ICIAM2023 Springer Series 3

Carol S. Woodward
Ulrike Meier Yang
Daniel F. Martin *Editors*

Numerical Software Libraries and their Impact on Scientific Applications



OPEN ACCESS

Springer

ICIAM2023 Springer Series

Volume 3

Editor-in-Chief

Hisashi Okamoto, Department of Mathematics, Gakushuin University, Toshima-ku, Japan

Series Editors

Pingwen Zhang, School of Mathematical Sciences, Peking University, Beijing, China

Ricardo H. Nochetto, Department of Mathematics, University of Maryland, College Park, USA

Carlos Parés, Math Analysis, Statistics and Applied Math, University of Malaga, Málaga, Spain

Giulia Di Nunno, Department of Mathematics, University of Oslo, Oslo, Norway

This series aims to publish some of the most relevant results presented at the ICIAM 2023 held in Tokyo in August 2023.

The series is managed by an independent Editorial Board, and will include peer-reviewed content only, including the Invited Speakers volume as well as books resulting from mini-symposia and collateral workshops.

The series is aimed at providing useful reference material to academic and researchers at an international level.

Carol S. Woodward · Ulrike Meier Yang ·
Daniel F. Martin
Editors

Numerical Software Libraries and their Impact on Scientific Applications

Springer

Editors
Carol S. Woodward
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA

Ulrike Meier Yang
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA, USA

Daniel F. Martin
Lawrence Berkeley National Laboratory
Berkeley, CA, USA



ISSN 3091-3098 ISSN 3091-3101 (electronic)
ICIAM2023 Springer Series
ISBN 978-981-92-0622-3 ISBN 978-981-92-0623-0 (eBook)
https://doi.org/10.1007/978-981-92-0623-0

Mathematics Subject Classification: 65Y15, 65-06, 65F05, 65F50, 65F55, 65M50, 68V35, 68W10, 65B99, 65F10, 65H10, 65M06, 65M20, 65M60, 65M70, 65Y05, 65-04, 65F08, 65M08, 65M22, 65M55, 65N55, 74F25, 74F99

This work was supported by Marcel Koch, Doru Popovici, Dan Martin, Sherry Li, David Schneider, Ulrike Yang, David Gardner and Nathan Roberts.

© The Editor(s) (if applicable) and The Author(s) 2027. This book is an open access publication.

Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.
The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.
Unless otherwise expressly agreed by the Publisher in writing, the Publisher does not authorize use or reproduction of all or any part of this work in any manner for the purpose of training, fine-tuning, evaluating, or developing artificial intelligence models, technologies, or systems. In accordance with Article 4(3) of the EU Directive 2019/790 on copyright and related rights in the Digital Single Market, the Publisher expressly reserves all rights with respect to this work under the text and data mining exception of the directive.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

If disposing of this product, please recycle the paper.

Preface

Some of the most significant products of applied mathematics are algorithms that enable faster and more accurate science and engineering simulations, or even the solution of problems that were previously impossible to model. These algorithms most often impact simulations through numerical software libraries. Numerous numerical software libraries have high-quality implementations of efficient algorithms which facilitate transfer of new algorithms into scientific applications, including those in laboratory and industry settings. Recently, use of these libraries within science and engineering applications has significantly increased, showing the value of these libraries more than ever before. Particularly visible is the context of the recent U.S. Department of Energy's Exascale Computing Project (ECP) where the use of libraries has enabled applications to migrate their simulation codes to take advantage of the newest exascale systems as well as improve performance and capabilities on more traditional architectures.

During the 2023 International Congress on Industrial and Applied Mathematics, a three-part minisymposium was held on the topic of, "Numerical Software Libraries Enabling Benefits to Scientific Applications." These sessions focused on numerical library activities undertaken to ensure value to applications and gave examples demonstrating the use of new algorithms as well as capability and speed improvements within many application areas. The sessions underscored the significant impact of applied mathematics on science and engineering applications.

This volume opens with a chapter describing the extreme-scale software development kit (xSDK), a collection of mathematical libraries and a subset of a larger software ecosystem developed within the ECP. Many large-scale applications, such as multi-physics simulations, require the use of a variety of different independently developed libraries in combination. Ensuring seamless builds and reliable performance of whole applications which utilize more than one of the xSDK libraries requires a healthy software ecosystem. Chapter one discusses the importance of such an ecosystem, including community-driven guidelines to facilitate sustainability and interoperability and the positive impact of such an ecosystem on applications, with focus on the xSDK.

Often applications require the solution of linear systems that are ill-conditioned or indefinite. Solving such problems requires direct factorization-based solvers or preconditioners, which are available through the SuperLU_DIST and STRUMPACK software libraries. The implementation of such algorithms on the current generation of exascale computers with GPUs is very challenging. The second chapter of this volume describes how these challenges can be overcome, and results of the use of these two libraries within fusion applications are presented.

The Ginkgo software library provides efficient sparse numerical linear algebra routines that can be used on a variety of different platforms. These routines allow applications that use such algorithms to transition to new architectures much faster than if such solvers were hand-coded specifically for the application. The third chapter describes the concepts, use, and results of Ginkgo within computational fluid dynamics, power grid, and electro-cardiophysiology simulations.

Achieving efficient performance on large-scale computers with many processors requires scalable algorithms. Algebraic multigrid (AMG) methods are linear solvers which can display such excellent scalability. The fourth chapter in this volume describes the *hypre* software library, which provides a variety of different AMG-based preconditioners whose performance is demonstrated within several applications, including combustion, reservoir simulation, and computational fluid dynamics.

The next chapter in this volume overviews the FFTB software framework for computation of distributed Fourier transforms. The software is demonstrated on problems from plane wave Density Functional Theory codes.

The sixth chapter in this volume describes the use of Anderson accelerated (AA) fixed point methods to solve coupled transport-gyrokinetic turbulence simulations from fusion applications. The AA method is applied from the SUNDIALS numerical library, and several optional variants of AA are explored. These explorations showcase the benefits of a numerical library in that algorithm choices are easily changed through parameter settings. Further, these settings were optimized through use of the GPTune machine learning-based package that optimizes algorithm choices based on a trained model.

The next chapter describes the Intrepid2 software package, a part of the Trilinos software suite, that provides highly efficient finite element and finite volume methods implemented for performance on a variety of computing platforms, including CPU, GPU, and hybrid systems. Results showing effectiveness of the package on plasma physics applications are included.

The eighth chapter discusses the preCICE library which provides methods for coupling individual physics components within multiphysics simulations. Results of a simulation of a human biceps muscle are presented. This simulation includes five coupled models and demonstrates effective data handling, the use of nonlinear solvers for consistency, and coupling schemes.

It is often the case that specific application use cases drive library development which then can be leveraged for use in other independent application areas. The last chapter in this volume describes the development of specific capabilities in the

Chombo framework to support the COGENT continuum gyrokinetic tokamak edge-modeling effort and how these new capabilities were then able to support other unrelated efforts.

Livermore, USA Carol S. Woodward
Livermore, USA Ulrike Meier Yang
Berkeley, USA Daniel F. Martin

Contents

An Ecosystem of Numerical Libraries to Support and Accelerate Scientific Discoveries 1
Hartwig Anzt, Satish Balay, Cody J. Balos, David J. Gardner, Pieter Ghysels, Tzanio Kolev, Xiaoye S. Li, Piotr R. Luszczek, Lois Curfman McInnes, Sarah Osborn, Viktor Reshniak, James M. Willenbring, Carol S. Woodward, and Ulrike M. Yang
1 Introduction 2
2 History 4
3 xSDK Community Policies 6
4 Interoperability 6
5 Release Process and Testing 8
6 Impact on Applications 11
7 Automatic Performance (co-)tuning 13
8 Conclusion 14
References 15

Application Impact of Exascale-Ready Direct Linear Solvers SuperLU_DIST and STRUMPACK 19
Pieter Ghysels and Xiaoye S. Li
1 Introduction 19
2 Algorithm Advances in SuperLU_DIST 21
3 Algorithm Advances in STRUMPACK 24
4 Impact on Applications 28
5 Conclusion 32
References 33

Software Development Aspects of Integrating Linear Algebra Libraries 37
Marcel Koch, Tobias Ribizel, Pratik Nayak, Fritz Göbel, Gregor Olenik, and Terry Cojean
1 Introduction 38
2 Related Work 38
3 The Ginkgo Library 39
4 Integration Approaches in Application Software 42
5 Case Studies 45
6 Summary 50
References 51

On the Use of Algebraic Multigrid in Applications on High Performance Computers 53
Robert D. Falgout, Rui Peng Li, Wayne Mitchell, Daniel Osei-Kuffuor, Victor A. P. Magri, and Ulrike M. Yang
1 Introduction 53
2 Interfaces and Solvers 54
3 Porting Strategies to Exascale Computers 59
4 Use of *hypre* in Applications 61
5 Conclusions 65
References 66

Flexible Multi-Dimensional FFTs for Plane Wave Density Functional Theory Codes 69
Doru Thom Popovici, Mauro Del Ben, Osni Marques, and Andrew Canning
1 Introduction 70
2 Background 71
3 A Flexible Framework for Multi-Dimensional Fourier Transforms 76
4 Experiments and Results 81
5 Conclusion 84
References 84

Towards the Use of Anderson Acceleration in Coupled Transport-Gyrokinetic Turbulence Simulations 87
David J. Gardner, Lynda L. LoDestro, and Carol S. Woodward
1 Introduction 88
2 Coupling Transport and Turbulence 88
3 Anderson Acceleration 91
4 Implementation 93
5 Numerical Results 94
6 Conclusions 101
References 102

Intrepid2: Structure-Preserving Optimizations for High-Order Finite Elements 105
Nathan V. Roberts, Mauro Perego, and Roger P. Pawlowski
1 Introduction 105
2 Structure Preservation Within Intrepid2 107
3 Sum Factorization Algorithm 110
4 Sum Factorized Assembly with Intrepid2 114
5 Performance Results for $H(curl)$ Assembly 116
6 Sum-Factorized Assembly for DPG Vlasov Solver 118
7 Conclusion 121
References 122

Partitioned Multiphysics Simulation of an Electrophysiological Three-Tendon-Biceps Model 125
Carme Homs-Pons, David Schneider, Frédéric Simonis, Miriam Schulte, and Benjamin Uekermann
1 Introduction 125
2 Volume Coupling of Muscle Fibers and Muscle Mechanics 129
3 Multi Coupling of Muscle Mechanics and Three Tendons 133
4 Compositional Coupling of All Five Participants 136
5 Conclusion 137
References 138

A COGENT Case Study: Supporting Applications with Chombo 141
Daniel F. Martin, Milo Dorr, Mikhail Dorf, and Lee F. Ricketson
1 Introduction 141
2 Modeling the Edge Region: The Gyrokinetic Approximation 142
3 The Chombo Framework 143
4 Higher-Order Mapped Multiblock Finite-Volume Discretizations in Chombo 144
5 Mixed-Dimensional Chombo 147
6 COGENT 149
7 Other Applications 151
8 Looking to the Future: Proto, AMR, Embedded Boundaries, and Sparse Grids 153
9 Conclusion 154
10 Code Availability 154
References 155

An Ecosystem of Numerical Libraries to Support and Accelerate Scientific Discoveries



Hartwig Anzt, Satish Balay, Cody J. Balos, David J. Gardner, Pieter Ghysels, Tzanio Kolev, Xiaoye S. Li, Piotr R. Luszczek, Lois Curfman McInnes, Sarah Osborn, Viktor Reshniak, James M. Willenbring, Carol S. Woodward, and Ulrike M. Yang

Abstract The emergence of heterogeneous computers with increasingly complex architectures necessitates continuous adaptation of software to take advantage of increased performance potential. Thus, the use of multiple mathematical libraries designed by expert mathematicians and software developers is crucial for scientific application codes. These libraries are often interconnected, and so, as each library is ported to a new computer architecture, it is also important that these libraries continue to work together. Maintaining such interoperability requires a healthy well-designed software ecosystem with engagement and coordination between library developers who share a common goal of improving software quality, usability, performance portability, and sustainability. This chapter discusses the importance of an ecosystem of high-performance numerical math libraries, including community-driven guidelines to facilitate long-term sustainability and interoperability, and the

S. Balay · L. C. McInnes
Argonne National Laboratory, Lemont, IL, USA
e-mail: balay@mcs.anl.gov

L. C. McInnes
e-mail: mcinnes@mcs.anl.gov

X. S. Li
Lawrence Berkeley Laboratory, Berkeley, CA, USA
e-mail: xsli@lbl.gov

C. J. Balos · D. J. Gardner (✉) · T. Kolev · S. Osborn · C. S. Woodward · U. M. Yang
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
e-mail: gardner48@llnl.gov

C. J. Balos
e-mail: balos1@llnl.gov

T. Kolev
e-mail: kolev1@llnl.gov

S. Osborn
e-mail: osborn9@llnl.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_1

positive impact of the ecosystem on applications; in particular, the focus is on the Extreme-Scale Scientific Software Development Kit (xSDK).

Keywords xSDK (Extreme-scale scientific software development kit) · Numerical software libraries · Software interoperability · High-Performance Computing (HPC) · Scientific software ecosystem · Performance autotuning

1 Introduction

Application scientists want to create larger and more detailed simulations of physical phenomena. Achieving these goals required the development of exascale computers, i.e., systems capable of exceeding a quintillion, or 10^{18}, floating point calculations per second. This has led to the rise of ever more complex computer architectures. The increase in computational power and the availability of more memory provided the opportunity for application scientists to solve larger more complex problems with higher fidelity. However, in order to take advantage of the full potential of these high-performance computers, a variety of challenges had to be overcome. Application codes and third-party libraries needed to be adapted to leverage the new computing capabilities, often requiring the introduction of new programming models. More complex science models required new and improved mathematical algorithms to enable the solution of harder problems. These challenges go beyond the ability of a single team of application developers, since they require expertise in a variety of different disciplines, including software development skills for new

C. S. Woodward
e-mail: woodward6@llnl.gov

U. M. Yang
e-mail: yang11@llnl.gov

P. R. Luszczek
MIT Lincoln Laboratory, LLSC, Lexington, MA, USA
e-mail: piotr.luszczek@ll.mit.edu

V. Reshniak
Oak Ridge National Laboratory, Oak Ridge, TN, USA
e-mail: reshniakv@ornl.gov

J. M. Willenbring
Sandia National Laboratories, Albuquerque, NM, USA
e-mail: jmwille@sandia.gov

H. Anzt
Technical University of Munich, Munich, Germany
e-mail: hartwig.anzt@tum.de

P. Ghysels
Advanced Micro Devices, Santa Clara, CA, USA
e-mail: pieter.ghysels@amd.com

computer architectures and deep understanding of mathematics in various areas, such as discretization of models on structured and unstructured grids, solution of linear and nonlinear systems, optimization, and so on. Dealing with these issues requires an ecosystem of libraries and tools that can support these application codes.

While the word "ecosystem" usually evokes images of an environment comprised of living organisms, like plants and animals, here it is more generally defined as *a group of independent but interrelated elements that comprise a unified whole*. One example of such an ecosystem that can support applications is E4S,[1] the Extreme-Scale Scientific Software stack [12, 21], which is comprised of packages from a large variety of different software categories, including programming models, runtime systems, development tools, numerical libraries, data and visualization software, and workflows. However, we focus here on a smaller ecosystem, which is also contained within E4S, the Extreme-Scale Scientific Software Development Kit, in short xSDK,[2] which consists of a variety of independently developed numerical libraries [6, 22, 24], including parallel, often GPU-enabled, software for solvers, meshes, discretizations, and more. It provides application developers a variety of the building blocks they need.

An efficient ecosystem requires that all its elements work well together, i.e., it needs to enable an application to smoothly build when using components of the system in combination. Achieving this cohesion presents several challenges, particularly when the individual libraries are managed by independent teams. To facilitate the combined build and use of multiple packages in the kit, the xSDK community developed a set of guidelines, known as the xSDK community policies, which are periodically reevaluated and updated. All xSDK member packages have at least one interoperable connection with another xSDK member, leading to additional challenges, particularly when testing the entire xSDK. A cohesive build of these packages is enabled by the Spack[3] package manager.

The rest of this chapter is organized as follows. We discuss the history of the xSDK in Sect. 2, the xSDK community policies in Sect. 3, and delve deeper into interoperability in Sect. 4, including the introduction of a suite of example codes demonstrating interoperability. Section 5 discusses the release process and testing challenges of the xSDK. Before concluding in Sect. 8, we highlight some examples of the impact the xSDK has made on applications in Sect. 6 and the benefits of automatic performance tuning of xSDK libraries in Sect. 7.

[1] E4S: https://e4s-project.github.io/.

[2] xSDK: https://xsdk.info/.

[3] Spack: https://spack.io/.

2 History

While there have been several efforts to address challenges inherent in a numerical software ecosystem in the past (e.g., see [11] and references therein), the xSDK was the first to truly deliver on this promise. The xSDK effort began in 2014 as part of the Interoperable Design of Extreme-scale Application Software (IDEAS) project [17].[4] While the original IDEAS project (now dubbed IDEAS-Classic) focused on use case requirements for multiphysics and multiscale terrestrial ecosystem modeling, the compatibility, usability, quality, and process improvements made for important DOE libraries–including *hypre*,[5] PETSc,[6] SuperLU,[7] and Trilinos[8]–were broadly beneficial and laid the groundwork for the expansion of the xSDK.

The first release of the xSDK in 2016 included only the four numerical libraries listed above and the two domain components Alquimia[9] and PFLOTRAN.[10] At the time of the release, this was a significant accomplishment, since previously it was not possible to link the four libraries into a single executable due to incompatibilities. The dependency graph for the IDEAS ecosystem of 2016 is shown in Fig. 1. The xSDK libraries are shown in green. Another significant benefit from the early days of the xSDK was breaking down barriers between the development teams of different libraries by establishing compatible goals as well as a common vocabulary. A tangible outcome of establishing these goals was the creation of the initial xSDK Community Policies, which are discussed in Sect. 3.

As a result of the early successes under the IDEAS project, the xSDK was funded as its own project as part of the Software Technologies focus area of the Exascale Computing Project (ECP) [8, 14] under the Math Libraries thrust, while the IDEAS-ECP project was funded to help mitigate the challenges of software development in emerging exascale environments and thereby to enhance the productivity and sustainability of the software ecosystem. The ECP brought together the developers of 70 distinct software technologies to create a rich and sustainable scientific software ecosystem [18] that supported a diverse suite of applications in chemistry, materials, energy, Earth and space science, data analytics, optimization, AI, and national security [1]. Projects targeted portable performance across multiple exascale computer architectures. ECP's aggressive goals required intensive software refactoring and development [3], involving more than 1,000 researchers across DOE labs and collaborating universities, as well as partnerships with DOE computing facilities, industry, and other US agencies.

---

[4] https://ideas-productivity.org/activities/ideas-classic/.

[5] https://llnl.gov/casc/hypre.

[6] https://petsc.org.

[7] http://portal.nersc.gov/project/sparse/superlu/.

[8] https://trilinos.org/.

[9] https://ideas-watersheds.github.io/software-ecosystem/codes/alquimia.

[10] https://pflotran.org.

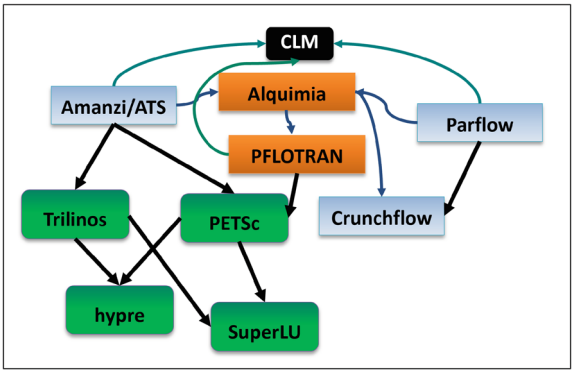


Fig. 1 Application packages (black, blue, and orange) and mathematical libraries (green) needed to generate multiscale, multiphysics integrated surface-subsurface hydrology models

During the ECP, xSDK membership grew over the years. It currently includes 26 math libraries: AMReX,[11] ArborX,[12], ButterFlyPACK,[13] DataTransferKit,[14] deal.ii,[15] ExaGO,[16] Ginkgo,[17] heFFTe,[18] HiOp,[19] *hypre*, libEnsemble[20] MAGMA,[21] MFEM,[22] Omega_h,[23] PETSc/TAO, PHIST,[24] PLASMA,[25] preCICE,[26] PUMI,[27] SLATE,[28] SLEPc,[29] STRUMPACK,[30] SUNDIALS,[31] SuperLU, TASMANIAN,[32] and Trilinos, and 2 application packages: PFLOTRAN, and Alquimia.

