An Interface from M++ to Ginkgo for Accelerated Linear Algebra

bwRSE4HPC Project Report

Assigned RSEs:	Tim Schrader[1], Marcel Koch[2]
Project Partners:	Niklas Baumgarten[3]
Scientific field:	Applied Mathematics
Subfield:	Numerical Linear Algebra and Uncertainty Quantification
Start:	23.06.2025
Duration:	6 months
Type of Work:	library integration, GPU support
License:	GPL
Link:	https://gitlab.kit.edu/kit/mpp/mpp
Programming Language:	C++
Technologies:	MPI, OpenMP, Cuda
Target Cluster:	HoreKa

1. Executive Summary

Problem Statement: Solving linear systems is often a dominant computational cost in the finite element library M++ [1], particularly for Uncertainty Quantification (UQ) [2]. The homegrown linear solvers in M++ cannot utilize GPUs, preventing the software from leveraging modern HPC clusters like HoreKa. Maintaining these custom solvers is a burden for the development team and limits access to advanced and highly configurable solvers and preconditioners.

Solution: We integrated the Ginkgo linear algebra library [3] as an optional, GPU-accelerated back end. We developed a highly efficient translation layer between M++'s overlapping MPI domain decomposition and Ginkgo's non-overlapping global indexing. We implemented a Ginkgo solver that integrates seamlessly into the existing M++ architecture and is fully configurable via JSON.

Impact: M++ can now offload distributed linear algebra operations to GPU architectures (CUDA, HIP, SYCL) and use Ginkgo's highly configurable solvers and preconditioners. This accelerates large-scale PDE and UQ simulations, enabling researchers to achieve higher accuracy by exploring larger systems or taking more samples. Section 2.4 shows how GPUs can outperform CPUs by 11 % in a typical use case. By keeping Ginkgo optional, we allow for gradual adoption without disrupting the work of the researchers using M++.

[1] SCC, KIT, Zirkel 2, 76131 Karlsruhe, Germany; tim.schrader@kit.edu
[2] SCC, KIT, Zirkel 2, 76131 Karlsruhe, Germany; marcel.koch@kit.edu
[3] Department of Mathematics, Universität Heidelberg, Im Neuenheimer Feld 205, 69120 Heidelberg, Germany; niklas.baumgarten@uni-heidelberg.de

2. Description of Work

2.1. Initial Problem

M++‘s homegrown linear solvers cannot utilize GPUs, preventing the software from making full use of modern HPC systems like HoreKa.

For the M++ team, there is a significant maintenance cost associated with these homegrown linear solvers. This burden can be reduced by using the dedicated linear algebra library Ginkgo.

For Uncertainty Quantification (UQ) and specifically SPDE sampling, solving the linear system is the dominant computational cost. Using GPU acceleration can make the calculations faster and more accurate, since the accuracy of a UQ computation has been shown to improve with invested compute power.

Furthermore, Ginkgo offers many preconditioner and solver configurations that cannot easily be replicated in M++.

2.2. Solution Design

We chose to integrate the Ginkgo library as an optional linear algebra back end to leverage its specialized GPU kernels and its highly configurable solvers. This approach was chosen over a complete removal of M++‘s native linear solvers and a shift towards Ginkgo data classes to allow for gradual adoption of Ginkgo.

The design centers on a translation layer, formalized in the `IGinkgoConverter` interface, which maps M++ data structures to Ginkgo’s distributed types. We specifically targeted distributed types to ensure that GPU-accelerated solvers remain scalable across multiple MPI ranks. We decided against the inclusion of Ginkgo’s non-MPI-compatible data types to reduce the code complexity and associated maintenance.

To enable efficient conversion between M++‘s overlapping local indices and Ginkgo’s non-overlapping global indices, we changed the memory layout in M++, as described in Section 2.3.6.

2.3. Contributions

2.3.1. Preliminary work

Preliminary work was done by Suryansh Chaturvedi and Niklas Baumgarten. This initial integration of the Ginkgo library laid the groundwork for the development. A `GinkgoSolver` class was written that could solve a linear algebra problem using Ginkgo without MPI.

