Towards Continuous Benchmarking: An Automated Performance Evaluation Framework for High Performance Software

Hartwig Anzt[*†], Yen-Chen Chen[‡], Terry Cojean[*], Jack Dongarra[†§¶], Goran Flegar[||],
Enrique S. Quintana-Ortí[||], Yuhsiang M. Tsai[‡], Weichung Wang[‡]
[*]Steinbuch Centre for Computing, Karlsruhe Institute of Technology, Germany
[†]Innovative Computing Lab (ICL), University of Tennessee, Knoxville, USA
[‡]Institute of Applied Mathematical Sciences, National Taiwan University, Taipei, Taiwan
[§]University of Manchester, Manchester, UK
[¶]Oak Ridge National Lab (ORNL), Oak Ridge, USA
[||]Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I Castellón, Spain
hartwig.anzt@kit.edu, yanjen224@gmail.com, terry.cojean@kit.edu, dongarra@icl.utk.edu, flegar@uji.es
quintana@uji.es, yhmtsai@gmail.com, wwangntu@gmail.com

***Abstract*—We present a framework that automates the process of testing and monitoring the performance of software libraries. Integrating this component into an ecosystem enables sustainable software development as a community effort via a web application for interactively evaluating the performance of individual software components. The performance evaluation tool is based exclusively on web technologies, which removes the burden of downloading performance data or installing additional software. The framework is currently integrated into the GINKGO sparse linear operator library, but allows for easy extension to cover other software projects. This enables the painless comparison of different high performance computing libraries.**

***Index Terms*—interactive performance visualization, automated performance benchmarking, continuous integration, healthy software lifecycle**

I. Introduction

Over the years, high performance computing (HPC) systems changed dramatically, and gradually became more complex. Current supercomputers typically consist of multiple layers of parallelism, heterogeneous compute nodes, and a complex cache hierarchy within every single processing unit. To effectively use these systems, high performance libraries have to reflect the complexity and heterogeneity of the architectures not only by providing back-ends for various hardware components, but also by integrating different programming models and algorithms that are suitable for the distinct hardware characteristics. In addition, the rapidly changing HPC landscape requires the software libraries to be amenable to modification and extension. These challenges increase the burden on the software developers. The growing size of developer teams can result in integration conflicts and increased complexity of the software stack.

The authors would like to thank the BSSw and the xSDK community efforts, in particular Mike Heroux and Lois Curfman McInnes, for promoting guidelines for a healthy software life cycle.

To increase productivity and software quality, tools for integration, testing, and code reviewing are constantly being improved. While the use of such tools has become the de-facto standard in industry, they are barely adopted by academic community. The primary reason is that academic software projects often arise as the by-product of a self-contained and limited research effort. At the same time, the lifespan of academic software regularly exceeds the duration of the specific research project, as the software gets extended beyond the original purpose. These extensions, however, are often workarounds to add functionality, or utilize existing software components that are not necessarily optimized for runtime performance. Hence, integrating gradually-extended software libraries into complex application codes can introduce performance bottlenecks that are difficult to track down. Therefore, it is important to provide the users of a library with easy access to performance analysis of the distinct software components.

In this work we design and deploy an interactive performance evaluation tool that propagates performance results via a web application. The data is automatically collected on HPC systems and archived in a remote repository — a strategy that allows to revisit "old" data. The performance evaluation framework builds upon open-source projects, and allows fine-grained analysis of performance data with respect to parameters and performance metrics customized by the user. The main contributions of the paper are:

- The design of a software development cycle featuring automatic performance evaluation on HPC systems and remote performance data archiving.
- The design and deployment of an automated performance evaluation tool that automatically retrieves performance data from a remote repository and allows to customize the analysis to the user requests.
- The design and deployment of a web application that

builds on web technology only, efficiently realizing the performance analysis as web service and removing the burden of downloading performance data to local disk or installing additional software.

d This work does not aim at developing an automated benchmark generator, which is a highly application-specific task requiring input from domain experts. Instead, the framework enables automatic scheduling of benchmark runs, as well as result data processing and its visualization. The motivation for such an automated / continuous benchmarking system is:

- **Reproducibility.** Reproducibility is a central pillar of natural science that is often left out in high performance benchmark studies. An automated performance evaluation system archiving all performance data along with execution parameters, plot scripts, machine settings and produced binary code allows to reproduce benchmark results at any point in the future.
- **Consistency.** Archiving performance results over the lifetime of a software library also allows to detect performance degradations and track down the modifications that triggered a performance loss. Coupling this with the Git Workflow and applying Continuous Benchmarking to all merged requests allows to tie every code change and feature addition to performance results. This is an important consideration in the development process of high performance software.
- **Usability.** An automated performance evaluation workflow allows launching benchmark tests with a few clicks in the web interface.

On top of a continuous benchmarking system, an open source web-based performance analysis framework like the GPE offers additional benefits:

- The design as a web service removes the need for downloading performance data to local disk or installing additional software.
- External software developers without access to HPC systems can contribute their code to the software library, invoke performance tests on an HPC platform (they do not have access to), and retrieve and analyze the benchmark results via the web service.
- A web-based performance explorer accessing open source repositories like the GPE enables easily comparison of the performance results from different software libraries by adapting the data access paths, since GPE relies on the JSON data format both for data storage and the JSONata transformation language for scripts.

In this work, we employ automated benchmarking framework and the interactive performance visualization for the GINKGO open-source software ecosystem, but the framework can be easily adopted by other software projects that employ a healthy software development cycle, similar to the one we present in Section II.

II. Software Development Ecosystem

In the past, software development was often a one-man-show or included only a small team of programmers. Nowadays, the increased hardware complexity and the demand for versatile software features requires software products to be developed as a team effort. The collaborative development of software is a challenge that requires the consideration of aspects such as sustainability, productivity, code readability and functionality, correctness, integration, and the synchronization of the distinct development efforts. Therefore, a healthy software life cycle employs an ecosystem where different tools used by developers are complemented with automatic features, all of them helping the development team to produce high quality software. Such ecosystems include code formatting tools, software versioning systems, automated compatibility and correctness checks, and community interaction tools. The Better Scientific Software initiative (BSSw [9]) aims to propagate measures and strategies to the scientific community that facilitate such a healthy software lifecycle. With a focus on high performance computing, resources like workshops, blogs, tutorials, and online learning materials are offered with the goal of improving the quality of scientific software, and simplifying its integration and interoperability. Software interoperability is also the main target of the xSDK [13] effort that aims to bundle existing software libraries into a coherent software stack. Its ultimate goal is to enable easy cross-compilation of different libraries, and to facilitate the combination of features taken from distinct software packages. For this purpose, the xSDK community has agreed on a set of policies that have to be adhered by all software packages part of the effort [13]. These policies, along with the sustainability measures propagated in the BSSw initiative, serve as guidelines for the development cycle that we employ for the GINKGO linear operator library. We provide an overview about GINKGO's software development cycle in Figure 1.

