Efficient Numerical Solvers for the Manifold Learning Framework datafold

Interdisciplinary Project
at the Department of Mathematics of the Technical University of München.

Supervised by Prof. Dr. Hans Joachim Bungartz
Dr. rer. nat. Tobias Neckel
Dr. rer. nat. Felix Dietrich
Department of Computer Science

Submitted by B.Sc. Michael Grad
B.Sc. Thomas Raith

Submittet on Munich, 31st of March 2021

Contents

1 Abstract ... 3
2 Problem Statement ... 4
2.1 Datafold Framework ... 4
2.1.1 Functionality and Use-Cases ... 5
2.1.2 Currently used Framework for Eigenproblem-Solving ... 5
2.2 SciPy Solver ... 6
2.2.1 Used Algorithms ... 6
2.2.2 Benchmark (Baselines) ... 6
3 Solution Space ... 9
3.1 Docker ... 9
3.2 SLEPc ... 10
3.2.1 SLEPc Framework (Use Cases, Background Info) ... 10
3.2.2 Technical Implementation ... 11
3.2.3 Used Algorithms (Benchmarks between Algorithms) ... 13
4 Performance Results and Comparison ... 16
4.1 Comparison between various SLEPc solvers ... 16
4.2 Comparison between SciPy Baseline and chosen SLEPc method ... 18
4.3 Results from HPC Cluster runs ... 19
4.3.1 Weak Scaling Benchmarks ... 20
4.3.2 Strong Scaling Benchmarks ... 21
5 Conclusion ... 24
A Dockerfile ... 27
A.1 Base Dockerfile ... 27
A.2 Benchmark Dockerfile ... 30
B Cluster Batch-Job File ... 31
C Benchmark Data ... 32
C.1 Singlecore Benchmark ... 32
C.2 Multicore Benchmark ... 33

1. Abstract

In this Interdisciplinary-Project, we augment the datafold framework for machine learning with an alternative eigensolver library, with the intend of runtime improvements and the ability to increase the feasible input dimension size respectively. While the current strategy utilizes the SciPy functionality for Eigenvalue and Eigenvector calculation, we incorporate the more specialized SLEPc framework. We therefor set up a Docker environment, that allows for a platform independent, automatic installation of all required components, which include an MPI distribution, PETSc as well as SLEPc. After implementing the SLEPc solver backend for sparse matrices, we assess the runtime improvement by benchmarking both variants on consumer grade laptop and desktop systems. Finally, we take a look at scalability by deploying and benchmarking our implementation on a Linux Cluster.

2. Problem Statement

Modern computer and sensor technology enable us to create more detailed and accurate models of scientific observations or more accurate simulations of physical conditions. Science aims to increase the quantity and dimension of the input dataset further to get even more details and more expressive insights of the problem they are facing. As datasets with billions of input points are not comprehensible for humans it is required to transform the input into something more tangible. This process is often limited by the projects budget and technological limits in terms of computational power and memory.

Datafold[1][LDKB20] tackles this problem by mapping the input dataset into a representation with smaller dimension. Therefore, the framework applies a machine learning approach that depends on the solutions of an eigenvalue problem of different size proportional to the input data size. Today datafold utilizes SciPy to solve the eigenvalue problems what leads to a bottleneck as the solvers that are included with SciPy are less optimized to be executed in high performance cluster environments or are not the best choice for every kind of problem (e.g. sparse input data).

We aim to introduce alternative eigenvalue solvers to datafold that are highly optimized for modern hardware architectures and cluster computing environments. The open source SLEPc[2] framework provides those advanced algorithms in an OpenMPI enabled implementation. As part of this interdisciplinary project we implement an adapter interface for the SLEPc framework that enables us to access its algorithms.

As second output of the IDP we want to proof that more advanced algorithms enable datafold to handle large data sets within a reasonable amount of time and that the execution utilizing more advanced algorithms have proportional runtime bounds to the size of the input dataset. This is going to be archived by executing benchmarks on consumer computer systems and the LRZ CoolMUC3 cluster.

2.1. Datafold Framework

Datafold, as presented in [LDKB20], is an open-source Python package that provides algorithms that map large-scale input datasets (represented as point clouds) to an explicit manifold representation. Additionally, it comes with methods that detect non-linear dynamical systems on manifolds. This process is backed on models that can be based on simple data structures up to complex machine learning algorithms.

The mapping is the output of a processing pipeline that is highly customizable to fit the needs

[1] https://datafold-dev.gitlab.io/datafold
[2] https://slepc.upv.es

of the respective problem which makes the framework universal for many data analytic and evaluation problems. To enhance extendibility datafold provides an API that abstracts defines the dataflow between pipeline and model modules. As data format it utilizes common data structures for scientific contexts (numpy and SciPy) structures.

2.1.1. Functionality and Use-Cases

While the size of datasets and their dimensionality is increasing due to more sophisticated sensors and the according data collection (amongst others), the demand for efficient processing arises consequently. In order to handle especially data with a higher input dimensionality, numerous approaches have been designed to reduce the dimensionality of the input data. While this effect may be achieved by random projection of the data points, this technique usually does not yield satisfying enough results. Hence, a number of linear dimensionalities reduction frameworks were developed, trying to preserve the desired structures of the underlying data. The datafold framework now builds upon this approach, generalizing its idea to support and detect non-linear structures in the data. Datafold therefor uses a diffusion maps approach, allowing for approximations on the data of e.g. the Laplace-Beltrami operator and out-of-sample interpolation.
The desired dimensionality reduction gained by finding an embedded manifold thereby reduces the data complexity e.g. for following machine learning applications. Also, since humans can find it hard to visualize data beyond the third dimension, this approach may also be useful for general visualization of data with more dimensions.

2.1.2. Currently used Framework for Eigenproblem-Solving

We start the project with a working version of datafold, that includes an eigensolver implementation based on the popular SciPy package. The "Python-based ecosystem of open-source software for mathematics, science, and engineering" [3], the SciPy stack consists of packages like NumPy (for n-dimensional arrays, etc.), Pandas (for Data-frames and -analysis), SciPy library (for scientific computing) and many more. Since especially NumPy has become more or less of a standard for handling n-dimensional objects in Python, SciPy is a widespread choice for scientific computing tasks, due to both, its compatibility with the aforementioned NumPy and ease of use. Offering a wide variety of functionality like statistical functions, signal processing, clustering and many more, the (sparse) linear algebra package conveniently comes with a collection of solvers for eigenvalue problems, as they are needed in this datafold project. More so, it allows for some optimization by offering different variations of said solvers, each specialized for various situations. Hence, the datafold version with which we start already differentiates between dense and sparse matrices, as well as Hermitian/non-Hermitian and chooses the best SciPy algorithm for the task respectively.

[3] https://www.scipy.org/index.html

2.2. SciPy Solver

2.2.1. Used Algorithms

Consequently, the used eigenproblem solvers are:

scipy.linalg.eig	For dense, non-Hermitian eigenvalue problems
scipy.linalg.eigh	For dense, Hermitian eigenvalue problems
scipy.sparse.linalg.eig	For sparse, non-Hermitian eigenvalue problems
scipy.sparse.linalg.eigh	For sparse, Hermitian eigenvalue problems

This differentiation takes place in the *datafold\pcfold\eigensolver.py* file which sets up the corresponding SciPy solver and contains the generalized interface function for a blackbox implementation. Since our primary goal is to reduce the calculation time of a given eigenproblem, we should also take a look at the specific implementations each framework offers and, as will become important later, choose the most suitable solver for our task.
According to its documentation both, **scipy.sparse.linalg.eigs**[4] as well as its hermitian counterpart **scipy.sparse.linalg.eigsh**[5], wrap the ARPACK functions *SSEUPD* and *DSEUPD* which use the **Implicitly Restarted Lanczos Method**.
Meanwhile, for the non-sparse cases, **scipy.linalg.eig**[6] relies on **LAPACK**, more specifically on the *geev* routine, which itself takes advantage of an implementation of the *QR-Algorithm*.
scipy.linalg.eigh[7] also calls **LAPACK** functions, however more dynamically, based on drivers, etc..

2.2.2. Benchmark (Baselines)

In order to be able to assess the performance of our work, we firstly take a series of baseline benchmark, running the currently used SciPy solvers. To do so, we generated a collection of benchmark cases so that we can ensure the same input data and parameters for the respective runs. These benchmark cases are derived from one of the tutorial Python-Notebooks[8] that come with datafold, namely the Swiss roll. Since we will later have multiple implementation compete for the best runtime performance, we scaled the aforementioned Swiss roll benchmark cases in multiple directions. Firstly we vary in the size of samples (15.000, 30.000 and 45.000) which should significantly increase the runtime. Secondly, we vary the amount of eigenvectors and -values (3, 9, 18) which we request from the respective solvers to asses the hit on runtime.
In order to make these benchmarks available for everyone without any major setup, we cre-

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigs.html
[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.linalg.eigsh.html
[6] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eig.html
[7] https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html
[8] https://datafold-dev.gitlab.io/datafold/tutorial_03_basic_dmap_scurve.html

ated a separate Docker container, that comes with a readily configured datafold environment as well as our benchmark cases, allowing you to run the benchmark on your own machine. Just checkout our Git repository, run the few command line arguments that are described in the readme and verify the results for yourself, or assess the performance of your machine. Using an adaptable Docker environment allows us to carefully control the runtime environment of our Docker installation and will later grant us the possibility to easily add frameworks and packages for various tasks during our implementation phase. All of the benchmarks (except for cluster related benchmarking) that are depicted were generated inside of Docker containers (see descriptions of the benchmark for details on which container exactly). Naturally this will impose somewhat of an overhead on top of our calculations, but since the baselines were also taken inside a container, these offset should put both results into perspective. Also note that, depending on the host machine, Docker restricts some of the usable resources for the container such as RAM, number of cores, etc.. Therefor, the settings of the corresponding docker container will be depicted with each benchmark result.

Having the aforementioned circumstances in mind, these are the numbers representing our baseline; the runtime of datafold using the SciPy solvers. The following representative runs are conducted on the same system as the ones in chapter 4.

