =
    local$disk_block_size[local$solver == "mumps/spido"]
```

Now, we can define the plot object. The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `tps_facto` representing factorization time in seconds,
- `n_blocks` giving the count of blocks the Schur complement matrix was split to,
- `size_schur` giving the Schur complement block size in number of matrix lines or columns,
- `solver` containing the names of the solvers featured in the coupling.

```
  plot = ggplot(
    local,
    aes(
      x = n_blocks,
      y = tps_facto,
      color = as.character(nbpts),
      shape = solver,
      linetype = solver
    )
  ) +
  geom_line() +
  geom_point(size = 2.5) +
```

The X-axis shows the count of unknowns in the linear system and the Y-axis shows the factorization time.

```
  scale_x_continuous(
    expression(n[b]),
    breaks = local$n_blocks,
    trans = "log10"
  ) +
  scale_y_continuous(
    "Factorization time [s]",
    labels = scientific,
    trans = "log10"
  ) +
```

Finally, we set the legend title, apply the custom theme and return the plot object.

```
  labs(
    shape = "Solver coupling",
    linetype = "Solver coupling",
    color = "\U1D441"
  ) +
  generate_theme(
    color_breaks = c(
      "250000",
      "5e+05",
      "1e+06",
```

```
      "1200000",
      "1400000",
      "1500000"
    ),
    color_labels = label_scientific,
    legend_rows_color = 2,
    shape_labels = label_solver,
    linetype_labels = label_solver
  )

  return(plot)
}
```

6.2.4.5 `multifacto_rss_by_nbpts_and_schur_size`

The function returns a plot of residual memory (RAM) usage peaks relatively to coupled linear system's unknown count and the number of blocks n_b the Schur submatrix is split into during the Schur complement computation phase when relying on the two-stage multi-factorization implementation scheme.

The Schur complement block size column is not the same for all the solver couplings we consider (column `disk_block_size` when SPIDO is used as dense solver and `schur_size` in case of HMAT), so we need to copy the data to the same column, `size_schur`.

```
multifacto_rss_by_nbpts_and_schur_size = function(dataframe) {
  local = dataframe
  local$size_schur[local$solver == "mumps/spido"] =
    local$disk_block_size[local$solver == "mumps/spido"]
```

Now, we can define the plot object. The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `rm_peak` giving RAM usage peaks, stored in mibibytes (MiB) but converted to gibibytes (GiB),
- `n_blocks` giving the count of blocks the Schur complement matrix is split to,
- `size_schur` giving the Schur complement block size in number of matrix lines or columns
- `solver` containing the names of the solvers featured in the coupling.

```
  plot = ggplot(
    local,
    aes(
      x = n_blocks,
      y = rm_peak / 1024.,
      color = as.character(nbpts),
      shape = solver,
      linetype = solver
    )
  ) +
  geom_line() +
  geom_point(size = 2.5) +
```

The X-axis shows the count of unknowns in the linear system and the Y-axis shows the RAM usage peaks.

```
  scale_x_continuous(
    expression(n[b]),
    breaks = local$n_blocks,
    trans = "log10"
  ) +
  scale_y_continuous(
    "Real memory (RAM) usage peak [GiB]",
    labels = function(label) sprintf("%.0f", label),
    limits = c(NA, 126),
    breaks = c(0, 30, 60, 90, 120)
  ) +
```

Finally, we set the legend title, apply the custom theme and return the plot object.

```
  labs(
    shape = "Solver coupling",
    linetype = "Solver coupling",
    color = "\U1D441"
  ) +
  generate_theme(
    color_breaks = c(
      "250000",
      "5e+05",
      "1e+06",
      "1200000",
      "1400000",
      "1500000"
    ),
    color_labels = label_scientific,
    legend_rows_color = 2,
    shape_labels = label_solver,
    linetype_labels = label_solver
  )

  return(plot)
}
```

6.2.4.6 `compare_coupled`

The function returns a plot of the best factorization times, including the Schur complement computation phase, relatively to coupled linear system's unknown count for every implementation scheme for solving coupled systems.

We begin by restraining the input data frame to the best results of the two-stage schemes and the results of the single-stage scheme implemented using HMAT. Note that as the latter only implies the use of one single solver, we need to specify the coupling method name manually to enable the data frame treatment.

```
compare_coupled = function(dataframe) {
  dataframe_best = dataframe
  dataframe_best$coupled_method = as.character(dataframe_best$coupled_method)
  dataframe_best$coupled_method[dataframe_best$solver == "hmat/hmat"] =
  "full-hmat"
  dataframe_best = merge(
    aggregate(
```

```
    tps_facto ~ coupled_method + nbpts + solver,
    min,
    data = dataframe_best
  ),
  dataframe_best
)
```

Now, we can define the plot object. The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `tps_facto` representing the factorization time in seconds,
- `coupled_method` giving the name of the scheme for solving coupled FEM/BEM systems, e. g. multi-solve,
- `solver` containing the names of the solvers featured in the coupling.

```
plot = ggplot(
  dataframe_best,
  aes(
    x = nbpts,
    y = tps_facto,
    color = coupled_method,
    shape = solver,
    linetype = solver
  )
) +
geom_line() +
geom_point(size = 2.5) +
```

The X-axis shows the count of unknowns in the linear system and the Y-axis shows the factorization time.

```
scale_x_continuous(
  "# Unknowns (\U1D441)",
  breaks = dataframe_best$nbpts,
  labels = scientific
) +
scale_y_continuous(
  "Factorization time [s]",
  labels = scientific,
  trans = "log10",
  limits = c(NA, NA)
) +
```

Finally, we set the legend title, apply the default theme and return the plot object.

```
labs(
  shape = "Solvers",
  linetype = "Solvers",
  color = "Implementation\nscheme"
) +
generate_theme(
  color_breaks = c("multi-solve", "multi-facto", "full-hmat"),
  color_labels = label_coupling,
```

```
    shape_breaks = c("mumps/spido", "mumps/hmat", "hmat/hmat"),
    shape_labels = label_solver,
    linetype_breaks = c("mumps/spido", "mumps/hmat", "hmat/hmat"),
    linetype_labels = label_solver,
    legend_rows = 3,
    legend_box = "horizontal"
  )

  return(plot)
}
```

6.2.4.7 `accuracy_by_nbpts`

The function returns a plot of relative error of the solution approximation according to linear system's unknown count.

The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `error` giving relative forward error of the solution approximations.