[11] https://amrex-codes.github.io/amrex/.
[12] https://github.com/arborx/arborx.
[13] https://github.com/liuyangzhuan/ButterflyPACK.
[14] https://github.com/ORNL-CEES/DataTransferKit.
[15] http://www.dealii.org/.
[16] https://github.com/ORNL/ExaGO.
[17] https://github.com/ginkgo-project/ginkgo.
[18] https://bitbucket.org/icl/heffte.
[19] https://github.com/LLNL/hiop.
[20] https://github.com/Libensemble/libensemble.
[21] http://icl.utk.edu/magma/.
[22] https://mfem.org/.
[23] https://github.com/sandialabs/omega_h.
[24] https://bitbucket.org/essex/phist.
[25] http://icl.utk.edu/plasma/.
[26] https://www.precice.org/.
[27] https://github.com/SCOREC/core.
[28] https://bitbucket.org/icl/slate.
[29] http://slepc.upv.es/.
[30] http://portal.nersc.gov/project/sparse/strumpack/.
[31] https://computing.llnl.gov/projects/sundials.
[32] https://ornl.github.io/TASMANIAN.

3 xSDK Community Policies

The ecosystem of numerical libraries for scientific applications is a confluence of the developer community with the hardware platforms and software tools they target. To address a range of technical and social challenges that arise when building and sustaining a rich software ecosystem a shared set of policies and practices agreed to by all members was developed. This helps maintain community cohesion while keeping the common goals in clear view of the members. Akin to a founding document, the policies first and foremost indicate that the community is in agreement on core principles which define expectations that must be fulfilled by all participating libraries.

Drawing on decades of open-source library development experience in the U.S. Department of Energy, the xSDK community policies are intended to create a highly productive, sustainable, and interoperable software library ecosystem. The short form of the latest version of the community policies is presented in Fig. 2 and a detailed version is available on GitHub.[33] The mandatory policies can be seen as quality assurance for participation in the xSDK ecosystem and is complemented by a set of recommended policies that have been identified to improve software productivity and sustainability but that cannot be fulfilled or do not apply to all software libraries. The policies are regularly reviewed and reevaluated as necessary to stay relevant with current software development and deployment best practices. Revisions or additions to the community policies can be suggested at anytime by opening a GitHub issue for discussion. Changes will then be voted on for inclusion by xSDK members.

4 Interoperability

For a software package to become an xSDK member it must comply with the mandatory community policies and have interoperability with at least one other xSDK member package. There are different levels of interoperability:

- Packages are used side by side in an application.
- Libraries exchange (or control) data with each other
- A library calls another library to perform a unique computation.

Today's complex multiphysics scientific and engineering simulation codes require a broad array of efficient implementations of numerical algorithms. As a result, many of these codes combine multiple numerical methods from different libraries to help achieve their goals. Interoperability between these libraries, each targeting a particular need in the overall simulation code, has proven to be a significant challenge with problems falling in two main areas. First, packages must be able to be compiled together when generating a single executable. Issues often arise through namespace

[33] https://github.com/xsdk-project/xsdk-community-policies.

Mandatory xSDK policies: must be satisfied

M1. Support portable installation through Spack (includes xSDK Spack variant guildelines)
M2. Provide a comprehensive test suite.
M3. Employ user-provided MPI communicator.
M4. Give best effort at portability to key architectures.
M5. Provide a documented, reliable way to contact the development team.
M6. Respect system resources and settings made by other previously called packages.
M7. Come with an open-source license.
M8. Provide a runtime API to return the current version number of the software.
M9. Use a limited and well-defined symbol, macro, library, and include file name space.
M10. Provide publicly available repository.
M11. Have no hardwired print or IO statements.
M12. Allow installing, building, and linking against an outside copy of external software.
M13. Install headers and libraries under <prefix>/include/ and <prefix>/lib/.
M14. Be buildable using 64-bit pointers. 32 bit is optional.
M15. All xSDK compatibility changes should be sustainable.
M16. Have a debug build option.
M17. Provide sufficient documentation to support use and further development.

Recommended xSDK policies: encouraged, but not required

R1. Provide at least one validation test that can be invoked through Spack.
R2. Possible to run test suite under valgrind in order to test for memory corruption issues.
R3. Adopt and document consistent system for error conditions/exceptions.
R4. Free all system resources it has acquired as soon as they are no longer needed.
R5. Provide a mechanism to export ordered list of library dependencies.
R6. Provide versions of dependencies.
R7. Have README, SUPPORT, LICENSE, and CHANGELOG file in top directory.
R8. Provide version comparison preprocessor macros.

Fig. 2 xSDK community policies version 1.0.0

clashes, specific dependencies on software versions or compilers and their flags, or assumptions made at development time, like floating point precision. Many of the xSDK Community Policies were designed to address these challenges. The second challenge is in developing deeper interoperabilities between packages and largely stem from creating interfaces between packages to allow use of one package from another. Here decisions on how data is passed from one package to the other have significant bearing on the overall efficiency of the resulting capabilities, especially when the simulation code requires flexibility in the target computing platform architecture and programming model. Deeper interoperabilities, e.g., a common interface to several solver libraries, provide added functionalities for application codes and can accelerate application development.

The xSDK Community Policies work in conjunction with the Spack package manager to address the challenge of compiling the packages together. For example,

policy M9 ensures namespace clashes do not happen, while policy R8 helps packages more easily support multiple versions of dependencies. With Spack, a package describes their dependencies, variants (compile-time options), and conflicts between variants, with specific versions of dependencies, compilers, etc. When installing a package with Spack, a user will tell Spack the abstract specification (spec) that they want to build and install, and Spack then turns this abstract spec into a concrete spec with all conflicts (to the extent possible based on the package description) resolved. The xSDK itself is a Spack meta-package which includes the individual xSDK packages as dependencies. The dependencies of the xSDK are described with abstract specs that are known to work together but are not unnecessarily descriptive. A typical dependency spec in the meta-package includes the version or version range of the dependency to be used, the floating point precision(s), the ordinal/index size(s), and relevant variants (such as GPU support) which may be enabled or disabled.

As noted above, to be a member package of the xSDK, there must be a deeper interoperability with at least one other member package. These interoperabilities have been greatly expanded in recent years to build greater flexibility and capability in the available numerical software stack. Figure 3 shows the current state of these deeper interoperabilities between the packages. A colored square in a row in this figure indicates that the package in the row name uses the package in the respective column name. In general, we see that discretization packages have fuller rows while solver packages have fuller columns. Colors indicate the maturity of the interoperability with magenta indicating that a deeper interoperability is in place, enabled in the xSDK Spack package, and regularly tested by the xSDK GitLab CI as discussed in Sect. 5. As the xSDK has matured, this matrix has become larger with more rows and columns and fuller with deeper interoperabilities in place.

A suite of example codes, called xsdk-examples,[34] demonstrate the use of various xSDK packages emphasizing the interoperability among packages to solve problems of interest. The examples provide both a training tool for users and a test suite as part of the testing of the xSDK that confirms correct xSDK builds. In its most recent release, xsdk-examples v0.4.0, the suite showcases 26 example codes, where 20 of the xSDK member packages are featured. Additionally, 10 examples are CUDA-enabled and 6 examples are HIP-enabled to demonstrate GPU interoperability between packages.

5 Release Process and Testing

Once a year, xSDK members create a new xSDK release that includes synchronized versions of all member packages and is installable with Spack. The most recent release, v1.0.0 from November 2023, can be installed via *spack install xsdk@1.0.0*. If one wants to build xSDK for NVIDIA, AMD, or Intel GPU architectures, additional specifications are required, as shown in the examples below:

[34] https://github.com/xsdk-project/xsdk-examples.

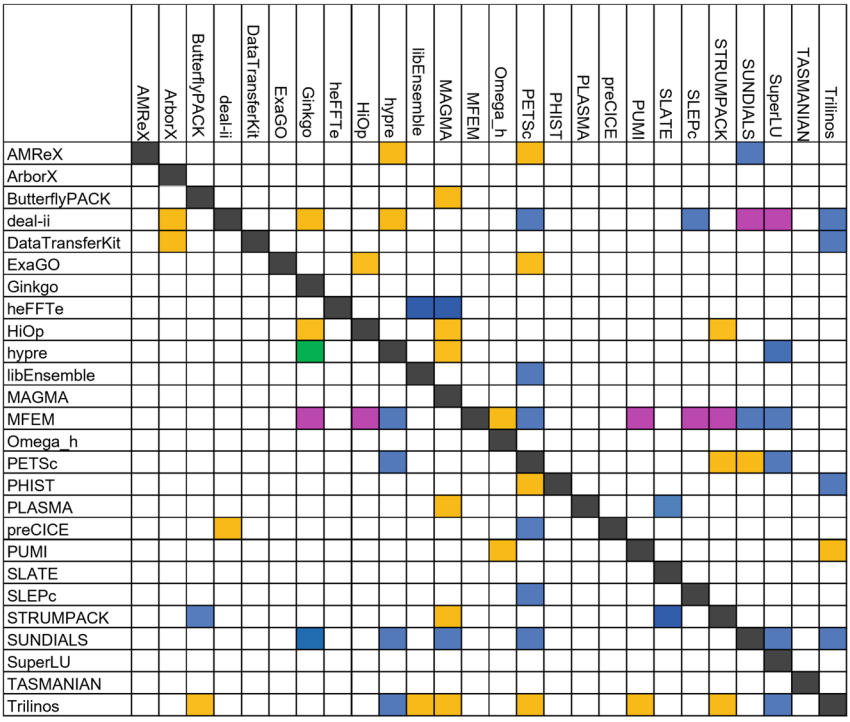


Fig. 3 Interoperability matrix for mathematical libraries in xSDK 1.0.0 (Nov. 2023). A filled in cell indicate that the row package can use capabilities from the column package e.g., AMReX can use linear solvers from *hypre* or PETSc or time integrators from SUNDIALS. Cell colors denote interoperability exists (yellow); interoperability exists and is enabled in the xSDK Spack package (blue); interoperability exists, is enabled in the Spack package, and is tested by the GitLab CI or in xsdk-examples (magenta); and interoperability is planned (green)

spack install xsdk@1.0.0 +cuda cuda_arch=70
spack install xsdk@1.0.0 +rocm amdgpu_target=gfx90a
spack install xsdk@1.0.0 +sycl ~trilinos ~dealii %oneapi@2023.2.0

As the xSDK member packages are independent projects with developers working towards the packages' primary goals and requirements, the generation of a new release requires increased coordination between the individual packages. Participation within xSDK requires a public recent release of the package to be included in the new xSDK release. As part of this release process, installation and interoperability are tested to make sure the xSDK, as a whole, is installable via Spack. This process includes portability testing across multiple compilers (e.g., GNU, Clang, and Intel), workstations (e.g., Intel, AMD, and ARM CPUs), GPU platforms (i.e., NVIDIA/CUDA, AMD/ROCm and Intel/SYCL), and leadership computing facility (LCF) machines at ALCF, OLCF, NERSC, and LLNL. Any problems that come up

need to be solved collectively by the xSDK project members. Each issue might need to be addressed and resolved by a different set of developers, thus improving the xSDK package ecosystem as a whole.

An early stage of xSDK release testing involves building development versions of packages against each other during their development cycles to identify regressions, API changes, or other updates that could potentially break interoperability between packages or the xSDK build as a whole. These jobs includes builds of different subsets of the xSDK that have dependencies between them. Detecting and fixing any problems that arise early in the release cycle can alleviate problems that would otherwise come up closer to the release date as packages coordinate on release versions. This testing is automated via GitLab CI which schedules and runs build tests daily.

As the xSDK release approaches, testing turns towards the full xSDK build and stabilization of individual package releases. This process is intertwined with the release process of individual packages with the goal of building the full xSDK collection to identify and address issues that need to be fixed before the release of individual member packages. This process includes changes to xSDK member package Spack recipes to enable building pre-release snapshots using the xSDK build process and is initially done with manual build testing on workstations. Once the builds can be automated, the tests are added to the GitLab CI. This automated testing captures new changes introduced in the member packages, the xSDK Spack meta package, or within Spack itself. This process is spread over several weeks with many xSDK member projects producing public releases at their convenient schedule, during this stage.

As the xSDK collection of package builds improves on workstations, it is also manually tested on LCF machines to find and address issues specific to those environments. LCF machines have more complex compiler and build software infrastructure compared to workstations as well as scheduler requirements, allocation limits, and wait times to access compute nodes. These extra considerations add to the complexity of testing xSDK builds on LCF systems. Moreover, xSDK builds are resource intensive. As some systems impose resource limits on the common use front-end nodes, xSDK builds are attempted on the corresponding compute nodes of such systems. Utilizing the compute nodes has the benefit of addressing potential differences in builds between compute and front end nodes. As member package developers might not have access to these resources, reproducing issues that come up during testing and addressing them further adds to the testing complexity.

After successful builds on workstations and LCF machines, the new xSDK version is released and made available via Spack. Once the new version is available via Spack, additional testing of this release is scheduled via GitLab CI in order to test it against the daily changes that are added to Spack. Again any issues that come up are addressed and resolved.

6 Impact on Applications

A variety of application codes—including ECP teams working in chemistry, materials, energy, Earth and space science, data analytics, optimization, AI, and national security [1] as well as many others in the broader community—have been and are benefiting from the greater availability of cutting edge numerical mathematical methods hardened into robust implementations through the establishment of the xSDK; whether it is from easier, more user-friendly builds of the whole application code or the availability of additional mathematical features from new library interoperability.

The first application to benefit from the xSDK was the one that sparked the whole endeavor, the Advanced Terrestrial Simulator or ATS[35] [19, 23], pictured in Fig. 1. It used the mathematical libraries *hypre*, Trilinos, PETSc, and SuperLU, and the domain components PFLOTRAN and Alquimia. The geochemistry reaction engine in PFLOTRAN was accessed through Alquimia. Updating the involved packages to adhere to the xSDK community policies removed version incompatibilities, namespace collisions, and other inter-package conflicts. The efforts also led to improved interoperability between some of the libraries and to the development of needed new features. With these improvements to the ecosystem, the application team was able to test a conceptual model of the Copper Creek catchment in the East River watershed, Colorado, in a quantitative manner with integrated-hydrology and a fully coupled surface/subsurface reactive transport model, see top left image in Fig. 4. In this model, they simulated the hydrology, solute transport, and chemical reactive processes of 17 major chemical components in pyrite oxidation and calcite dissolution reactions along with more than 40 secondary species.

Another project that was able to take advantage of the xSDK is the ECP ExaWind team. This projects examines the flow physics governing whole wind plant performance, including wake formation, complex terrain impacts, and turbine-turbine interaction effects, see the top right image in Fig. 4. The ExaWind team utilizes xSDK libraries AMReX, *hypre*, and Trilinos. The interoperability of xSDK libraries of Trilinos and *hypre* enabled the team to discover that *hypre* was able to solve some of their problems more efficiently.

The PeleLMeX[36] code [9] for simulating low-mach reacting flows, see the bottom image in Fig. 4, also benefits from the xSDK ecosystem. PeleLMeX is built on top of the AMReX software framework for block-structured adaptive mesh refinement (AMR) applications and GPU portability. It utilizes AMReX's interoperability with *hypre* to solve elliptic systems that arise in a projection method to update advection velocities and with SUNDIALS for evolving numerous chemical reaction systems in time. When implicit time integration methods in SUNDIALS are employed to handle stiff chemistry mechanisms, PeleLMeX also utilizes SUNDIALS' interoperability with MAGMA and Ginkgo for solving linear systems. With the help from these

[35] https://amanzi.github.io/.

[36] https://github.com/Pele-Suite/PeleLMeX.

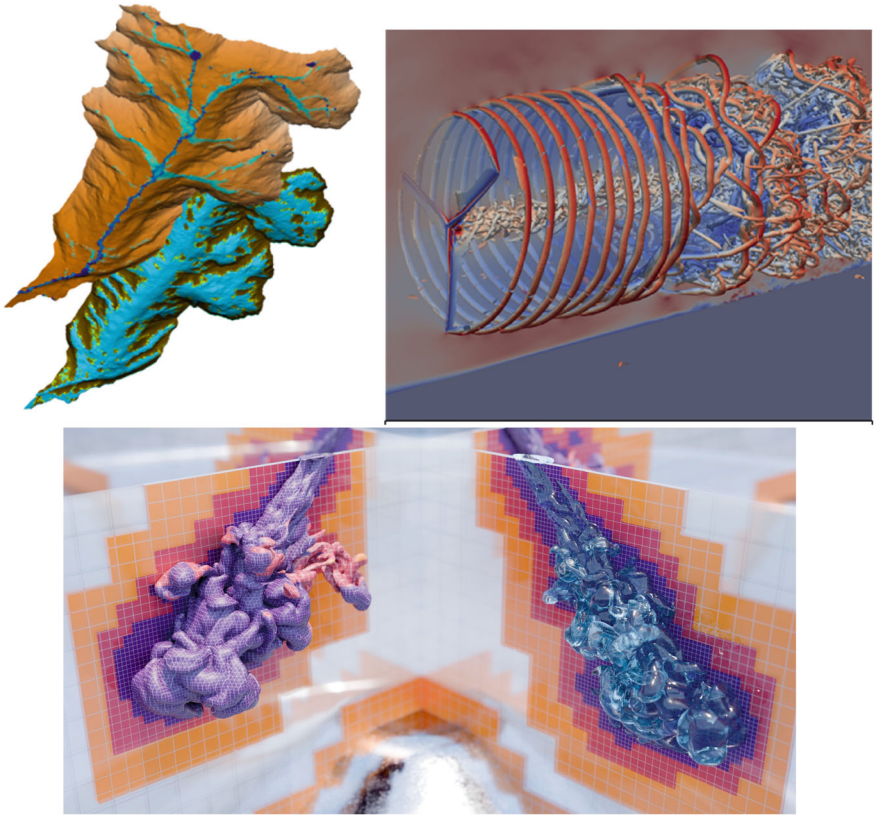


Fig. 4 Top left: Integrated hydrology simulation of Copper Creek shows the ponded depth of water on the surface (visualized above the subsurface), and the water saturation in the subsurface (Credit: David Moulton and Zexuan Xu). Top right: Flowfield, isosurfaces of Q-criterion colored by votricity magnitude and a plane with vorticity-magnitude iso-contours, for the NREL 5-MW rotor with rigid blades operating in uniform inflow of 8 m/s (Credit: Shreyas Ananthan and Ganesh Vijayakumar). Bottom: Turbulent combustion of n-dodecane in a high-pressure piston chamber computed using PeleLMeX with an adaptive mesh hierarchy on Frontier (Credit: Nicholas Brunhart-Lupo and Marc Day)

libraries in the xSDK, PeleLMeX has demonstrated good scaling on the Frontier exascale supercomputer while simulating combustion processes typical of modern combustion systems [5].

In addition, xSDK played a critical role in the CEED[37] ECP co-design center, where *hypre* was vital in preconditioning matrix-free high-order finite element discretizations in NekRS and MFEM-based applications, including ExaSMR.[38] In addition to MFEM, the CEED software and applications also utilized GPU dense linear

[37] https://ceed.exascaleproject.org.

[38] https://www.exascaleproject.org/research-project/exasmr/.

algebra routines from MAGMA, meshing algorithms from PUMI, linear algebra from PETSc and additional solvers from Gingko and STRUMPACK. See [2, 13, 20] for more details.

7 Automatic Performance (co-)tuning

In the xSDK ecosystem, apart from installation, testing and interoperability, another critical element is performance optimization—such as runtime, memory usage, energy usage—on an actual HPC machine. The performance of each library is a complex function depending on the input problems and the underlying computer architectures, including the inter-node communication network. Each library usually exposes a set of parameters to the users that can influence the library's performance. Most of the time the users simply apply default parameter settings which may be far from optimum. Worse yet, a bad choice of parameters may lead to non-convergence with solver libraries. When an application code uses multiple libraries, the dimension of the parameter space becomes intractably large for manually searches for the best configuration. We have been developing an autotuning tool to ease performance optimization tasks for users. Given the large number of libraries in the xSDK collection that provide diverse functionalities, the best strategy is to treat the performance optimization as "black-box" optimization. In particular, we employ the statistical Bayesian optimization framework to build performance surrogate models via Gaussian Process (GP) regression. The surrogate models are based on the execution results of the program collected on the actual machines with an automatically-chosen subset of parameter configurations. The tool is called GPTune. It features multitask/transfer-learning algorithms [15], multi-objective optimization, multi-fidelity tuning [25], clustered GP for non-smooth functions, hybrid models for mixture of real, integer and categorical variables [16], and a history database for crowd tuning.