To our surprise we observe, that requesting more Eigenpairs from SciPy does not significantly penalize runtime, but rather seems to produce faster runs. Also, the single core Benchmark, as depicted in 2 reveals faster execution times than the Multicore variant. A possible explanation for that might be, that the SciPy implementation greatly benefits from CPU core clock rather than more cores. Hence this run might have been positively influenced by CPU Boost, enabling faster CPU clock rates for one single core.

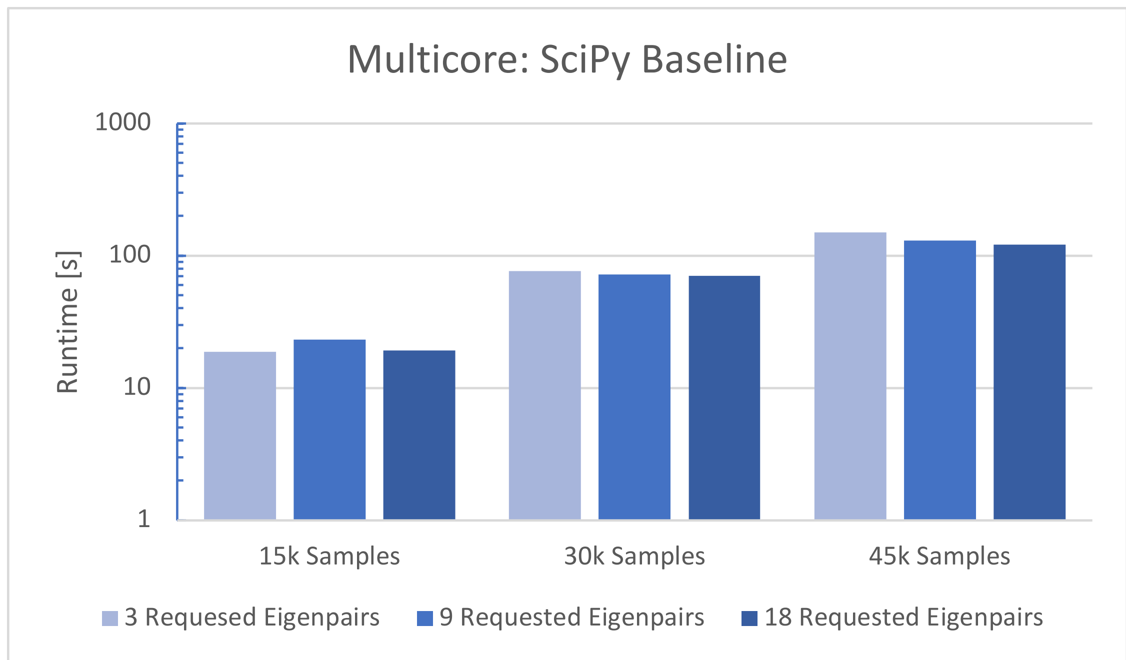

Figure 1 Baseline Benchmarks: Performance of Multicore SciPy Benchmark. Data represents the execution times of 15k, 30k and 45k on a consumer grade system. As expected, the execution time rises with increasing problem size. In general, requesting more eigenpairs does not elevate runtime.

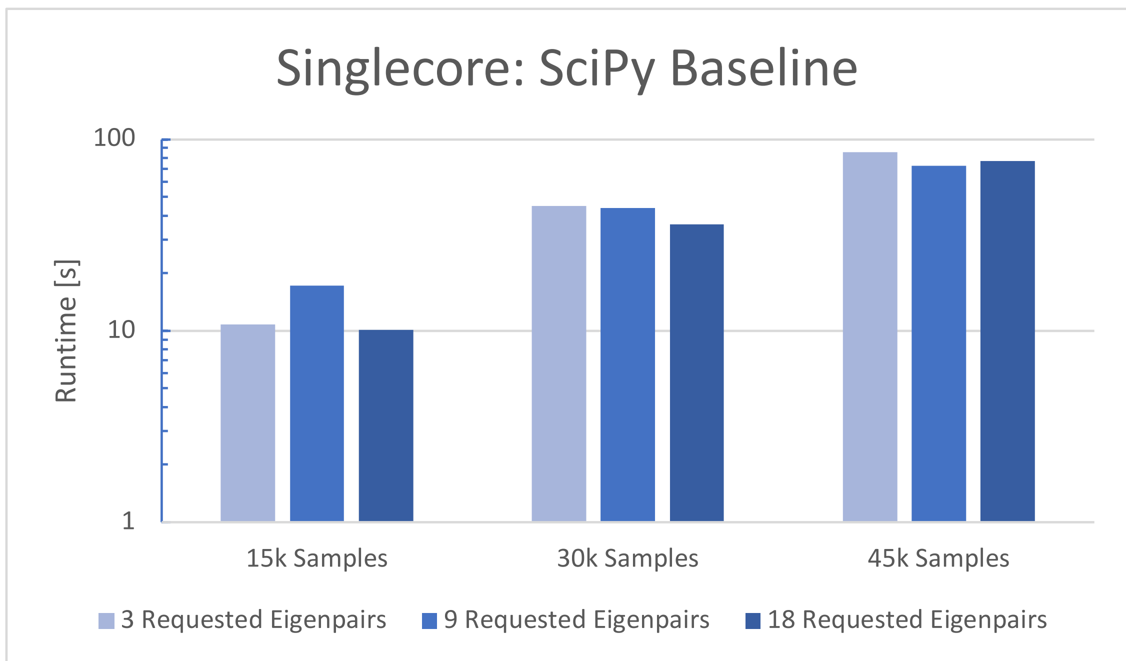

Figure 2 Baseline Benchmarks: Performance of Singlecore SciPy Benchmark. The results of Singlecore execution show similar behavior to the Multicore runs. It is remarkable that the overall runtime of these runs is lower.

3. Solution Space

In this section we cover our implementation and benchmarking efforts. To ensure equal conditions on each development machine we started off with the development of Docker[1] images that bundle dependencies and implementation into a file structure that can be collaboratively accessed by utilizing common VCSs (Version Control Systems).

One milestone is the optimized benchmark Docker image that we used to retrieve benchmark results on single socket systems (e.g. workstations). Another milestone is the OpenMPI[2] powered implementation for multi-node clusters that does not utilize any abstraction layer between system environment and application execution environment.

3.1. Docker

Docker is an open source framework that allows developers and publishers to deliver software in packages called containers. Therefore, docker uses OS-level virtualization techniques that allow resource efficient execution of (multiple) containers as they share the services of the host operating system kernel. Containers bundle the software, its dependencies (e.g. external libraries or software components) and configuration files and isolate them from each other. In addition, communication between containers and the host system is only possible over well-defined channels.

Docker containers are the final outcome of the docker deployment process that starts with a configuration file - the so called Dockerfile[3] - that is built into a Docker image. Docker images can be shared via hubs (e.g. Docker Hub[4]) and executed on any system with a Docker daemon installation. If an image gets executed by the docker daemon, the running instance is called Docker container.

We decided to utilize Docker during the implementation process of our project as it reduces organizational overhead for dependency management to a minimum and by that it speeds up the implementation process. As we used git[5] repositories for collaborative development we added a Dockerfile to our code base that contains all dependencies and required installation steps to execute our implementation.

This enables us to start right after repository checkout with development work. After a checkout a developer rebuilt the locally saved Docker image based on the Dockerfile that was checked out and the implementation was ready for execution on the development machine

[1] https://www.docker.com
[2] https://www.open-mpi.org
[3] see Appendix Section A for our Dockerfile
[4] https://hub.docker.com
[5] https://git-scm.com

independent of the installed operating system what was important as we used to have a mix of Apple MacOS, Linux (Ubuntu) and Microsoft Windows.

In addition to our development image we developed a benchmark image to ease up benchmark runs on single-socket workstations. The image helped us to retrieve multiple benchmark results on different host systems. As the performance impact of Docker is neglectful on private desktop systems and notebooks we decided to run small benchmarks by utilizing this image (all benchmark results captured by using the Docker image are marked throughout this report).

3.2. SLEPc

As a first task, we decided to concentrate on the runtime optimization of sparse matrices, as our docker preparations would more easily allow us to get started using SLEPc than with the framework we planned to use for dense problems.

3.2.1. SLEPc Framework (Use Cases, Background Info)

The SLEPc[6] framework (shortened from: "Scalable Library for Eigenvalue Problem Computations") is a free software library under BSD license developed by the Universitat Politecnica de Valencia in Spain. It is specifically designed for solving large, sparse eigenvalue problems and optimized for parallel execution. SLEPc supports real and complex arithmetic from single up to quadruple precision.Therefor, the library builds upon the PETSc framework (short for: Portable, Extensible Toolkit for Scientific Computation). PETSc herein provides a collection of data structures, routines and for various use cases, such as linear system solvers. In its focus on large scale parallel use, it implements the MPI standard for communication, as well as CUDA or OpenGL for GPU support. Hereby SLEPc inherits said data structures and means of parallelization from PETSc, however adds various algorithms and solvers specifically for sparse eigenvalue problems. It is capable of solving linear eigenvalue problems in both, standard and generalized form, both real and complex as well as nonlinear eigenvalue problems in polynomial or general form. Beside eigenvalue problems the frameworks offers support for problems like SVD (Singular Value Decomposition) or the PEP (Polynomial Eigenvalue Problem).

SLEPc is heavily focused on sparse problems. Therefore, it includes various algorithms and solvers in its components (e.g. eigensolvers like Krylovschur and the CISS (Contour Integral Spectrum Slicing) method and spectral transformations such as shift-and-invert).

[6] https://slepc.upv.es

3.2.2. Technical Implementation

A main goal of our project is the integration and evaluation of SLEPc into the datafold framework as an alternative solver backend. We expect to observe faster computation times as SLEPc is capable of more optimized algorithms for multi dimensional problems.

Before we started implementing the SLEPc solver into datafold, the eigensolver part of the Python code had already been handled as a "black box" component, meaning, that the previously used SciPy solver was embedded in a strategy pattern that allowed the specification of a backend which then takes over the task to find the requested eigenvalues and -vectors.

Technically our implementation introduces a new handler function that maps datafold's interfaces to the interface of PETSc. The solving process itself is handled by the SLEPc framework. As SLEPc is optimized for parallel computing we designed the handler to use MPI for managing multiple parallel processes.