```
accuracy_by_nbpts = function(dataframe) {
  plot = ggplot(
    dataframe,
    aes(
      x = nbpts,
      y = error,
      color = as.character(desired_accuracy),
      shape = solver,
      linetype = solver
    )) +
    geom_line() +
    geom_point(size = 2.5) +
    scale_x_continuous(
      "# Unknowns (\U1D441)",
      trans = "log10",
      breaks = dataframe$nbpts,
      labels = scientific
    ) +
```

For the Y-axis, we define the tick values manually for better comparison to the associated relative error values.

```
    scale_y_continuous(
      expression(paste("Relative error (", E[rel], ")")),
      limits = c(min(dataframe$error), max(dataframe$error)),
      breaks = c(1e-13, 1e-10, 1e-6, 1e-3),
      labels = scientific,
      trans = "log10"
    ) +
```

Finally, we set the legend labels, apply the common theme parameters and return the plot object.

```
    labs(color = "\U1D700", shape = "Solver", linetype = "Solver") +
    generate_theme(
      color_labels = label_epsilon,
      shape_labels = label_solver,
      linetype_labels = label_solver
    )

  return(plot)
}
```

6.2.4.8 `rss_peaks_by_nbpts`

The function returns a plot of residual memory (RAM) and disk space usage peaks relatively to linear system's unknown count. Exceptionally, this function can take two extra arguments, `limit_max` and `tick_values`, allowing to redefine respectively the default Y-axis maximum and tick values.

The column names featured in this plotting function are:

- `nbpts` giving linear system's unknown count,
- `rm_peak` representing real memory usage peaks, stored in mibibytes (MiB) but converted to gibibytes (GiB),
- `hdd_peak` representing disk space usage peaks, stored in MiB but converted to GiB.

At first, we convert the data frame from wide to long format (see Section 6.2.4.1).

```
rss_peaks_by_nbpts = function(dataframe, limit_max = 126,
                              tick_values = c(0, 30, 60, 90, 120)) {
  dataframe_long = gather(
    dataframe,
    key = "memory_type",
    value = "memory_usage",
    c("rm_peak", "hdd_peak")
  )
```

Also, we fix the order of the values in `memory_type` so that real memory (RAM) usage peaks come first before disk usage peaks in the facet grid.

```
dataframe_long$memory_type = factor(
  dataframe_long$memory_type,
  c('rm_peak', 'hdd_peak')
)
```

Then, we configure the plot object itself and handle different values of the low-rank compression threshold ϵ parameter by adding shape aesthetics.

```
  plot = ggplot(
    dataframe_long,
    aes(
      x = nbpts,
      y = memory_usage / 1024.,
      color = as.character(desired_accuracy),
```

```
      shape = solver,
      linetype = solver
    )
  ) +
  geom_line() +
  geom_point(size = 2.5) +
  scale_x_continuous(
    "# Unknowns (\U1D441)",
    trans = "log10",
    breaks = dataframe_long$nbpts,
    labels = scientific
  ) +
```

On the Y-axis, we round the values to 0 decimal places, set the axis top limit to 126 GiB and set the ticks manually for better visibility.

```
  scale_y_continuous(
    "Storage usage peak [GiB]",
    labels = function(label) sprintf("%.0f", label),
    limits = c(NA, limit_max),
    breaks = tick_values
  ) +
```

Storage types are distinguished using a facet grid.

```
  facet_grid(
    . ~ memory_type,
    labeller = labeller(memory_type = label_storage)
  ) +
```

We end by setting legend titles, applying the common theme parameters and returning the plot object.

```
  labs(color = "\U1D700", shape = "Solver", linetype = "Solver") +
  generate_theme(
    color_labels = label_epsilon,
    shape_labels = label_solver,
    linetype_labels = label_solver
  )

  return(plot)
}
```

6.2.4.9 `rss_by_time`

The function returns a plot of residual memory (RAM) usage relatively to the execution time. The first argument of the function corresponds to the data frame containing the resource consumption data. The function can take three extra arguments. `limit_max` and `tick_values` allow to redefine respectively the default Y-axis maximum and tick values. Also, if the function is used to visualize the results of benchmarks on coupled systems, `timeline` may provide a data frame to plot a series of vertical lines corresponding to selected phases of the computation from the associated execution timeline trace.

The column names featured in this plotting function are:

- `time` giving the time elapsed since the beginning of the execution t_0,
- `rss` representing RAM usage in mibibytes (MiB) but converted to gibibytes (GiB).

We begin directly by defining the plot object.

```
rss_by_time = function(dataframe, limit_max = 126,
                       tick_values = c(0, 30, 60, 90, 120),
                       timeline = NULL) {
  plot = ggplot(
    dataframe,
    aes(
      x = time,
      y = rss / 1024.,
      fill = aesthetics
    )
  ) +
```

We draw the RAM usage using an area chart.

```
  geom_area()
```

In the case the extra `timeline` argument is present, we include additional vertical lines in the plot as well as segments corresponding to selected memory usage peak values using the `geom_vline` and `geom_text` layers. Note that for the `geom_text` layer we also need to adjust the text position and rotation. To enable the processing of the labels as math expressions, we use the `parse` parameter.

```
  if(is.data.frame(timeline)) {
    plot = plot +
      geom_vline(
        data = timeline,
        aes(xintercept = time),
        linetype = "dotted",
        size = 0.4
      ) +
      geom_text(
        data = timeline,
        aes(
          x = middle,
          y = 0.75 * limit_max,
          label = label
        ),
        parse = T,
        angle = 90,
        vjust = "center"
      )
```

Before drawing the segment lines, we have to convert the `timeline` data frame to the long format (see Section 6.2.4.1). This allow us to use different sizes to distinguish different segments based on the type of peak value they represent. Notice that the `peak_non_symmetric_factorization` column is not always present in the data frame.