2.3.2. Architecture

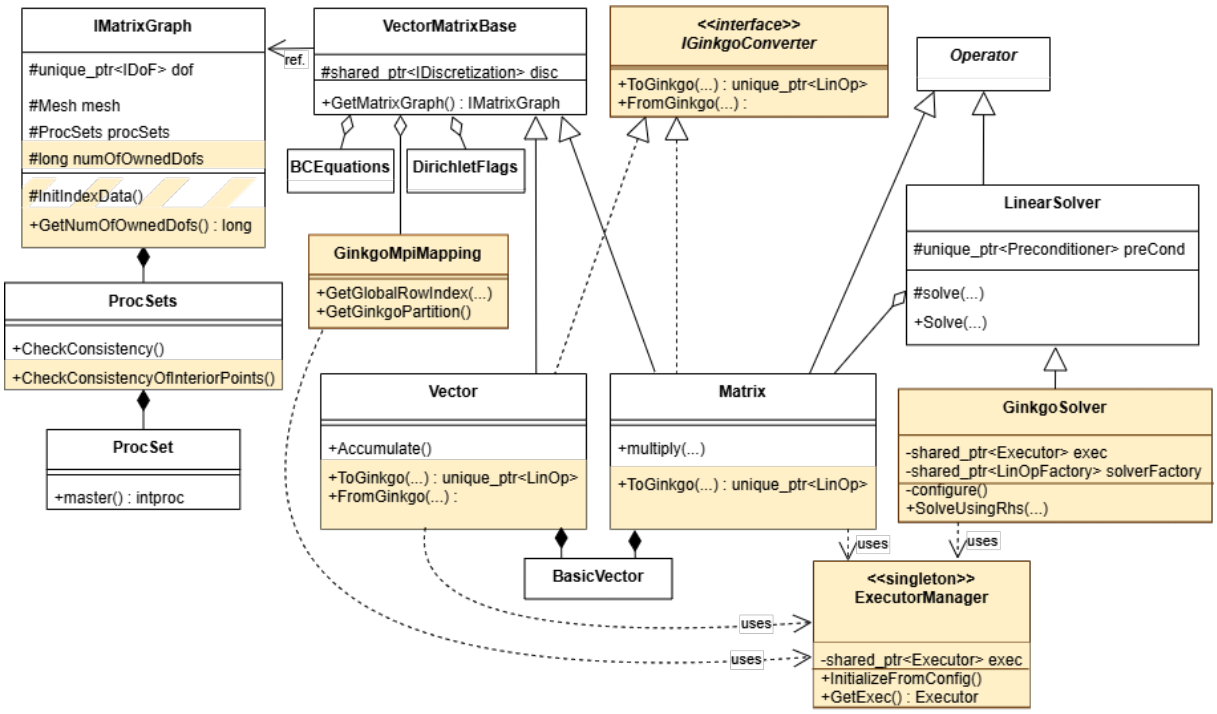

Figure 1: Diagram of some classes in M++. Additions for Ginkgo are marked in yellow.

As Figure 1 shows, the changes to the M++ class structure are largely additive to ensure that Ginkgo remains optional during compilation. The `IMatrixGraph` is built from the distributed mesh, and the ownership information is stored in `ProcSets`. When a `Vector` is first built, it uses this `IMatrixGraph` and constructs its `GinkgoMpiMapping`. Every subsequent `Vector` or `Matrix` can now use that `GinkgoMpiMapping` for its `ToGinkgo` and `FromGinkgo` methods. When the `GinkgoSolver` is chosen as the `LinearSolver`, it can use these conversion methods.

2.3.3. GinkgoSolver

We improved the `GinkgoSolver` so that it inherits from `LinearSolver` and can, in principle, be used by any `PDESolver`. The `LinearSolver` solves

$$Au = b \quad \text{with } A \in \mathbb{R}^{N\times N} \text{ and } u, b \in \mathbb{R}^N$$

for u by iteratively computing approximations u_k that reduce the residual $r = b - Au_k$. A challenge here was that M++ native linear solvers are passed the residual, but Ginkgo solvers expect the original right-hand side b. The `GinkgoSolver` reconstructs the right hand side with $b = r + Au_0$. The constructor of `GinkgoSolver` calls the `GinkgoSolver::configure` method to let Ginkgo configure the linear solver using a JSON object.