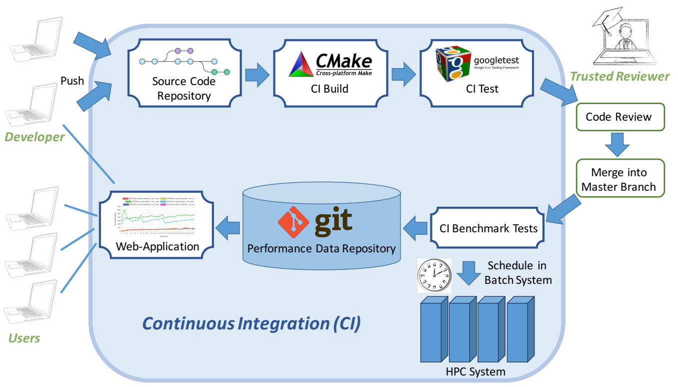


Fig. 1. The software development ecosystem of the GINKGO library.

We note that GINKGO is distributed as open-source software under the BSD 3-clause license, and that the complete software development ecosystem builds upon open source tools. The library itself has no external dependencies, and the extra components used for testing and benchmarking are licensed under either the MIT or BSD 3-clause license. However, these additional components can be manually deactivated, without removing any of the library's core functionalities.

GINKGO's source code is version-controlled using git. Git has established itself as the de-facto standard version control system for tracking changes in computer files and coordinating work on those files among multiple developers [3]. As a distributed revision control system, it particularly focuses on data integrity and speed. While git does support decentralized management of collaborative software efforts, most modern workflows, like Gitflow[1] used by GINKGO, assume a central repository available at all times. These workflows are supported by web services which provide hosting of git repositories. Among the most popular ones are GitHub [4], GitLab [5] and Bitbucket [1]. All of them offer hosting of open-source projects free of charge. These services also integrate community features facilitating collaborative development, such as pull/merge requests incorporating code review, issue and bug tracking, wiki pages and project website hosting. The GINKGO project uses both GitHub and GitLab to host its repositories. A public repository available to the wider community is hosted on GitHub[2], while a private version used for ongoing, unpublished research is hosted on GitLab. In both cases, GINKGO relies heavily on the above mentioned community features to organize the development effort.

Recent software development trends pursue the automation of an increased number of housekeeping tasks associated with software development, and bundle them in Continuous Integration (CI) systems [16]. They provide non-trivial computational capabilities to the otherwise static repository hosting, and can either be integrated in the hosting service (e.g., GitLab CI/CD), or realized as a separate service that communicates with repository hosting (e.g., Jenkins, Travis CI, AppVeyor). Usually, CIs are used to verify the integrity of the software after each change in the source code by compiling the software on a set of supported architectures and using different configurations. CIs can also be employed for testing the software's functionality using the utilities bundled with the source repository. At the same time, CI systems can also be used for other functionalities. GINKGO employs the GitLab CI/CD service for building the library, to run its unit tests, to synchronize between the public and the private repository, and to automatically generate and publish the user documentation. In this work we extend the CI pipelines to automatically run GINKGO's benchmark suite on predefined HPC systems and publish the collected results in a publicly available performance data repository, see Figure 1.

Software developers that want to add their components to the GINKGO software stack can create a fork of the public repository on GitHub, and submit a pull request with their changes. The CI system then selects a variety of compiler/hardware configurations, and tries to compile the source code in these environments, using GINKGO's CMake-based build system [19]. This automated workflow ensures a conflict-free compilation across a variety of supported platforms. All functionalities in GINKGO are covered by unit tests. Unit tests check the correctness of the smallest software entities and allow to quickly track down software bugs [20]. If the CI system succeeds in compiling on a specific hardware/-compiler configuration, the unit tests are invoked to check correctness. Writing the unit tests is facilitated via the Google Test framework [11]. Once all tests have passed, a member of the GINKGO core development team performs a manual code review. The reviewer ensures that all code is correct, follows GINKGO's code style guide, is well documented using Doxygen [21], adds useful functionality, and fits the scope of the software effort. In addition to reviewer's comments, a significant portion of GINKGO's code style is enforced by the clang-format [2] tool, which is integrated into GINKGO's build system.

The reviewer also has an important role as a gatekeeper: after the code is merged into the GINKGO software stack, benchmark tests on an HPC cluster are invoked to evaluate the code's performance. Running externally-contributed code on an HPC system poses a high security risk, and the reviewer approving the merge request has to carefully check the code for malware. Therefore, the trusted reviewer is someone with access privileges to the HPC system, and by approving the merge, he takes the responsibility for the code's integrity. Once approved, the continuous integration system inserts the benchmark tests into the cluster's scheduling system, and (once the tests have completed), collects the performance results. Those are archived in a distinct git repository that is designed as a comprehensive collection of performance characteristics. Archiving the performance results allows to monitor the performance of individual functionalities over the software's lifetime and the detection of possible performance degradations.

III. PERFORMANCE EVALUATION

The performance evaluation on a specific HPC system is automated via a series of jobs defined in GINKGO's CI system configuration file (`.gitlab-ci.yml` by default). These performance benchmark jobs are defined as "scheduled", which means they are not invoked automatically at every repository update, but can be set to execute at fixed intervals via GitLab's web interface.[3]

Once the benchmark runs are invoked, the CI server establishes an SSH connection to the target HPC system. The GINKGO repository is cloned to the server, and the library is compiled using its build system. Next, a set of benchmark tests is submitted to the HPC system's job scheduler. The exact sequence of commands to facilitate this depends on the scheduler employed by the system, and is fully configurable in the CI configuration file. Finally, once all benchmarks are completed, the CI job collects the results and uploads them into the performance data repository.