```
    columns <- c()
    if("peak_non_symmetric_factorization" %in% colnames(timeline)) {
      columns <- c("assembly_estimation", "peak_symmetric_factorization",
```

```
                    "peak_non_symmetric_factorization", "schur_estimation")
    } else {
      columns <- c("assembly_estimation", "peak_symmetric_factorization",
                    "schur_estimation")
    }
    timeline_long <- gather(
      timeline,
      key = "peak_kind",
      value = "peak_value",
      columns
    )
    plot <- plot +
      geom_spoke(
        data = timeline_long,
        aes(
          x = middle - 0.5 * (time - middle),
          y = peak_value,
          linetype = peak_kind,
          radius = time - middle
        ),
        angle = 2 * pi
      )
  }
```

Next, we set up the axis and their scaling.

```
  plot = plot +
    scale_x_continuous("Execution time [s]") +
    scale_y_continuous(
      "Real memory (RAM) usage [GiB]",
      labels = function(label) sprintf("%.0f", label),
      limits = c(NA, limit_max),
      breaks = tick_values
    )
```

We end by configuring the visual aspect of the plot according to the above.

```
  if(is.data.frame(timeline)) {
    plot <- plot +
      labs(
        fill = "Implementation scheme",
        linetype = "Estimated peak usage"
      ) +
      generate_theme(
        color_labels = label_coupling,
        linetype_labels = label_peak_kind,
        legend_rows_linetype = length(columns)
      )
  } else {
    plot <- plot +
      labs(fill = "Implementation scheme") +
      generate_theme(color_labels = label_coupling)
  }

  return(plot)
}
```

6.2.4.10 `scalability`

The function returns a plot of factorization and solve computation times relatively to available processor core count.

The column names featured in this plotting function are:

- `p_units` giving the count of cores,
- `tps_facto` representing factorization time in seconds,
- `tps_solve` representing solve time in seconds,
- `mapping` providing the mapping configuration of MPI processes.

Then, data frame is converted to long format (see Section 6.2.4.1) so we are able to combine times of both phases using a facet grid. Also, we force the order of facet labels for the `solver` column by refactoring the latter.

```
scalability = function(dataframe) {
  dataframe_no_NA = subset(dataframe, !is.na(dataframe$nbpts))

  dataframe_long = gather(
    dataframe_no_NA,
    key = "computation_phase",
    value = "time",
    c("tps_facto", "tps_solve")
  )

  dataframe_long$solver = factor(
    dataframe_long$solver,
    levels = c("spido", "hmat-bem", "hmat-fem", "hmat-fem-nd", "mumps")
  )

  plot = ggplot(
    dataframe_long,
    aes(
      x = p_units,
      y = time,
      group = mapping,
      color = mapping
    )
  ) +
  geom_line() +
  geom_point(size = 2.5, shape = 16) +
```

As there may be a considerable scale difference between factorization and solve time values, we use $\log_{10}$ scale on the Y-axis to be able to combine both metrics on the same axis without loosing on clarity.

```
  scale_x_continuous(
    "# Cores (\U1D450)",
    breaks = (dataframe_long$p_units)
  ) +
  scale_y_continuous(
    "Computation time [s]",
    labels = scientific,
    trans = "log10"
  ) +
```

Finally, using facet grid, we distinguish between factorization and solve times on horizontal and between each solver considered on vertical.

For the sake of clarity, we specify that scales on Y-axis can be different. It is useful when multiple solvers are considered having each considerably different execution time values.

Also, before returning the plot, we apply the common theme parameters. For scalability and for parallel efficiency plots, we split the legend items to two lines as the labels are too long.

```
  facet_grid(
    computation_phase ~ solver + nbpts,
    labeller = labeller(
      solver = label_solver,
      computation_phase = phase.labs,
      nbpts = label_nbpts
    ),
    scales = "free"
  ) +
  labs(color = "Parallel\nconfiguration") +
  generate_theme(
    color_breaks = c("node", "socket", "numa", "core"),
    color_labels = label_mapping,
    legend_rows = 4
  )

  return(plot)
}
```

6.2.4.11 `efficiency`

The function returns a plot of parallel efficiency, e. g. the percentage of computational resource usage, relatively to available processor core count.

The column names featured in this plotting function are:

- `p_units` giving the count of cores,
- `efficiency_facto` representing factorization phase efficiency,
- `efficiency_solve` representing solve phase efficiency,
- `mapping` providing the mapping configuration of MPI processes.

The function itself follows the exact same logic as in the case of scalability plotting function (see Section 6.2.4.10).

```
efficiency = function(dataframe) {
  dataframe_no_NA = subset(dataframe, !is.na(dataframe$nbpts))

  dataframe_long = gather(
    dataframe_no_NA,
    key = "computation_phase",
    value = "efficiency",
    c("efficiency_facto", "efficiency_solve")
  )

  dataframe_long$solver = factor(
    dataframe_long$solver,
    levels = c("spido", "hmat-bem", "hmat-fem", "hmat-fem-nd", "mumps")
```

```
  )

  plot = ggplot(
    dataframe_long,
    aes(
      x = p_units,
      y = efficiency,
      group = mapping,
      color = mapping
    )
  ) +
  geom_line() +
  geom_point(size = 2.5, shape = 16) +
  geom_hline(yintercept = 1) +
  scale_x_continuous(
    "# Cores (\U1D450)",
    breaks = (dataframe_long$p_units)
  ) +
```

There is only one additional step, i. e. to convert efficiency values from x such as $0 \leq x \leq 1$ to percentages using the `label_percentage` function (see Section 6.2.1). Also, we fix our own Y-axis tick values (breaks).