2.3.4. Config

M++ uses `.conf` files to set and store configurations in the `Config` singleton class for all parts of the program, including its linear solvers. To use the `GinkgoSolver`, four options are required:

```
LinearSolver = ginkgo
Preconditioner = NoPreconditioner
```

```
executor = reference # or omp, cuda, hip, sycl
GinkgoConfigFile = someGinkgoConfig.json
```

These `.conf` files use a strictly flat hierarchy, making them incompatible with the nested and hierarchical Ginkgo solver configurations. Instead, the `Config` class was extended to use `nlohmann/json` to read in a JSON file that allows for the exact configuration of the Ginkgo solver.

An example of such a JSON config file is:

```
{
  "type": "solver::Gmres",
  "criteria": [
    {
      "type": "Iteration",
      "max_iters": 10000
    },
    {
      "type": "ResidualNorm",
      "reduction_factor": 1e-12
    },
    {
      "type": "ResidualNorm",
      "baseline": "absolute",
      "reduction_factor": 1e-14
    }
  ],
  "preconditioner": {
    "type": "preconditioner::Schwarz",
    "local_solver": {
      "type": "preconditioner::Jacobi"
    }
  }
}
```

Only Schwarz and Multigrid can be used as top-level preconditioners, because only they support MPI distributed solves. However, all available preconditioners can be used as the local solvers for the Schwarz preconditioner.

2.3.5. Algebra Classes and their Ginkgo Interface

We added methods to the `Vector` and `Matrix` classes that enable us to convert their data to/from Ginkgo formats. This interface is formalized in the `IGinkgoConverter` class. Ginkgo contains both distributed and local data types. We decided to only convert to the distributed types to reduce the volume and complexity of the code.

The conversion to and from the Ginkgo type in the Vector class has been designed to rarely require a copy and to let Ginkgo handle the MPI communication. As explained in Section 2.3.6, the data contained in the `Vector` are ordered such that the owned data comes first and is then followed by the non-owned data. This means that in the `Vector::toGinkgoDistrVec` method, a view can be created of only the owned data, which are contiguous in memory. From this view, Ginkgo can then create the distributed vector

either using the same memory for the CPU or by moving it to the GPU. For debugging, the method `Vector::PrintValuesWithGlobalIndex` was added.

The data needed for the conversion are created and stored in the `GinkgoMpiMapping` class, to which the `VectorMatrixBase` class holds a shared pointer. The creation is based on the graph `IMatrixGraph`, so it has to be done only once for each set of vectors and matrices that are based on the same graph and `VectorMatrixBase`.

The `GinkgoMpiMapping` constructor establishes a unified global indexing. It first performs an MPI_Allgather and a partial sum to define contiguous global index ranges for each rank, which are then used to build a Ginkgo Partition object. Finally, it uses an `ExchangeBuffer` to MPI-communicate these global indices between ranks, ensuring that M++ ghost/shared rows are correctly mapped to their corresponding global indices on owning ranks.

The `Matrix::toGinkgoDistrMat` method converts an M++ matrix into the distributed Ginkgo format. It first iterates through local rows to collect non-zero entries into a temporary host-side coordinate format, specifically handling Dirichlet boundary conditions by injecting identity values on owning ranks while zeroing relevant off-diagonal contributions. This host data is then converted to a structure-of-arrays layout and passed, together with the partition object, to Ginkgo's `assemble_rows_from_neighbors`, which use MPI to reconcile matrix rows that are split across different ranks. Finally, the method moves the assembled data to the target execution device, such as a GPU, where it sorts the entries.

2.3.6. IMatrixGraph and ProcSet

M++ and Ginkgo use different ways of partitioning the data across MPI ranks. M++ uses an overlapping domain decomposition with local indices. Information on shared rows is stored in the `ProcSet` class. Ginkgo uses a non-overlapping partitioning with global indices. This difference is illustrated in Figure 2.