There exist different strategies to detect the completion of the performance tests. A first startegy keeps the SSH connection to the HPC system for the duration of the benchmark

[1] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

[2] https://github.com/ginkgo-project

[3] https://docs.gitlab.com/ee/user/project/pipelines/schedules.html

execution. This is an adequate solution if the network is guaranteed to be stable, e.g. if the CI server and the HPC system are located in the same local network. When connecting to a remote system, the assumption of a stable network connection may not be realistic. An alternative strategy closes the connection to the HPC system as soon as the benchmarks are submitted to the server's job scheduling system, and a separate job is used to collect the benchmark results and upload them into the performance data repository. This job can be triggered from the script running on the HPC cluster via a GitLab trigger[4], by sending a `POST` message to the GitLab web API. In case the HPC system does not allow web access, the job can be configured to check the completion status at regular intervals. This last strategy is the one we currently use.

The output format in which the performance data is stored has to allow for easy interaction between the benchmark runners, the web application, and third-party applications. To that end, the data exchange format should be chosen carefully with respect to support for low-level programming languages as well as scripting languages used for web development. In our ecosystem, we choose the JSON [14] data format as it has become the de-facto standard for web applications and has native support in most higher-level languages (Python, MATLAB, Javascript). Furthermore, libraries providing JSON interfaces are available for low-level programming languages such as C and C++. In GINKGO, we employ RapidJSON [22] to generate JSON files in the benchmark suite.

IV. PERFORMANCE VISUALIZATION

While the previous steps of the performance benchmarking workflow were assembled by using existing open-source components, we were unable to identify a suitable tool to enable rapid performance visualization. Such a tool has to quickly provide library developers with insight about the behavior of their algorithms. In addition, it should offer useful information about the library's performance to existing and prospective users. Ideally, the users do not have to install any additional software nor manually download performance results.

To fulfill these requirements, we developed the "GINKGO performance explorer"[5] (GPE). This web application automatically retrieves the data from the performance data repository, and visualizes it in a web browser. This implies that a web browser alone is sufficient to access and analyze the performance results: the web-based performance analysis framework does not require the installation of additional software or downloading performance data. GPE works on all major operating systems (Linux, Mac and Windows) and we tested the correct functionality using current versions of the Firefox, Chrome, and Safari browsers.

We employed the Angular framework [8] to implement the application logic as well as the interaction in-between the distinct components and the communication with the rest of the web (e.g., to download performance data). We used Angular's Material user interface (UI) components for the application's layout and form controls. We also provide a component that visualizes the data retrieved from the JSON performance files, using the Chart.js plotting library [10].

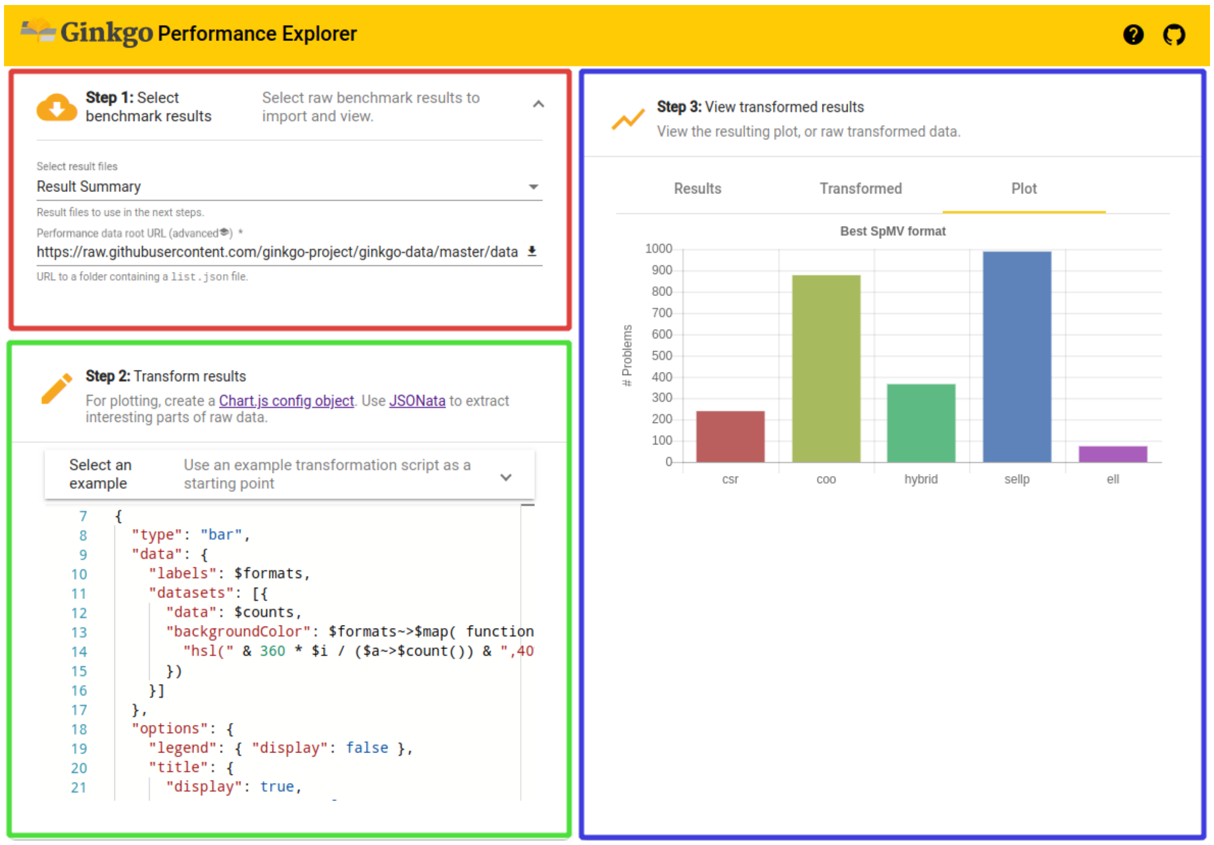
Library developers likely need the flexibility to customize performance graphs to focus on a specific aspect. To enable this flexibility we decided to embed a powerful domain-specific scripting language which can be used to extract subsets of the raw data and to transform them into a format that can be used as input for the visualization. JSONata [6] is a scripting language designed for acting on JSON data. A JSONata open-source compiler in the form of a web component is also available. We adopted both, the JSON scripting language along with the compiler in GPE to provide the required scripting capabilities. For convenience, GPE also features the open-source Monaco editor web component [12], which allows to develop JSONata scripts directly inside the web interface. Interconnecting it with the JSONata compiler enables "as-you-type" syntax checking, result transformation, and plotting.