```
  scale_y_continuous(
    "Parallel efficiency [%]",
    breaks = c(0.0, 0.2, 0.4, 0.6, 0.8, 1.0),
    labels = label_percentage
  ) +
  facet_grid(
    computation_phase ~ solver + nbpts,
    labeller = labeller(
      solver = label_solver,
      computation_phase = phase.labs,
      nbpts = label_nbpts
    )
  ) +
  labs(color = "Parallel\nconfiguration") +
  generate_theme(
    color_breaks = c("node", "socket", "numa", "core"),
    color_labels = label_mapping,
    legend_rows = 4
  )

  return(plot)
}
```

6.2.5 Preparing global data frame

Benchmark results are collected into a one global `*.csv` file we use to construct an R data frame from. Prior to generating the plots, we have to import the `*.csv` file and format the results.

Notice that the `noweb` references (see Section 3) `«includes»`, `«styles»` and `«plotting-functions»` in the following code block point to the source code described in sections 6.2.1, 6.2.2 and 6.2.4 respectively.

```
<<includes>>
<<styles>>
<<plotting-functions>>
```

```
data = read.csv(
  file = "rr-2020.csv",
  header = TRUE
)
```

Names of columns holding the same information, such as factorization and solve time as well as the low-rank compression threshold ϵ, vary depending on the solver. To ease further data treatment, we fusion these into common columns.

```
data$tps_facto = ifelse(
  is.na(data$tps_cpu_facto_mpf),
  data$tps_cpu_facto,
  data$tps_cpu_facto_mpf
)

data$tps_solve = ifelse(
  is.na(data$tps_cpu_solve_mpf),
  data$tps_cpu_solve,
  data$tps_cpu_solve_mpf
)

data$desired_accuracy = ifelse(
  !is.na(data$mumps_blr_accuracy),
  data$mumps_blr_accuracy,
  NA
)

data$size_schur = ifelse(
  is.na(data$size_schur) & data$coupled_method == "multi-facto",
  data$disk_block_size,
  data$size_schur
)

data$desired_accuracy = ifelse(
  (data$solver == "hmat-bem" | data$solver == "hmat-fem" |
   data$solver == "hmat-fem-nd") &
  !is.na(data$h_assembly_accuracy),
  data$h_assembly_accuracy,
  data$desired_accuracy
)
```

Then, we must remove from the data frame the lines corresponding to unfinished jobs, e. g. lines without computation time information.

```
data = subset(data, !is.na(tps_facto))
data = subset(data, !is.na(tps_solve))
```

Some required columns are not present in the data frame by default, so we have to compute them based on existing data.

The total of cores used for computation is the combination of MPI process count and thread count.

```
data$p_units = data$omp_thread_num * data$processes
```

The value of `n_blocks` represents the number of blocks the dense submatrix A_{ss} is split into during the Schur complement computation in the two-stage implementation multi-factorization

scheme for solving coupled systems. When, this value is determined automatically by the solver, `n_blocks` equals to `auto`. Although, for the plotting functions to operate correctly, we need to recompute the corresponding numerical values.

```
data$n_blocks = ifelse(
  data$n_blocks == "auto",
  ceiling(data$nbem / data$size_schur),
  as.numeric(data$n_blocks)
)
```

Factorization and solve efficiency computation is split into several steps:

- gathering the best sequential execution times for each solver,

```
sequentials = subset(
  data,
  select = c("solver", "tps_facto", "tps_solve", "nbpts"),
  data$p_units == 1
)

sequentials = merge(
  aggregate(tps_facto ~ solver + nbpts, min, data = sequentials),
  sequentials
)

sequentials = sequentials %>% dplyr::rename(
                                    tps_facto_seq = tps_facto,
                                    tps_solve_seq = tps_solve
                                  )
```

- adding a column containing the sequential time for every benchmark in the data frame,

```
data = dplyr::left_join(data, sequentials, by = c("solver", "nbpts"))
```

- computing the efficiency itself.

```
data$efficiency_facto = data$tps_facto_seq / (data$p_units * data$tps_facto)
data$efficiency_solve = data$tps_solve_seq / (data$p_units * data$tps_solve)
```

6.2.6 Data frame sub-setting

Before using the generic plotting functions (see Section 6.2.4), the global data frame (see Section 6.2.5) needs to be split into multiple smaller data frames depending on the solver used, the linear system kind and so on. We end up having separate data frames containing:

- the results of benchmarks of the solvers SPIDO and HMAT on BEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_dense_parameters = subset(
  data,
  (solver == "spido" | solver == "hmat-bem") & variation == "parameters"
)
```

- the results of scalability benchmarks of the SPIDO and HMAT solvers on BEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_dense_scalability = subset(
  data,
  (solver == "spido" | solver == "hmat-bem") & variation == "scalability"
)
```

- the results of benchmarks of the MUMPS and HMAT solvers on FEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_sparse_symmetric_parameters = subset(
  data,
  ((solver == "mumps" & is.na(symmetry)) | (solver == "hmat-fem" &
                                            symmetry == "symmetric")) &
  variation == "parameters"
)
```

- the results of benchmarks of the HMAT solver with or without ND on non-symmetric FEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_sparse_non_symmetric_parameters = subset(
  data,
  (solver == "hmat-fem" | solver == "hmat-fem-nd" ) &
  (symmetry == "non-symmetric" | symmetry == "asymmetric")
)
```

- the results of scalability benchmarks of the MUMPS and HMAT solvers on FEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_sparse_scalability = subset(
  data,
  (solver == "mumps" | solver == "hmat-fem") & variation == "scalability"
)
```

- the results of benchmarks of existing implementation schemes for solving coupled FEM/BEM systems,

```
<<global-data-frame-rr-2020>>
dataframe_coupled = subset(
  data,
  solver == "mumps/spido" | solver == "mumps/hmat" | solver == "hmat/hmat"
)
```

- the results of benchmarks of the MUMPS solver on FEM systems of size matching the count of unknowns related to FEM discretization in case of the coupled FEM/BEM system benchmarks.

```
<<global-data-frame-rr-2020>>
dataframe_coupled_reference = subset(
  data, solver == "mumps" | !is.na(symmetry)
)
```

6.2.7 Reading a resource consumption data frame

In addition to the global data frame (see Section 6.2.5), we define a couple of auxiliary data frames based on the resource consumption data (see Section 5.6). During a benchmark execution, we measure the consumption of residual memory (RAM) periodically each second. All of these measures are stored into log files where each line corresponds to one measure and the very last line corresponds to the peak value.