We started by encapsulating our SLEPc implementation in its own file in order not to clutter the existing eigensolver.py file. This way, we can simply import our added functionality and store our implementation elsewhere, improving code readability and reusability. By re-routing all function calls to the eigensolver from SciPy to our implementation, we were able to test and troubleshoot our code while not changing much of the existing code-base.
Our goal was to add fully parallel SLEPc solving functionality to datafold, allowing for efficient scaling, having in mind possible cluster deployments of the framework in the future. Firstly, we started by setting up a single-core variant of a SLEPc solver, as the code would be mostly reusable and it gave the chance to gradually gain experience with SLEPc, PETSc without having to worry about errors due to parallelism.

Having implemented a single core variant, we then further added multicore MPI support to the solver, allowing us to specify the number of cores involved in the solving as well as the added functionality of MPI's ability to expand to distributed systems, such as clusters, etc.. This was also the reason, why we decided to launch said solver via a subprocess, as it allows us for example to easily modify MPI's command line parameters. Different to the already working SciPy based solver, SLEPc requires binary, file based data input. As we support parallel processing it is required to split the input problem into partial problems and input files for each process. Our implementation generates multiple intermediate results (1 output file per process). After all calculation processes have settled, the handler combines all generated intermediate results into the final result. All these steps add additional processing time to the overall runtime of the solver as the segmentation and consolidation are serial processes.

As *datafold* is a Python[7] package we use the Python bindings *petsc4py*[8] and *slepc4py*[9]

[7] https://www.python.org
[8] https://www.mcs.anl.gov/petsc/petsc4py-current/docs
[9] https://slepc.upv.es/slepc4py-main/docs/slepc4py.pdf

provided by the development teams of the PETSc and SLEPc framework to interact with the C (PETSc) and C++/Fortran (SLEPc) based frameworks.[Dal] As we use OpenMPI to parallelize the solving process, we also require access to the MPI child process environment. Therefore we make use of the *mpi4py*[10] package. This library enables us to access information like process rank and communication channels with the MPI host process.

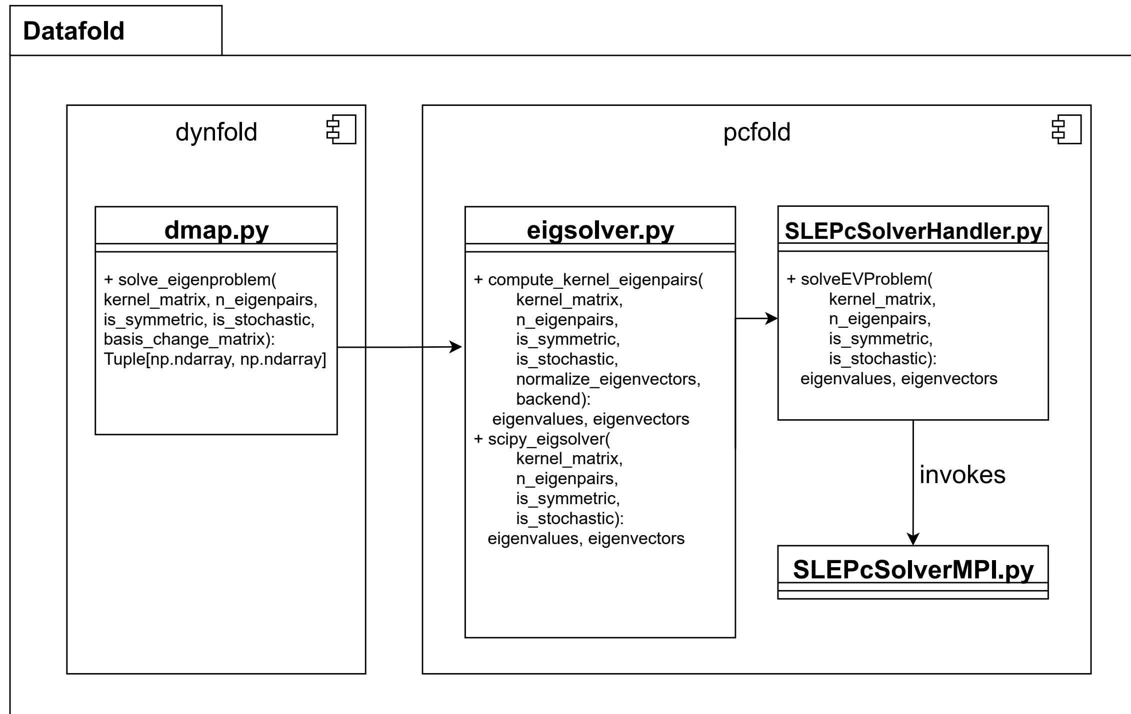

Figure 3 An excerpt from the internal structure of datafold. Most parts are omitted in this overview as it only depicts the functions relevant for eigenproblem solving and their files.

Figure 3 demonstrates how our SLEPc integration is fitted into the datafold project as a "black box" usage, as SLEPcSolverHandler.py implements a general eigensolver that returns the according eigenvalues and vectors, irrespective of any datafold specificities.

The sequence diagram in Figure 4 shows which functions are invoked when and gives a rough idea about the roles of each part in the call-stack.

To ensure an equal environment on all development machines during the development phase we started off with creating a basic Docker image that encapsulates a Python3 installation with the required environment (e.g. petsc4py and slepc4py packages), native dependencies of SLEPc and PETSc and a configured OpenMPI installation. As we modify the datafold framework we decided to mount all datafold related components from a directory into the container. This ensures equal execution conditions while allowing easy access to the files framework that needs to be modified.

Our image is based on plain Ubuntu[11]. Based on environment variables in the Dockerfile we

[10] https://bitbucket.org/mpi4py/mpi4py
[11] https://ubuntu.com

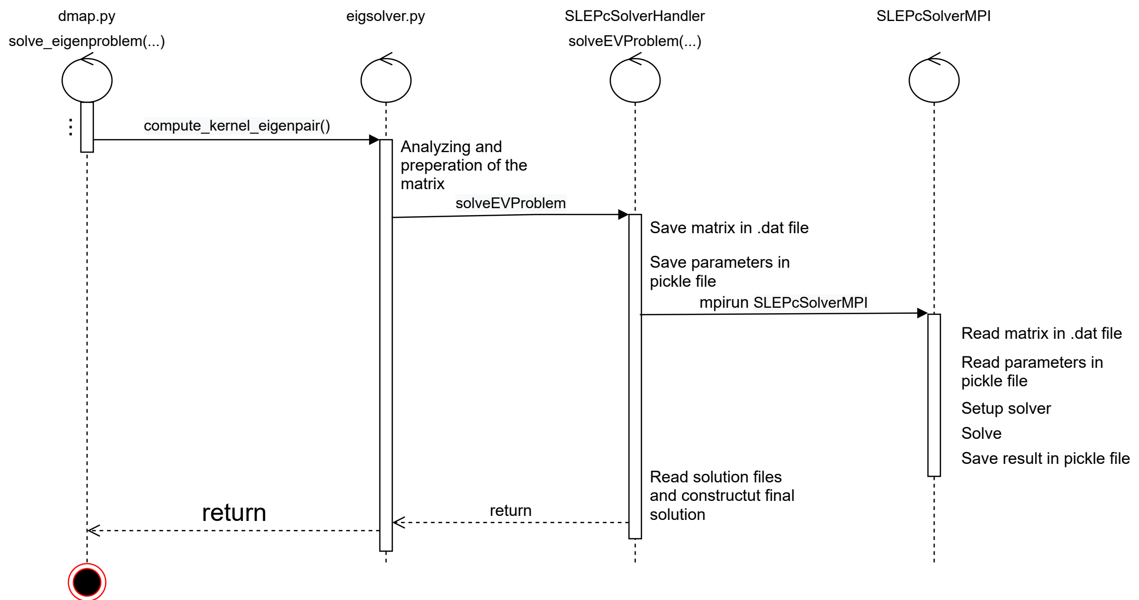

Figure 4 This sequence diagram explains further how the call-stack looks like for eigenvalue solving while some notes explain what is done in certain functions.

download, build and install different versions of OpenMPI, SLEPc and PETSc directly from the projects repository. This enables us to quickly adapt to newer versions and avoids version lock-ins.

In addition to our base image we created a benchmark image. This image is based on our base image but does not have the datafold mounting directory dependency as we include a fully running and modified version with our new handler. This image is intended to be used for running benchmarks on single-socket workstations and notebooks. Since these computers are regularly in day-to-day use and are therefore not optimized for efficient calculations (e.g. unnecessary background processes, anti-virus software etc.), we accept the minimal impact introduced by the Docker virtualization layer.

3.2.3. Used Algorithms (Benchmarks between Algorithms)

With the integration of SLEPc we made a new set of linear eigenvalue solver algorithms available to datafold. The SLEPc framework includes 7 linear eigenvalue solver variants that are more or less optimized in terms of computing time and memory requirements for our use case. To evaluate the performance of these solvers we decided to take an experimental approach and to run benchmarks on problems of increasing size.

Due to IDP time constraints, we agreed on scoping the project to mainly 3 solvers that we are going to evaluate in the following sections.