While providing additional convenience, a downside of "as-you-type" is a noticeable performance degradation when working with large datasets, as the compilation and execution of the script is currently handled by the thread in charge of the UI and the visualization process. A temporary workaround is to deliberately introduce a syntax error when writing the script. This will cause the low-cost compilation process to terminate, and the costly dataset transformation will not be invoked. Once the script is finished, the syntax error should be removed, and the whole compilation and the JSON transformation process will take place. Additionally, the tool currently requires an active web connection to obtain the datasets from the repository. However, once the datasets are retrieved, the connection is no longer required. Both problems described can be solved via Service Workers[6]. They are implemented as separate processes from the UI thread, and can be used to offload computation, which solves performance degradation problems. Service Workers can also intercept network requests, and serve a cached version of the data as response to data requests, which allows the use of GINKGO without web access. We are currently adopting Service Workers in the GPE framework.

By combining the features described in this section, GPE provides the flexibility of tools like MATLAB or Python, while efficiently decreasing the effort for library users and developers:

1) No additional software has to be installed;
2) No performance data has to be downloaded to local disk;
3) The raw performance data is automatically retrieved from the repository;
4) Using a language specifically designed to transform JSON files, the data extraction scripts are simpler than their MATLAB / Python counterparts; and

[4] https://docs.gitlab.com/ee/ci/triggers/
[5] https://ginkgo-project.github.io/gpe
[6] https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API


Ginkgo Performance Explorer

Step 1: Select benchmark results
Select raw benchmark results to import and view.

Select result files
Result Summary
Result files to use in the next steps.
Performance data root URL (advanced) *
https://raw.githubusercontent.com/ginkgo-project/ginkgo-data/master/data
URL to a folder containing a `list.json` file.

Step 2: Transform results
For plotting, create a Chart.js config object. Use JSONata to extract interesting parts of raw data.

Select an example — Use an example transformation script as a starting point

```
{
  "type": "bar",
  "data": {
    "labels": $formats,
    "datasets": [{
      "data": $counts,
      "backgroundColor": $formats~>$map( function
        "hsl(" & 360 * $i / ($a~>$count()) & ",40
      })
    }]
  },
  "options": {
    "legend": { "display": false },
    "title": {
      "display": true,
```

Step 3: View transformed results
View the resulting plot, or raw transformed data.

Results | Transformed | Plot

Fig. 2. GINKGO Performance Explorer layout. Red box: *dataset selection dialog*; Green box: *transformation script editor*; Blue box: *data and plot viewer*.

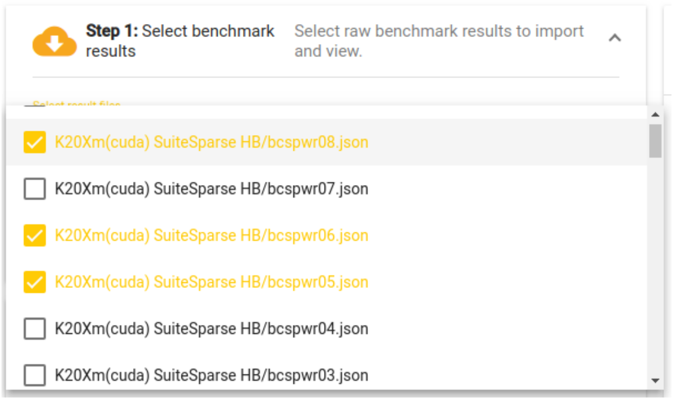
5) The visualization of the converted data is automated in the web application.

V. OVERVIEW OF “GINKGO PERFORMANCE EXPLORER”

This section provides a step-by-step user tutorial of GPE. Hands-on experience is enabled by accessing GPE on GINKGO’s GitHub pages.[7] For those interested in extending the capabilities of the web application, the source code is also available on GitHub[8] under the MIT license.

The web application is divided into three components, as shown in Figure 2. On the top left (and marked in red) is the *data selection dialog*. The dialog is used to retrieve the raw performance data from the performance repository. Clicking on the “Select result files” control opens a multiple select pop–up dialog listing available performance data. By default, the application uses GINKGO’s performance data repository[9] to populate the list of available performance result files. However, an alternative performance database location (e.g. containing performance data for a different library) can be provided via the “Performance root URL”. The value of this control can be changed, and after clicking the download button on the right of the control, GPE will try to read the `list.json` file from the provided URL. This

[7] https://ginkgo-project.github.io/gpe/
[8] https://github.com/ginkgo-project/gpe
[9] github.com/ginkgo-project/ginkgo-data


Step 1: Select benchmark results
Select raw benchmark results to import and view.

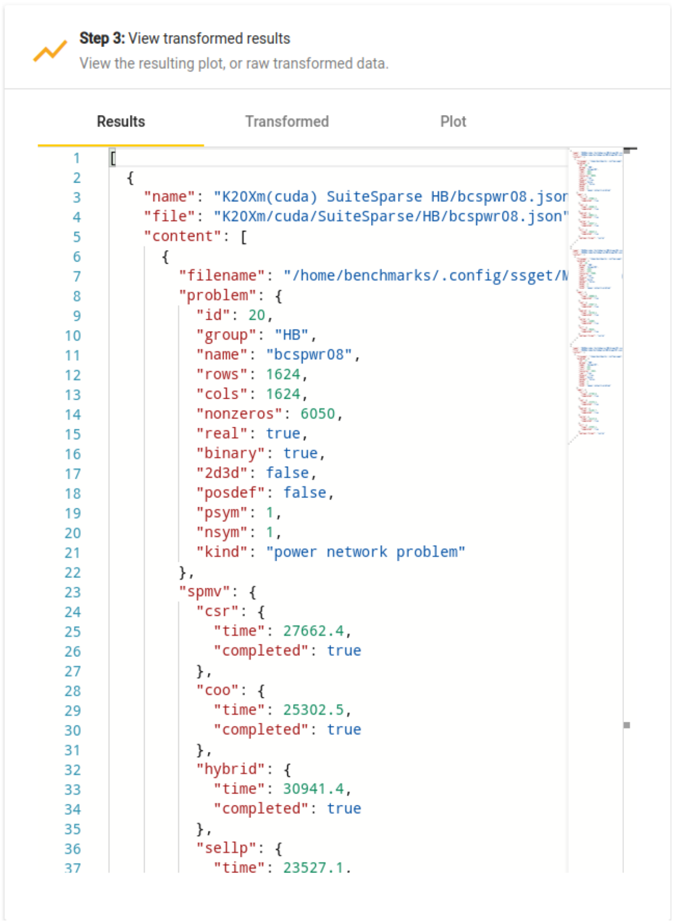
- [x] K20Xm(cuda) SuiteSparse HB/bcspwr08.json
- [ ] K20Xm(cuda) SuiteSparse HB/bcspwr07.json
- [x] K20Xm(cuda) SuiteSparse HB/bcspwr06.json
- [x] K20Xm(cuda) SuiteSparse HB/bcspwr05.json
- [ ] K20Xm(cuda) SuiteSparse HB/bcspwr04.json
- [ ] K20Xm(cuda) SuiteSparse HB/bcspwr03.json