We have applied GPTune to optimize parameters for the math libraries in xSDK as well as real simulation codes. Figure 5 shows the tuning results of two codes. The left subfigure shows pareto optimization for SuperLU_DIST with four parameters and two objectives: time and memory, for a sparse matrix Si2 from the PARSEC quantum chemistry application [15], using eight Cori nodes at NERSC with 256 MPI tasks. The default parameter setting is about 5× slower and uses 14× more memory compared to the tuned setting. The right subfigure shows the tuning of the multiphysics simulation code NIMROD to predict microscopic MHD instabilities of burning plasma in the ITER fusion reactor. The code uses SuperLU_DIST as a preconditioner for the GMRES iterative linear solver. We co-tuned five parameters, 2 from NIMROD and 3 from SuperLU_DIST, and achieved an approximately 10% performance gain on 16 Cory nodes with 512 MPI tasks. The plot also shows that the multi-fidelity algorithm GPTuneBand is superior to the vanilla single-fidelity GPTune [25]. Another demonstration of GPTune can be seen in the paper on the application of Anderson acceleration for fusion applications in this volume [10].

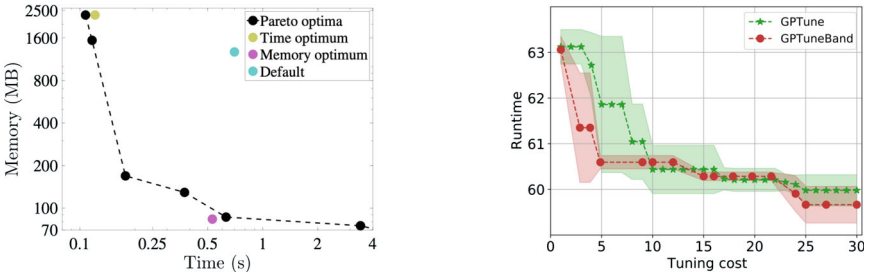


Fig. 5 GPTune tunning results: (left) two-objective tuning for SuperLU_DIST [15, Figure 7]; (right) two algorithms in GPTune used to tune NIMROD for ITER two-fluid flow simulation [25, Figure 6]

8 Conclusion

This article discusses the motivation for and many benefits of an ecosystem of high-performance numerical mathematics libraries for scientific applications, as achieved through the xSDK. Development of the xSDK has been a critical step in getting new, highly performant, and robust implementations of mathematical methods into the hands of applications scientists to use in their simulations. These implementations are the products of decades of numerical methods research. However, as noted throughout this paper, there are significant challenges in ensuring these implementations work well together. The xSDK and the work discussed in this paper on policies, interoperabilities, testing, releases, deploying to applications, and performance tuning are critical to ensuring new numerical methods are highly performant on today's computing platforms, performing as an ecosystem of numerical libraries. With the conclusion of the DOE Exascale Computing Project in December 2023, work on the stewardship of numerical libraries and other ECP software technologies is transitioning to the newly established Consortium for the Advancement of Scientific Software (CASS) [7]. To address the continuing challenges of heterogeneous computer architectures at all scales of computing and the increasing complexities of application codes, we see a growing need for even broader collaboration on numerical software ecosystems [4] throughout the international community.

Acknowledgements This work has been supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC (LLNL-JRNL-864981), by Lawrence Berkeley National Laboratory under contract No. DE-AC02-05CH11231.

Research was sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525. This written work is co-authored by an employee of NTESS. The employee, not NTESS, owns the right, title and interest in and to the written work and is responsible for its contents. Any subjective views or opinions that might be expressed in the written work do not necessarily represent the views of the U.S. Government. The publisher acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this written work or allow others to do so, for U.S. Government purposes. The DOE will provide public access to results of federally sponsored research in accordance with the DOE Public Access Plan.

We would like to thank the users of xSDK libraries for motivating advances in library functionality and the developers of all the xSDK libraries for engaging in collaboration on community policies and other software advances. We also thank the reviewers for their helpful comments on how to improve the paper.

References

1. Alexander, F., et al.: Exascale applications: skin in the game. Philos. Trans. R. Soc. A Math. Phys. Eng. Sci. **378**(2166), 20190056 (2020). https://doi.org/10.1098/rsta.2019.0056
2. Andrej, J., Atallah, N., Dobrev, V., Kolev, T., Pazner, W., Tomov, V.: High-performance finite elements with MFEM. Int. J. High Perform. Comput. Appl. (2024)
3. Anzt, H., Huebl, A., Li, X.S.: Then and now: improving software portability, productivity, and 100× performance. IEEE Comput. Sci. Eng. **25**(6) (2023). https://doi.org/10.1109/MCSE.2024.3387302
4. Bader, M., Ltaief, H., McInnes, L., Yokota, R.: How we can all play the high-performance game: From laptops to supercomputers. SIAM News (2023)
5. Balos, C.J., Day, M., Esclapez, L., Felden, A.M., Gardner, D.J., Hassanaly, M., Reynolds, D.R., Rood, J., Sexton, J.M., Wimer, N.T., Woodward, C.S.: SUNDIALS time integrators for exascale applications with many independent ODE systems. Int. J. High Perform. Comput. Appl. **39**(1) (2025). https://doi.org/10.1177/10943420241280060
6. Bartlett, R., Demeshko, I., Gamblin, T., Heroux, M., Johnson, J., Klinvex, A., Li, X., McInnes, L., Moulton, J., Osei-Kuffuor, D., Sarich, J., Smith, B., Willenbring, J., Yang, U.: xSDK foundations: towrd an extreme-scale scientific development kit. Supercomput. Front. Innov. **4**, 69–82 (2017)
7. Consortium for the Advancement of Scientific Software (CASS) (n.d.). https://cass.community
8. DOE Exascale Computing Project (ECP) (n.d.). https://www.exascaleproject.org
9. Esclapez, L., Day, M., Bell, J., Felden, A., Gilet, C., Grout, R., de Frahan, M.H., Motheau, E., Nonaka, A., Owen, L., Perry, B., Rood, J., Wimer, N., Zhang, W.: PeleLMeX: An AMR low mach number reactive flow simulation code without level sub-cycling. J. Open Source Softw. **8**(90), 5450 (2023). https://doi.org/10.21105/joss.05450
10. Gardner, D.J., LoDestro, L.L., Woodward, C.S.: Towards the use of Anderson acceleration in fusion simulations. In: This volume (2024)
11. Henderson, M.E., Anderson, C.R., Lyons, S.L.: Object Oriented Methods for Interoperable Scientific and Engineering Computing: Proceedings of the 1998 SIAM Workshop, vol. 99, SIAM (1999)
12. Heroux, M., Willenbring, J., Shende, S., Coti, C., Spear, W., Peyralans, J., Skutnik, J., Keever, E.: E4S: extreme-scale scientific software stack. In: 2020 Collegeville Workshop on Scientific Software Whitepapers (2020)

13. Kolev, T., Fischer, P., Min, M., Dongarra, J., Brown, J., Dobrev, V., Warburton, T., Tomov, S., Shephard, M.S., et al.: Efficient exascale discretizations: high-order finite element methods. Int. J. High Perform. Comput. Appl. 527–552 (2021). https://doi.org/10.1177/10943420211020803
14. Kothe, D., Lee, S., Qualters, I.: Exascale computing in the United States. Comput. Sci. Eng. **21**(1), 17–29 (2019). https://doi.org/10.1109/MCSE.2018.2875366
15. Liu, Y., Sid-Lakhdar, W., Marques, O., Zhu, X., Meng, C., Demmel, J., Li, X.: GPTune: multitask learning for autotuning exascale applications. In: PPoPP '21: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 234–246, San Francisco, USA (2021). https://doi.org/10.1145/3437801.3441621
16. Luo, H., Cho, Y., Demmel, J.W., Li, X.S., Liu, Y.: Hybrid parameter search and dynamic model selection for mixed-variable Bayesian optimization. Comput. Graph. Stat. (2024). https://doi.org/10.1080/10618600.2024.2308216
17. McInnes, L.C., Heroux, M.A., Bernholdt, D.E., Dubey, A., Gonsiorowski, E., Gupta, R., Marques, O., Moulton, J.D., Nam, H.A., Norris, B., Raybourn, E., Willenbring et al., J.: A cast of thousands: How the IDEAS productivity project has advanced software productivity and sustainability. IEEE Comput. Sci. Eng. **25**(6) (2023). https://doi.org/10.1109/MCSE.2024.3383799
18. McInnes, L.C., Heroux, M.A., Draeger, E.W., Siegel, A., Coghlan, S., Antypas, K.: How community software ecosystems can unlock the potential of exascale computing. Nat. Comput. Sci. **1**, 92–94 (2021). https://doi.org/10.1038/s43588-021-00033-y
19. Molins, S., Svyatsky, D., Xu, Z., Coon, E., Moulton, J.: A multicomponent reactive transport model for integrated surface-subsurface hydrology problems. Water Resour. Res. **58** (2022). https://doi.org/10.1029/2022WR032074
20. Vargas, A., Stitt, T., Weiss, K., Tomov, V., Camier, J.S., Kolev, T., Rieben, R.: Matrix-free approaches for GPU acceleration of a high-order finite element hydrodynamics application using MFEM, Umpire, and RAJA. Int. J. High Perform. Comput. Appl. **36**(4), 492–509 (2022)
21. Willenbring, J.M., Shende, S.S., Gamblin, T.: Providing a flexible and comprehensive software stack via Spack, E4S, and SDKs. IEEE Comput. Sci. Eng. **25**(6) (2023). https://doi.org/10.1109/MCSE.2024.3395016
22. xSDK: Extreme-scale Scientific Software Development Kit (n.d.). https://xsdk.info/
23. Xu, Z., Molins, S., Dwivedi, I., Svyatsky, D., Moulton, J., Steefel, C.: Understanding the hydrogeochemical response of a mountainous watershed using integrated surface-subsurface flow and reactive transport modelingWater Resour. Res. **58** (2022). https://doi.org/10.1029/2022WR032075
24. Yang, U., McInnes, L.: xSDK: building an ecosystem of highly efficient math libraries for exascale. SIAM News **54**, 8–10 (2021)
25. Zhu, X., Liu, Y., Ghysels, P., Bindel, D., Li, X.: GPTuneBand: multi-task and multi-fidelity autotuning for large-scale high performance computing applications. In: SIAM PP22: Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing (2022)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Application Impact of Exascale-Ready Direct Linear Solvers SuperLU_DIST and STRUMPACK



Pieter Ghysels and Xiaoye S. Li

Abstract Many high fidelity simulations and data analysis involve large-scale multiphysics and multiscale modeling problems that generate highly ill-conditioned and indefinite algebraic equations. Factorization based algorithms are indispensable building blocks in the solver stack for these numerically challenging problems. We will highlight how factorizations and low-rank approximate factorizations can be effectively used as direct solvers or as preconditioners for iterative solvers. This chapter discusses recent advances in SuperLU_DIST and STRUMPACK targeting exascale machines and exascale applications.

Keywords Sparse direct solver · Sparse preconditioner · Low-rank structure · Supernodal sparse LU · Multifrontal sparse LU

1 Introduction

The US Exascale Computing Project (ECP) [13] funded two general-purpose mathematical libraries for factorization based sparse solvers and preconditioners. They encompass the two widely used algorithm variants: supernodal (SuperLU) and multifrontal (STRUMPACK).

SuperLU is a library of a sparse direct solver using a DAG based supernodal approach. It can be used for any sparse linear system and is particularly efficient for non-symmetric sparsity. In addition to sparse LU and triangular solves, the library provides both normwise and componentwise error bounds, condition estimation, and an easy-to-use interface for repeated solutions with the same sparsity pattern

X. S. Li (✉)
Lawrence Berkeley Laboratory, Berkeley, CA, USA
e-mail: xsli@lbl.gov

P. Ghysels
Advanced Micro Devices, Santa Clara, CA, USA
e-mail: pieter.ghysels@amd.com

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_2

and/or similar numerical values. It is parallelized with MPI, OpenMP, and three types of GPUs: AMD, Intel, NVIDIA. This chapter only discusses SuperLU_DIST, the distributed memory version.

STRUMPACK is a library of direct solvers and preconditioners for sparse and dense rank-structured linear systems. The dense solvers and sparse preconditioners rely on hierarchically low-rank matrix representations, achieving near linear complexity; the classical dense direct solvers scale cubically. The sparse solver uses a tree based multifrontal approach. The dense frontal matrices can be compressed with the above hierarchical matrix techniques, resulting in strong preconditioners for iterative solvers to handle highly ill-conditioned linear systems.

Both libraries are purely algebraic and broadly usable in many application domains. We address several exascale challenges, with the following focused areas: (1) Develop novel approximation algorithms that have lower arithmetic and communication complexity with respect to the size of the input matrix; (2) Develop new parallelization strategies that reduce inter-process communication and expose task parallelism and vectorization for irregular computations involving sparse data structures to better use on-node resources; (3) Integrate our software into higher-level algebraic solvers such as hypre, PETSc & Trilinos, and collaborate with the ECP teams for application-specific and hardware-specific tuning of the parameters to achieve optimal parallel efficiency.

Our solver technology is essential for ECP, because many codes expected to run on the exascale machines need to solve sparse algebraic systems, and many high-fidelity simulations involve large-scale multiphysics and multiscale modeling problems that generate highly ill-conditioned and indefinite algebraic equations, for which pure iterative methods cannot converge to the solutions. The factorization-based algorithms being developed herein represent an important class of methods that are indispensable building blocks for solving those numerically challenging problems. Our software can often be used as a reliable standalone solver, or as a preconditioner for Krylov methods, or as a coarse grid solver in multigrid methods.

Key Challenges

At exascale we need to address several major challenges, such as decreasing amount of memory per core, increasing impact of communication cost and load imbalance, and increasing architectural heterogeneity. Our new design of algorithms and codes must focus not only on floating point throughput, but more on reducing communication and synchronization and task scheduling. In sparse factorization methods, we expect new bottlenecks in parts of the code that previously received little attention. For example, the preprocessing step involves numerical pivoting for selecting stable pivots and symbolic factorization, which do not yet parallelize well on manycore architectures with fine-grained parallelism. At exascale, direct solvers are more likely to be used in a preconditioning strategy, for example, in block Jacobi preconditioning, in domain decomposition methods or as coarse-grid solvers in algebraic multigrid, which requires repeated triangular solves. The challenge here is to mitigate the low arithmetic intensity and high degree of data dependency.

Compared to iterative methods, the primary bottleneck of direct solvers is the asymptotically higher growth in memory need and floating point operations, especially for problems from three-dimensional geometry. It is imperative to develop new factorization methods that require much less memory and data movement.

Solution Strategy

We address these challenges in several thrust areas. New techniques have been implemented in both SuperLU_DIST and STRUMPACK. In both cases, we spent a great deal of effort developing various algorithm components for the GPU architectures, since more than 90% of the computing power of both DOE's exascale machines, Frontier and Aurora, come from the GPUs. Our codes are portable to the three GPU environments: NVIDIA's CUDA, AMD's HIP/ROCm and Intel's SYCL/DPC++.

The improvements for SuperLU_DIST are mainly in three areas: (1) develop the communication-avoiding 3D factorization and triangular solve algorithms and codes that have provably lower communication complexity; (2) develop a synchronization-avoiding triangular solve code to enable more overlap of communications of different processes at different substitution steps; (3) develop new multi-GPU codes for both symbolic preprocessing step and numerical factorization and solve steps.

In addition to exploiting structural sparsity as SuperLU does, STRUMPACK also exploits data sparseness in the dense blocks of sparse factors using low-rank representations, which leads to linear scaling $O(n)$ or $O(n \log n)$ memory and arithmetic complexity for PDEs with smooth kernels. The developments for STRUMPACK are primarily in three areas: (1) development of an advanced preconditioner based on approximate multifrontal LU factorization, combining small dense fronts, medium sized Block Low-Rank (BLR) fronts (reducing the complexity pre-factor) and large fronts compressed using the Hierarchically Off-Diagonal Butterfly (HODBF) representation (leading to nearly linear scaling for large high-frequency wave equations); (2) porting the BLR algorithms, including several variants like left and right-looking, low-rank update and accumulation, to GPUs, which are also included in the sparse preconditioner; and (3) develop sparse direct solvers and preconditioners that are fully resident on the GPU, including the preprocessing steps.

The rest of the chapter is organized as follows. Section 2 summarizes the algorithm advances made in SuperLU_DIST. Section 3 summarizes the algorithm advances made in STRUMPACK. Section 4 highlights the impact of the solvers' new features on the performance improvement of several large-scale scientific and machine learning applications. Section 5 concludes this chapter with an outlook of future directions.

2 Algorithm Advances in SuperLU_DIST

In this section we summarize several major algorithm innovations since the inception of the ECP project.

2.1 Communication-Avoiding 3D Parallelization Strategy

On any large scale machines with thousands of nodes, the strong-scalability of many linear algebra algorithms is fundamentally limited by the cost of inter-node communication. The problem is particularly acute in sparse matrix algorithms, whereby the arithmetic intensity is asymptotically lower than the dense counterparts. That is, the ratio of arithmetic operations to data access is much smaller. As an example in an earlier version of SuperLU_DIST, for a large fixed-size model problem derived from 5-point 2D finite difference discretization, the sparse LU speeds up by only 2× when increasing the process count by 32× (from 24 to 768 processes).

The SuperLU team developed the first-of-its-kind sparse LU and sparse triangular solution algorithms with asymptotically lower communication than the previous state-of-the-art algorithms. The communication-avoiding ideas trade off extra memory for reduced communication, from well-studied cases in dense linear algebra to the difficult case of direct methods for sparse linear algebra, thereby significantly extending the regime of strong-scalability for such methods. The novelty lies in the new 3D process grid organization for mapping the elimination tree to the processes. Each 2D slice of the 3D grid is assigned a distinct subtree, and the third dimension accumulates the partial Schur complement updates in parallel from the 2D process slices. In theory, for a planar graph with n vertices, the 3D sparse LU reduces communication volume asymptotically by a factor of $O\left(\sqrt{\log n}\right)$ and latency by a factor of $O(\log n)$. The 3D sparse triangular solve reduces communication volume by $O\left(n^{\frac{1}{4}}\right)$. For non-planar cases, the 3D sparse LU can reduce the per-process communication volume by 3× and latency by $O\left(n^{\frac{1}{3}}\right)$ times. The 3D sparse triangular solve reduces communication volume by $O\left(n^{\frac{1}{6}}\right)$. In practice, the 3D sparse LU achieves speedups up to 27× for planar graphs and up to 3.3× for non-planar graphs over the baseline 2D algorithms when run on 24,000 cores of a Cray XC30 [39]. The 3D sparse triangular solve achieved up to 7.2× speedup on 12,000 cores [40]. For the same model problem as in the previous paragraph, the 3D sparse LU achieves a 23× speedup instead of just 2×.

2.2 Synchronization Reduction in Sparse Triangular Solve

The sparse triangular solve (SpTRSV) is an indispensable task in building sparse solvers and preconditioners, as well as in numerical simulations and machine learning. SpTRSV is notoriously hard to parallelize, since its arithmetic intensity is very low, of $O(1)$; it is all about communication cost. The new 3D algorithm framework in Sect. 2.1 reduced communication via some memory replication along the 3rd process dimension. However, the scalability bottleneck remains in the 2D process slices while performing partial SpTRSV corresponding to independent subtrees. The communication pattern can be succinctly modeled as a directed acyclic graph (DAG). For

such irregular communication patterns, the commonly used send-receive two-sided MPI communication paradigm misses the opportunity of overlapping communications occurring on different DAG edges, due to excessive synchronizations among the MPI tasks. To speed up the communication along the DAG, the SuperLU team restructured the algorithms to enable the use of one-sided communication. One-sided communication paradigms enjoy higher effective network bandwidth and lower synchronization costs compared to the two-sided method. We leverage the foMPI library, which is a fast, MPI-3.0 RDMA library interface on the Cray systems. When tested on 256 to 4096 MPI processes, the new one-sided code achieved between 1.5× to 2.5× reduced communication time for a range of sparse matrices [12].