Arnoldi Method

The Arnoldi method is an iterative approach to approximate eigenvalues and their eigenvectors of general matrices. This is archived by constructing an orthonormal basis of the Krylov

matrix of the matrix $A \in C^{n \times n}$ and an arbitrary initial vector $q \in C^n$.[Ste02][CR16b]

$$K_m = span\{q, Aq, A^2q, ..., A^{m-1}q\}$$

In general this matrix is not orthogonal, but by applying the Gram–Schmidt orthogonalization method it is possible to extract the orthogonal basis. The resulting vectors are an orthogonal basis of the Krylov subspace $\mathcal{K}_n$ and span good approximations of the eigenvectors that belong to the n largest eigenvalues.[Ste02]

The algorithm is defined by the pseudo code listed in algorithm 1 (with $A \in C^{n \times n}$ and an arbitrary vector $q_1 \in C^n$ with $\|q_1\| = 1$:

Algorithm 1 Arnoldi Method [Huh]

1: Start with $q_1 \in C^n$ with $\|q_1\| = 1$
2: **for** $k = 2, 3, \ldots$ **do**
3: $\quad q_k \leftarrow Aq_{k-1}$
4: $\quad$ **for** $j = 1, 2, \ldots, k-1$ **do**
5: $\quad\quad k_{j,k-1} \leftarrow q_j^* q_k$
6: $\quad\quad q_k \leftarrow q_k - h_{j,k-1} q_j$
7: $\quad$ **end for**
8: $\quad h_{k,k-1} \leftarrow \|q_k\|$
9: $\quad g_k \leftarrow \frac{q_k}{h_{k,k-1}}$
10: **end for**

Lanczos Method

The Lanczos method is an iterative algorithm that calculates eigenvalues and their corresponding eigenvectors of matrices beside its capability to approximate solutions of linear equation systems. The method is based on projections of Krylov subspaces. In general the algorithm is based on 2 Krylov subspaces $\mathcal{K} = \mathcal{K}(A, v)$ and $\hat{\mathcal{K}} = \mathcal{K}(A^H, w)$ with $v, w \in C^n$ and $w^H v = 1$ (v and w are biorthogonal to one another). The algorithm biorthogonalizes the the bases of the Krylov spaces against each other by applying a two-sided variant of the Gram-Schmidt method.[CR12][Kre05]

Krylov-Schur Method

The Krylov-Schur method is a more advanced evolution of the Lanczos method and only targets large eigenvalue problems. It generalizes the the Arnoldi decomposition.[Kre05][HRTV07][CR16a]

Benchmarks

We started with running benchmarks on problems consisting of point clouds with 5 dimensions and 45.000 points. The reason for choosing relatively small problems with only 45.000 points was that we executed those benchmarks on regular consumer workstations and notebooks.

The next section presents the results of our benchmarks and shows which of the methods available performs best for the use with the datafold framework.

4. Performance Results and Comparison

This section covers our benchmarking process and the results we retrieved. We decided to go with a staged benchmark approach. The first step represents benchmarks on customer computer systems for finding a baseline for runtimes of algorithms and their behavior in multicore environments. These insights helped us during the job planning phase for our second stage: Benchmarks on the LRZ CoolMUC-3 cluster. Each stage consists of individual substages as the respective environment allows us to apply different benchmarking strategies.

During the first stage we utilized our docker image to execute multiple runs on consumer systems. We spitted this step into 2 substages where the first stage represents multi core benchmarks and the second stage single core benchmarks for reference if the algorithms benefit of parallel processing. The second stage covers our benchmarks retrieved on the LRZ[1] (Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities) CoolMUC-3[2] cluster. To create our benchmark job files, we rely on the results from stage 1 to approximate planned runtimes. Further we decided to centralize our scope to the algorithm that proofs as most performant in terms of runtime and multi core scaling behavior during stage 1. Therefore, all cluster environment benchmarks were retrieved by utilizing the Krylovschur algorithm.

In the following subsections we discuss our results and evaluate the scalability of various algorithms in modern computer environments. The system specifications of our benchmark systems are:

- **Consumer Computer**
  - Intel Core i7-8565U (4 cores, 8 threads)
  - 16GB RAM (DDR4)
- **CoolMUC-3 (data per node)**
  - 3,1 TFlop/s
  - 64 cores (4 threads per core)
  - 96GB RAM (DDR4)

4.1. Comparison between various SLEPc solvers

This step was accomplished in in single core mode and multi core mode on a consumer computer system. We evaluated 3 algorithm implementations provided by SLEPc:

[1] https://www.lrz.de
[2] https://doku.lrz.de/display/PUBLIC/CoolMUC-3

- Arnoldi Method
- Jacobi–Davidson Algorithm (JD)
- Krylovschur Algorithm

To get insights on how the algorithms scale, we selected 2 input parameters for variation:

- Targeted number of eigenvalues (3, 9, 18 eigenvalues)
- Input dataset size (15k, 30k, 45k samples)

The benchmarks were taken by using out benchmark docker image and the Python profiler cProfile.

We started with multi core benchmarking. After the benchmark groups 3 and 9 eigenvalues we could already assume that Krylovschur algorithm performs better that the other algorithms in terms of runtime and scaling behavior. The third run with 18 targeted eigenvalues confirmed our assumptions.

Figure 5 shows the average runtime results of our benchmarks. It aggregates the runtime of all runs per algorithm and shows the average scaling factor (AVSF) from increasing the problem size. The AVSF represents the algebraic average factor by that runtime increases from one problem size to the next problem size over all runs of the benchmark protocol: for n runs with $runtimes = \{t_1, t_2, \ldots, t_n\}$:

$$AVSF = \frac{1}{n-1} \sum_{i=2}^{n} \frac{t_i - t_{i-1}}{2}$$

Figures 14 - 16 show all runs in detail and compares their respective runtimes. The benchmark data gives us the following insights:

Algorithm	**Average Runtime [s]**	**Average Scaling Factor**	**Scaling Factor Trend**
Krylovschur	17,5056	2,0365	equal
JD	79,3092	1,9040	up
SciPy	75,6541	2,7205	up
Arnoldi	720,1428	4,0885	equal

Table 1 Average Performance of Multicore SLEPc Benchmarks with Scaling Factor Trend. The Scaling Factor Trend represents the change of $\frac{t_i - t_{i-1}}{2}$ from the AVSF equation for rising i. An upward trend represents $\frac{t_{i-1} - t_{i-2}}{2} \ll \frac{t_i - t_{i-1}}{2} \ll \ldots$. An equal trend represents $\frac{t_{i-1} - t_{i-2}}{2} \approx \frac{t_i - t_{i-1}}{2} \approx \ldots$

The best overall runtime was observed by using the Krylovschur algorithm whilst the smallest scaling factor was shown by the JD method. As the Arnodi method performed the worst and

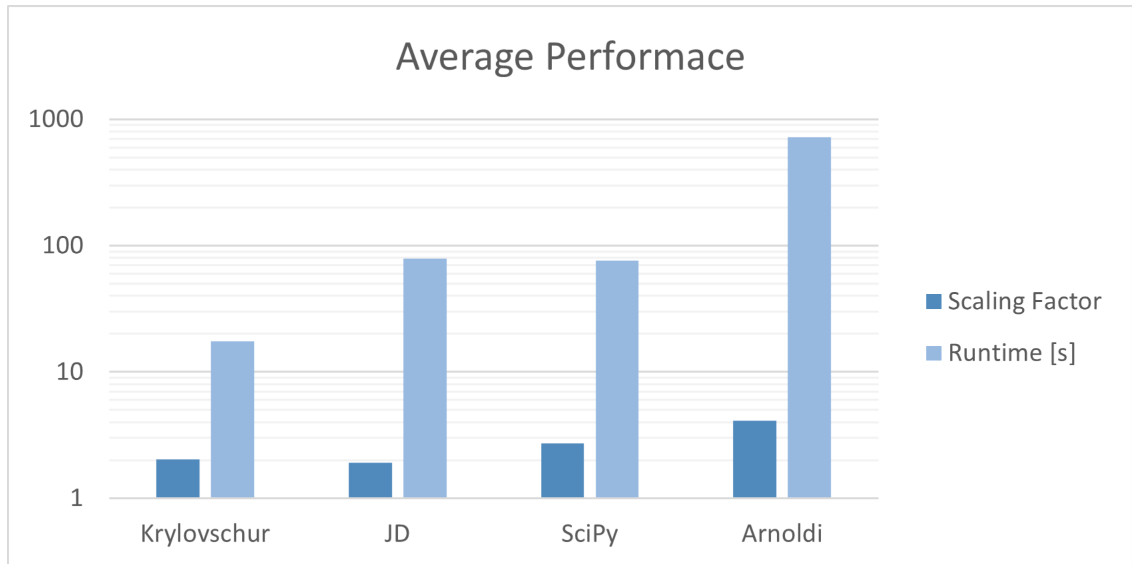

Figure 5 Average Performance of Multicore Benchmarks. Conducted runs show that the Krylovschur method outperforms all other methods in terms of runtime. It is remarkable that the scaling factor of Krylovschur is slightly higher than the factor of JD.

the scaling factor is more than double the factors of Krylovschur and JD we decided to drop it for stage 2. We had to decide between the 2 remaining algorithms provided by SLEPc for the one that is our candidate for stage 2. Krylovschur shows the best runtime and a scaling factor like the JD method. So, we decided to execute upcoming benchmarks with the Krylovschur algorithm as it tends to be the most performant algorithm that can benefit the most on multicore environments.

4.2. Comparison between SciPy Baseline and chosen SLEPc method

This section covers the comparison between the currently implemented default SciPy based solver of datafold with Krylovschur algorithm provided by SLEPc. For direct comparison we started off with single core runs on our consumer computer system:

Algorithm	**Average Runtime [s]**	**Average Scaling Factor**
Krylovschur	24,4684	2,6507
SciPy	44,2520	2,6648

Table 2 Average Performance of Singlecore Benchmarks (Comparison Baseline results with SLEPc results). Data proofs our assumption, that the Krylovschur method provided by SLEPc finishes faster than the SciPy implementation.