Fig. 3. The performance result selection pop-up dialog.

```
[{ "name": "A", "file": "path/to/A.json" },
 { "name": "B", "file": "path/to/B.json" }]
```

Fig. 4. Example `list.json` file.

file lists the names and locations of the performance results. For example, if a database contains two data files located at “`http://example.com/data/path/to/A.json`” and “`http://example.com/data/path/to/B.json`,” then a “`list.json`” file with content as


Step 3: View transformed results
View the resulting plot, or raw transformed data.
Results
Transformed
Plot

```
[
  {
    "name": "K20Xm(cuda) SuiteSparse HB/bcspwr08.jso
    "file": "K20Xm/cuda/SuiteSparse/HB/bcspwr08.json"
    "content": [
      {
        "filename": "/home/benchmarks/.config/ssget/M
        "problem": {
          "id": 20,
          "group": "HB",
          "name": "bcspwr08",
          "rows": 1624,
          "cols": 1624,
          "nonzeros": 6050,
          "real": true,
          "binary": true,
          "2d3d": false,
          "posdef": false,
          "psym": 1,
          "nsym": 1,
          "kind": "power network problem"
        },
        "spmv": {
          "csr": {
            "time": 27662.4,
            "completed": true
          },
          "coo": {
            "time": 25302.5,
            "completed": true
          },
          "hybrid": {
            "time": 30941.4,
            "completed": true
          },
          "sellp": {
            "time": 23527.1,
```

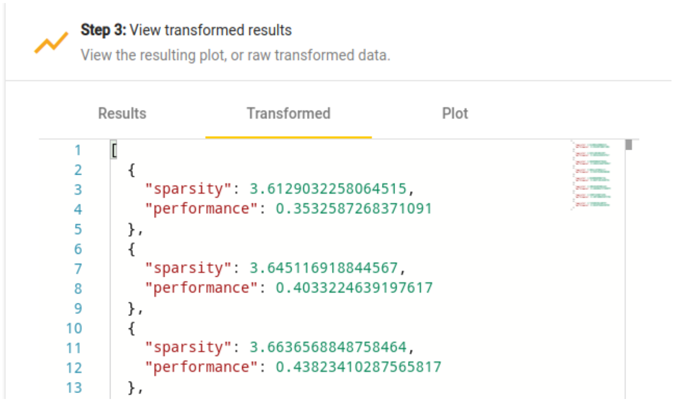
Fig. 5. Raw performance results viewer.

```
content.{
    "sparsity": problem.(nonzeros / rows),
    "performance": 2 * problem.nonzeros /
                        spmv.csr.time
}
```

Fig. 6. JSONata script that computes the performance of the CSR SpMV kernel and the nonzero-per-row average.

shown in Figure 4 has to be available at "`http://example.com/data/list.json`". Afterwards, the "Performance root URL" in GPE is changed to `http://example.com/data`, and the application will retrieve the data from the chosen location. Once the performance results are loaded, they can be viewed in the "Results" tab of the *data and plot viewer* (the blue box on the right-hand side in Figure 2). An example of raw data that is retrieved by GPE is shown in Figure 5. All accessed result files are combined into one single JSON array of objects. Each object consist of properties such as the name and relative path to the result file, as well as its content.

Collecting useful insights from raw performance data is usually difficult, and distinct values need to be combined or aggregated before drawing conclusions. This is enabled by providing a script in the *transformation script editor*, marked


Step 3: View transformed results
View the resulting plot, or raw transformed data.
Results
Transformed
Plot

```
[
  {
    "sparsity": 3.6129032258064515,
    "performance": 0.3532587268371091
  },
  {
    "sparsity": 3.645116918844567,
    "performance": 0.4033224639197617
  },
  {
    "sparsity": 3.6636568848758464,
    "performance": 0.43823410287565817
  },
```

Fig. 7. Transformed data viewer.

```
($transformed := content.{
  "sparsity": problem.(nonzeros / rows),
  "performance": 2*problem.nonzeros /
                 spmv.csr.time
}; {
  "type": "scatter",
  "data" : {
    "datasets": [{
      "label": "CSR",
      "data": $transformed.{
        "x": sparsity,
        "y": performance
      },
      "backgroundColor": "hsl(38,93%,54%)"
    }]
  }
})
```

Fig. 8. JSONata example script that plots the performance of the CSR SpMV kernel in relation to the nonzero-per-row average.

with a green box on the left bottom in Figure 2. For example, the data in Figure 5 shows raw performance data of various sparse matrix-vector multiplication kernels (SpMV) on a set of matrices from the SuiteSparse matrix collection. It may be interesting to analyze how the performance of the CSR-based SpMV kernel depends on the average number of nonzeros per row. Neither of these quantities is available in the raw performance data. However, by following the tree of properties "`content > problem > nonzeros`", "`content > problem > rows`" and "`content > spmv > csr > time`", the total number of nonzeros in a matrix, the number of rows in a matrix, and the runtime of the CSR SpMV kernel can be derived. Since these are the only quantities needed to generate the comparison of interest, the transformation script editor can be used to write a suitable JSONata script[10]; see Figure 6. The script is in real-time applied to the input data, and the result is immediately available in the "Transformed" tab of the *data and plot viewer*, as shown on Figure 7.