In the current section, we define the function `read_rss` allowing to construct an R data frame based on a RAM consumption `log_file`. If the optional `aesthetics` argument is provided, the function adds an extra constant column in the resulting data frame initialized with the value of the argument. Adding such column may be useful later for using an aesthetics function (see Section 6.2.2) when drawing a plot based on the output data frame.

The first step is to read the data from `log_file`.

```
read_rss = function(log_file, aesthetics = NA) {
  log = read.table(log_file)
```

Then, we strip the very last line containing the peak value

```
  log = head(log, -1)
```

and add column name.

```
  colnames(log) = c("rss")
```

Finally, we insert a column into the data frame that shall contain the timestamp in seconds since the beginning of the execution, the time zero t_0.

```
  log$time = 0:(nrow(log) - 1)
```

Also, if the `aesthetics` argument is provided, we include also a constant column containing the value of the argument.

```
  log$aesthetics = rep(as.character(aesthetics), nrow(log))

  return(log)
}
```

6.2.8 Reading a timeline trace

The function `read_timeline` allows us to read a `test_FEMBEM` timeline trace `trace`, extract information on selected `events` from the latter and return it in form of a data frame suitable for drawing axis intercepting lines with the associated labels using the appropriate plotting function (see Section 6.2.4.9).

Note that `events` represents a list of lists each describing one event to extract. For example, to extract the entering time of the second call to the function `foo()`, one should include in `events` the list `{list("ENTER", "foo()", 2)`.

We begin by reading the timeline trace `trace`.

```
read_timeline <- function(trace, events) {
  complete_timeline <- read.delim(trace, header = FALSE)
```

For an easier column access, we add column names to the newly created data frame.

```
  colnames(complete_timeline) <- c("rank", "time", "type", "duration", "path")
```

Next, we construct the resulting `timeline` data frame containing only the data we are interested in (see the `events` argument). The new data frame shall contain three columns:

- `time` providing the corresponding event timestamp,
- `zero` representing a constant zero column useful for plotting vertical or horizontal axis intercepts,
- `middle` providing the values representing the middles of the intervals defined by the values of the `time` column.

Notice that the `middle` column is useful to center the labels plotted using the `geom_text` function.

```
  timeline <- data.frame(time = double(), zero = double(), middle = double())
  for(i in 1:length(events)) {
    time <- complete_timeline[
      complete_timeline$type == events[[i]][1] &
      str_detect(complete_timeline$path, paste0(events[[i]][2], "$")),
      "time"
    ]
    time <- time[[as.integer(events[[i]][3])]]
    middle <- ifelse(
      i > 1,
      time - ((time - timeline[i - 1, 1]) / 2),
      time / 2
    )
    event <- data.frame(time = time, zero = .0, middle = middle)
    timeline <- rbind(timeline, event)
  }
```

At the end, we return the `timeline` data frame.

```
  return(timeline)
}
```

6.2.9 StarVZ configuration

To plot FXT traces, we need to know the path to the current benchmark session and the `ggplot2`, `starvz` and `gridExtra` R packages. The variable `starvz_config` holds the name of the Yaml configuration file instrumenting the plot generation by the StarVZ package (see Section 6.2.3).

The StarVZ configuration begins by determining the path to the latest `gcvb` benchmark session directory.

```
output = try(
  system("ls -t ~/benchmarks/rr-2020/results | head -n 1", intern = TRUE)
)

latest_session_path = paste(
  "~/benchmarks/rr-2020/results",
  output,
  sep = "/"
)
```

Then, we import the required R libraries and set the `starvz_config` variable holding the path to the Yaml configuration file of the StarVZ package.

```
library(ggplot2)
library(starvz)
library(grid)

starvz_config = "./tangle/starvz-rr-2020.yaml"
```

6.2.10 Generating plots

Finally, relying on the `noweb` syntax (see Section 3), we generate the resulting plots based on the formatted data frames (see sections 6.2.6, 6.2.7, 6.2.8), using the common visual style (see Section 6.2.1) and our plotting functions (see Section 6.2.4).

```
<<dataframe-dense-parameters>>
times_by_nbpts(dataframe_dense_parameters)
```

Listing 2: Figure 18 in [17]

```
<<dataframe-dense-parameters>>
rss_peaks_by_nbpts(dataframe_dense_parameters)
```

Listing 3: Figure 19 in [17]

```
<<dataframe-dense-parameters>>
accuracy_by_nbpts(dataframe_dense_parameters)
```

Listing 4: Figure 20 in [17]

```
<<dataframe-dense-scalability>>
scalability(dataframe_dense_scalability)
```

Listing 5: Figure 21 in [17]

```
<<dataframe-dense-scalability>>
efficiency(dataframe_dense_scalability)
```

Listing 6: Figure 22 in [17]

```
<<dataframe-sparse-symmetric-parameters>>
times_by_nbpts(dataframe_sparse_symmetric_parameters)
```

Listing 7: Figure 24 in [17]

```
<<dataframe-sparse-non-symmetric-parameters>>
times_by_nbpts(dataframe_sparse_non_symmetric_parameters)
```

Listing 8: Figure 26 in [17]

```
<<dataframe-sparse-symmetric-parameters>>
accuracy_by_nbpts(dataframe_sparse_symmetric_parameters)
```

Listing 9: Figure 27 in [17]

```
<<dataframe-sparse-symmetric-parameters>>
rss_peaks_by_nbpts(dataframe_sparse_symmetric_parameters)
```

Listing 10: Figure 25 in [17]

```
<<dataframe-sparse-scalability>>
scalability(dataframe_sparse_scalability)
```

Listing 11: Figure 29 in [17]

```
<<dataframe-sparse-scalability>>
efficiency(dataframe_sparse_scalability)
```

Listing 12: Figure 30 in [17]

```
<<dataframe-coupled>>
compare_coupled(dataframe_coupled)
```

Listing 13: Figure 38 in [17]