This restructuring of communication patterns paves the way for the multi-GPU SpTRSV using one-sided GPU communication layers such as NVSHMEM on NVIDIA or ROCSHMEM on AMD, see Sect. 2.3.

2.3 Multi-GPU Sparse LU and Triangular Solve

The SuperLU team was the first to attempt sparse LU factorization on the GPUs. In order to use multiple GPUs, our strategy is to let each MPI process offload certain amount of work to one GPU that is connected to the CPU of the MPI process. Our earlier work started with a simple scheme whereby only BLAS-3 GEMM (matrix multiplication) is offloaded to GPU [38] at each elimination step. Even though we developed a sophisticated algorithm to overlap the GPU GEMM operation with the data transfer via the PCIe bus between CPU and GPU, the speedup over the CPU-only code is limited to a factor of 2–3×. The team continued to add new GPU algorithms for the Schur complement update [37] and the supernodal factorization [40], which led to almost an order of magnitude speedup compared to the CPU-only factorization.

Our earlier attempt to develop GPU-capable SpTRSV uses the aforementioned MPI driven GPU offload mode. Due to excessive communication in SpTRSV, the initial multi-GPU code hardly achieved any speedup using any numbers of GPUs. Taking the lessons learned in the one-sided MPI paradigm Sect. 2.2, we leveraged the advantage of GPU-initiated data transfers of NVSHMEM to implement and evaluate a multi-GPU SpTRSV algorithm. We created a novel producer-consumer paradigm to manage the computation and communication and implemented it using two CUDA streams. This multi-GPU SpTRSV implementation achieved a 3.7× speedup when using twelve GPUs (two nodes) relative to a single GPU, and up to 6.1× faster than NVIDIA's `cusparse_csrsv2()` over the range of one to eighteen GPUs [11].

2.4 Mixed-Precision Algorithms

In recent years, a new hardware trend, especially for accelerators like GPUs, is to employ low-precision special-function units tailoring to the demands of AI work-

loads. The lower-precision floating point arithmetic can be done several times faster than their higher-precision counterparts. For direct linear solvers, a well-understood method is to use lower precision to perform factorization (expensive) and higher precision to perform iterative refinement (IR) to recover accuracy (cheap) [4, 9, 10]. For a typical sparse matrix resulting from the 3D finite difference discretization of a regular mesh, sparse LU needs $O\left(n^2\right)$ flops while each IR step needs only $O\left(n^{4/3}\right)$ flops (including SpTRSV and sparse matrix-vector multiplication). Therefore, as long as the number of IR steps is small, we should see faster overall time.

In SuperLU_DIST, we provide two mixed-precision driver routines: `psgssvx_d2` and `pdgssvx_d2`. In the first case, the user input/output are in IEEE single precision. In the second case, the input/output are in IEEE double precision. In both cases, the internal LU factorization is performed in single precision, while the IR is performed in double precision. In addition, the routines also return both backward error and forward error bound, both normwise and componentwise. On 8 nodes of the Summit machine at ORNL, using 6 CPUs and 6 GPUs per node, `psgssvx_d2` achieved up to 1.65× speedup over the pure double precision code [21].

2.5 Batched Sparse Direct Solver

Over the course of interactions with applications team in the ECP project, the need for batched sparse linear algebra (LA) functions has emerged in order to make more efficient use of the GPUs for many small and sparse linear algebra problems. These systems arise from combustion modeling, bio-chemical modeling, fusion plasma modeling, and nuclear physics simulations, to name a few. The dimension of each matrix is of order 1000 s to 10000 s. The SuperLU team led the development of the batched sparse direct solver interface and the prototype implementation in the SuperLU_DIST code base. A salient feature of the new interface is to allow heterogeneous batch structure in that each linear system can have different size, sparsity pattern and numerical property. Our interface allows each system in the batch to have different preprocessing: equilibration, row and column reordering, depending on the numrical values and sparsity pattern of each individual system. Our initial results with a couple of combustion and fusion applications demonstrate that our new batched sparse direct solver can be order of magnitude faster than the batched banded solver or the batched dense solver [3].

3 Algorithm Advances in STRUMPACK

STRUMPACK version 1.0.0 was released in May 2016 – not long before the inception on the ECP project – providing a multi-threaded and distributed memory sparse direct (multifrontal [14]) solver and preconditioner with hierarchically semi-separable

matrix approximation. The multifrontal variant of sparse LU relies on a (binary) tree data-structure called the supernodal or multifrontal tree, where each node τ of the tree stores a dense frontal matrix $F_\tau = [F_{11}F_{12}; F_{21}F_{22}]$. The lower/upper triangular parts of the F_τ contain the blocks making up the sparse triangular factors L and U. During numerical factorization, the tree is traversed from the leaves to the root, performing LU factorization of F_{11} and computing the (temporary) Schur complement in F_{22}, which is then passed on to the next level in the tree using a sparse scatter operation.

3.1 Multi-GPU Sparse Multifrontal Solver

In STRUMPACK, the multifrontal tree is distributed over multiple MPI ranks. The root front is partitioned using a 2D block cyclic layout (as used by ScaLAPACK) on a 2D process grid defined by the user-specified MPI communicator. This communicator is then split proportionally to the estimated amount of work (or memory usage) of the two subtrees.

In the original CPU-only implementation the multifrontal tree is traversed recursively using dynamically scheduled OpenMP tasks. Unfortunately, this does not map well to modern GPU architectures. Hence, the GPU implementation traverses the local tree level-by-level. Note that different fronts on the same level can be handled concurrently. We implemented this level-by-level traversal first using unoptimized batched CUDA kernels [17] for the smallest fronts ($F_{11} \leq 32 \times 32$) and using cuBLAS/cuSOLVER for the larger fronts. Later, we worked closely with the MAGMA [42] team on a more general variable sized batched dense LU factorization [1]. This allows for very high performance on the GPU for the larger fronts while eliminating the kernel launch overhead by batching many small matrix operations. Like the rest of STRUMPACK, MAGMA supports CUDA (NVIDIA), ROCm (AMD) and SYCL (Intel). We also developed batched GPU kernels in CUDA, HIP and SYCL for the sparse scatter (extend-add) operation to pass the Schur complements from one level to the next. Since the pattern of the factors can be known before the numerical factorization starts, we allocate sufficient GPU memory with one single allocation for the factors and one allocation for the temporary work memory. However, if the factors do not fit on the device they are copied back to host memory asynchronously and concurrently with the Schur updates. For the distributed fronts, we replace ScaLAPACK with the GPU-capable SLATE [15], also using a 2D block cyclic layout. However, when using SLATE on the GPU, we select a much larger block size to achieve better GPU performance. Overall we have observed up to 16× speedup with the GPU code compared to the CPU only solver.

3.2 Approximate Factorization Preconditioners

While the numerical robustness and the black-box nature are the main benefits of sparse direct solvers, the drawbacks are the asymptotic scaling with problem size. For

instance, for a 7 point stencil on a regular 3D $N = k^3$ mesh, the number of operations for the factorization scales as $O\left(N^2\right)$ and the memory usage as $O\left(N^{3/2}\right)$. In order to reduce this complexity, while at the same time keeping as much as possible the benefits of sparse direct solvers, we apply rank-structured matrix compression to the fronts. The matrix representations used in STRUMPACK are special cases of the general hierarchical or $\mathcal{H}$-matrix format introduced by [20], as illustrated in Fig. 1.

From the first release STRUMPACK provided support for the hierarchically semi-separable format, which is based on a 1D recursive partitioning of the degrees of freedom. This leads to a format where all off-diagonal blocks are approximated as low rank, and the factors are further compressed using a nested decomposition over the multiple levels. The diagonal blocks correspond to self interactions and are stored as dense blocks. There exists theory [43] that motivates the use of this type of approximation for frontal matrices arising from different PDE problems, leading to a sparse direct solver of near-linear complexity. For HSS construction STRUMPACK relies on randomized sketching [28], which does not require first explicitly forming the front. The HSS-based factorization uses the ULV decomposition [43], where U and V are orthogonal matrices and L is a lower triangular matrix. STRUMPACK's HSS

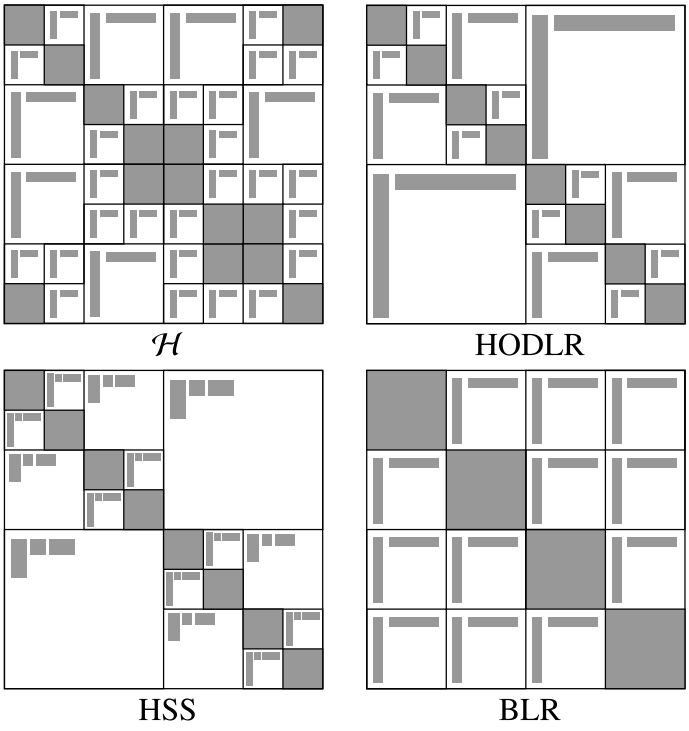


Fig. 1 Various rank-structured matrix formats. STRUMPACK natively supports HSS (Hierarchically Semi-Separable) and BLR (Block Low Rank), with GPU support, as well as HODLR (Hierarchically Off-Diagonal Low Rank). The TikZ package was used to draw these figures

algorithms are described in detail in [35], the shared memory HSS-enabled sparse solver in [16]. In [18] this solver is compared to a number of other preconditioners. We developed a more robust adaptive rank determination scheme for randomized HSS construction in [19].

To overcome the main bottleneck of the HSS format – the potentially large off-diagonal ranks especially for high-dimensional problems and discretization of high-frequency PDEs – we added support in STRUMPACK for hierarchically off-diagonal butterfly (HODBF) [26]. HODBF is like HODLR (Fig. 1), but with low rank replaced with the more general butterfly decomposition [30], as implemented in ButterflyPACK [24]. The butterfly decomposition is a multilevel matrix decomposition algorithm well-suited for representing highly oscillatory operators. The HODBF scheme proved very effective in compressing large fronts from highly oscillatory problems. However, due to the complexity, GPU acceleration is extremely challenging.

We added support in STRUMPACK for yet another compression scheme, called block low rank (BLR) [8] (as also used in the MUMPS sparse solver [29]). BLR is easier to implement and off-load to the GPU, mainly because it is not a hierarchical format. Unlike HSS and HODLR/HODBF, BLR supports strong admissibility (not all off-diagonal blocks are compressed), which makes BLR a very practical format for a wide range of problems despite the theoretically worse complexity. STRUMPACK has support for several variants of BLR compression/factorization, including left and right-looking, and with compression before or after all Schur updates are applied, see [8]. We recently added multi-GPU BLR support to STRUMPACK [7]. For a 3D high-frequency visco-acoustic wave propagation problem on a 250^3 mesh, on 32 nodes of Perlmutter, the factorization on GPU (128 NVIDIA A100s) is 5.1× faster than the CPU-only code (2048 cores), while the BLR-enabled sparse solver is 10.6× faster than the CPU-only sparse direct solver with the sparse factors taking only 36% of the memory of the exact solver and GMRES converging in 6 iterations.

Finally, in [8], multiple compression schemes are combined: HODBF compression for the largest fronts for it's low complexity, BLR for medium sized fronts, and the smaller fronts are stored using a lossy compression scheme from the ZFP [23] or SZ3 [22] libraries. The lossy compression does not reduce the operation count, but does reduce the peak memory usage.

3.3 Rank-Structured Matrix API

The rank-structured matrix formats used in the sparse preconditioners can also be used directly for a variety of applications leading to large scale dense matrix problems. STRUMPACK provides an API for these various formats (HSS, HODLR, HODBF, BLR, LR, lossy, lossless), for the construction, matrix-vector product, factorization and other routines. For construction, there are multiple scenarios, depending on the use case. The user can either provide (1) a dense matrix, (2) a matrix vector product, (3) an element evaluation routine, (4) both (2) and (3), or (5) point

coordinates and a kernel function. Note that not every format supports all types of construction algorithms or operations. STRUMPACK also provides several clustering algorithms [33] to build the matrix hierarchy. See Sect. 4.4 for an example use case for kernel ridge regression, where we were able to reduce the algorithmic complexity from the traditional cubic scaling to linear scaling.

4 Impact on Applications

In this section, we discuss several large scale scientific and machine learning applications that benefited from our solver advancement.

4.1 Stability Study of Plasma Fusion Reactors

The study of fusion physics applications is important in discovering how to harness fusion, creating clean energy using nearly inexhaustible hydrogen as the fuel. In particular, the largest fusion reactor called ITER is being constructed in France based on the tokamak concept, burning plasmas at a very high temperature. In order to optimize the operation and output of ITER, fusion scientists have created the multiphysis modeling codes M3D-C1 and NIMROD codes to investigate macroscopic dynamics of magnetized plasmas in confinement. The instabilities may lead to topological changes in the magnetic field which may damage the reactor. Both codes solve the extended-magnetohydrodynamic (MHD) equations, which is a model that describes plasma as electrically conducting fluids of ions and electrons.

Tokamaks are 3D torus-shaped vessels filled with plasmas, and the two modeling codes use finite element or spectral element discretization of the PDEs at the poloidal planes and along the toroidal direction, leading to large sparse linear systems of dimension tens to hundreds of millions. The global matrix has a block diagonal structure (poloidal planes), plus the off-diagonal entries connecting the planes in the toroidal direction. The only successful solver combination for both codes is to use SuperLU_DIST to factorize the diagonal blocks, which is used as a block diagonal preconditioner for the GMRES iterative solver for the entire linear system.

In this usage scenario, the most time-consuming computation is in the parallel sparse triangular solve (SpTRSV) needed in each GMRES iteration – usually 50 to 100 SpTRSVs are needed for one LU factorization. The SpTRSV time can be 40–50% of the entire simulation time. Our recent work to improve SpTRSV is particularly applicable to this application. For example, for a poloidal plane matrix of dimension about 200k, our one-sided communication algorithm discussed in Sect. 2.2 is over 1.5× faster using 256 to 1024 MPI processes (see matrix A30 in [12]). For another poloidal matrix of dimension 253k (see matix s1_mat_0_253872 in [25]), our combined 3D communication-avoiding and GPU algorithm discussed in Sect. 2.1 and

Sect. 2.3 is 6.5× faster than the earlier code using 16 A100 GPUs on Pelmutter at NERSC.

4.2 Radio-Frequency Power in Fusion Plasmas

STRUMPACK is used in the Center for Integrated Simulation of Fusion Relevant RF Actuators (RF-SciDAC), a DOE funded SciDAC partnership project[1]. The capability to reliably couple megawatts of radio-frequency (RF) power in the ion-cyclotron, lower-hybrid, and electron-cyclotron ranges of frequencies to magnetically confined fusion plasmas for the purposes of heating and current drive is essential for steady state operation of future fusion reactors. STRUMPACK is used within a state-of-the-art field solver capable of simulating wave fields produced by RF actuators in a full 3D torus, while accurately retaining geometric details of the engineering CAD models.

STRUMPACK is used for the efficient solution of the linear operator resulting from a finite element discretization of Maxwell's equations modeling the plasma dielectric response. The field solvers Petra-M [41] and Stix [31] are based on the finite element library MFEM [2], which has a direct interface to STRUMPACK.

Table 1 shows results from an indefinite Maxwell simulation on a model of the National Spherical Torus Experiment-Upgrade (NSTX-U), the primary fusion experiment at PPPL, using Petra-M, on top of MFEM, using STRUMPACK to solve the resulting linear system. The largest problem, a 4th order Nédélec finite element discretization with a total of 46M DoFs, is solved using 1024 NVIDIA A100 GPUs on 256 nodes of the Perlmutter system at NERSC.

Table 1 NSTX-U 180°C antenna phasing

	Problem 1	Problem 2
FEM element order	3	4
DoFs	21,469,920	46,371,976
Nonzeros	1,347,479,136	5,223,641,048
Perlmutter nodes	32	256
A100 GPUs	128	1024
ParMETIS time (s)	10.4	38.7
Symbolic analysis (s)	8.5	2.6
Factor time (s)	84.8	101.1
Factor flop rate (TFlop/s)	93.8	433.8
Factor memory (GB)	1034.8	3806.0
Solve time (s)	22.3	40.7

[1] https://www.rfscidac4.org/.

4.3 Singularly Perturbed PDEs

We consider the PDE $-\delta^2 \Delta u + u = f$, on $\Omega = (0, 1)^3$ with $u(\partial\Omega) = g$, where g and f are given functions and δ is a small and positive perturbation parameter. This is a three dimensional singularly perturbed reaction-diffusion PDE, which is found in fluid dynamics, computational chemistry, and biological applications [34]. This problem has proven [27] to be challenging for sparse direct solvers due to the occurrence of many subnormal numbers in the factors, for which floating point operations are expensive. Figure 2 (from [8]) shows the results for solving this singularly perturbed PDE with $\delta = 10^{-4}$ on a regular mesh from 64^3 to 512^3, with a 7-point finite difference scheme. We compare different composite compression schemes, as discussed in Sect. 3.2, including BLR, ZFP (lossy compression), HODBF-BLR as well as HODBF-BLR-ZFP, with the compression tolerance set to $\epsilon = 10^{-2}$, and ZFP (in fixed precision mode) using 16 bitplanes. Note that we now recommend, and by default use the ZFP fixed accuracy mode instead. Figure 2 shows that when adding HODBF compression to the sparse solver, we achieve $O\left(N \log^2 N\right)$ computational scaling and $O(N)$ memory complexity. Adding lossy compression in addition to BLR and HODBF slightly reduces memory usage, but increases the solve time per iteration.

4.4 Kernel Ridge Regression Using HSS Acceleration

In [5], we developed a scalable and memory-efficient framework for kernel ridge regression. The resulting algorithm is $O(r^2 N)$ with N the number of training samples and r the low-rank structure of the underlying kernel. In contrast, kernel methods typically rely on $O(N^3)$ dense matrix factorization. The HSS construction is based on a skeletonization approach using approximate nearest neighbor information from the input data points. The approximate nearest neighbors can be found efficiently by merging nearest neighbors found in the leaves of multiple different instances of a random projection tree on the input data. Our experiments include Gaussian, Laplacian and ANOVA ($p = 2$) kernels for 7 popular datasets with up to 4.5M data points. The kernel width h and the kernel ridge regression hyperparameter λ are tuned with the off-the-shelf OpenTuner framework to minimize the classification error. The final classification error is competitive with other approaches such as FALKON [36], INV-ASKIT [6] and the exact kernel method as implemented in scikit-learn [32]. The new fast kernel algorithms are written in C++ and part of STRUMPACK, but we provide a C interface, and a Python interface on top of that, which is compatible with the scikit-learn classifiers and regressors. Figure 3a shows the relative error for the HSS compression for a given low rank compression tolerance. Figure 3b shows the achieved compression ratio for the compression tolerance. Note that typically binary classification does not require high accuracy of the kernel matrix. Figure 3c compares the runtime of the new HSS-ANN algorithm with the scikit-learn kernel ridge regression. Note the better scaling of the HSS algorithms, and the ability to scale to distributed memory and much larger problems.