The missing step is the visualization of the performance data. For that purpose, the data has to be transformed into a format that is readable for Chart.js, i.e. it has to be a

[10]Full JSONata user guide describing the syntax in detail is available at https://docs.jsonata.org

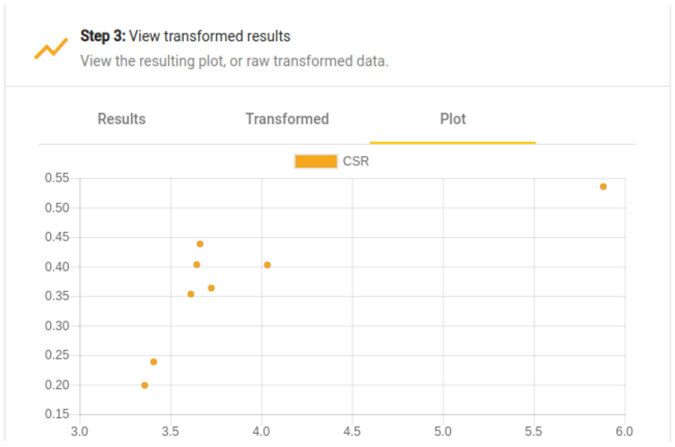


Fig. 9. Plot generated by the example script given in Figure 8.

Chart.js configuration object (as described in the Chart.js documentation[11]). Figure 8 provides a minimal extension of the script to generate a Chart.js configuration object. The visualized data is then available in the "Plot" tab of the *data and plot viewer*; see Figure 9.

For first-time users, or to get a quick glance of the library's performance, we provide a set of predefined JSONata scripts which can be used to obtain some performance visualizations without learning the language. These can be accessed from the "Select an example" dropdown menu of the transformation script editor. By default, the example scripts are retrieved from the GINKGO performance data repository. However, the script location can be modified in the same way like the dataset location.

VI. EXAMPLES

To conclude the presentation of GPE, we present an end-to-end usage example that demonstrates the capabilities of GPE in analyzing performance data. The scenario we consider involves a user of the GINKGO library that is interested in finding out which of the sparse matrix vector product (SpMV) kernels has the best overall performance for a wide range of problems. To that end, we look at the performance of GINKGO's SpMV kernels on the entire Suite Sparse matrix collection [7]. Even though the whole dataset contains results for various architectures, we exclusively focus on the performance of GINKGO's CUDA executor on a V100 GPU. Thus, as a first step, the results are filtered to include only this architecture:

```
$data := content[dataset.(
  system = "V100_SXM2" and executor = "cuda")]
```

In the following visualization examples we particularly focus on how to realize the data transformations needed to extract interesting data. The specific visualization configurations to generate appealing plots (including the labeling of the axes, the color selection, etc.), are well-documented and easy to integrate [10]. The full JSONata scripts used to generate the graphs in this section are available as templates in the example script selector of GPE.

[11] https://www.chartjs.org/docs/latest/getting-started/usage.html

A. Fastest matrix format

In a first example, we identify the "best" SpMV kernel by inspecting the number of problems for which that particular kernel is the fastest. To that end, we first extract the list of available kernels. Then, we split the list of matrices into sublists, where every sublist contains the matrices for which one of the kernels is the fastest. From this information, the numbers can be accumulated and arranged in a Chart.js configuration object. The JSONata script and the resulting plot are given in Figure 10.

B. A more detailed analysis

The results in Figure 10 provide a summery, but no details about the generality of the kernels. Each kernel "wins" for a portion of matrices, but it is impossible to say which kernel to choose for a specific matrix. Since the SpMV kernel performance usually depends on the number of nonzeros in the matrix, we next visualize the performance of the distinct SpMV kernels depending on the nonzero count. From the technical point of view, different SpMV kernels have to be identified in the set, and the relevant data has to be extracted from the dataset as shown on the left side in Figure 11. To distinguish the performance of the distinct SpMV kernels data in the scatter plot, we encode the kernels using different colors. This is realized via a script that defines a helper `$getColor` function which selects a set of color codes that are equally-distant in the color wheel.

Figure 11 reveals more details about the performance of the distinct SpMV kernels. Inside the GPE application, the points representing distinct kernels can be activated and deactivated by clicking on the appropriate label in the legend. We note that this plot contains about 15,000 individual data points ($> 3,000$ test matrices, 5 SpMV kernels), which makes the interactive analysis very resource-demanding. Figure 11 indicates that the COO, the CSR, and the hybrid kernels achieve good performance for a wide range of problems. Omitting the hybrid kernel for a moment, we investigate the performance ratio between the CSR and the COO kernel.

C. Comparison of CSR and COO formats

From comparing the performance of the CSR and the COO kernel in Figure 11, we conclude that the CSR format achieves better peak performance than COO. However, the COO performance seems more consistent as (for large enough matrices), it never drops below 5 GFLOP/s. This suggest that there exist matrices for which the CSR kernel is not suitable. We may assume that load balancing plays a role, and the regularity of matrices having a strong impact on the performance of the CSR kernel. Indeed, the CSR kernel distributes the matrix rows to the distinct threads, which can result in significant load imbalance for irregular matrices. The COO kernel efficiently adapts to irregular sparsity patterns by balancing the nonzeros among the threads [15].

```
$formats  := $data.spmv~>$keys();
$counts   := $formats~>$map(function ($v) {
  $data.optimal[spmv = $v]~>$count()
});
{
  "type": "bar",
  "data": {
    "labels": $formats,
    "datasets": [{ "data": $counts }]
  },
  "options": { "scales": { "yAxes": [{
    "ticks": { "beginAtZero": true }
  }]}}
}
```

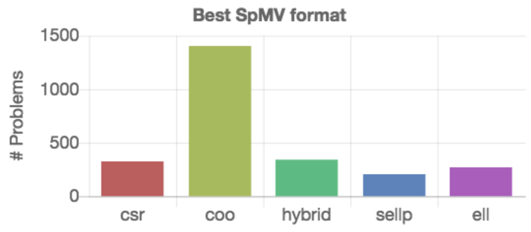


Fig. 10. Left: JSONata script for creating a bar plot visualizing the number of problems for which an SpMV kernel is the fastest. Right: The graph generated by the script (after adding some visualization options).

```
$getColor := function($n, $id) {
  "hsl(" & $floor(360 * $id / $n)
         & ",40%,55%)"
};

$formats := $data.spmv~>$keys();
$plot_data := $formats~>$map(function($v, $i) {{
  "label": $v,
  "data": $data.{
    "x": problem.nonzeros,
    "y": 2 * problem.nonzeros /
         (spmv~>$lookup($v)).time
  },
  "backgroundColor":
    $formats~>$count()~>$getColor($i)
}});

{
  "type": "scatter",
  "data": { "datasets": $plot_data },
  "options": { "scales": { "xAxes": [{
    "type": "logarithmic"
  }]}}
}
```