We may need to further restrict the data frame used to generate a plot. The R function `subset` allows us to subset a data frame based on one or more conditions.

```
<<dataframe-coupled>>
multisolve_times_by_nbpts_and_nbrhs(
  subset(
    dataframe_coupled,
    coupled_method == "multi-solve"
  )
)
```

Listing 14: Figure 32 in [17]

```
<<dataframe-coupled>>
multisolve_rss_by_nbpts_and_nbrhs(
  subset(
    dataframe_coupled,
    coupled_method == "multi-solve"
  )
)
```

Listing 15: Figure 33 in [17]

```
<<dataframe-coupled>>
multifacto_times_by_nbpts_and_schur_size(
  subset(
    dataframe_coupled,
    coupled_method == "multi-facto"
  )
)
```

Listing 16: Figure 35 in [17]

```
<<dataframe-coupled>>
multifacto_rss_by_nbpts_and_schur_size(
  subset(
    dataframe_coupled,
    coupled_method == "multi-facto"
  )
)
```

Listing 17: Figure 36 in [17]

Sometimes, none of our generic plotting functions fits our needs. Therefore, we have to redefine the plot object manually.

```
<<dataframe-sparse-scalability>>
dataframe_mumps_scalability_ram = subset(
  dataframe_sparse_scalability,
  solver == "mumps" & nbpts == 4e+06
)

ggplot(
  dataframe_mumps_scalability_ram,
  aes(x = p_units, y = rm_peak / 1024., group = mapping, color = mapping)) +
  geom_line() + geom_point(size = 2.5, shape = 16) +
  scale_x_continuous("Core count (\U1D450)",
                     breaks = (dataframe_mumps_scalability_ram$p_units)) +
  scale_y_continuous("Real memory (RAM) usage peaks [GiB]",
                     labels = function(label) sprintf("%.0f", label),
                     limits = c(NA, 126), breaks = c(0, 30, 60, 90, 120)) +
  facet_grid(. ~ solver + nbpts,
             labeller = labeller(solver = label_solver, nbpts = label_nbpts)) +
  labs(color = "Parallel\nconfiguration") +
  generate_theme(color_breaks = c("node", "socket", "numa", "core"),
                 color_labels = label_mapping, legend_rows = 4)
```

Listing 18: Figure 28 in [17]

To generate figures based on FXT execution traces (see Section 6.2.3), we make use of the StarVZ package for R, the associated configuration (see Section 6.2.9) and visual style (see Section 6.2.3). In the code blocks below, the combination of StarVZ functions `panel_st_runtime` and `starvz_read` allows us to read an FXT trace and generate the corresponding plot. For lack of style adjustment options of the StarVZ package, we have to edit the resulting plot objects manually in order to tweak their visual aspect and arrange them in a grid view eventually.

Notice that the `noweb` reference `«starvz-header-rr-2020»` points to the source code described in Section 6.2.9.

```
<<starvz-header-rr-2020>>
hmat_bem_node = panel_st_runtime(starvz_read(
    paste(latest_session_path, "fxt-scalability-hmat-bem-node-1x4-25000",
          sep = "/"), starvz_config))

hmat_bem_socket = panel_st_runtime(starvz_read(
    paste(latest_session_path, "fxt-scalability-hmat-bem-socket-2x2-25000",
          sep = "/"), starvz_config))

hmat_bem_core = panel_st_runtime(starvz_read(
    paste(latest_session_path, "fxt-scalability-hmat-bem-core-4x1-25000",
          sep = "/"), starvz_config))

hmat_bem_node = hmat_bem_node + theme_bw() +
  theme(text = element_text(family = "Arial", size = 16))

hmat_bem_node$scales$scales[[1]]$range = c(NA, 44000)
hmat_bem_node$scales$scales[[1]]$limits = c(NA, 44000)
hmat_bem_node$theme$legend.position = "none"
hmat_bem_node$theme$axis.title.x = element_blank()
hmat_bem_node$theme$axis.text.x = element_blank()
hmat_bem_node$theme$axis.ticks.x = element_blank()

hmat_bem_socket = hmat_bem_socket + theme_bw() +
  theme(text = element_text(family = "Arial", size = 16))

hmat_bem_socket$scales$scales[[1]]$range = c(NA, 44000)
hmat_bem_socket$scales$scales[[1]]$limits = c(NA, 44000)
hmat_bem_socket$theme$legend.position = "none"
hmat_bem_socket$theme$axis.title.x = element_blank()
hmat_bem_socket$theme$axis.text.x = element_blank()
hmat_bem_socket$theme$axis.ticks.x = element_blank()

hmat_bem_core = hmat_bem_core + theme_bw() +
  theme(
    text = element_text(family = "Arial", size = 16),
    legend.text = element_text(family = "Arial", size = 14))

hmat_bem_core$scales$scales[[1]]$range = c(NA, 44000)
hmat_bem_core$scales$scales[[1]]$limits = c(NA, 44000)
hmat_bem_core$theme$legend.title = element_blank()
hmat_bem_core$theme$legend.position = "bottom"

grid.newpage()
grid.draw(rbind(ggplotGrob(hmat_bem_node), ggplotGrob(hmat_bem_socket),
                ggplotGrob(hmat_bem_core), size = "max"))
```