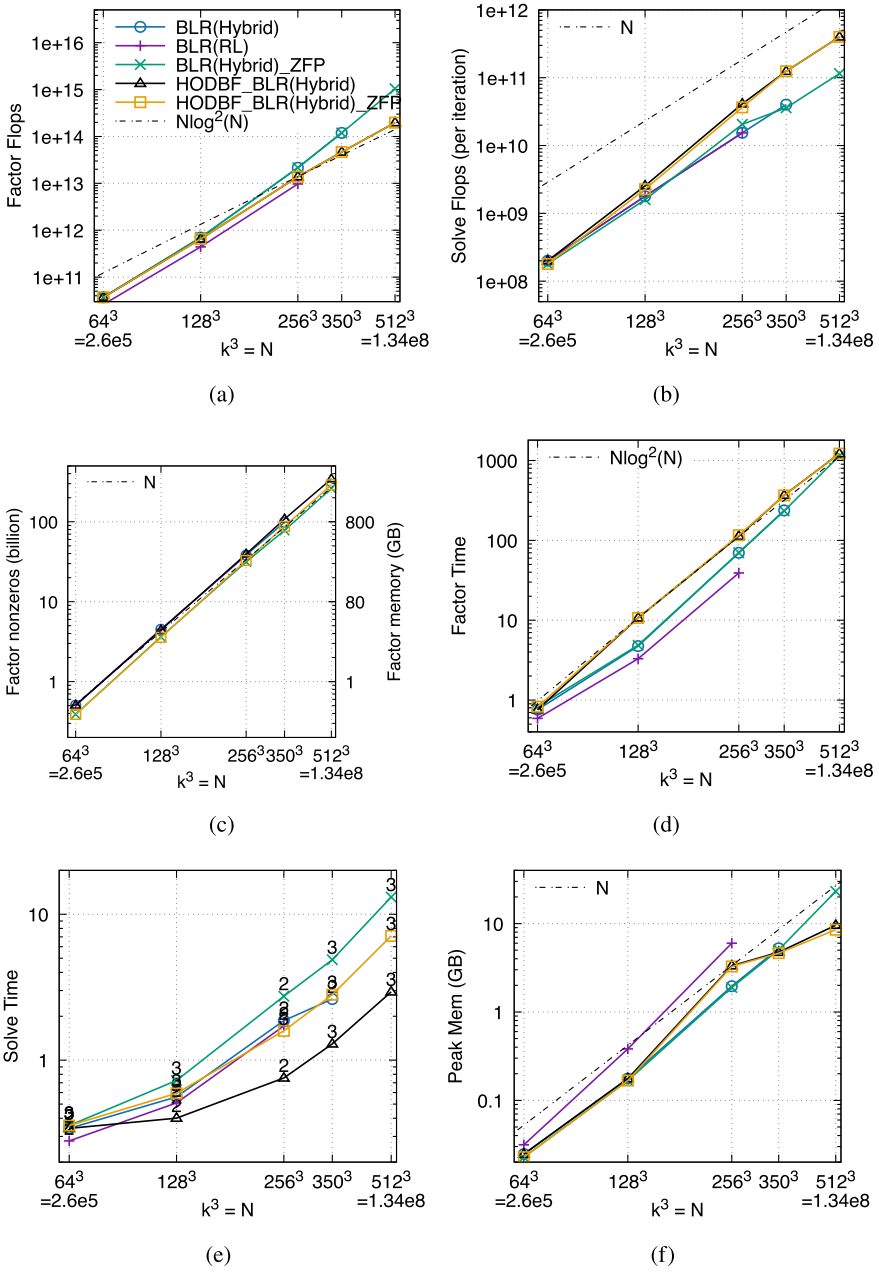


Fig. 2 Results for 3D singularly perturbed differential equations (SPDE) using the BLR, BLR-ZFP, BLR-HODBF and BLR-HODBF-ZFP (10^{-2}) multifrontal solver with GMRES. **a** Flop counts for factorization. **b** Flop counts for solve per iteration in GMRES. **c** Total memory usage for the factors. **d** CPU time for factorization. **e** CPU time for solve. **f** Peak working memory per MPI. We use 64 compute nodes, with 2 MPI ranks per node and 16 OpenMP threads per MPI process. (Reproduced from [8, Fig. 9])

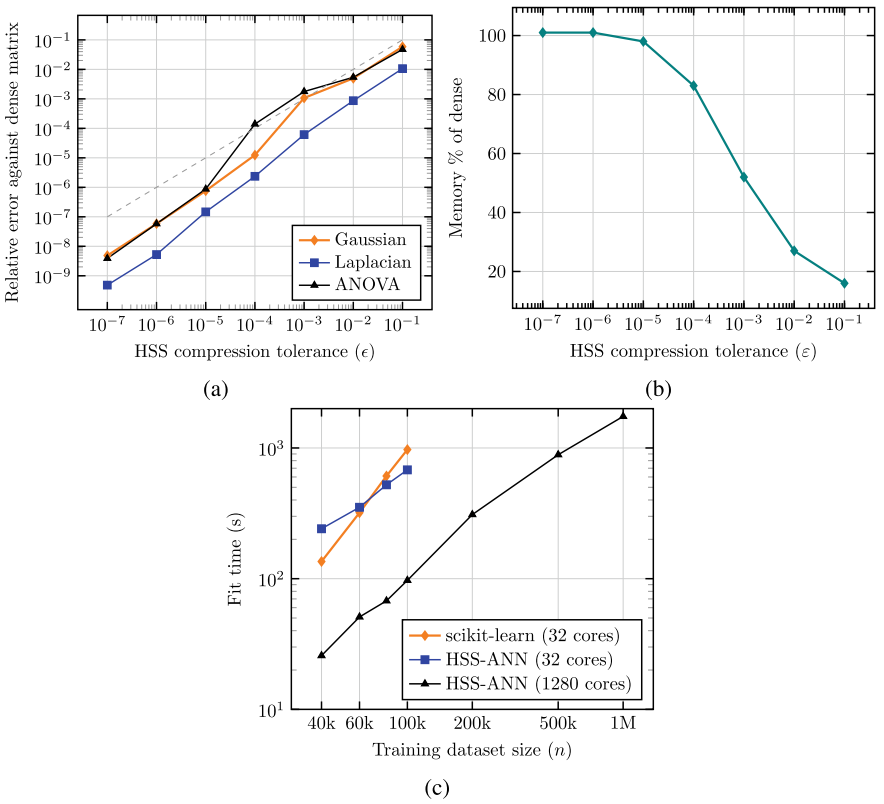


Fig. 3 **a** The relative error between the original (dense) kernel matrix and its corresponding HSS approximation is always smaller than ϵ. **b** Memory savings (in percentage) compared to the dense storage of the kernel matrix for different ϵ. **c** Time comparison between scikit-learn and HSS-ANN using the SUSY dataset. (Results taken from [5, Figs. 5, 6, 8].)

5 Conclusion

This chapter summarizes our ECP efforts and results of novel algorithm developments in SuperLU_DIST and STRUMPACK. The novel 3D communication-avoiding algorithms improved the strong scaling from hundreds of cores to tens of thousands of cores, compared to the pre-ECP 2D algorithms. We observed that the combined advances in communication reduction and GPU acceleration led to one to two orders of magnitude runtime reduction. The hierarchical low-rank compression algorithms achieved more than an order of magnitude reduction of memory footprint.

Looking to the future, we will investigate the use of more randomized algorithms for building fast preconditioners, scalable symmetric indefinite solvers for the systems arising from numerical optimization, extension to tensor computations, and efficient sparse operations needed for building the scientific ML models.

Acknowledgements This work has been supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Berkeley National Laboratory under contractNo. DE-AC02-05CH11231.

Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this chapter.

References

1. Abdelfattah, A., Ghysels, P., Boukaram, W., Tomov, S., Li, X.S., Dongarra, J.: Addressing irregular patterns of matrix computations on GPUs and their impact on applications powered by sparse direct solvers. In: SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–14. IEEE (2022). https://doi.org/10.1109/SC41404.2022.00031
2. Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.S., Cerveny, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Dahm, J., Medina, D., Zampini, S.: MFEM: a modular finite element methods library. Comput. Math. Appl. (2020)
3. Boukaram, W., Hong, Y., Liu, Y., Shi, T., Li, X.S.: Batched Sparse Direct Solver Design and Evaluation in SuperLU_DIST. Int. J. High Perform. Comput. Appl. (2024)
4. Carson, E., Higham, N.J.: Accelerating the solution of linear systems by iterative refinement in three precis ions. SIAM J. Sci. Comput. **40**(2), A817–A847 (2018)
5. Chávez, G., Liu, Y., Ghysels, P., Li, X.S., Rebrova, E.: Scalable and memory-efficient kernel ridge regression. In: 2020 IEEE International parallel and distributed processing symposium (IPDPS), pp. 956–965. IEEE (2020)
6. Chenhan, D.Y., March, W.B., Biros, G.: An n log n parallel fast direct solver for kernel matrices. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 886–896. IEEE (2017)
7. Claus, L., Ghysels, P., Boukaram, W.H., Li, X.S.: A GPU accelerated sparse direct solver and preconditioner with block low rank compression. Submitt. Int. J. High Perform. Comput. Appl. ECP Special Issue (2024)
8. Claus, L., Ghysels, P., Liu, Y., Nhan, T.A., Thirumalaisamy, R., Bhalla, A.P.S., Li, S.: Sparse approximate multifrontal factorization with composite compression methods. ACM Trans. Math. Softw. **49**(3), 1–28 (2023)
9. Demmel, J., Hida, Y., Kahan, W., Li, X., Mukherjee, S., Riedy, E.: Error bounds from extra-precise iterative refinement. ACM Trans. Math. Softw. **32**(2), 325–351 (2006)
10. Demmel, J., Hida, Y., Riedy, E., Li, X.: Extra-precise iterative refinement for overdetermined least squares problems. ACM Trans. Math. Softw. (TOMS) **35**(4), 28 (2009)
11. Ding, N., Liu, Y., Williams, S., Li, X.S.: A message-driven, multi-gpu parallel sparse triangular solver. In: Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21), pp. 147–159 (2021). https://doi.org/10.1137/1.9781611976830.14
12. Ding, N., Williams, S., Liu, Y., Li, X.S.: Leveraging one-sided communication for sparse triangular solvers. In: Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Com Puting (PP), pp. 93–105 (2020). https://doi.org/10.1137/1.9781611976137.9
13. DOE Exascale Computing Project (ECP) (n.d.). https://www.exascaleproject.org
14. Duff, I.S., Reid, J.K.: The multifrontal solution of indefinite sparse symmetric linear. ACM Trans. Math. Softw. (TOMS) **9**(3), 302–325 (1983)
15. Gates, M., Kurzak, J., Charara, A., YarKhan, A., Dongarra, J.: SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In: Proceedings of the International

Conference for High Performance Computing, Networking, Storage and Analysis, SC '19. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3295500.3356223
16. Ghysels, P., Li, X.S., Rouet, F.H., Williams, S., Napov, A.: An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling. SIAM J. Sci. Comput. **38**(5), S358–S384 (2016)
17. Ghysels, P., Synk, R.: High performance sparse multifrontal solvers on modern GPUs. Parallel Comput. **110**, 102897 (2022)
18. Ghysels, P., Xiaoye, S.L., Gorman, C., Rouet, F.H.: A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 897–906. IEEE (2017)
19. Gorman, C., Chávez, G., Ghysels, P., Mary, T., Rouet, F.H., Li, X.S.: Robust and accurate stopping criteria for adaptive randomized sampling in matrix-free hierarchically semiseparable construction. SIAM J. Sci. Comput. **41**(5), S61–S85 (2019)
20. Hackbusch, W.: A sparse matrix arithmetic based on-matrices. part I: Introduction to-matrices. Computing **62**(2), 89–108 (1999)
21. Li, X.S., Lin, P., Liu, Y. and Sao, P.: Newly released capabilities in the distributed-memory superlu sparse direct solver. ACM Trans. Math. Softw. **49**(1) (2023). https://doi.org/10.1145/3577197
22. Liang, X., Zhao, K., Di, S., Li, S., Underwood, R., Gok, A.M., Tian, J., Deng, J., Calhoun, J.C., Tao, D., et al.: SZ3: A modular framework for composing prediction-based error-bounded lossy compressors. IEEE Trans. Big Data **9**(2), 485–498 (2022)
23. Lindstrom, P.: Fixed-rate compressed floating-point arrays. IEEE Trans. Visual Comput. Graphics **20**(12), 2674–2683 (2014)
24. Liu, Y.: Butterflypack (2018). https://portal.nersc.gov/project/sparse/butterflypack/
25. Liu, Y., Ding, N., Sao, P., Williams, S., Li, X.S.: Unified communication optimization strategies for sparse triangular solver on cpu and gpu clusters. In: Proceedings of the International Conference for High Performance Computing, Netwo rking, Storage and Analysis, SC '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3581784.3607092
26. Liu, Y., Ghysels, P., Claus, L., Li, X.S.: Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations. SIAM J. Sci. Comput. **43**(5), S367–S391 (2021)
27. MacLachlan, S., Madden, N.: Robust Solution of Singularly Perturbed Problems Using Multigrid Methods. SIAM J. Sci. Comput. **35**(5), A2225–A2254 (2013). https://doi.org/10.1137/120889770
28. Martinsson, P.G.: A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. SIAM J. Matrix Anal. Appl. **32**(4), 1251–1274 (2011)
29. Mary, T.: Block low-rank multifrontal solvers: complexity, performance, and scalability, Université Paul Sabatier-Toulouse III, Ph.D. thesis (2017)
30. Michielssen, E., Boag, A.: Multilevel evaluation of electromagnetic fields for the rapid solution of scattering problems. Microw. Opt. Technol. Lett. **7**(17), 790–795 (1994)
31. Migliore, C., Wright, J., Stowell, M., Bonoli, P.: Using the stix finite element rf code to investigate operation optimization of the icrf antenna on alcator c-mod. Nucl. Fusion **63**(10), 106006 (2023)
32. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V.: Scikit-learn: Machine learning in Python. the. J. Mach. Learn. Res. **12**, 2825–2830 (2011)
33. Rebrova, E., Chávez, G., Liu, Y., Ghysels, P., Li, X.S.: A study of clustering techniques and hierarchical matrix formats for kernel ridge regression. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 883–892. IEEE (2018)
34. Roos, H.G., Stynes, M., Tobiska, L.: Robust numerical methods for singularly perturbed differential equations, In: Springer Series in Computational Mathematics, vol. 24, 2nd edn. Springer-Verlag, Berlin (2008)

35. Rouet, F.H., Li, X.S., Ghysels, P., Napov, A.: A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. ACM Trans. Math. Softw. (TOMS) **42**(4), 1–35 (2016)
36. Rudi, A., Carratino, L., Rosasco, L.: FALKON: An optimal large scale kernel method. Adv. Neural Inf. Process. Syst. 30 (2017)
37. Sao, P., Liu, X., Vuduc, R., Li, X.: A sparse direct solver for distributed memory xeon phi-accelerated systems. In: 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS). Hyderabad, India (2015)
38. Sao, P., Vuduc, R., Li, X.: A distributed CPU-GPU sparse direct solver. In: Proceeding of Euro-Par 2014. LNCS, vol. 8632, pp. 487–498. Porto, Portugal (2014)
39. Sao, P., Vuduc, R., Li, X.: A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. J. Parallel Distrib. Comput. (2019)
40. Sao, P., Vuduc, R., Li, X.: A Communication-Avoiding 3D Sparse Triangular Solver. In: ICS 2019: International Conference on Supercomputing, Phoenix, AZ (2019)
41. Shiraiwa, S., Wright, J.C., Bonoli, P.T., Kolev, T., Stowell, M.: RF wave simulation for cold edge plasmas using the MFEM library. EPJ Web Conf. **157**, 03048 (2017)
42. Tomov, S., Dongarra, J., Baboulin, M.: Towards dense linear algebra for hybrid GPU accelerated manycore systems. Parallel Comput. **36**(5–6), 232–240 (2010). https://doi.org/10.1016/j.parco.2009.12.005
43. Xia, J.: Randomized sparse direct solvers. SIAM J. Matrix Anal. Appl. **34**(1), 197–227 (2013)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Software Development Aspects of Integrating Linear Algebra Libraries



Marcel Koch, Tobias Ribizel, Pratik Nayak, Fritz Göbel, Gregor Olenik, and Terry Cojean

Abstract Many scientific discoveries are made through, or aided by, the use of simulation software. These sophisticated software applications are not built from the ground up, instead they rely on smaller parts for specific use cases, usually from domains unfamiliar to the application scientists. The software library Ginkgo is one of these building blocks to handle sparse numerical linear algebra on different platforms. By using Ginkgo, applications are able to ease the transition to modern systems, and speed up their simulations through faster numerical linear algebra routines. This paper discusses the challenges and benefits for application software in adopting Ginkgo. It will present examples from different domains, such as CFD, power grid simulation, as well as electro-cardiophysiology. For these cases, the impact of the integrations on the application code is discussed from a software engineering standpoint, and in particular, the approaches taken by Ginkgo and the applications to enable sustainable software development are highlighted.

M. Koch (✉) · P. Nayak · F. Göbel · G. Olenik · T. Cojean
Scientific Computing Center (SCC), KIT, Karlsruhe, Germany
e-mail: marcel.koch@kit.edu

P. Nayak
e-mail: pratik.nayak@kit.edu

F. Göbel
e-mail: fritz.goebel@kit.edu

G. Olenik
e-mail: gregor.olenik@kit.edu

T. Cojean
e-mail: terry.cojean@kit.edu

T. Ribizel
Computational Mathematics, TUM, Heilbronn, Germany
e-mail: tobias.ribizel@tum.de

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_3

1 Introduction

Integrating numerical linear algebra libraries has considerable impact for application software. They provide new functionality, or optimize existing ones. Typically, application developers have little expertise in numerical linear algebra. Thus, through the integrations, the application software can benefit from functionality developed by experts in this field. The integrations come at a cost, though. They require development time, which is usually scarce for application scientists.

Both application and library developers have to ensure that the development cost stays minimal. This paper considers efforts within the numerical linear algebra library Ginkgo, and selected applications, with integrations of it, taken to minimize development and maintenance effort. This paper will first give a brief overview of the related work in Sect. 2. Then, in Sect. 3 Ginkgo's main concepts regarding integrations are presented. A corresponding view on integration from the application side is discussed next in Sect. 4. Several case studies are presented in Sect. 5. These highlight the key points from the two previous sections. Finally, a summary of the main points is provided in Sect. 6.

2 Related Work

Many software libraries provide numerical linear algebra functionality. Some of the most versatile libraries are PETSc [3] and Trilinos [11]. Both offer features for linear algebra, but they also encompass more areas, such as non-linear solver, mesh handling, parallel communication, or discretization frameworks. As such, they are attractive to a variety of applications. The PETSc user manual, for example, has more than 4000 citations on Google Scholar and Trilinos close to 1500.[1] Most of these are application papers that usually present only the results of integration of one of these libraries, and don't discuss their approach to the integration in detail. Papers on the library themselves, on the other hand, e.g. PETSc's AMG implementation [14] or an overlapping Schwarz method in Trilinos [10], often focus on new algorithmic improvements, leaving out again details on the integration aspects.

In [8, 19] the history of important numerical software libraries is presented. Dongarra's Turing lecture [8] focuses on the developments of LAPACK and adjacent packages, e. g. BLAS, ScalLAPACK, or SLATE, while [19] discusses the mathematical software library HSL. Both articles show how libraries can adapt to emerging needs of the community to stay relevant. Since the articles present the rich history of longstanding libraries, little place is left for technical details on integration. Nevertheless, Dongarra also notes the funding and personnel cost for maintaining such long-standing efforts.

The performance impacts of the integrations in Sect. 5 have already been published. Ginkgo's batched solver, which provides most of the performance benefits for

[1] From https://scholar.google.com/ accessed on 2024.05.06.

PeleLM, are described in [1, 2]. The paper [20] shows the performance of Ginkgo's direct solver in the context of the ExaSGD application. Additionally, the summary paper [6] provides an overview of the performance gains of multiple applications from the Exascale Computing Project (ECP)[2] that use Ginkgo, including PeleLM and ExaSGD. The OpenFOAM results using OGL are available in [16]. All of these papers, again, provide little consideration to the integration effort. Only the papers on OGL are an exception since they discuss the design of OGL which impacts the integration.

3 The Ginkgo Library

Ginkgo is a software library for sparse numerical linear algebra. It started as part of the ECP with the goal of providing an efficient and portable library for heterogeneous hardware. In particular, Ginkgo runs on all major GPU architectures. The respective backends are written in their native language to gain the most performance, i.e. CUDA for NVIDIA GPUs, HIP for AMD GPUs, and SYCL for Intel GPUs. On the algorithmic side, the main focus lies on iterative and direct solvers for sparse linear systems. Together with preconditioners and reorderings, Ginkgo is capable of solving complex problems efficiently on a variety of different architectures. This section introduces the core concepts of Ginkgo and discusses how they facilitate the integration of Ginkgo into application software.