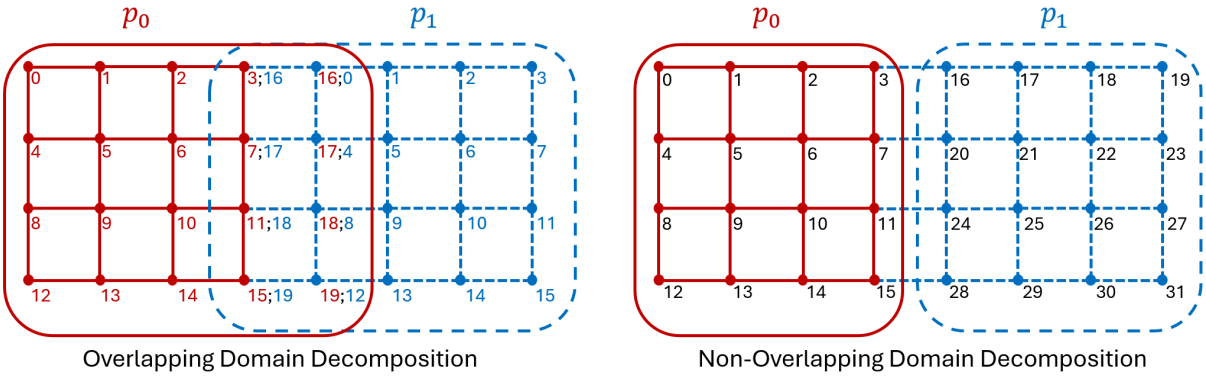

Figure 2: Comparison between overlapping domain decomposition of a 2D mesh with local indices and non-overlapping domain decomposition with global indices. Color and line type indicate ownership, large rounded boxes indicate that a rank (process p) holds that information.

The `ProcSet` class stores on every rank, for every non-owned row, the information about the owner of that row. During development, the `ProcSet::CheckConsistency` method was a valuable tool to debug inconsistencies between ranks. It, however, only checks rows for which `ProcSet` information exists. If two ranks both believe themselves to be owner of

a row, neither of them will have `ProcSet` information about that row. To help debug that, we added the `ProcSet::CheckConsistencyOfInteriorPoints(const std::vector<Point>& allLocalPoints)` method. It checks the consistency between ranks of the ownership information of all rows, not just the ones for which `ProcSet` information exists.

The `IMatrixGraph` class was modified to change the memory layout. We updated `InitIndexData()` to use a `std::stable_partition` on the row IDs, ensuring that all degrees of freedom (DoFs) owned by the current rank are assigned contiguous indices starting from zero. This reordering is the prerequisite for zero-copy views. We also added `GetNumOfOwnedDofs()` to expose this partitioning information to the converter classes.

2.3.7. ExecutorManager

The `ExecutorManager` singleton class was added to provide the correct Ginkgo executor (`reference`, `omp`, `hip`, `cuda`, `sycl`) for the selected hardware back end. One executor exists for every MPI rank. If the selected back end is a GPU, it can happen that there are more MPI ranks than GPUs available. In that case, the method `gko::experimental::mpi::map_rank_to_device_id` can oversubscribe the GPUs using

$$d_{\mathrm{GPU}} = p_{\mathrm{MPI}} \bmod N_{\mathrm{GPUs}}$$

where d_{GPU} is the assigned device ID, p_{MPI} is the MPI process or rank, and N_{GPUs} is the total number of available GPUs. This can have some negative performance impact. The sharing and scheduling of a GPU for multiple processes is handled by the GPU programming models; for example, Nvidia-MPS[4] can be used for a CUDA back end.

2.3.8. VectorMatrixBase

One minor refactoring for improved code clarity was that `Matrix` no longer holds a reference to the `Vector` from which it was created. Instead, shared information (BCEquations, DirichletFlags, GinkgoMpiMapping) is accessible through the `VectorMatrixBase`. Methods that used to need `Matrix::GetVector()` had their argument type changed to `VectorMatrixBase` so that they can now simply accept the Matrix as an argument.