Direct comparison shows that the Krylovschur algorithm outperforms the SciPy solver in terms of runtime. In singlecore environments both methods scale with a similar factor around 2,6 which allows us to follow that the Krylovschur methods can take advantage of multiprocessing as it decreases the scaling factor whereas the SciPy methods scaling factor increases

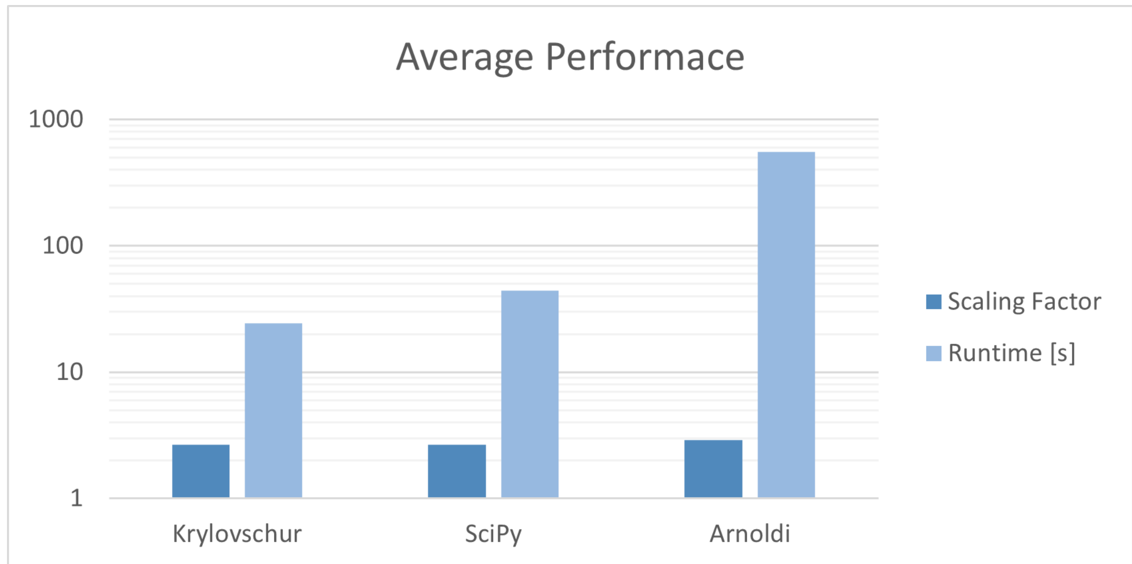

Figure 6 Comparison of the average Performance of Singlecore Benchmarks conducted with the SciPy implementation and SLEPc (Krylovschur algorithm and Arnoldi method). It shows graphically that Krylovschur performs best.

in multicore environments:

Algorithm	Singlecore		Multicore	
	Average Runtime [s]	Average Scaling Factor	Average Runtime [s]	Average Scaling Factor
Krylovschur	24,4684	2,6507	17,5056	2,0365
SciPy	44,2520	2,6648	75,6541	2,7205

Table 3 Comparison SciPy solver with SLEPc in Singlecore and Multicore environments.

These results show that the solving process of datafold ca be optimized by using the Krylovschur solver provided by SLEPc. We expect that our solver implementation based on SLEPc extends the possibilities of datafold and enables more efficient processing of large datasets.

4.3. Results from HPC Cluster runs

For our benchmarks on the LRZ CoolMUC-3 cluster we dropped our docker approach, as we do not have the need of simulating an MPI environment and docker does not support multi node computing without additional tooling. Since the cluster offers most of the base dependencies (Python3, pip, anaconda) for our project as modules and a fully configured OpenMPI installation, our setup efforts are reduced to loading the respective modules and creating an anaconda python environment with the required python packages.

Additionally, we replaced cProfiler with time variables to further reduce unnecessary overhead

and to get more and better insights into how long which part of the code needs for execution. In the final benchmark version of the code we are able to measure every function call of the datafold pipeline. That allows us to break down overall runtime into data generation, data preparation, eigenvalue solving process and eigenvalue selection process or even more detailed.

On the cluster we changed our benchmark approach to standardized procedures for better result comparison: weak scaling and strong scaling. All runs were conducted with 7 dimensions with a dataset size between 12.500 and 3.200.000 samples on 1 to 265 cores.

4.3.1. Weak Scaling Benchmarks

Weak scaling denotes a benchmarking protocol that increases compute power proportional with the problem size. For our benchmarks we started with 1 core/12.500 samples and doubled it each run until our last one with 256 cores/3.200.000 samples. We performed this protocol for both the Gaussian kernel and the CKNN kernel to investigate if these kernel types show different scaling behavior. Table 4 shows an overview of the results (find complete benchmark results in our git repository[3]).

Kernel			**Gaussian**		**CKNN**	
Cores	**Samples [x10³]**	**Dim.**	**Overall Runtime**	**Runtime Eigen-solver**	**Overall Runtime**	**Runtime Eigen-solver**
1	**12,5**	**7**	00:00:49	00:00:17	00:00:50	00:00:16
2	**25**	**7**	00:01:03	00:00:21	00:01:07	00:00:16
4	**50**	**7**	00:02:05	00:00:32	00:01:44	00:00:16
8	**100**	**7**	00:03:42	00:00:50	00:03:30	00:00:30
16	**200**	**7**	00:08:25	00:01:23	00:07:43	00:00:43
32	**400**	**7**	00:16:48	00:02:09	00:17:34	00:01:16
64	**800**	**7**	00:38:33	00:04:21	00:31:58	00:02:02
128	**1.600**	**7**	01:20:45	00:10:01	01:14:59	00:04:10
256	**3.200**	**7**	03:24:46	00:30:11	02:40:36	00:09:08
Average Scaling Factor			2,0286	1,8527	1,9599	1,6054

Table 4 Weak Scaling Results

[3] https://gitlab.lrz.de/thomasraith/idp-datafold

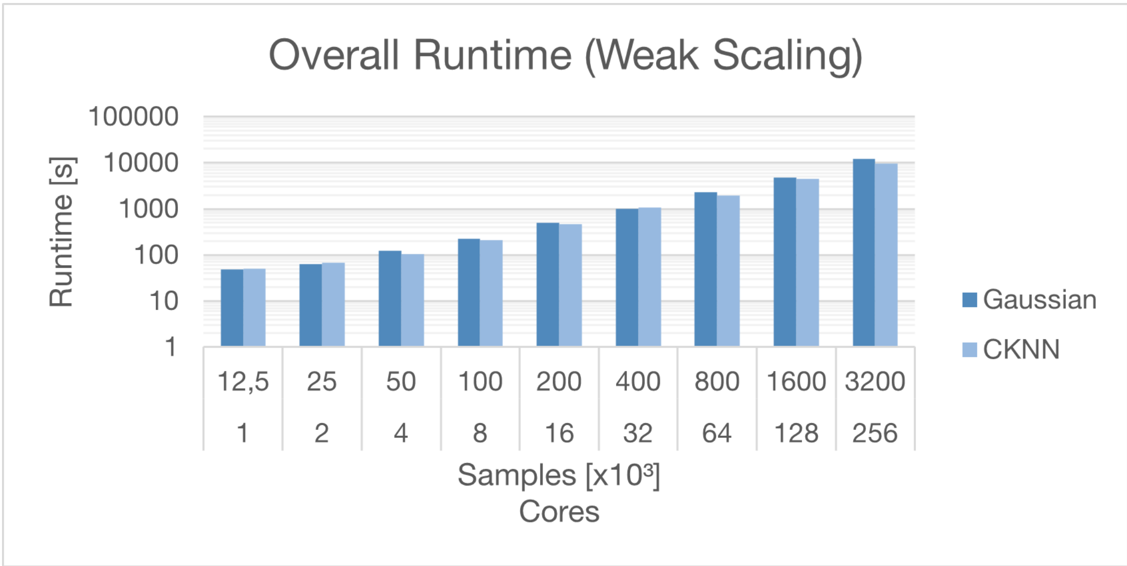

Figure 7 Benchmark Weak Scaling Overall Runtime Results. It shows that runtime increases proportional to the problem size.

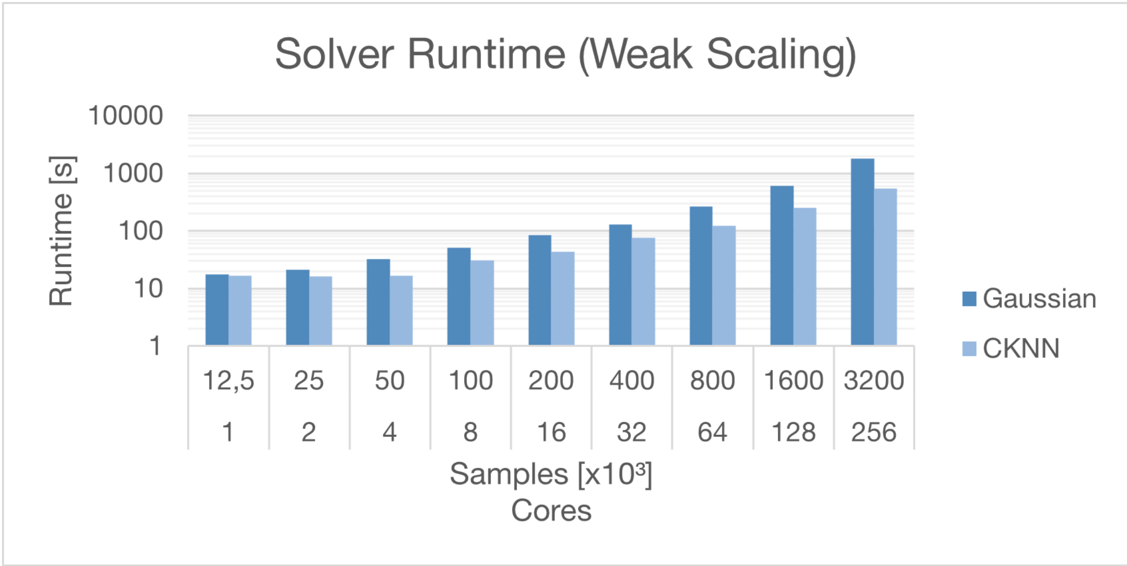

Figure 8 Benchmark Weak Scaling Solver Runtime Results. It shows that runtime increases proportional to the problem size and that the use of the CKNN kernel reduces runtimes in comparison to the Gaussian kernel.

The benchmarks show that the CKNN kernel mode performs better than the Gaussian kernel mode by comparing the actual solving process with an approx. 15% lower scaling factor. Comparing the runtime of the complete datafold pipeline the performance equalizes. Both modes show scaling factors around 2, with CKNN mode still has a scaling factor approx. 3,5% lower than Gaussian mode.

4.3.2. Strong Scaling Benchmarks

Strong scaling denotes a benchmarking protocol that increases compute power that processes a problem of fixed size. We selected our problem size to be 700.000 samples/7 dimension and deducted runs with 1 core to 256 cores. 700.000 samples/7-dimension is a problem of relatively small size compared to the 3.500.000 samples/7-dimension problem

used during our weak scaling runs. This was decided for purpose as we wanted to detect the point where the problem starts being over scaled on the LRZ cluster. This point represents the border where further scaling increases the runtime instead of decreasing it because of the additional overhead added though scaling. Table 5 shows an overview of the results (find complete benchmark results in our git repository[4]).