Listing 19: Figure 23 in [17]

```
<<starvz-header-rr-2020>>
hmat_fem_node = panel_st_runtime(starvz_read(
  paste(latest_session_path, "fxt-scalability-hmat-fem-node-1x4-50000",
        sep = "/"), starvz_config))

hmat_fem_socket = panel_st_runtime(starvz_read(
  paste(latest_session_path, "fxt-scalability-hmat-fem-socket-2x2-50000",
        sep = "/"), starvz_config))

hmat_fem_core = panel_st_runtime(starvz_read(
  paste(latest_session_path, "fxt-scalability-hmat-fem-core-4x1-50000",
        sep = "/"), starvz_config))

hmat_fem_node = hmat_fem_node + theme_bw() +
  theme(text = element_text(family = "Arial", size = 16))

hmat_fem_node$scales$scales[[1]]$range = c(NA, 101000)
hmat_fem_node$scales$scales[[1]]$limits = c(NA, 101000)
hmat_fem_node$theme$legend.position = "none"
hmat_fem_node$theme$axis.title.x = element_blank()
hmat_fem_node$theme$axis.text.x = element_blank()
hmat_fem_node$theme$axis.ticks.x = element_blank()

hmat_fem_socket = hmat_fem_socket + theme_bw() +
  theme(text = element_text(family = "Arial", size = 16))

hmat_fem_socket$scales$scales[[1]]$range = c(NA, 101000)
hmat_fem_socket$scales$scales[[1]]$limits = c(NA, 101000)
hmat_fem_socket$theme$legend.position = "none"
hmat_fem_socket$theme$axis.title.x = element_blank()
hmat_fem_socket$theme$axis.text.x = element_blank()
hmat_fem_socket$theme$axis.ticks.x = element_blank()

hmat_fem_core = hmat_fem_core + theme_bw() +
  theme(text = element_text(family = "Arial", size = 16),
        legend.text = element_text(family = "Arial", size = 14))

hmat_fem_core$scales$scales[[1]]$range = c(NA, 101000)
hmat_fem_core$scales$scales[[1]]$limits = c(NA, 101000)
hmat_fem_core$theme$legend.title = element_blank()
hmat_fem_core$theme$legend.position = "bottom"

grid.newpage()
grid.draw(rbind(ggplotGrob(hmat_fem_node), ggplotGrob(hmat_fem_socket),
                ggplotGrob(hmat_fem_core), size = "max"))
```

Listing 20: Figure 31 in [17]

Eventually, prior to plot a memory consumption evolution figure using the `rss_by_time` function (see Section 6.2.4.9), we have to read the entire memory log and the execution timeline. This way, we can highlight different computation phases in the resulting figure using labelled vertical lines. Furthermore, we represent peak memory consumption estimation for selected base operations thanks to horizontal lines. The required data, determined from logs or computed manually, is passed to `rss_by_time` through its `timeline` argument.

```
<<dataframe-coupled-reference>>
<<reading-rss>>
<<reading-timeline>>
peak_factorization = dataframe_coupled_reference[
  dataframe_coupled_reference$nbpts == 235165 &
  dataframe_coupled_reference$symmetry == "symmetric", "rm_peak"]
peak_factorization <- peak_factorization / 1024.

schur_estimation =
  ((((250000 - 235165) * (250000 - 235165)) / 2) * 16) / 1024. / 1024. / 1024.

log = read_rss("./rss-0-multi-solve-1-mumps-spido-256-250000.log",
               "multi-solve")

timeline = read_timeline(
  "./timeline-multi-solve-1-mumps-spido-256-250000.log",
  events = list(list("ENTER", "mpf_solvegemm\\(\\)", 1),
                list("EXIT", "mpf_solvegemm\\(\\)", 1),
                list("ENTER", "MatFactorPartial_META\\(\\)", 1),
                list("EXIT", "MatFactorPartial_META\\(\\)", 1)))

timeline$aesthetics = head(log$aesthetics, 4)
timeline$label = c("Linear~system~assembly",
                   "Block-wise~computation~of~italic(S)", NA,
                   "Factorization~of~italic(S)")

assembly_estimation <- log[log$time == floor(timeline$time[1]), "rss"]
assembly_estimation <- assembly_estimation / 1024.

timeline$assembly_estimation = c(NA, assembly_estimation, NA,
                                 assembly_estimation)
timeline$peak_symmetric_factorization = c(NA, assembly_estimation +
                                            peak_factorization, NA, NA)
timeline$schur_estimation = c(NA, schur_estimation, NA, schur_estimation)

rss_by_time(log, limit_max = 14, tick_values = c(0, 4, 8, 12),
            timeline = timeline)
```

Listing 21: Figure 34 in [17]

```
<<dataframe-coupled-reference>>
<<reading-rss>>
<<reading-timeline>>
peak_symmetric_factorization = dataframe_coupled_reference[
  dataframe_coupled_reference$nbpts == 235165 &
  dataframe_coupled_reference$symmetry == "symmetric", "rm_peak"]
peak_symmetric_factorization <- peak_symmetric_factorization / 1024.

peak_non_symmetric_factorization = dataframe_coupled_reference[
  dataframe_coupled_reference$nbpts == 235165 &
  dataframe_coupled_reference$symmetry == "non-symmetric", "rm_peak"]
peak_non_symmetric_factorization <- peak_non_symmetric_factorization / 1024.

schur_estimation =
  ((((250000 - 235165) * (250000 - 235165)) / 2) * 16) / 1024. / 1024. / 1024.

log = read_rss("./rss-0-multi-factorization-1-mumps-spido-250000-7418.log",
               "multi-facto")

timeline = read_timeline(
  "./timeline-multi-factorization-1-mumps-spido-250000-7418.log",
  events = list(list("ENTER", "CreateSchurComplement", 1),
                list("ENTER", "CreateSchurComplement", 2),
                list("ENTER", "CreateSchurComplement", 3),
                list("EXIT", "CreateSchurComplement", 3),
                list("ENTER", "MatFactorPartial_META\\(\\)", 1),
                list("EXIT", "MatFactorPartial_META\\(\\)", 1)))

timeline$aesthetics = head(log$aesthetics, 6)
timeline$label = c(
  "Linear~system~assembly", "Computation~of~italic(S[11])",
  "Computation~of~italic(S[21])", "Computation~of~italic(S[22])",
  "Factor.~of~italic(A[vv])~and~solve~of~italic(A[vv]^{-1}*b[v])",
  "Factorization~of~italic(S)")

assembly_estimation <- log[log$time == floor(timeline$time[1]), "rss"]
assembly_estimation <- assembly_estimation / 1024.

timeline$assembly_estimation = c(NA, assembly_estimation, assembly_estimation,
                                 assembly_estimation, assembly_estimation,
                                 assembly_estimation)
timeline$peak_symmetric_factorization = c(NA, NA, NA, NA,
                                          assembly_estimation +
                                          peak_symmetric_factorization, NA)
timeline$peak_non_symmetric_factorization = c(NA,
  assembly_estimation + peak_non_symmetric_factorization,
  assembly_estimation + peak_non_symmetric_factorization,
  assembly_estimation + peak_non_symmetric_factorization, NA, NA)
timeline$schur_estimation = c(NA, schur_estimation, schur_estimation,
                              schur_estimation, schur_estimation,
                              schur_estimation)

rss_by_time(log, limit_max = 20, tick_values = c(0, 3, 6, 9, 12, 15, 18),
            timeline = timeline)
```