3.1 Concepts

At its core, Ginkgo is a modern C++ library that values performance, flexibility, and sustainability. The aforementioned hardware specific backends offer excellent performance through optimized low-level kernels, while the clear type hierarchy offers flexibility in terms of functionality for the users. Through modern C++ patterns (requiring the standard C++14), the library provides a sustainable design. Finally, exhaustive testing and benchmarking ensure the correctness of the library.

Kernel Backends

The library is split into a core library and the different backend libraries. Performance critical kernels are implemented in the backends, and the higher level algorithmic structure is in the core library. The separation enables Ginkgo to directly target the hardware in question. This allows Ginkgo to easily implement complex algorithms on an abstract level, while gaining performance through hardware-optimized kernels. Compatibility layers, such as Kokkos [4], typically add some overhead compared to native implementations as shown in [7], which Ginkgo circumvents.

[2] https://www.exascaleproject.org/.

The separation also benefits encapsulation within Ginkgo. Any dependencies on the vendor specific languages are restricted to their respective backend and not propagated through the whole library. An application thus only needs a standard C++ compiler to compile with Ginkgo, even if the application will use for example Ginkgo's CUDA kernels.

The backend design not only offers performant kernels, it also provides flexibility in terms of the used hardware. Ginkgo can be built with all backends enabled at the same time. This makes it not a compile-time choice, which hardware should be used and which not, but a runtime choice instead. Which backend is chosen only depends on the implementation of the `Executor` interface used at runtime. The following example shows how different executors are used, depending on a runtime parameter:

```
void app_run(std::shared_ptr<gko::Executor> exec, ...);
// ...
{
  auto exec = cuda ? gko::CudaExecutor::create(...)
                   : gko::OmpExecutor::create(...);
  app_run(exec, ...);
}
```

Any Ginkgo operation on `app_run` will then use the kernels belonging to the dynamic type of `exec`. Furthermore, as `app_run` only depends on the abstract `Executor` interface, it is not necessary to recompile this function when switching between executors.

Linear Operators

The Ginkgo library is mainly designed around the linear operator concept. The type `LinOp` represents an abstract interface for the mathematical notion of a linear operator. This encompasses of course matrices, but also preconditioners and linear solvers, although they satisfy linearity only up to the achieved solving precision. All Ginkgo types implementing these concepts are implementations of the abstract `LinOp` type. The primary requirement is that the implementations provide a mapping between vector spaces.

The abstract linear operator concept offers great flexibility for choosing different algorithms. Through polymorphism, different implementations of, for example, matrix formats can be used interchangeably in algorithms that only rely on the operator application. This is true for most Krylov methods. So, users can easily switch out the matrix format to be used in a Krylov solver, and pick the format that is most suitable for their specific application. However, the flexibility is not restricted to matrix formats. The Krylov (and direct) solvers themselves, as well as the preconditioners are also implementations of the `LinOp` interface. Thus, users can easily try out a wide range of functionality, for example switching between applying a matrix or its inverse approximated by a solver:

```
void operate(const gko::LinOp*, ...);
// ...
{
  if(invert){
    auto solver = gko::solver::Cg<>::build()->generate(mtx);
    operate(solver.get(), in, out);
  } else {
    operate(mtx.get(), in, out);
  }
}
```

Design Patterns

Ginkgo uses design patterns to simplify the code, convey intent, and provide safe code. A key pattern is the use of smart pointers. These enable behavior close to automatic garbage collection, while still having completely deterministic behavior. Through them, the lifetime and ownership of polymorphic objects is managed. Either an object only has a single owner, then a `std::unique_ptr` keeps the object alive as long as the pointer stays in scope. The following gives a short example using one of Ginkgo's matrix types:

```
{
  std::unique_ptr<gko::LinOp> mtx = gko::matrix::Csr::create(exec, ...);
  // operate on mtx
  // ...
} // after leaving the scope mtx is automatically deleted
```

Alternatively, an object might have multiple owners, e.g. a matrix that is owned by a solver and a preconditioner. In this case, a `std::shared_ptr` keeps the object alive while at least one owner stays in scope. Once all owners are out-of-scope, the object is finally deleted. This creates a semi-automatic garbage collector behavior that prevents common memory issues, such as leaks, or double frees, while using a simple API with deterministic behavior and minimal overhead.

Lastly, Ginkgo is designed to offer great compatibility with other libraries through its transparent API. Part of the transparent API is to provide access to the stored data of the previously mentioned data structures through raw pointers. In contrast to smart pointers, these don't modify the ownership, but this introspection of the data layout allows other libraries to operate directly on data owned by Ginkgo. Compatibility in the other direction is also supported, since most data structures can be created by passing in raw pointers to existing data, resulting in Ginkgo objects that don't take ownership of the data. Both options are presented in the following snippet:

```
// extract data (and number of non-zeros) from a Ginkgo CSR matrix
double* values = mtx->get_values();
gko::size_type nnz = mtx->get_num_stored_elements();

// lend data to Ginkgo
double* app_ptr = ...; // given from application
auto view = gko::make_array_view(exec, app_size, app_ptr);
```

3.2 Facilitating Integrations

The aforementioned design concepts provide distinct benefits when integrating Ginkgo. Besides offering high-performance kernels, the runtime selectable backends provide great flexibility when interacting with other realizations of platform portability. Many portability approaches are configurable only at compile-time. This includes portability layers like Kokkos. An application that uses such a compile-time approach has to make sure that all components use the same configured backend. Since the backend choice for Ginkgo is delayed until runtime, it's easy to ensure that it uses the correct backend.

Furthermore, Ginkgo's type hierarchy, especially the `LinOp` interface, offers a flexible and extensible API. Not only can existing Ginkgo algorithms be switched out easily, applications can also provide their own implementation of Ginkgo's interface and then use their types in Ginkgo's algorithms. For example, this allows users to implement an operator for matrix-free stencil applications. The matrix-free operator can then be used in Ginkgo's Krylov solvers. Thus, applications gain flexible access to algorithms through Ginkgo, with a simple extension interface to inject specialized operations.

Finally, Ginkgo's transparent API greatly helps the data exchange with application code. By exposing raw pointers, an application can easily pass Ginkgo data, such as vectors, to other components without them relying also on Ginkgo, while the ownership stays with Ginkgo. Additionally, data that is owned by the application can be lent to be used within Ginkgo. This helps the application to create clear boundaries between their internal components without requiring expensive copies.

4 Integration Approaches in Application Software

The previous section presented approaches within Ginkgo to ease the effort of integrating it into an application. Now, the application side of the integration is considered. In general, an application can follow two different approaches to the integration. First, the application can use tight coupling with Ginkgo. The other option is loose coupling. These approaches are not specific to Ginkgo, but rather common approaches in the design of software applications. They reflect how deeply intertwined the application becomes with a library.

The strongest intertwining is the tight coupling. In this case, the application uses the Ginkgo data structures directly and might even use them outside the linear solver setting. Loose coupling leads to less intertwining by relying on abstract interfaces instead of Ginkgo directly, which can be considered as an application of the facade design pattern as mentioned in [9]. Ginkgo supports both modes, and the examples in Sect. 5 show different variations on them. The OpenFOAM example uses tight coupling, and openCARP transitioned from tight coupling with PETSc to loose

coupling for both PETSc and Ginkgo. Both ExaSGD and PeleLM use loose coupling, although in different variations.

4.1 Tight Coupling

Ginkgo provides the basic data structures necessary for linear algebra. Both dense vectors and sparse matrices with different formats are available to use as the backbone of an application's linear algebra component. Additionally, Ginkgo offers a storage type for one-dimensional arrays. The array type is directly linked to the `Executor` concept, which allows allocating memory on different devices besides the CPU. Together with simple APIs for copying data between arrays of different executors, Ginkgo provides the means for arbitrary storage using different devices.

In contrast to other popular numerical linear algebra packages, Ginkgo's scope is limited to linear algebra. Applications might require other concepts that are not strictly part of linear algebra, mesh handling for example. Longstanding libraries such as PETSc or Trilinos provide these additional functionalities. Although Ginkgo is more focused than those, it might still be used as the sole linear algebra backend. Since Ginkgo allows for direct data exchange without overhead, it can easily be used in conjunction with other libraries that cover the missing areas.

The Example in Listing 1 shows how an application could implement an implicit Euler time stepper by using a tight coupling with Ginkgo. The first thing to note is that the application class stores Ginkgo objects directly. The `advance` function then uses these objects and calls methods directly on them. Additionally, as the function expects Ginkgo objects as parameters, the calling code must also use Ginkgo objects directly. Thus, the dependency of Ginkgo extends further than this class.

The example highlights some of the advantages and disadvantages of the tight coupling. As Ginkgo types are used directly, no additional type hierarchy on the application side is necessary. But the implementation on the application side now deeply depends on Ginkgo. Generally speaking, implementations using tight coupling have their lifetime tied to the integrated libraries. If the integrated library stops receiving updates, the application is stuck with potentially out-of-date code. Either the application would have to update the library or replace it. Both would be major undertakings for the application.

Another not yet mentioned benefit is the simpler dependency management. The application always depends on the integrated library, so no special handling has to be developed if a library is missing. This will usually simplify the application's build system.

```
using Dense = gko::matrix::Dense<>;

class ImplicitEuler{
public:
  void advance(Dense* u_n, Dense* dx){
    // assemble right-hand-side f_ based on u_n

    Ainv_->apply(f_, x_);

    u_n->add_scaled(dx, x_);
  }

private:
  std::unique_ptr<Dense> x_;
  std::unique_ptr<Dense> f_;
  std::unique_ptr<gko::LinOp> Ainv_;
}
```

Listing 1: A tight-coupling application code example that implements an Euler time-stepper using Ginkgo functionality.

4.2 Loose Coupling

Compared to tight coupling, loosely coupled integrations will use Ginkgo only through indirection. First, the application needs abstract interfaces for its linear algebra component. The interfaces represent the boundary between the application code and the external libraries, such as Ginkgo. Interface code is called with application data, and expects application data as output if applicable. But internally, implementations of the interface will use Ginkgo types and functions. This might incur some overhead due to copying. Although both sides of the interface use the same conceptual representation, e.g. a matrix in CSR format, the underlying data layouts might differ, which requires copies. On the other hand, if the underlying data layouts are the same, no copy should be necessary. Thus, if libraries can communicate the used data layouts through their APIs, then interoperability can be improved through the elimination of copies, as will be seen in the case studies from Sect. 5.1.

In the context of linear algebra, these interfaces usually involve vectors, matrices, and solvers. These interfaces can provide multiple purposes. The solver interface provides the solution to a linear system, and the vector and matrix interface can provide conversions between the application's native format and Ginkgo, or other features such as assembly. Applications might provide multiple implementations of an interface for Ginkgo. If they differentiate between Krylov solvers on a type level, for example, then multiple implementations of the solver interface might be necessary to cover different Ginkgo solvers.

The interfaces might also provide simpler access to complex functionality. A solver interface could offer a method to regenerate the solver with a new matrix. The Ginkgo implementation for this would contain multiple lines of code, with different

Ginkgo functionality used, because Ginkgo doesn't provide the regenerate functionality through a simple API. Thus, the interface can hide complexity in Ginkgo, making it easier to use for the application specific purposes.

It is not necessary to provide all the mentioned interfaces. If only the solution to a linear system is required, without additional matrix or vector algebra, then the solver interface would be enough. The conversions between application and Ginkgo data are then handled purely internally. Ultimately, it's up to the application if all three interfaces are necessary, or if a subset suffices.

The example in Listing 2 shows an implementation of a contrived abstract solver interface. In this case, the application only provides the solver interface, so the in- and outputs are defined as application types. During the constructor, an application matrix is provided. This is converted into a Ginkgo matrix using an explicit copy, since it is assumed that different data layouts are used. For the solution method, application vectors are provided, but no copy is performed. Now, the assumption is that both Ginkgo and application vectors have the same data layout. This is typically easier to guarantee since Ginkgo uses a contiguous data layout with strides, which is widely used, e.g. in the BLAS API, or for Kokkos Views.

The only Ginkgo specific type in the interface is the executor `exec`. If an application is already using different execution spaces, e.g. CPU execution, and GPU execution, then a mapping between the applications' notion of execution spaces and Ginkgo's executors has to be defined. In the special case of Kokkos' execution spaces, Ginkgo already provides these mapping. For applications that were not using GPUs before, this might consist in mapping `std::strings` to Ginkgo's executors.

Compared to the tight coupling, loose coupling exhibits advantages that support the applications' freedom of choice. As the dependency is hidden through an interface, it's easy to change, remove, or add external libraries implementations. Furthermore, the dependency is restricted to a clear set of implementation files. Another benefit of the interfaces is the restriction of the integrated library to only the necessary functionality. On the other hand, the interface introduces an additional class hierarchy to manage, as well as boilerplate code in some instances. Also, managing the dependencies in the build system is more complex.

5 Case Studies

In the following section, examples are brought up to highlight the improvements applications can gain from integrating Ginkgo, as well as the differences in their integration approaches. First, two ECP applications, ExaSGD and PeleLM, are discussed. Both use loose coupling to integrate Ginkgo. While ExaSGD benefits from the direct solvers in Ginkgo, PeleLM exploits the batched iterative solvers. The third application is openCARP, which transitioned from a tight coupling with PETSc to a loose coupling with both PETSc and Ginkgo. It benefits from the enabled GPU acceleration. Finally, the OGL library uses tight coupling to provide Ginkgo solvers

```
class GinkgoCG: public AbstractSolver{
  using Dense = gko::matrix::Dense<>;
  using Csr = gko::matrix::Csr<>;
  using Cg = gko::solver::Cg<>;

public:
  void solve(const AppVector& b, AppVector& x) override
  {
    // Create views on the application data.
    // If the application data uses host memory, then instances of exec_
    // needs to be replaced by exec_->get_master().
    auto gko_x = Dense::create(
                    exec_, gko::dim<2>{b.num_rows(), b.num_cols()},
                    gko::make_array_view(exec_, b.total_size(), b.data()),
                    b.stride());
    // This is the const-correct version of the call above.
    auto gko_b = Dense::create_const(
                    exec_, gko::dim<2>{b.num_rows(), b.num_cols()},
                    gko::make_const_array_view(exec_, b.total_size(),
                                               b.data()),
                    b.stride());
    solver_->apply(gko_b, gko_x);
    // since views are used, the AppVector x already contains the result
    // otherwise an extra copy is necessary
  }

  GinkgoCG(std::shared_ptr<const gko::Executor> exec, AppMatrix& A,
           int max_it, double eps)
    : AbstractSolver(), exec_(exec)
  {
    gko::matrix_data<> md(gko::dim<2>{A.num_rows(), A.num_cols()};
    // Fill md by extracting the triplet (row-index, col-index, value) for
    // each non-zero in A and pass it to
    // md.nonzeros.emplace_back(row, col, value)
    auto gko_A = Csr::create(exec);
    gko_A->read(md);
    solver_ = Cg::build()
                .with_criteria(gko::stop::Iteration::build()
                                 .with_max_iters(max_it),
                               gko::stop::ResidualNorm<>::build()
                                 .with_reduction_factor(eps))
                ->generate(std::move(gko_A));
  }

private:
  std::shared_ptr<const gko::Executor> exec_;
  std::unique_ptr<Cg> solver_;
};
```

Listing 2: A loose-coupling application integration example, implementing a GinkgoCG solver by following an abstract solver interface.

for OpenFOAM. Again, through these solvers, GPU acceleration is enabled for parts of computational fluid dynamics (CFD) simulations using OpenFOAM.

5.1 ECP Applications

Both the ExaSGD and PeleLM applications benefited greatly from their integration with Ginkgo. As the major findings have been presented in the papers [1, 2, 20], and additionally in the summary paper [6], the performance gain for the applications are not discussed here. Instead, the focus lies on the integration approaches. The summary paper also mentions integrations with MFEM, and XGC. These are not followed up on here, since XGC is very similar to PeleLM in how Ginkgo is used, and MFEM doesn't classify as a science application, since it is again a library that other applications will integrate.

The **ExaSGD** project simulates the power grid of the US in order to make projections for future developments. At its core, it solves non-linear optimization problems, which is done through HiOP [17]. The resulting linear systems tend to be highly ill-conditioned, requiring the use of a direct solver. In addition, these systems have little structure, which makes prominent direct solvers such as SuperLU less effective. A new direct solver has been co-developed within Ginkgo to address these issues. The authors provide a detailed overview in [18]. By integrating Ginkgo into HiOP, platform portable direct solvers became available, as discussed in [20].

As mentioned earlier, ExaSGD (through HiOP) integrates with Ginkgo through loose coupling. More specifically, it only provides a solver interface for Ginkgo to implement. Thus, the integration requires a few lines of code, in total about 470 lines. Together with a few outside changes, e.g. to the build system or code examples, the initial implementation was done in 700 line of code (LOC) changes.[3] This marks a relatively low burden on HiOP's code base.

The integration is very similar to the example provided in Sect. 4.2. In both cases, an application matrix is copied over to a Ginkgo matrix during initialization, while views on application vectors are used in the solver application phase. Additionally, more application-specific interfaces are provided. These concern updating the values of a matrix, without changing the structure, and recomputing the factorization. Since the direct solver classes in Ginkgo don't provide this functionality directly, the implementation complexity is hidden by the interface.

During the interface initialization, the Ginkgo objects can be customized. Since HiOP only requires a specific subset of Ginkgo, the customization is rather limited. Only five options are provided in a mapping from HiOP options to Ginkgo customization. Besides that, many Ginkgo settings are fixed. The solver is always an LU solver, possibly wrapped in GMRES depending on the options, and MC64 is always used as an apriori reordering of the system.

[3] https://github.com/LLNL/hiop/pull/433/.

The **PeleLM** application [15] is used for CFD simulation, specifically for reactive flow in the Low Mach number regime. The discretization approach requires solving many independent ODEs. These are solved using Sundials [12], which requires the solution of many small independent linear systems. To solve these efficiently, batched iterative solvers have been developed in Ginkgo. The papers [1, 2] offer more details on the performance aspects.

Although the PeleLM integration through Sundials shares many similarities to the HiOP integration, it had a vastly different starting point. In contrast to HiOP, Sundials is mainly a C code. This introduces additional friction, as many modern C++ design principles which Ginkgo uses are not compatible with C design patterns. Integrating Gingko using a C API would lead to struggles due to C++ features that are not part of C, mostly RAII-based object management and templating. To alleviate the differences, Sundials introduced a new C++ API instead. This API takes advantage of templating to let users decide which concrete Ginkgo type should be used in the implementation of the interface.

The extra work is reflected in the integration effort. About 8000 LOC additions and 2500 LOC deletions were necessary in the initial pull request[4] (PR), which included the new C++ API. Despite the large-scale API changes, the changes solely related to Ginkgo are in the same order as HiOP. The Ginkgo implementation of the linear solver C++ API requires about 350 LOC. In addition, the matrix interface is also implemented for Ginkgo, again, in about 350 LOC.

Compared to HiOP's implementations, the Sundials ones are more minimal. Besides the expected apply and solve method, the matrix and solver interfaces only provide conversions to the Sundials C API, as well as access to the underlying Ginkgo objects. Similar to HiOP the apply and solve methods work with views on the application data. However, in contrast to HiOP, no configuration options are available, either as methods or as constructor arguments. Instead, the Ginkgo interfaces are constructed from existing Ginkgo objects. This allows users of Sundials to use every possible Ginkgo configuration option, without adding a burden to the interface by translating these options into Sundials options. Thus, users profit from Ginkgo's flexibility, while the maintenance effort on Sundials is minimal.

5.2 openCARP

openCARP is a simulator for cardiac electrophysiology. It provides tools for a wide range of simulations with different physical models. The simulations are discretized with finite element discretization, which, in the end, leads to solving linear systems. Through integrating Ginkgo, openCARP is capable of offloading the linear systems to GPUs [5].

Through the integration of Ginkgo, openCARP transitioned from a tight coupling to a loose coupling. Before Ginkgo was used, openCARP solely relied on PETSc

[4] https://github.com/LLNL/sundials/pull/205.

for their linear algebra need. As a consequence, openCARP was tightly coupled with PETSc, using their data types throughout their codebase. Components that are not related to linear algebra, for example the physics module, contained hard dependencies on PETSc.[5] These dependencies had to be weakened for Ginkgo to be utilized.