Kernel			**Gaussian**		**CKNN**	
Cores	**Samples [x10³]**	**Dim.**	**Overall Runtime**	**Runtime Eigen-solver**	**Overall Runtime**	**Runtime Eigen-solver**
1	**200**	**7**	00:17:41	00:11:35	00:09:50	00:03:22
2	**200**	**7**	00:11:58	00:05:02	00:08:21	00:01:47
4	**200**	**7**	00:10:46	00:03:20	00:08:49	00:01:11
8	**200**	**7**	00:08:30	00:02:04	00:07:33	00:00:47
16	**200**	**7**	00:08:56	00:01:30	00:06:56	00:00:39
32	**200**	**7**	00:07:48	00:01:04	00:07:18	00:00:35
POINT OF OVERSCALING						
64	**200**	**7**	00:08:00	00:01:06	00:08:06	00:00:40
128	**200**	**7**	00:08:15	00:01:18	00:07:37	00:00:51
256	**200**	**7**	00:11:24	00:02:21	00:09:58	00:01:22
Average Scaling Factor			0,9659	0,897	1,0112	0,9498

Table 5 Strong Scaling Benchmark Results. At 32 cores the last stage is reached that reduces runtime (point of overscaling). All stages above 32 cores add administrative overhead to the execution so that the runtime extends.

As expected, the benchmarks show the phenomenon of over scaling for both kernel modes. Until reaching 32 cores the runtime decreases for each run. From 64 cores and up the runtime increases for each run with a rising factor. Figure 10 shows that the solving process is more affected than the overall runtime. This is due to the increased overhead caused by the orchestration of MPI processes. Additionally, the file based data interface between datafold/PETSc to SLEPc is adding additional overhead. The number of file system read/write operations rise proportional with the number of MPI solving processes as each process writes its own output file. Those files must be combined afterwards which is a sequential process.

[4] https://gitlab.lrz.de/thomasraith/idp-datafold

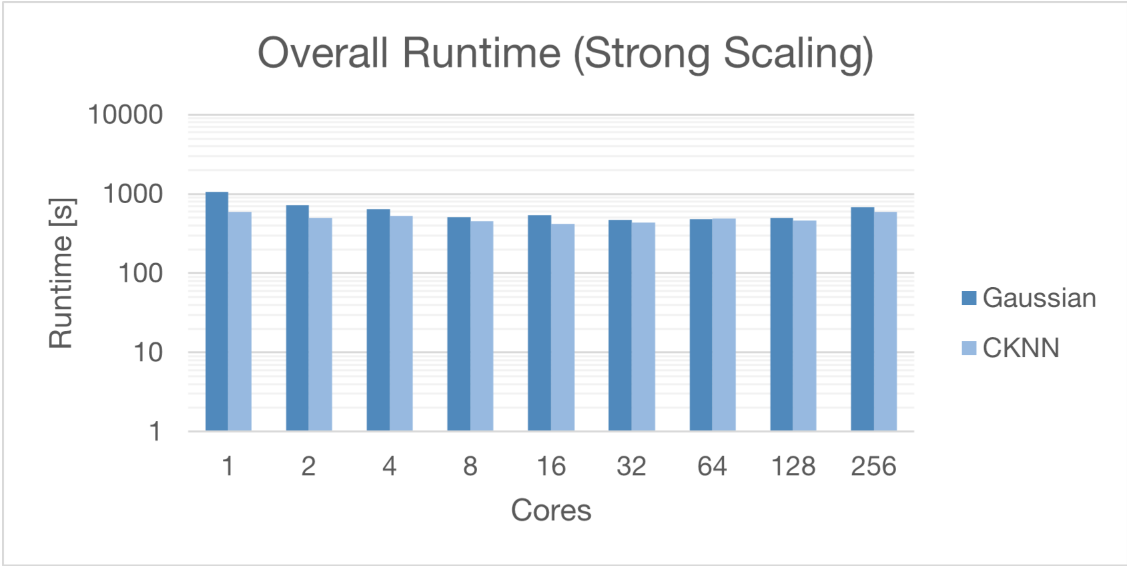

Figure 9 Benchmark Strong Scaling Overall Runtime Results. This chart visualizes that utilizing the CKNN kernel delivers better runtime results compared to the Gaussian kernel.

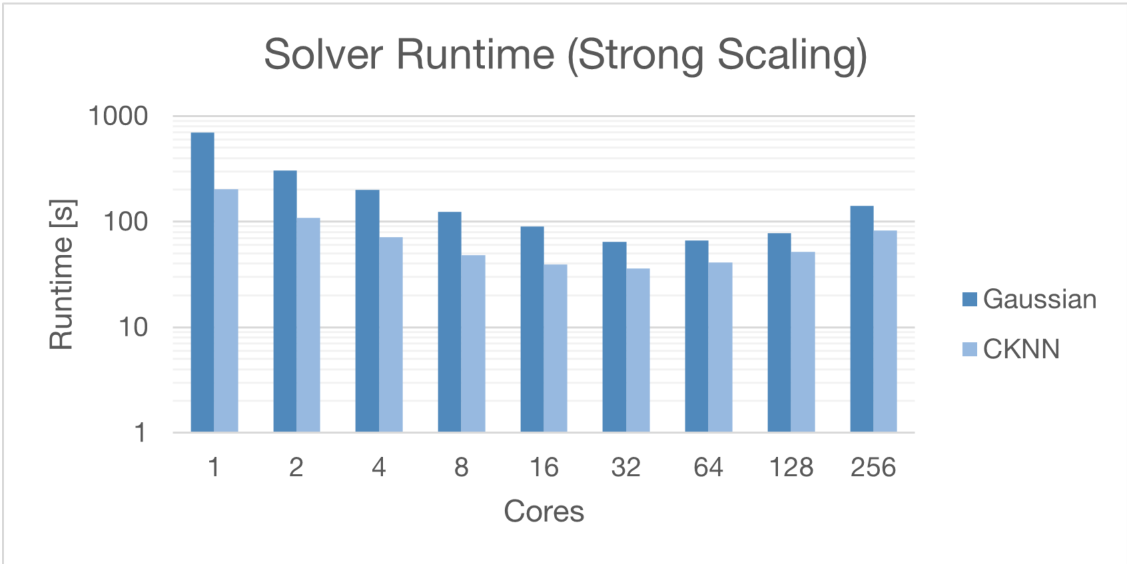

Figure 10 Benchmark Strong Scaling Solver Runtime Results. The point of overscaling between 32 cores and 64 cores is clearly visible in the solver runtimes as they represent the part of the highest parallelism.

5. Conclusion

The outcome of the IDP presents the superiority of SLEPc over the solving methods currently available in datafold. In terms of increasing input data sizes SLEPc's enhanced scalability properties allow faster computation processes in multicore environments (12+ cores). For low-sized problems on consumer grade multicore systems the solving performance of SciPy can compete with the SLEPc algorithms in terms of runtime and memory consumption. Considering the setup process of the SLEPc framework, time savings might get eaten up increased administrative overhead of environment preparation and configuration for SLEPc.

Our project showed an approach to ease up SLEPc provisioning on workstation environments by utilizing docker and its container technology. While our benchmark image was optimized for harvesting runtime insights, it is possible to create a size and interface optimized image distributed via a docker registry (e.g. Docker Hub) that reduce the setup procedure to single use of the docker run command.

By encapsulating SLEPc into a docker image there is an easy-to-use possibility to include SLEPc into the datafold processing pipeline to gain access to its performant solving algorithms. For cluster environments and high performance requirements we recommend not to use dockerized versions of SLEPc to avoid the virtualization overhead introduced by the docker toolset and to allow better scaling properties on cluster system topologies.

Acknowledgment

The authors gratefully acknowledge the Leibniz Supercomputing Centre for funding this project by providing computing time on its Linux-Cluster.

Bibliography

[CR12] C. Campos and J. E. Roman. Strategies for spectrum slicing based on restarted Lanczos methods. *Numer. Algorithms*, 60(2):279–295, 2012.

[CR16a] C. Campos and J. E. Roman. Parallel Krylov solvers for the polynomial eigenvalue problem in SLEPc. *SIAM J. Sci. Comput.*, 38(5):S385–S411, 2016.

[CR16b] C. Campos and J. E. Roman. Restarted Q-Arnoldi-type methods exploiting symmetry in quadratic eigenvalue problems. *BIT Numer. Math.*, 56(4):1213–1236, 2016.

[Dal] Lisandro Dalcin. Slepc for python.

[HRTV07] V Hernández, J E Román, A Tomás, and V Vidal. Scalable library for eigenvalue problem computations krylov-schur methods in slepc, 2007.

[Huh] Marko Huhtanen. Arnoldi methods for the eigenvalue problem, generalized or not.

[Kre05] Daniel Kressner. *Numerical Methods for General and Structured Eigenvalue Problems*, volume 46. 2005.

[LDKB20] Daniel Lehmberg, Felix Dietrich, Gerta Köster, and Hans-Joachim Bungartz. datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283, 2020.

[Ste02] G. W. Stewart. A krylov-schur algorithm for large eigenproblems. *SIAM Journal on Matrix Analysis and Applications*, 23:601–614, 7 2002.