Listing 22: Figure 37 in [17]

7 Conclusion

Throughout this technical report we provided an exhaustive description of the experimental environment used to realize the study presented in [17]. This includes our software choices, source codes, scripts and configurations we relied on to design, define, build and use the environment. The report may be considered as a result of our efforts to keep our work reproducible but also as a support for anyone facing the challenge of reproducible research.

8 Appendix

```
Testing : with MPF matrices.
...
[mpf] OpenMP thread number = 24
[mpf] MKL thread number = 24
...
Testing: BEM (dense matrix).
Reading nbPts = 25000
Reading radius = 2.000000
Reading height = 4.000000
...
<PERFTESTS> ScalarType = DOUBLE COMPLEX
...
<PERFTESTS> StepMesh = 4.483993e-02
<PERFTESTS> NbPts = 25000
<PERFTESTS> NbPtsBEM = 25000
<PERFTESTS> NbRhs = 50
<PERFTESTS> nbPtLambda = 1.000000e+01
<PERFTESTS> Lambda = 4.483993e-01
Computing RHS...

**** Computing Classical product...
BEM:   0% ......... 10% ... 90% ......... 100%  Done.
<PERFTESTS> TpsCpuClassic = 1.653491
  Size of thread block : [358 x 358]
  Size of proc block : [3572 x 3572]
  Size of disk block : [3572 x 3572]
MatAssembly_SPIDO :   0% ......... 10% ... 90% ......... 100%  Done.
Assembly done. Making the matrix symmetric...
<PERFTESTS> TpsCpuMPFCreation = 6.426428
VecAssembly_EMCP2 :   0% ......... 10% ... 90% ......... 100%  Done.
VecAssembly_EMCP2 :   0% ......... 10% ... 90% ......... 100%  Done.
MatFactorLDLt_SPIDO :   0% ......... 10% ... 90% ......... 100%  Done.
MIN = 2.4028e+00        MAX = 3.5494e+00        COND = 1.4772e+00
<PERFTESTS> TpsCpuFactoMPF = 85.717232
<PERFTESTS> TpsCpuSolveMPF = 5.684338

**** Comparing results...
<PERFTESTS> Error = 6.8930e-15
...
```

Listing 23: Excerpt of a `test_FEMBEM` output

References

[1] *FxT, Fast User/Kernel Tracing, a library for efficient trace recording.* `http://savannah.nongnu.org/projects/fkt`.

[2] *(g)enerate (c)ompute (v)alidate (b)enchmark, a Python 3 module aiming at facilitating non-regression, validation and benchmarking of simulation codes.* `https://github.com/felsocim/gcvb`.

[3] *ggplot2, a system for declaratively creating graphics.* `https://ggplot2.tidyverse.org/`.

[4] *GNU Emacs: An extensible, customizable, free/libre text editor — and more.* `https://www.gnu.org/software/emacs/`.

[5] *GNU Guix software distribution and transactional package manager.* `https://guix.gnu.org`.

[6] *Intel(R) Math kernel library.* `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`.

[7] *OpenBLAS, an optimized BLAS library.* `https://www.openblas.net/`.

[8] *Org mode documentation (Evaluating source code).* `https://orgmode.org/manual/Evaluating-Code-Blocks.html`.

[9] *Org mode documentation (Exporting).* `https://orgmode.org/manual/Exporting.html`.

[10] *Org mode documentation (Extracting source code).* `https://orgmode.org/manual/Extracting-Source-Code.html`.

[11] *Org mode documentation (Noweb Reference Syntax).* `https://orgmode.org/manual/Noweb-Reference-Syntax.html`.

[12] *PlaFRIM: Plateforme fédérative pour la recherche en informatique et mathématiques.* `https://plafrim.fr/`.

[13] *Quick start user's guide for slurm.* `https://slurm.schedmd.com/quickstart.html`.

[14] *slurm workload manager.* `https://slurm.schedmd.com`.

[15] *test_FEMBEM, a simple application for testing dense and sparse solvers with pseudo-FEM or pseudo-BEM matrices.* `https://gitlab.inria.fr/solverstack/test_fembem`.

[16] *The R project for statistical computing.* `https://www.r-project.org/`.

[17] E. Agullo, M. Felšöci, and G. Sylvand, *A comparison of selected solvers for coupled FEM/BEM linear systems arising from discretization of aeroacoustic problems*, Rapport de recherche RR-9412, Inria, June 2021.

[18] C. Augonnet, S. Thibault, and R. Namyst, *StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*, Rapport de recherche RR-7240, INRIA, Mar. 2010.

[19] C. Dominik, *The Org Mode 9.1 Reference Manual*, 12th Media Services, 2018.

[20] A. Falco, *Combler l'écart entre H-Matrices et méthodes directes creuses pour la résolution de systèmes linéaires de grandes tailles*, thèse de doctorat, Université de Bordeaux, June 2019.

[21] V. Garcia Pinto, L. Mello Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, *A visual performance analysis framework for task-based parallel applications running on hybrid clusters*, Concurrency and Computation: Practice and Experience, 30 (2018), p. e4472. e4472 cpe.4472.

[22] D. E. Knuth, *Literate programming*, Comput. J., 27 (1984), p. 97–111.

[23] V. G. Pinto, L. Stanisic, A. Legrand, L. M. Schnorr, S. Thibault, and V. Danjean, *Analyzing dynamic task-based applications on hybrid platforms: An agile scripting approach*, in 2016 Third Workshop on Visual Performance Analysis (VPA), 2016, pp. 17–24.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour

33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803