The transition to a loose coupling introduced abstract interfaces for vectors, matrices, and solvers. This required a large-scale effort. About 8000 LOC were changed in the initial PR[6] due to the dependency on PETSc in many modules. Since openCARP doesn't have a native vector or matrix type like HiOP or Sundials, the interfaces provide a multitude of functionality to cover the previous usage. Most essential are the typical BLAS level 1 and 2 type operations. Additionally, methods to help the assembly are present. Still, even with these additions, the resulting interfaces are relatively small compared to the initial effort, only approximately 300–500 LOC per interface were necessary.

Now, the openCARP codebase has become more sustainable. The linear algebra backend implementations are properly separated from other modules, and their dependencies are restricted to the linear algebra module. It has become possible to easily add other implementations for this component. As both the PETSc and Ginkgo implementation have a manageable size, 1600 lines, and 1000 lines respectively, the expected effort for additional implementations would again be of moderate size.

5.3 OpenFOAM

OpenFOAM (Open Field Operation And Manipulation) is a C++ framework for the development of numerical solvers and pre-/post-processing utilities for continuum mechanics problems, especially computational fluid dynamics (CFD) [13]. While various efforts were made to provide GPU support to OpenFOAM no commonly accepted approach exists [16] to this date. One approach is to offload certain parts, e.g. the linear solver, to GPUs via dedicated plugins. These plugins are separate from OpenFOAM. Their respective libraries are loaded at runtime to provide access to the additional functionality. Thus, OpenFOAM interacts through loose coupling with the plugins, while the plugins themselves are implemented using tight coupling. The OpenFOAM Ginkgo Layer (OGL) [16][7] is one of the plugins, providing linear solvers from the Ginkgo library.

OGL functionality is split into three parts. First, it provides a mapping between OpenFOAM and Ginkgo data structures, especially for sparse matrices. For these Ginkgo data structures, OGL has to ensure that they persist through the whole simulation. Lastly, the Ginkgo solvers and preconditioners can be configured at runtime

[5] See for example the file `physics/electrics.cc` in the release `v11.0`.

[6] https://git.opencarp.org/openCARP/openCARP/-/merge_requests/79.

[7] https://github.com/hpsim/OGL/.

through OGL. All combined, OGL allows OpenFOAM to efficiently use the platform portable solvers of Ginkgo.

Since OGL is a standalone library, it needs a considerably larger effort than the previously discussed integrations. The library consists roughly of 7000 lines of C++ code. Currently, the largest part of that is handling the object persistency. This requires knowledge of the underlying object storage as defined in Ginkgo, which is directly exposed in Ginkgo API. Without tight coupling, more complex interfaces and hierarchies would be required to describe the storage dynamically for different libraries. In general, the type hierarchy is shallow, since very little abstraction is necessary.

6 Summary

Many papers discuss the functionality gain that library integrations offer to application software. Although these gains are the reason why the integration is considered in the first place, the consequences on the development effort for the application are rarely discussed. This paper offers a discussion on these kinds of impacts. By the example of the library Ginkgo and its integration into multiple applications, the effects on application software are highlighted.

While Ginkgo offered each application clear gains, e.g. in the form of a platform portable linear algebra component or specialized batched solvers, those came at a cost on the application side. The Ginkgo integration had to be developed and needed to be maintained, which occupies valuable application developer time. But under the right circumstances, the integration effort is minimized. The key factor here is the loose coupling through well-defined interfaces on the application side. Applications that pursued this approach could integrate Ginkgo in less than 500 LOC per interface, while otherwise a large effort was required by the application, either to switch to loose coupling or build the tight coupling from the ground up.

For the examples given here, loose coupling offered the best approach for application developers. However, the examples only represent a very limited subset. Further evaluation of different integration approaches is necessary to provide both library and application developers with concise guidelines. Another aspect to further evaluate is the extension to other library types. While this paper only considers linear algebra libraries, the integration of libraries with different focuses might lead to other conclusions.

Acknowledgements This work was supported by the “Impuls und Vernetzungsfond of the Helmholtz Association” under grant VH-NG-1241. This work was supported by the European High-Performance Computing Joint Undertaking EuroHPC under grant agreement No 955495 (MICROCARD). Further, it was supported by BMBF through the SCALEXA initiative and the ExaSim (16ME0676K) and PDExa (16ME0641) projects. This research was also supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

References

1. Aggarwal, I., Kashi, A., Nayak, P., Balos, C.J., Woodward, C.S., Anzt, H.: Batched sparse iterative solvers for computational chemistry simulations on GPUs. In: 2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), pp. 35–43 (2021). 10/gn3xcg. https://ieeexplore.ieee.org/document/9652814
2. Aggarwal, I., Nayak, P., Kashi, A., Anzt, H.: Preconditioners for batched iterative linear solvers on GPUs. In: Doug, K., Al, G., Pophale, S., Liu, H., Parete-Koon, S. (eds.) Accelerating Science and Engineering Discoveries Through Integrated Research Infrastructure for Experiment, Big Data, Modeling and Simulation, Communications in Computer and Information Science, pp. 38–53. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-23606-8_3
3. Balay, S., Abhyankar, S., Adams, M.F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W.D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M.G., Kong, F., Kruger, S., May, D.A., McInnes, L.C., Mills, R.T., Mitchell, L., Munson, T., Roman, J.E., Rupp, K., Sanan, P., Sarich, J., Smith, B.F., Zampini, S., Zhang, H., Zhang, H., Zhang, J.: PETSc/TAO users manual. Tech. Rep. ANL-21/39-Revision 3.21, Argonne National Laboratory (2024). https://doi.org/10.2172/2205494
4. Carter Edwards, H., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. (2014). https://doi.org/10.1016/j.jpdc.2014.07.003
5. Cojean, T., Göbel, F., Houillon, M.: Porting the opencarp cardiac electrophysiology simulation framework to GPU architectures (2022)
6. Cojean, T., Nayak, P., Ribizel, T., Beams, N., Tsai, Y.H.M., Koch, M., Gobel, F., Grützmacher, T., Anzt, H.: Ginkgo—a math library designed to accelerate ECP science applications. Int. J. High Perform. Comput. Appl. (Forthcom.)
7. Deakin, T., McIntosh-Smith, S., Price, J., Poenaru, A., Atkinson, P., Popa, C., Salmon, J.: Performance portability across diverse computer architectures. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 1–13 (2019). https://doi.org/10.1109/P3HPC49587.2019.00006
8. Dongarra, J.J.: The evolution of mathematical software. Commun. ACM **65**(12), 66–72 (2022)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional (1994)
10. Heinlein, A., Klawonn, A., Rheinbach, O.: A parallel implementation of a two-level overlapping schwarz method with energy-minimizing coarse space based on trilinos. SIAM J. Sci. Comput. **38**(6), C713–C747 (2016)
11. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the trilinos project. ACM Trans. Math. Softw. **31**(3), 397–423 (2005)
12. Hindmarsh, A.C.: SUNDIALS: suite of nonlinear/differential/algebraic equation solvers. Tech. Rep. UCRL-JC-149711, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2002). https://doi.org/10.1145/1089014.1089020. https://www.osti.gov/biblio/15002968. ISSN: 0098-3500 Issue: 3 Volume: 31
13. Jasak, H., Jemcov, A., Tukovic, Z., et al.: Openfoam: A c++ library for complex physics simulations. In: International Workshop on Coupled Methods in Numerical Dynamics, vol. 1000, pp. 1–20 (2007)
14. May, D.A., Sanan, P., Rupp, K., Knepley, M.G., Smith, B.F.: Extreme-scale multigrid components within PETSc. In: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '16. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2929908.2929913

15. Nonaka, A., Day, M.S., Bell, J.B.: A conservative, thermodynamically consistent numerical approach for low Mach number combustion. Part I: Single-level integration. Combust. Theory Model. **22**(1), 156–184 (2018). https://doi.org/10.1080/13647830.2017.1390610. Publisher: Taylor and Francis _eprint
16. Olenik, G., Koch, M., Boutanios, Z., Anzt, H.: Towards a platform-portable linear algebra backend for OpenFOAM. Meccanica (2024). https://doi.org/10.1007/s11012-024-01806-1
17. Petra, C.G., Chiang, N., Wang, J.: HiOp—User Guide. Tech. Rep. LLNL-SM-743591, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory (2018)
18. Ribizel, T., Anzt, H.: Parallel symbolic cholesky factorization. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W '23, pp. 1721–1727. Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624253
19. Scott, J.: HSL@60: a brief history of the HSL mathematical software library. Tech. rep., STFC Rutherford Appleton Laboratory (2023). https://doi.org/10.5286/stfctr.2023002. http://purl.org/net/epubs/work/54066639
20. Świrydowicz, K., Koukpaizan, N., Ribizel, T., Göbel, F., Abhyankar, S., Anzt, H., Peleš, S.: GPU-resident sparse direct linear solvers for alternating current optimal power flow analysis. Int. J. Electr. Power & Energy Syst. **155**, 109517 (2024)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



On the Use of Algebraic Multigrid in Applications on High Performance Computers



Robert D. Falgout, Rui Peng Li, Wayne Mitchell, Daniel Osei-Kuffuor, Victor A. P. Magri, and Ulrike M. Yang

Abstract The *hypre* software library provides a variety of parallel linear solvers designed for high performance computers. It focuses on algebraic multigrid methods (AMG), which provide excellent scalability. With the expanding use of accelerators in current and future high-performance computers, new programming models have been added to take advantage of the increased performance potential of graphics processing units (GPUs). This chapter will discuss porting strategies and present results for *hypre*'s GPU-enabled multigrid solvers within three application codes.

Keywords Algebraic multigrid · High performance computing · Linear solvers · Preconditioning · Scalability · HPC software · GPU computing

1 Introduction

The solution of large linear systems is an essential and often computationally expensive component of many large-scale simulations on high-performance computing (HPC) machines. Such simulations would not be feasible without linear solvers

R. D. Falgout (✉) · R. P. Li · W. Mitchell · D. Osei-Kuffuor · V. A. P. Magri · U. M. Yang
Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, Livermore, CA, USA
e-mail: falgout2@llnl.gov

R. P. Li
e-mail: li50@llnl.gov

W. Mitchell
e-mail: mitchell82@llnl.gov

D. Osei-Kuffuor
e-mail: oseikuffuor1@llnl.gov

V. A. P. Magri
e-mail: paludettomag1@llnl.gov

U. M. Yang
e-mail: yang11@llnl.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_4

that are algorithmically scalable and tailored to particular HPC architectures. These demands have only increased in the new age of exascale computers, where performance is achieved through vast parallelism and highly specialized GPU architectures.

The *hypre* [12, 18] library of high-performance solvers and preconditioners is developed and maintained at Lawrence Livermore National Laboratory (LLNL) and provides scalable and portable algorithms that are relied upon by many large-scale application codes around the world. The goals of *hypre* are three-fold: first, to provide a variety of general and adaptable interfaces that can be easily used across many disparate application areas (Sect. 2); second, to provide scalable and effective algorithms for a wide set of applications and problem sizes (Sect. 2); and finally, to provide performant and portable implementations of these solvers on a wide variety of computing architectures (Sect. 3). Through pursuing these goals, *hypre* has seen widespread success in application codes, and three examples are given here (Sect. 4). The applications cover three scientific areas: combustion, subsurface flow, and computational fluid dynamics (CFD); they use different *hypre* interfaces and solvers and run on HPC systems, including the exascale computer Frontier at Oak Ridge National Laboratory (ORNL), with scalability demonstrated up to tens of thousands of GPUs. The paper concludes with a synthesis of the findings and future research directions in Sect. 5.

2 Interfaces and Solvers

The *hypre* library provides three different interfaces [12] to allow users to represent their problems in a way they prefer: a structured interface that uses grids and stencils, a semi-structured interface for problems that are mostly structured, such as adaptive mesh refinement grids, or block-structured grids, defined with stencils or finite elements, and a linear-algebraic interface, where problems are expressed in terms of matrices and vectors.

Figure 1 shows the notion of *hypre*'s conceptual interfaces and how they connect to solvers. The top row shows various concepts, such as structured grids, composite grids, block-structured grids, unstructured grids, or standard matrices. The second row lists various solvers and preconditioners. These require different information from the user provided through the interfaces. The bottom row lists different data layouts. The most efficient solvers, such as PFMG, can only be used through the structured interface on the left. In contrast, unstructured solvers such as (Boomer)AMG can be used with all interfaces, albeit at a cost, since they use less efficient data layouts but can solve more complex problems.

Figure 2 shows a list of available solvers in *hypre* and indicates the interfaces that can be accessed along with the data layouts used for their implementation. The term 'matrix free' means the solver can be run with any of the three data layouts, whereas all other solvers have been implemented using the data layout mentioned in Fig. 2. Most of *hypre*'s solvers are multigrid methods. Exceptions are the Krylov solvers, the incomplete LU factorization preconditioner ILU, and the sparse approximate

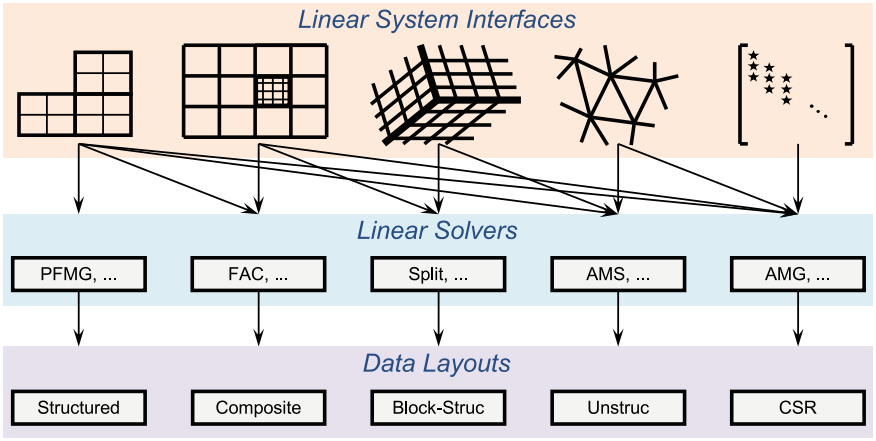


Fig. 1 Illustration of the idea behind *hypre*'s conceptual interfaces

Solvers

Linear solver availability via *hypre*'s conceptual interfaces for selected solvers

Data Layouts	Solvers	Interfaces		
		Struct	SStruct	IJ
Structured	SMG	√	√	
	PFMG	√	√	
Semi-Structured	Split		√	
	SysPFMG		√	
Sparse Matrix	ADS		√	√
	AMS		√	√
	BoomerAMG		√	√
	FSAI		√	√
	ILU		√	√
	MGR		√	√
	pAIR		√	√
`Matrix free'	BiCGSTAB	√	√	√
	GMRES	√	√	√
	PCG	√	√	√
	Hybrid	√	√	√

Fig. 2 Selected *hypre* solvers and their association with interfaces and data layouts

inverse preconditioner FSAI. These methods have been included for difficult systems that cannot be solved with multigrid methods and can also be used as smoothers within *hypre*'s unstructured AMG solver BoomerAMG for more robust convergence if needed. All methods in Fig. 2 have been GPU-enabled for NVIDIA, AMD, and Intel GPUs via CUDA, HIP, and SYCL. However, not all options and configurations of these solvers are fully ported to GPUs. SMG, PFMG [3], SysPFMG, pAIR [15, 23], and BoomerAMG [16] can be described by the general AMG method described in Sect. 2.1, whereas ADS, AMS and MGR are more specialized solvers that internally use BoomerAMG. AMS [19] has been developed for systems derived from H-curl discretizations of the Maxwell equations, ADS [20] for H-div systems, and MGR [27], described in Sect. 2.2, for block linear systems. The solvers used in the applications considered in Sect. 4 are PFMG, BoomerAMG, and MGR, which are described in more detail below.

2.1 Algebraic Multigrid

AMG methods consist of a setup phase and a solve phase. The setup phase consists of the coarsening algorithm that determines the variables to be used on the next coarser level, the generation of restriction and prolongation operators, $R^{(m)}$ and $P^{(m)}$, needed to move between different levels, m and $m+1$, and the generation of the coarse grid operator, which is generally accomplished by the Galerkin product, $A^{(m+1)} = R^{(m)} A^{(m)} P^{(m)}$, where $A^{(0)} = A$ is the original linear system to be solved. Often restriction is defined as the transpose of interpolation, $R^{(m)} = (P^{(m)})^T$, particularly when dealing with a symmetric matrix, A. Figure 3 shows two levels of an AMG V-cycle, beginning with a few steps of smoothing followed by a restriction of the residual to the next level. This process is then applied recursively until the coarsest level, where the system can be solved directly. The coarse-grid error correction, e^{m+1}, is then interpolated to the next finer level and added to the approximation, u^m, before performing some additional smoothing.

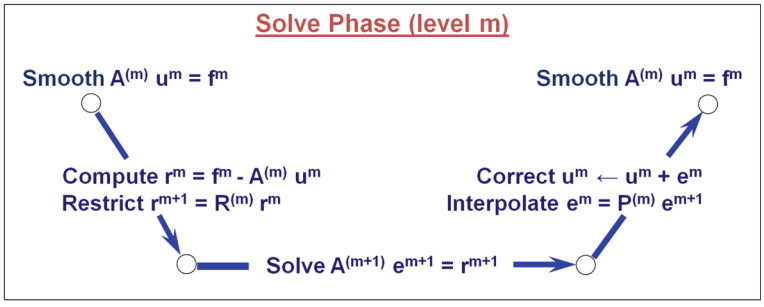


Fig. 3 Two levels of an AMG V-cycle

PFMG, *hypre*'s most efficient solver, is based on the structured data layout, which is defined by grids and stencils and is well suited for GPUs. It uses semi-coarsening and a simple two-point interpolation operator. This strategy is highly efficient, since it limits stencil sizes on lower levels. The solve phase uses an efficient point smoother.

BoomerAMG, *hypre*'s most widely used preconditioner, is based on a parallel compressed sparse row (CSR) matrix data layout. It has a large variety of different options. Default settings for GPUs include PMIS coarsening [11] and interpolation operators based on matrix operations [21]. Smoothing is generally done with simple methods, such as L1-Jacobi [4], but more complex smoothers such as ILU or FSAI can also be employed.

2.2 Multigrid Reduction Framework

Multigrid reduction (MGR) is an approximation of the total reduction method to yield an efficient iterative solver. Originally introduced by Ries, Trottenberg and Winter in 1983 [24], various adaptations of the algorithm have been developed, including [2, 13, 22, 27, 28]. The MGR framework developed in *hypre* is designed to enable the efficient solution of systems of coupled equations, which are known to be problematic for scalar multigrid methods. The approach generalizes multi-stage and block preconditioners, typically used in multiphysics applications, in a standard multigrid framework [27]. Following standard multigrid convention, consider a splitting of a coupled system of discretized partial differential equations (PDEs) into two sets of C-points and F-points:

$$A = \begin{pmatrix} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{pmatrix} = \begin{pmatrix} I_{FF} & 0 \\ A_{CF}A_{FF}^{-1} & I_{CC} \end{pmatrix} \begin{pmatrix} A_{FF} & 0 \\ 0 & S_* \end{pmatrix} \begin{pmatrix} I_{FF} & A_{FF}^{-1}A_{FC} \\ 0 & I_{CC} \end{pmatrix}, \tag{1}$$

where I_{CC} and I_{FF} are identity operators and $S_* = A_{CC} - A_{CF}A_{FF}^{-1}A_{FC}$ is the Schur complement. The idea is to construct approximate restriction and interpolation operators, R and P, so that $RAP \approx S_*$. The resulting two-grid error propagation operator for the MGR solution can be expressed as

$$I - M_{MGR}^{-1}A = (I - PM_C^{-1}RA)(I - M_F^{-1}A), \tag{2}$$

which describes the complementary processes of F-relaxation, with smoother M_F^{-1}, followed by coarse grid correction, with $M_C^{-1} \approx (RAP)^{-1}$. For a detailed description and theory of the two-grid approach, as well as equivalences to multi-stage and block preconditioners used in the multiphysics community, see [27].

Algorithm 1 describes the multigrid reduction's solve phase, which extends the two-level method in (2) via recursion. The subscript l denotes the operator at level l of the multilevel hierarchy. The algorithm presents a general approach, which includes pre-smoothing by a global smoother at each level. The global smoother may be used

Algorithm 1 Multigrid Reduction Solve Phase.