A. Dockerfile

A.1. Base Dockerfile

```
FROM ubuntu:18.04

ENV DF_MODE="ALL"

ENV OPENMPI_VERSION=2.1.1
ENV PETSC_VERSION=3.13.3
ENV SLEPC_VERSION=3.13.4
ENV PETSC_ARCH="arch-linux-c-debug"
ENV DATAFOLD_MOUNT_DIR="/datafold_mount_dir"
ENV MPICC="/opt/openmpi/2.1.1/bin/mpicc"

ENV NB_USER="usr_nb_petsc"
ENV NB_UID="1000"
ENV NB_GID="100"
ARG HOME="/home/$NB_USER"

VOLUME benchmarkOut:$DATAFOLD_MOUNT_DIR/BenchmarkResult

# Install Ubuntu dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    git \
    ca-certificates \
    wget \
    python3.8 \
    python3-pip \
    python3-dev\
    openssh-client\
    libopenmpi-dev \
    cmake \
    bzip2 \
    curl \
    pkg-config \
    libblas-dev \
    liblapack-dev \
    \
    openmpi-bin \
    libopenmpi2 \
    libopenmpi-dev && \
    \
    apt-get clean && \
    apt-get autoremove

```

```
44  # Download, Build and Install OpenMPI
45  WORKDIR /tmp
46
47  RUN curl
    ↪ https://download.open-mpi.org/release/open-mpi/v2.1/openmpi-$OPENMPI_VERSION.tar.bz2
    ↪ | tar xjvf - && \
48      cd openmpi-$OPENMPI_VERSION && \
49      ./configure --prefix=/opt/openmpi/${OPENMPI_VERSION}
        ↪ --enable-orterun-prefix-by-default --enable-mpirun-prefix-by-default
        ↪ --enable-static --enable-shared --with-verbs && \
50      make -j2 && make install && \
51      export PATH=\$PATH:\$HOME/opt/openmpi/$OPENMPI_VERSION/bin && \
52      export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/opt/openmpi/$OPENMPI_VERSION/lib
53
54  WORKDIR /tmp
55  RUN git clone --single-branch --branch petsc https://gitlab.com/tobinec/datafold.git
    ↪ datafold
56  WORKDIR /tmp/datafold
57
58  RUN pip3 install --upgrade pip setuptools && \
59      python3 setup.py install && \
60      pip3 install -r requirements-dev.txt && \
61      pip3 install pandas==1.0.5 && \
62      pip3 install mpi4py && \
63      pip3 install psutil && \
64      pip3 install GPUtil
65
66  # Clean up (and remove aptitude repository lists)
67  RUN rm -rf /tmp/* /var/tmp/* && \
68      rm -rf /var/lib/apt/lists/*
69
70  COPY fix-permissions /usr/local/bin/fix-permissions
71  RUN chmod a+rx /usr/local/bin/fix-permissions
72
73  RUN echo "auth requisite pam_deny.so" >> /etc/pam.d/su && \
74      useradd -m -s /bin/bash -N -u $NB_UID $NB_USER && \
75      chmod g+w /etc/passwd && \
76      fix-permissions $HOME
77
78  USER $NB_USER
79  WORKDIR $HOME
80
81  RUN export PATH=\$PATH:\$HOME/opt/openmpi/$OPENMPI_VERSION/bin && \
82      export LD_LIBRARY_PATH=\$LD_LIBRARY_PATH:\$HOME/opt/openmpi/$OPENMPI_VERSION/lib
83
84  # Installing petsc
85  ENV PETSC_USR_BUILD_PATH="$HOME/petsc-build"
86  # create directory as user usr_exec, such that this user has writing rights to the
    ↪ directory.
87  RUN mkdir $PETSC_USR_BUILD_PATH
88  WORKDIR $PETSC_USR_BUILD_PATH
89
```

```
90  RUN wget -q -O-
    ↪ http://ftp.mcs.anl.gov/pub/petsc/release-snapshots/petsc-lite-${PETSC_VERSION}.tar.gz
    ↪ -O - | tar xvz
91
92  ENV PETSC_DIR="$PETSC_USR_BUILD_PATH/petsc-$PETSC_VERSION"
93
94  RUN cd petsc-$PETSC_VERSION && \
95      export PETSC_DIR=$PETSC_USR_BUILD_PATH/petsc-$PETSC_VERSION && \
96      ./configure && \
97      make PETSC_DIR=${PETSC_DIR} PETSC_ARCH=${PETSC_ARCH}
98
99  # Installing slepc
100 ENV SLEPC_USR_BUILD_PATH="$HOME/slepc-build"
101 # create directory as user usr_exec, such that this user has writing rights to the
    ↪ directory.
102 RUN mkdir $SLEPC_USR_BUILD_PATH
103 WORKDIR $SLEPC_USR_BUILD_PATH
104
105 ENV SLEPC_DIR="$SLEPC_USR_BUILD_PATH/slepc-$SLEPC_VERSION"
106
107 RUN wget -q -O- https://slepc.upv.es/download/distrib/slepc-${SLEPC_VERSION}.tar.gz -O
    ↪ - | tar xvz
108
109 RUN cd slepc-$SLEPC_VERSION && \
110     export SLEPC_DIR=$SLEPC_USR_BUILD_PATH/slepc-$SLEPC_VERSION && \
111    ./configure && \
112    make SLEPC_DIR=${SLEPC_DIR} PETSC_DIR=${PETSC_DIR} PETSC_ARCH=${PETSC_ARCH}
113
114 ENV SLEPC_DIR="$SLEPC_USR_BUILD_PATH/slepc-$SLEPC_VERSION"
115 RUN pip3 install --user petsc4py==3.13.0 && \
116     pip3 install --user slepc4py==3.13.0
117
118 CMD [ "/bin/bash" ]
```

A.2. Benchmark Dockerfile

```
FROM dfbase

ENV DF_MODE="ALL"

ENV OPENMPI_VERSION=2.1.1
ENV PETSC_VERSION=3.13.3
ENV SLEPC_VERSION=3.13.4
ENV PETSC_ARCH="arch-linux-c-debug"
ENV DATAFOLD_MOUNT_DIR="/datafold_mount_dir"
ENV MPICC="/opt/openmpi/2.1.1/bin/mpicc"

ENV NB_USER="usr_nb_petsc"
ENV NB_UID="1000"
ENV NB_GID="100"

ENV PETSC_USR_BUILD_PATH="$HOME/petsc-build"
ENV PETSC_DIR="$PETSC_USR_BUILD_PATH/petsc-$PETSC_VERSION"
ENV SLEPC_USR_BUILD_PATH="$HOME/slepc-build"
ENV SLEPC_DIR="$SLEPC_USR_BUILD_PATH/slepc-$SLEPC_VERSION"
ENV SLEPC_DIR="$SLEPC_USR_BUILD_PATH/slepc-$SLEPC_VERSION"

VOLUME benchmarkOut:$DATAFOLD_MOUNT_DIR/BenchmarkResult

USER root
COPY /copyToImage $DATAFOLD_MOUNT_DIR
WORKDIR $DATAFOLD_MOUNT_DIR
RUN fix-permissions $DATAFOLD_MOUNT_DIR
RUN chmod +x ./benchmark.sh
USER $NB_USER


RUN python3 setup.py install --user
CMD [ "./benchmark.sh" ]
```

B. Cluster Batch-Job File

```bash
#!/bin/bash
#SBATCH --job-name idp_datafold_benchmarks_320_7_128_gaussian
#SBATCH -o ./%x.%j.%N.out
#SBATCH -D ./
#SBATCH --get-user-env
#SBATCH --clusters=mpp3
#SBATCH --partition=mpp3_batch
#SBATCH --nodes=1
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=1
# 56 is the maximum reasonable value for CooLMUC-2
#SBATCH --mail-type=all
#SBATCH --mail-user=raith@in.tum.de
#SBATCH --export=NONE
#SBATCH --time=00:02:00
module load slurm_setup
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export TUM_NTASKS=$SLURM_NTASKS
export TUM_DIMENSIONS=7
export TUM_SAMPLES=3200000
export TUM_KERNEL_TYPE=gaussian

module load python

cd datafold

source activate py38

python setup.py install --user

# python tutorials/scurve.py
python tutorials/hypercube.py
```