2.3.9. Tests

Tests were added for all changes to the source code. Further, the testing setup was extended with the CMake function `add_mpi_gko_test_for_multiple_exec` that can replace `add_mpi_test`. If the test that is added in this way builds its `MppTest` using `GinkgoTestUtils::ExecNameFromPreprocessor()` , then out of a single source file, one test executable is generated for each available executor. If in one file one or more `TEST(...){...}` are defined, then the main function might look like:

```
int main(int argc, char **argv) {
  GinkgoTestUtils::skipTestIfNoDevice();
  return MppTest(MppTestBuilder(argc, argv)
    .WithConfigEntry("executor", GinkgoTestUtils::ExecNameFromPreprocessor())
    .WithConfigEntry("GinkgoConfigFile", "ginkgoDefault.json")
    .WithPPM())
```

[4] Nvidia Multi-Process Service https://docs.nvidia.com/deploy/mps/index.html

```
    .RUN_ALL_MPP_TESTS();
}
```

2.3.10. CI pipeline

Ginkgo was removed as a submodule and instead included as a library. This massively improved compilation times and reduces maintenance work. For the CI pipeline, that meant that, in the adjacent repository for docker images[5], Ginkgo was included in the used docker files. These are then used by the Gitlab runners to run the compilation, tests, and some benchmarks.

The Gitlab pipeline is directly linked to the HoreKa HPC[6]. To install Ginkgo on HoreKa a special CI job was created that downloads and compiles Ginkgo and makes it available as a module in user space. This job is not part of a regular CI pipeline and does not use the `.mpp-ci-templates`. It is run with:

```
ci.variable="COMPILE_GINKGO_ON_HOREKA=true" .
```

For developers and other users, installation instructions are provided in the README file.

2.4. Evaluation

The first goal of this evaluation was to estimate the negative impact of the conversion overhead. The second goal was to show under what circumstances a GPU-based back end becomes more efficient compared to a CPU-based back end. The benefits of the more configurable Ginkgo solver are not evaluated here, as it is highly dependent on the particular research problem.

For the evaluation, the `BenchSPDEHex` benchmark was implemented which uses the `SPDESamplingOnHexahedron` class. This defines a problem on a cube (three dimensional hexahedron) and computes statistical aggregates of random solution samples. By subdividing the mesh according to the `Level` parameter, the size of this problem can be adjusted, and the number of samples guarantees the reliability of the results. This is similar to problems of active research [4].

Figure 3 shows the results of a comparison for 256 samples of a Level 5 cube being solved by a CG solver with a Jacobi preconditioner. All back ends scale well in this strong scaling benchmark. Compared to the M++ native implementation, the `GinkgoSolver` on CPUs is 16 % slower, likely due to the conversion overhead. A single heavily oversubscribed GPU can make up that overhead and therefore performs on par with the native implementation. Due to the relatively small problem size of 35 937 rows, two GPUs are not faster than one GPU. They do, however, allow the use of 64 MPI ranks in parallel, with each GPU oversubscribed 32-fold.

[5] https://gitlab.kit.edu/kit/mpp/docker_base_images

[6] “Hochleistungsrechner Karlsruhe” (HoreKa) https://www.nhr.kit.edu/userdocs/horeka/