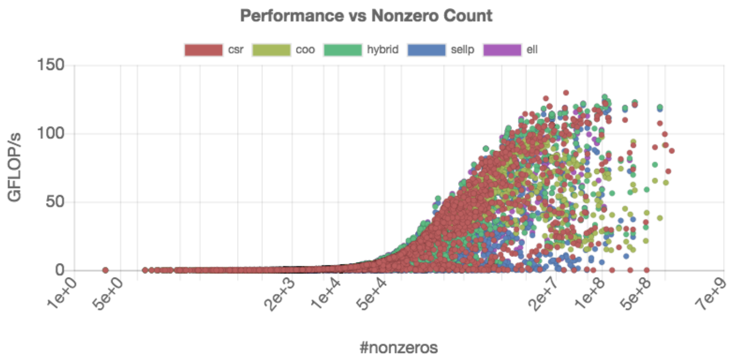


Fig. 11. Left: JSONata script crating a performance vs. nonzeros graph for different SpMV kernels. Right: The graph generated by the script (after adding visualization options).

```
$plot_data := $data[problem.nonzeros > 100000].{
  "x": problem.row_distribution.(
         $sqrt(variance) / median),
  "y": spmv.(csr.time / coo.time)
};
{
  "type": "scatter",
  "data": { "datasets": [{
      "label": "COO is faster",
      "data": $plot_data[y >= 1],
      "backgroundColor": "hsl(0,40%,55%)"
    }, {
      "label": "CSR is faster",
      "data": $plot_data[y < 1],
      "backgroundColor": "hsl(120,40%,55%)"
  }]},
  "options": { "scales": {
    "xAxes": [{ "type": "logarithmic" }],
    "yAxes": [{ "type": "logarithmic" }]
  }}
}
```

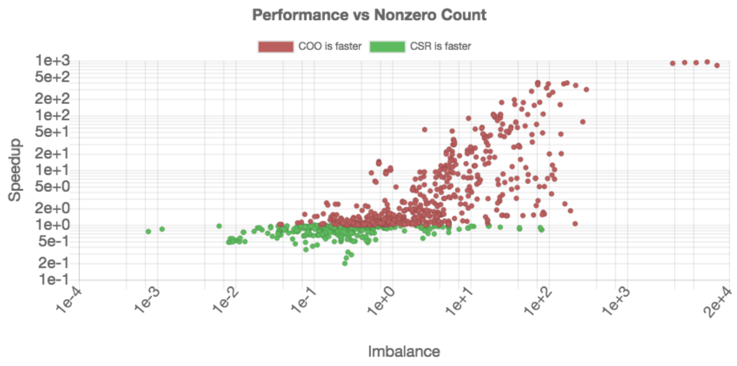


Fig. 12. Left: JSONata script for a visualizing the speedup of the COO kernel over the CSR kernel. Right: The graph generated by the script (after adding visualization options).

To analyze this aspect, we create a scatter plot that relates the speedup of COO over CSR to the "sparsity imbalance of the matrices." We derive this metric as the ratio between the standard deviation and the arithmetic mean of the nonzero-per-row distribution. We expect to see a slowdown (speedup smaller than one) for problems with low irregularity, and a speedup (larger than one) for problems with higher irregularity. The previous analysis in Figure 11 included problems that are too small to generate useful performance data. In response, we restrict the analysis to problems containing at least $100,000$ nonzeros. The script for realizing the performance comparison and the resulting graph indicating the validity of the assumption are given in Figure 12.

D. The performance profile of different formats

Before concluding that the COO format is the most general format in terms of cross-problem performance, Figure 13 visuallizes which strategy renders the best performance in terms of the "*performance profile*" [17]. The performance profile is a visualization ideal for comparing the performance of algorithms on problem sets that are otherwise difficult to illustrate (e.g. they are too large, or there is no reasonable metric to determine how difficult each of the problems is). Each algorithm A from the set $\mathbb{A}$ of all algorithms being compared on a problem set Φ is represented via its *performance function* $f_A = f_{A,\mathbb{A},\Phi}$. The value $f_A(t)$ in point t of the performance function is defined as the percentage of problems $\phi \in \Phi$ where the performance of A is not more than t times worse than the performance of the best algorithm in $\mathbb{A}$. Formally, if $p(A, \phi)$ is the performance of the algorithm $A \in \mathbb{A}$ on the problem $\phi \in \Phi$, the performance function is defined as:

$$f_A(t) := \frac{|\{\phi \in \Phi \mid t \cdot p(A, \phi) \geq \max_{B \in \mathbb{A}} p(B, \phi)\}|}{|\Phi|}. \quad (1)$$

The performance profile is a set $P_{\mathbb{A},\Phi} := \{f_A \mid A \in \mathbb{A}\}$ of all performance functions of $\mathbb{A}$, and is usually visualized by sampling the performance functions at fixed intervals and plotting them as lines in a line plot. See the book of D. Higham and N. Higham [18] for more details about performance profiles. To increase the significance of the performance profile, we consider only test matrices with at least $100,000$ nonzero elements. The script for visualizing the performance profile is available as an example in the GPE web application, but omitted in this paper for brevity. In Figure 13 we identify the COO and the hybrid kernels as the overall winners in generality. The other kernels win fewer cases (have a lower leftmost value $f(t)$ for $t = 1$) and are less general (exhibit a milder slope). Ginkgo's ELL and SELLP formats have very similar performance profiles, with small advantages on the SELLP side.

VII. Using the Framework in Other Projects

The entire workflow is designed to allow for the easy adoption by other software projects. Since the majority of components are open-source tools, the adoption of GPE mostly consists of configuring these components. The setup described here assumes that the project adopting the workflow is hosted in a publicly available git repository (e.g., on GitHub, GitLab or Bitbucket).