C. Benchmark Data

C.1. Singlecore Benchmark

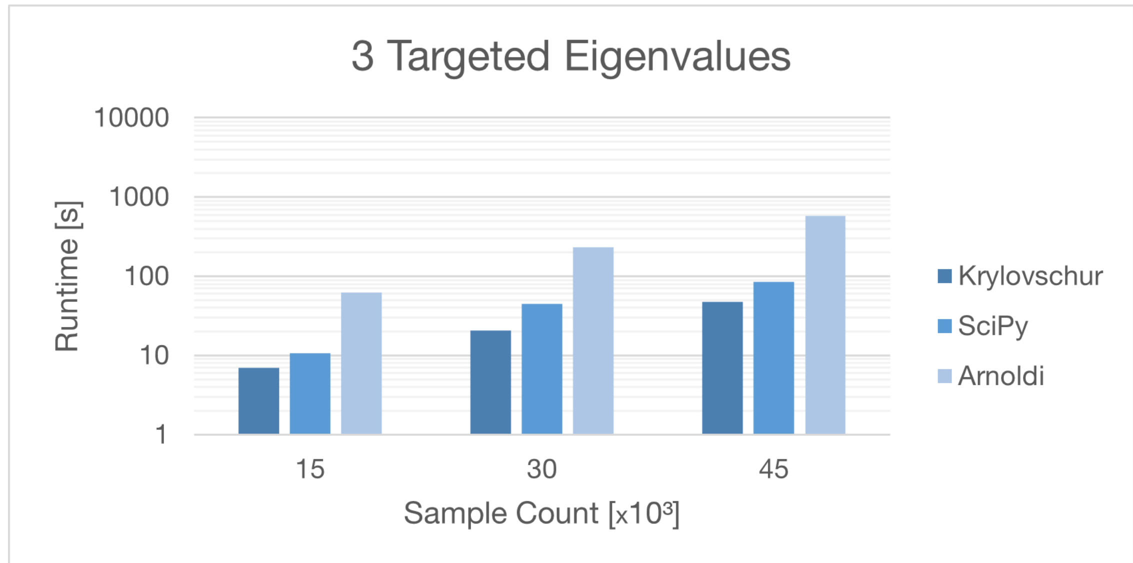

Figure 11 Singlecore Benchmark Results with 3 targeted Eigenvalues

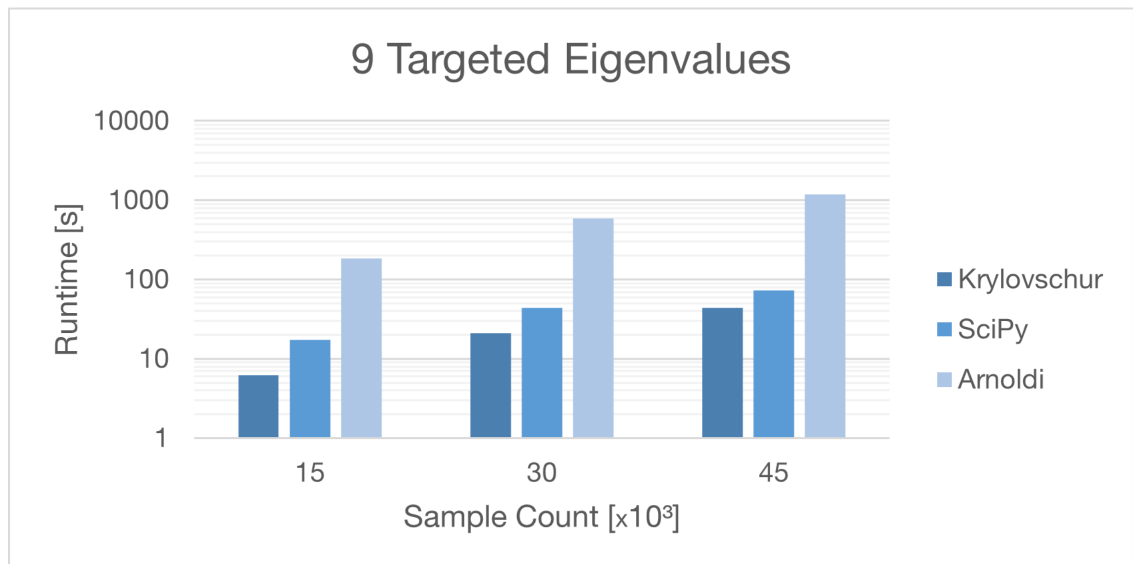

Figure 12 Singlecore Benchmark Results with 9 targeted Eigenvalues

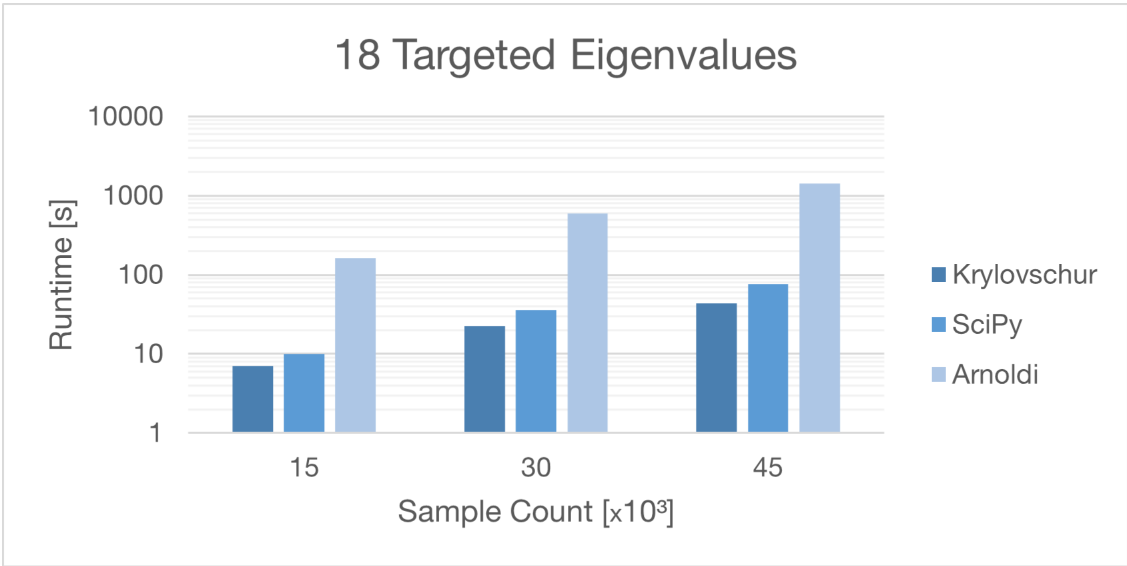

Figure 13 Singlecore Benchmark Results with 18 targeted Eigenvalues

C.2. Multicore Benchmark

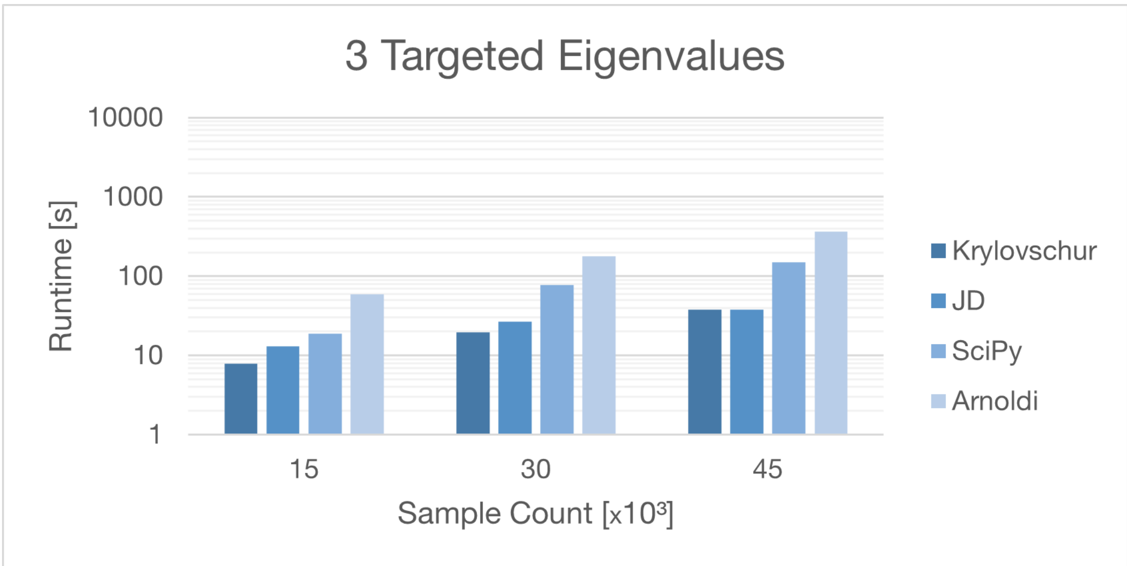

Figure 14 Multicore Benchmark Results with 3 targeted Eigenvalues

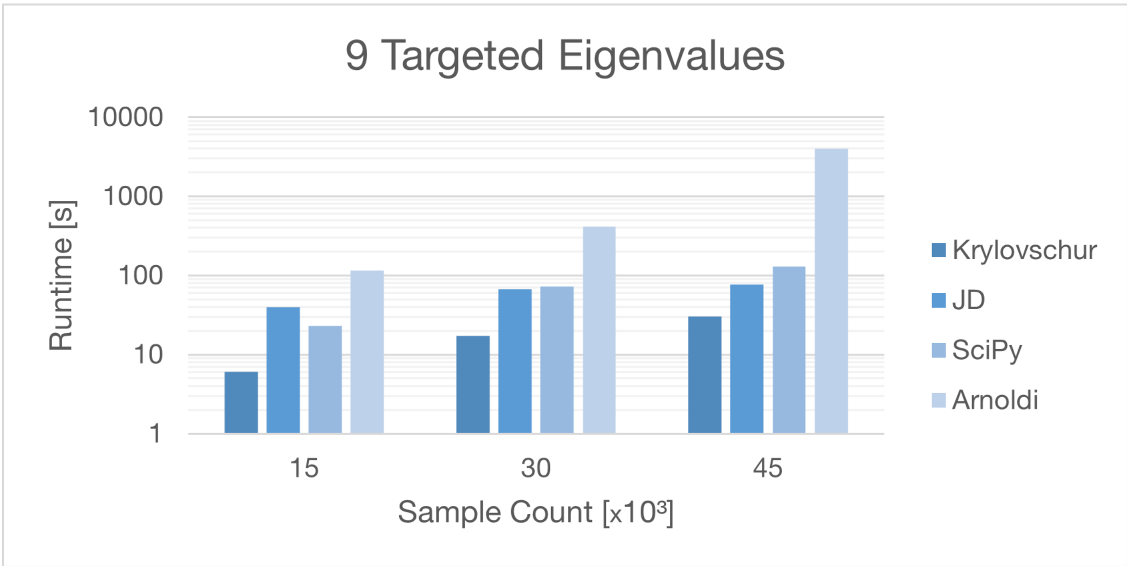

Figure 15 Multicore Benchmark Results with 9 targeted Eigenvalues

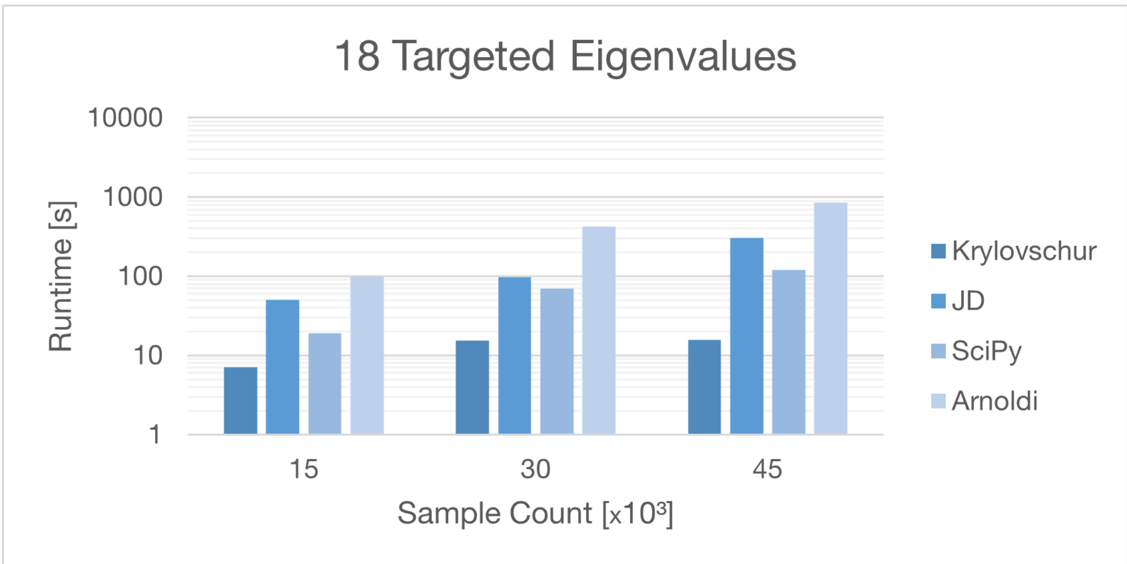

Figure 16 Multicore Benchmark Results with 18 targeted Eigenvalues