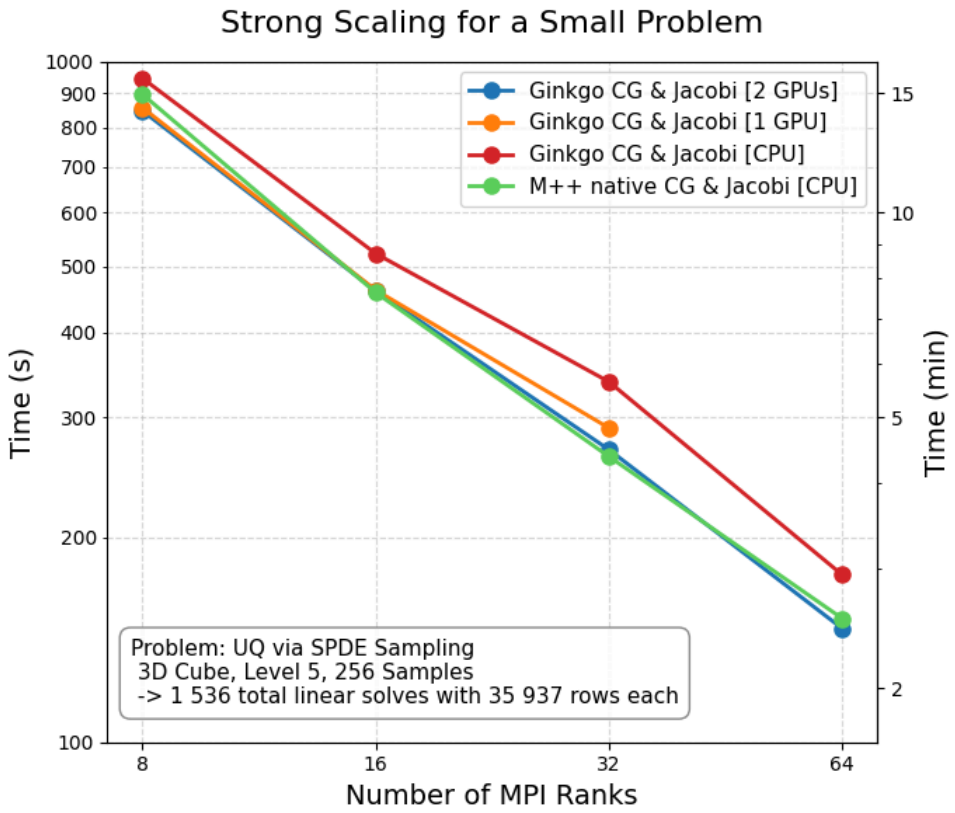

Figure 3: Strong scaling benchmark for a small, three dimensional problem.

The results for a larger, Level 6 problem are shown in Figure 4. Compared to the M++ native implementation of the CG solver and Jacobi preconditioner, the `GinkgoSolver` has a 15 % overhead on the CPU. On two GPUs, however, the `GinkgoSolver` is 11 % faster and even outperforms M++'s implementation of the CG solver with a SuperLU preconditioner by 4 %. With a larger problem size, the GPUs become more efficient compared to the CPUs.

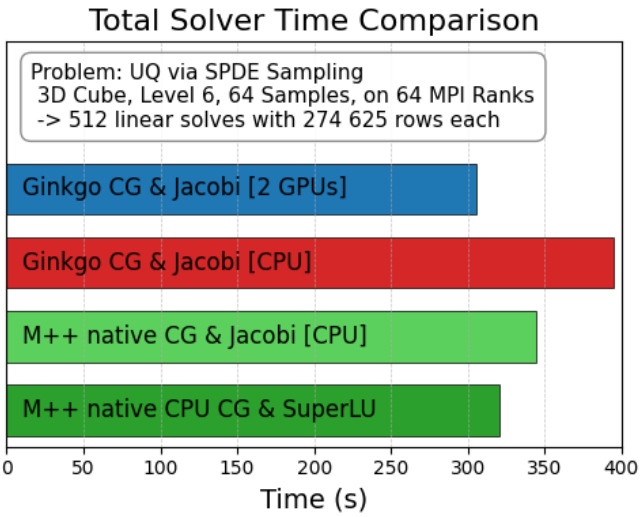

Figure 4: Performance comparison for a Level 6 problem with 64 samples.

While conversion overhead makes the CPU-based Ginkgo solver slightly slower than the native implementation, the GPU back end achieves a performance crossover at Level 6, resulting in an 11% speedup over native CPU solvers.

3. Possible Follow-Ups

The project partners agree that a follow-up project would be very valuable and might happen as part of a DFG project. Such a larger project would focus on Optimal Control and Data Assimilation with Uncertainty Quantification.
This research project would benefit greatly from a budgeted multilevel sampling method that uses not only the CPUs [4] or the GPUs but both. In a hybrid approach, the majority of linear solves could be done on the GPU, while the bias-correcting high-level solves could be done on the CPU. Since these solves do not depend on one another, they could be done asynchronously.
Another area where the current interface can be improved is the handling of MPI ranks. Instead of oversubscribing GPUs, a future interface might use as many MPI ranks on the CPUs as there are cores available and then repartition them into as many ranks as there are GPUs available. Such a repartitioning would have to combine the data from multiple ranks to populate the new ranks in a highly efficient manner. This would eliminate the oversubscription overhead.
A challenge is that multilevel sampling inherently generates linear problems of vastly different sizes. Ginkgo, however, is currently best used with few, large problems or many batched small problems. By changing the way medium-sized problems are handled in M++ or Ginkgo, significant performance improvements could be gained.

The project partners intend to pursue these questions in a proposal for a 3-year PhD position and one year of an RSE position. The proposal will be drafted around April 2026, with a possible start date of October 2026.