First, a public repository used to store the performance database has to be created on a web-accessible server. The only requirement is that the raw files stored in the repository can be accessed over the `http`/`https` protocol, which is true for all repository hosting services mentioned above. Then, a new "CI/CD project for an external repository" has to be set up on GitLab.[12] This project will be used to run the automated CI jobs. Most likely, projects want to set up a custom account for the performance data repository that will be used by the CI system to publish new results. Then, GINKGO's CI configuration file[13] should be copied into the project source repository and customized to fit the project's needs. This includes changing the URLs to connected repositories (e.g., the performance data repository), and the sequence of commands used to build the project, to run the unit tests, to connect to the HPC system, and to run the benchmarks. If some of the steps are not needed, they can either be deleted or commented out. For security reasons, the authentication details should not be stored directly in the publicly available CI configuration file, but as protected variables in the GitLab CI system. This way, they are only available when running the jobs in a protected branch, which can only be modified by trusted developers. For example, The frequency of benchmark runs can be configured on the GitLab CI/CD schedules menu.[14] After this setup is complete, GitLab will automatically mirror the repository, run the configured build and unit tests at every commit, and schedule the benchmarks on the HPC system at regular intervals.

Interactive performance visualization via GPE can be enabled in two ways. The simplest approach is to just use the version of GPE hosted on GINKGO's GitHub pages, and change the data and the plot URLs to appropriate values for the project's performance data repository, as explained in Section V. However, if more customization and improved user experience is needed, the GPE repository on GitHub can be fork and a custom version of the application built for the adopting project. The default database location can be changed in `src/app/default-form-values.ts`. The color scheme of the web application can be modified to match the signature colors of the project by updating the color definitions in `src/ginkgo-theme.scss`, and the logo can be replaced with the project's logo by providing another `src/assets/logo.png` file. Once the customization is completed, the application can be compiled using the script `scripts/build.sh`. To complete this step, `angular-cli`, `node` and `npm` have to be installed on the system[15]. Finally, the application can be hosted on the GitHub page of the fork by invoking the `scr/deploy.sh` script.

[12] https://docs.gitlab.com/ee/ci/ci_cd_for_external_repos
[13] https://github.com/ginkgo-project/ginkgo/blob/develop/.gitlab-ci.yml
[14] https://docs.gitlab.com/ce/user/project/pipelines/schedules.html
[15]Refer to https://angular.io/guide/quickstart for more details

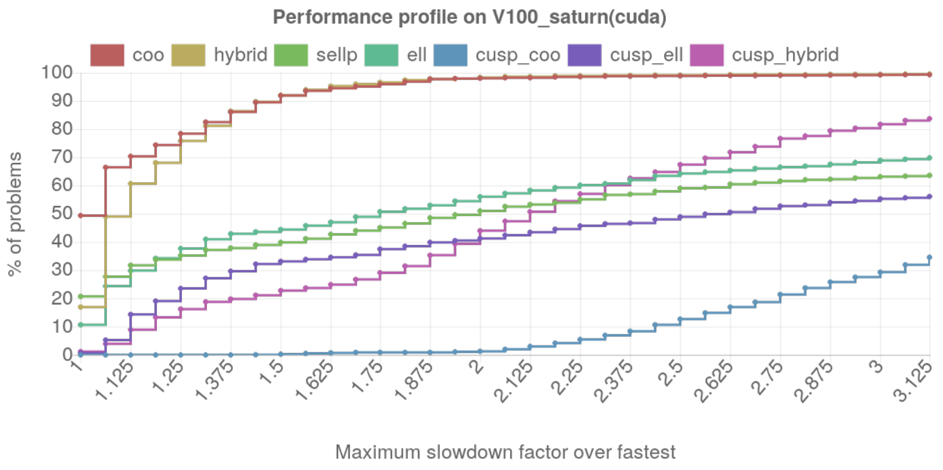


Fig. 13. Performance profile comparing a list of SpMV kernels.

VIII. Summary and Outlook

We have presented a framework for the automatic performance evaluation of the Ginkgo linear operator library. The integrated Ginkgo performance explorer (GPE) allows to retrieve and interactively analyze data from a repository containing performance results collected on HPC platforms. Designing GPE as a web application removes the burden of installing additional software or downloading performance results. The framework is amenable to extension to other software efforts. Consequently, we hope other software libraries will adopt the framework, and we envision the establishment of a global performance database. Such a database will allow the quick and painless performance comparison of distinct libraries and software components.

References

[1] Bitbucket https://bitbucket.org/.
[2] ClangFormat https://clang.llvm.org/docs/ClangFormat.html.
[3] Git https://git-scm.com/.
[4] GitHub https://github.com/.
[5] GitLab https://gitlab.com/.
[6] JSONata — JSON query and transformation language https://docs.jsonata.org/.
[7] Suitesparse matrix collection. https://sparse.tamu.edu, 2018. [Online, Accessed: 2018-08-12].
[8] Angular: One framework. mobile & desktop.,angular.io, accessed in August 2018.
[9] Better Scientific Software (BSSw) https://bssw.io/, accessed in August 2018.
[10] Chart.js: Simple yet flexible javascript charting for designers & developers, chartjs.org, accessed in August 2018.
[11] Google Test https://github.com/google/googletest, accessed in August 2018.
[12] Monaco editor: A browser based code editor, https://microsoft.github.io/monaco-editor, accessed in August 2018.
[13] xSDK: Extreme-scale Scientific Software Development Kit https://xsdk.info/, accessed in August 2018.
[14] Ecma International. *The JSON Data Interchange Syntax*, 2 edition, 12 2017.
[15] Goran Flegar and Hartwig Anzt. Overcoming load imbalance for irregular sparse matrices. In *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, pages 2:1–2:8, New York, NY, USA, 2017. ACM.
[16] M. Fowler and M. Foemmel. Continuous integration, http://www.martinfowler.com/articles/continuousIntegration.html, 2005.
[17] Nicholas Gould and Jennifer Scott. A note on performance profiles for benchmarking software. *ACM Trans. Math. Softw.*, 43(2):15:1–15:5, August 2016.
[18] D. Higham and N. Higham. *Matlab Guide*. Society for Industrial and Applied Mathematics, 2005.
[19] Kitware, Inc. CMake. http://cmake.org, 2012.
[20] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.
[21] Dimitri van Heesch. Doxygen: Source code documentation generator tool, 2008.
[22] Milo Yip. RapidJSON: A fast JSON parser/generator for C++ with both SAX/DOM style API, rapidjson.org, accessed in August 2018.