4. Acknowledgements

The authors gratefully acknowledge the computing time provided on the high-performance computer HoreKa by the National High-Performance Computing Center at KIT (NHR@KIT). This center is jointly supported by the Federal Ministry of Education and Research and the Ministry of Science, Research and the Arts of Baden-Württemberg, as part of the National High-Performance Computing (NHR) joint funding program (https://www.nhr-verein.de/en/our-partners). HoreKa is partly funded by the German Research Foundation (DFG).

5. Reproducibility Note

5.1. Hardware

Benchmarks were done on
HoreKa Blue, dev_cpuonly partition, 2x Intel Xeon Platinum 8368 CPUs/node with 76 cores/node total, $\geq$ 256 GiB Memory

and
HoreKa Green, dev_accelerated partition, 2x Intel Xeon Platinum 8368 CPUs/node with 76 cores/node total, 512 GiB Memory, 4x NVIDIA A100-40 with 40 GB/GPU .

5.2. Software

- GNU version 13

- CUDA version 12.4
- OpenMPI version 5.0
- Ginkgo version 1.10.0, SWHID: swh:1:dir:30b5c614d2271fec002dd79ab076a6e558af6657, compile options are available in the compile script with the full data.
- M++, SWHID: swh:1:dir:34f5ec89905bd13e3f6f193d754bc61caa353ef5, compile options are available in the compile script with the full data.

5.3. Data

Full data is available at: doi.org/10.5281/zenodo.19098128

Table 1: Benchmarking data

ID	Setup	MPI Ranks	Level	Samples	Time [s]
80	Ginkgo CG & Jacobi [CPU]	8	5	256	945
79	Ginkgo CG & Jacobi [CPU]	16	5	256	522
78	Ginkgo CG & Jacobi [CPU]	32	5	256	338
56	Ginkgo CG & Jacobi [CPU]	64	5	256	176
39	M++ native CG & Jacobi [CPU]	8	5	256	895
40	M++ native CG & Jacobi [CPU]	16	5	256	457
41	M++ native CG & Jacobi [CPU]	32	5	256	262
42	M++ native CG & Jacobi [CPU]	64	5	256	152
22	Ginkgo CG & Jacobi [1 GPU]	8	5	256	856
23	Ginkgo CG & Jacobi [1 GPU]	16	5	256	461
24	Ginkgo CG & Jacobi [1 GPU]	32	5	256	289
26	Ginkgo CG & Jacobi [2 GPUs]	8	5	256	850
27	Ginkgo CG & Jacobi [2 GPUs]	16	5	256	460
28	Ginkgo CG & Jacobi [2 GPUs]	32	5	256	270
32	Ginkgo CG & Jacobi [2 GPUs]	64	5	256	228
31	Ginkgo CG & Jacobi [2 GPUs][7]	64	5	256	147
66	Ginkgo CG & Jacobi [2 GPUs]	64	6	64	305
68	M++ native CPU CG & SuperLU	64	6	64	320
81	Ginkgo CG & Jacobi [CPU]	64	6	64	394
82	M++ native CG & Jacobi [CPU]	64	6	64	344

Bibliography

[1] N. Baumgarten and C. Wieners, “The parallel finite element system M++ with integrated multilevel preconditioning and multilevel Monte Carlo methods,” *Comput. Math. Appl.*, vol. 81, pp. 391–406, 2021, doi: 10.1016/j.camwa.2020.03.004.

[7] Uses a wrapper script to pin MPI rank to GPU; is used instead of ID 32

[2] N. Baumgarten, S. Krumscheid, and C. Wieners, “A Fully Parallelized and Budgeted Multilevel Monte Carlo Method and the Application to Acoustic Waves,” *SIAM/ASA Journal on Uncertainty Quantification*, vol. 12, no. 3, pp. 901–931, 2024.

[3] H. Anzt *et al.*, “Ginkgo: a modern linear operator algebra framework for high performance computing,” *ACM Trans. Math. Software*, vol. 48, no. 1, pp. Art.2, 33, 2022, doi: 10.1145/3480935.

[4] N. Baumgarten and D. Schneiderhan, “Multilevel Stochastic Gradient Descent for Optimal Control Under Uncertainty,” *arXiv preprint arXiv:2506.02647*, 2025.