1: $MGR(l)$
2: **if** l is coarsest level, L **then**
3: solve coarse grid system $A_L e_L = r_L$ by classical AMG
4: **else**
5: Let $r = b$ and $e_0 = 0$
6: **Global Relaxation:** $e_l \leftarrow e_l + M_l^{-1} r$, where $M_l \approx A_l$
7: **F-Relaxation:** $e_l \leftarrow e_l + {M_F}_l^{-1}(r_l - A_l e_l)$
8: **Restriction:** $r_{l+1} = R_l(r_l - A_l e_l)$
9: **Recursion:** $e_{l+1} = MGR(l+1)$
10: **Update solution:** $e_l \leftarrow e_l + P_l e_{l+1}$
11: **end if**

instead of F-relaxation (as a more potent smoother) or as an additional smoothing step. In general, block relaxation schemes such as Jacobi or hybrid Gauss-Seidel are sufficient as a global smoother. However, ILU smoothers are more effective for problems with complicated coupling of unknowns. The framework provides a flexible interface to prescribe the splitting of unknowns into C-points and F-points for various block linear systems. Assuming this splitting is done based on the different physical variables in the system, a typical approach reduces a physical variable at each level until a scalar problem is reached at the coarsest level. In practice, however, variables with similar physical properties can be prescribed together at the same level and treated as a scalar problem. The implementation admits various compositions of smoothers to handle different physical regimes across the levels. Typically, using cheaper yet more efficient smoothers for F-relaxation and a more robust solver for the coarse grid correction leads to a reasonable preconditioner. The framework supports standard smoothers such as Jacobi, Chebyshev, and ILU; multilevel options like AMG; and direct solvers at each level. At the coarsest level, the coarse grid correction uses classical AMG, which is important to maintain good scalability for large-scale simulations. Thus, it is favorable to have a reduction strategy that has good M-matrix properties on the coarsest grid. At each level in the hierarchy, the coarse grid matrix can be constructed via a Galerkin product (RAP) or a non-Galerkin approach, which provides additional flexibility in approximating the Schur Complement. The MGR framework offers GPU support for various solver options and linear algebra operations. It has been successfully applied to large-scale applications of subsurface flow with geomechanical processes [7, 8, 27]. Numerical examples in Sect. 4.2.2 will demonstrate the performance of MGR on a multiphase subsurface flow problem with thermal effects and poromechanics.

3 Porting Strategies to Exascale Computers

Adapting *hypre* for GPU-based exascale machines necessitates the migration of operations within the algorithms from CPUs to GPUs, provided they remain suitable for the fine-grained parallelism inherent to GPUs. Otherwise, entirely new algorithms may need to be developed specifically for GPUs. For example, porting the original extended and extended+i interpolation algorithms [10] directly to GPUs poses various challenges. Consequently, new versions of the algorithms have been devised, formulated as sparse matrix-matrix multiplications (SpGEMM) [21]. This approach not only leverages existing high-performance SpGEMM kernels but also maintains the interpolation properties in the original forms.

Our porting strategies to develop high-performance GPU-accelerated AMG code adopt the following 3 approaches in sequence:

1. Utilize existing dense or sparse kernels from vendor libraries: cuBLAS, cuSPARSE, cuSOLVER (NVIDIA); rocBLAS, rocSPARSE, rocSOLVER (AMD); oneAPI, oneMKL (Intel).
2. Leverage high-level libraries that provide parallel primitives on GPUs of the algorithms in the C++ Standard library: thrust (CUDA); rocprim (HIP); oneDPL (SYCL). These primitives often times serve as building blocks of complicated AMG algorithms.
3. As necessary, develop custom device functions.

This sequence balances the extent of new code development while achieving optimal performance portability across GPUs and multicore CPUs. More details on the porting strategies of *hypre* to GPUs can be found in [14].

An example of the initial approach pertains to a critical operation, specifically sparse matrix-vector product (SpMV), which is essential in the AMG solve phase. Notably, sparse linear algebra libraries offered by vendors all include highly efficient algorithms for matrices in the CSR format.

As an illustrative example of the second approach, consider a fundamental operation used at many places in constructing AMG: transforming the row pointer array of the CSR matrix into the array of row indices of all nonzeros of a matrix. The following implementation, utilizing a double for-loop, can efficiently execute on CPUs, where `nrows` is the number of rows, `RowPtr` is the row pointer array of CSR, and `RowIdx` is the output array of row indices of all the nonzeros. To further improve performance on multi-core architectures, multi-threaded parallelism can be implemented for the outer for-loop.

```
#pragma omp parallel for
for (i = 0; i < nrows; i++)
  for (j = RowPtr[i]; j < RowPtr[i+1]; j++) { RowIdx[j] = i; }
```

Alternatively, for a GPU implementation aimed at maximizing parallelism, particularly in this case with respect to the number of nonzero coefficients, an efficient approach involves leveraging three parallel primitives: `fill`, `scatter_if`, and `inclusive_scan`. The first two primitives are in the C++ Standard Library, while

the last one is available in the Boost C++ Libraries [25], and all of them are in the C++ parallel algorithms library Thrust [1]. Here, `empty_row` is a straightforward function that returns true if a row contains no nonzeros.

```
fill(RowIdx, RowIdx + nnz, 0);
scatter_if(counting_iterator<int>(0), counting_iterator<int>(nrows), RowPtr,
transform_iterator(zip_iterator(make_tuple(RowPtr, RowPtr+1)), empty_row()),
  RowIdx);
inclusive_scan(RowIdx, RowIdx + nnz, RowIdx, maximum<int>());
```

Compared to the CPU implementation, this approach offers several advantages when running on GPUs: (1) enhanced granularity of parallelism, particularly on a per-nonzero basis within the first and last primitives; (2) improved memory coalescing patterns from adjacent threads; (3) better workload balancing, especially when the distribution of nonzeros among rows is far from an even distribution. Seamless unification of primitive function calls across different languages (i.e., CUDA, HIP and SYCL) can be achieved effortlessly through straightforward macro definitions within distinct namespaces.

The final approach of writing custom device functions is commonly employed when developing complicated AMG algorithms that are challenging to organize into standard linear algebra operations or even into standard parallel primitives. Implementations of these algorithms often diverge significantly from parallel CPU implementations for multi-core CPUs. Consequently, porting these algorithms presents nontrivial challenges, stemming from factors like thread granularity and the partitioning of computations among threads. We carefully implemented these device functions by exploiting low level hardware and language support, e.g., on-chip shared memory, warp-level primitives, thread synchronizations. Unifying device function implementations for CUDA and HIP is straightforward due to the similarity and compatibility of these two languages, requiring only the resolution of minor discrepancies such as the size of warps and maximum shared memory size. The syntax for launching device kernel functions remains identical across both languages. On the other hand, transitioning to SYCL presents remarkable challenges. While the mapping of functions between thrust and rocprim is straightforward, manual translation is often required for oneDPL. Additionally, while OneMKL provides sparse BLAS routines, caution must be exercised to navigate potential pitfalls, such as the matrix sorting requirement in the SpGEMM routine. Moreover, differences in kernel specification and launch syntax add further complexity to the process. To facilitate code reuse and efficiency, we defined macros as follows.

```
sycl::range<1> : gridsize, blocksize
void hypreGPUKernel_* (sycl::nd_item &item, ...) {
    get_grid_thread_id<1, 1>(item) }
#define HYPRE_GPU_LAUNCH {sycl::queue->submit([&](sycl::handler& cgh) {
cgh.parallel_for (sycl::nd_range<1>(gridsize*blocksize, blocksize),[=](sycl::
  nd_item<1> item)[[intel:: reqd_sub_group_size(HYPRE_WARP_SIZE)]] {(
  kernel_name)(item, __VA_ARGS__);});}).wait_and_throw();}
```

Finally, basic infrastructure elements, such as SYCL devices and warp/group-level functions, differ and thus necessitate re-implementation.

4 Use of *hypre* in Applications

Hypre's recent advances, especially its GPU capabilities, have significantly enhanced the ability of application codes to leverage modern supercomputers. In this section, we demonstrate the use of *hypre* in three application codes, emphasizing its critical role in achieving scalability and reducing overall execution times.

4.1 Uintah[1]

The Uintah framework is utilized for simulating boilers and conducting combustion research at the University of Utah. The code employs *hypre*'s PFMG to solve pressure equations. Figure 4 shows a scaling study of the ARCHES turbulent combustion component that simulates a helium plume on a single-level structured grid, which is a cube divided into patches. The experiment was performed for three problem sizes using two different patch sizes on the Summit computer at ORNL and used both GPUs and CPUs for the Uintah components of the code through Kokkos enabling OpenMP threading for the CPU cores and CUDA for the GPUs. The *hypre* library was exclusively run on GPUs using CUDA, with PFMG and micro-kernel fusion for enhanced performance [17]. The results, shown in Fig. 4, demonstrate excellent weak and strong scaling. Larger patch sizes yield better performance, while smaller patch sizes allow for a higher degree of fine-grained parallelism. Approximately 50 % of the total simulation time is attributed to *hypre*, indicating its significant contribution to the achieved scalability. Additionally, the Uintah team has successfully run their framework with *hypre* on two exascale computers: Frontier at ORNL (utilizing HIP for *hypre*) and Aurora at Argonne National Laboratory (utilizing SYCL for *hypre*).

4.2 GEOS[2]

GEOS is an open-source and GPU-enabled simulation code written in C++ specifically designed to solve tightly-coupled multiphysics problems, with a primary focus on subsurface reservoir applications. It offers robust capabilities for modeling compositional flow, poromechanics, faults and fractures slip, and thermal effects in underground reservoirs, setting it apart from traditional simulators. The mathematical models in GEOS typically result in challenging linear systems of equations. These systems are primarily solved using *hypre*, employing its MGR preconditioner, or the

[1] http://uintah.utah.edu.

[2] https://www.geos.dev/.

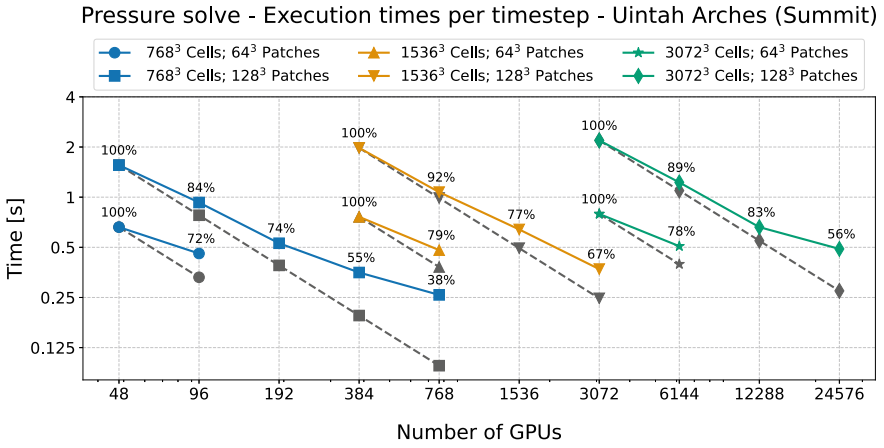


Fig. 4 Helium plume run to 24,576 NVIDIA V100 GPUs and 8,192 IBM POWER9 processors with 172,032 CPU threads using MPI+Kokkos for Uintah components and CUDA for PFMG

BoomerAMG preconditioner for simpler single-phase flow scenarios. In this section, we present weak and strong scalability results for two simulations performed with GEOS, highlighting *hypre*'s performance and efficiency.

4.2.1 Weak Scaling Single-Phase Flow Model

We used GEOS to simulate fluid flow in the vicinity of a wellbore, a scenario that is important for optimizing extraction processes and enhancing reservoir management. The model considers a single fluid type and solves for the pressure distribution in a radially structured mesh. The resulting linear systems generated in the transient simulation are solved with *hypre*'s GMRES preconditioned with BoomerAMG.

Figure 5 presents a weak scalability study in the Frontier supercomputer. We show average execution times per time step of the simulation and distinguish between *hypre* and GEOS times. The x-axis represents the number of GPUs[3] used followed by total number of degrees of freedom (DOFs) in millions (M) and billions (B). As the number of GPUs increases from 4 to 16384, corresponding to an increase in DOFs from 6.6 million to 27 billion, the execution times for both the overall GEOS simulation and the *hypre* solver components exhibit moderate increases. Specifically, the total GEOS execution time rises from 0.06 to 0.08 s, demonstrating the framework's efficiency and scalability in handling larger problem sizes. *hypre*'s (BoomerAMG) setup time increases from 0.12 to 0.47 s, reflecting the additional distributed communica-

[3] In the context of the Frontier system, the term "GPU" refers to a Graphics Compute Die (GCD) and not a physical GPU card. Each AMD MI250X GPU contains 2 GCDs, and with 4 MI250X cards per node, this results in 8 GCDs per node, each treated as a separate GPU in system configurations and programming.

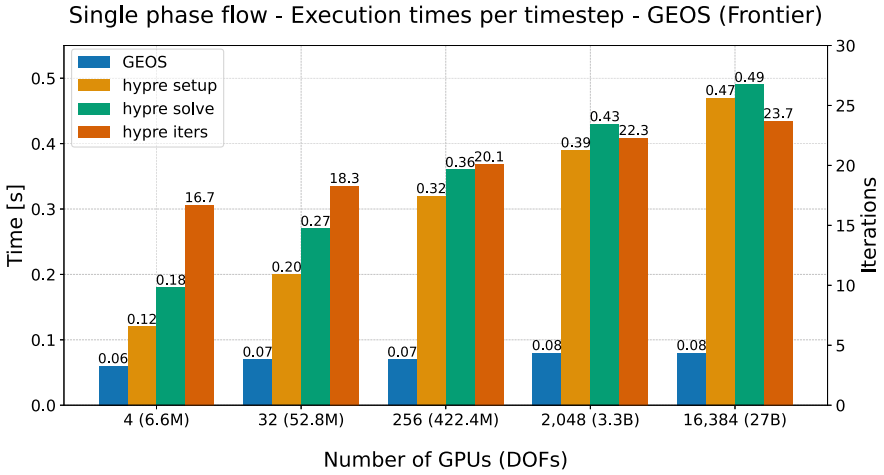


Fig. 5 Average execution time (left y-axis) and iteration count (right y-axis) per time step for a single-phase flow simulation in GEOS. As the number of GPUs and DOFs increase by a factor of 4096, both *hypre* setup and solve times increase slightly by a factor less than 4, demonstrating the excellent weak scalability of *hypre*'s algebraic multigrid solver (BoomerAMG)

tion overhead required to build the preconditioner with larger compute node counts. Lastly, *hypre*'s (BoomerAMG-GMRES) solve time shows a gradual increase from 0.18 to 0.50 s, maintaining a relatively consistent performance despite the growing complexity. We highlight that the largest scale run used a little less than one-forth of the whole machine.

4.2.2 Strong Scaling Thermo-Hydro-Mechanical (THM) Model

This test problem uses *hypre*'s MGR as a preconditioner to GMRES to solve a real-world simulation of underground carbon storage and sequestration (CSS) [9, Sect. 4.4] including thermal effects. The primary variable types of this multiphysics problem are seven: three displacement degrees of freedom, two component densities (water and gas), one pressure, and one temperature. The MGR strategy consists of four levels. In the first level, the displacement degrees of freedom are eliminated, followed by the water component density, and then gas component density. At the last level, pressure and temperature are treated together since, in this test problem, they share elliptic properties that are amenable to AMG. To address the pressure-temperature coarse grid problem, we apply a single V(1,1)-cycle of an unknown-based BoomerAMG [5]. This approach improves setup and solution times by creating an algebraic multigrid hierarchy with interpolation operators that do not take into account inter-variable couplings, in this case, pressure and temperature degrees of freedom. For more information on the isothermal version (without temperature) of this MGR preconditioning technique, see [9, Sect. 3].

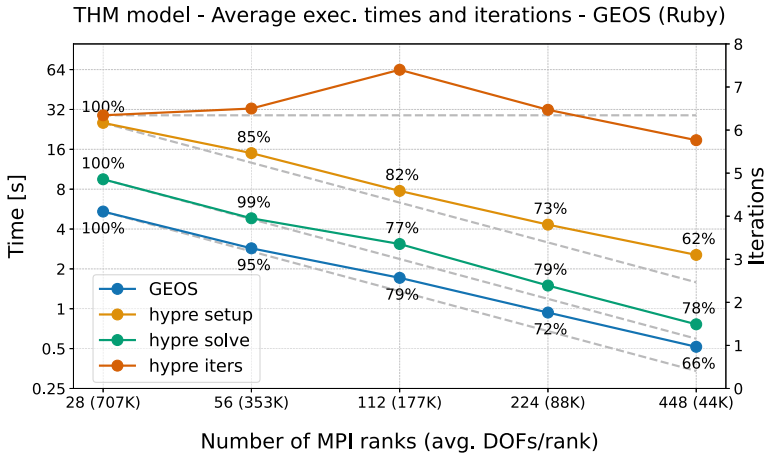


Fig. 6 Strong scaling results for a THM simulation in GEOS, showing average execution times (left y-axis) and iterations (right y-axis) for different numbers of MPI ranks on Ruby. Both *hypre* phases and GEOS exhibit parallel efficiencies of up to nearly 70% after a 16-fold increase in computational resources

Figure 6 shows strong scaling results obtained using Ruby, an HPC cluster at LLNL equipped with 2 × 28-core Intel® Xeon® Platinum 8276 CPUs. The results indicate that as the number of MPI ranks increases, there is a significant reduction in the total execution time for the application and *hypre* times (setup and solve), demonstrating efficient parallel scalability. Notably, the execution time decreases almost linearly with the increase in MPI ranks, maintaining high efficiency percentages across different configurations. For instance, the *hypre* setup time drops from approximately 32 s with 28 ranks to around 2 s with 448 ranks, reflecting a 62% efficiency at the highest rank count. Similarly, the *hypre* solve and GEOS times exhibit substantial reductions, achieving around 66 and 78% parallel efficiency, respectively, at the maximum rank count (448). The average number of iterations required for convergence remains relatively stable, with a slight decrease, highlighting the robustness of the MGR preconditioner. These results demonstrate *hypre*'s scalability and effectiveness in leveraging high-performance computing resources for large-scale multiphysics simulations.

4.3 Chombo[4]

Chombo is a software framework designed for CFD simulations on block-structured and adaptively refined grids. Chombo is capable of handling complex geometries and dynamic mesh refinement, making it suitable for high-resolution simulations

[4] https://commons.lbl.gov/display/chombo.

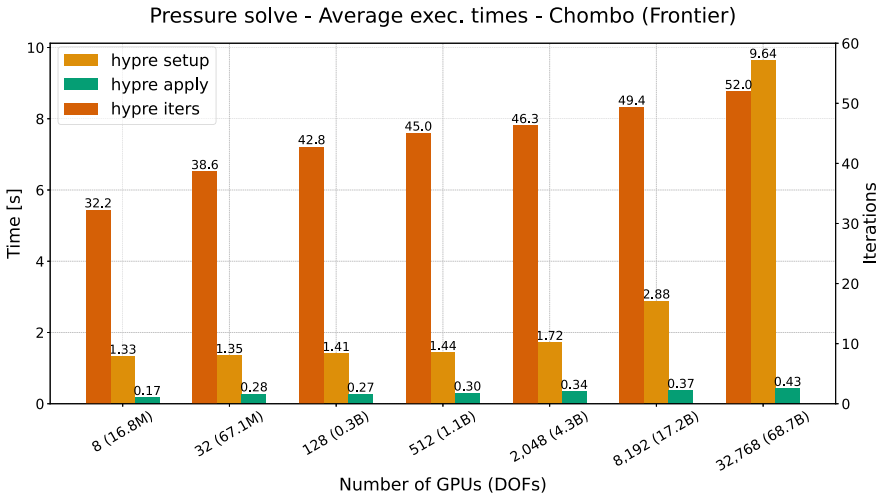


Fig. 7 Average execution time (left y-axis) and iteration count (right y-axis) per linear system for the pressure projection solve in Chombo. As the problem size scales up, the execution times for both setup and solve phases show moderate increase, highlighting *hypre*'s performance and scalability in handling large-scale elliptic systems arising from computational fluid dynamics simulations

at various Reynolds numbers capturing both laminar and turbulent flow regimes. BoomerAMG is used by Chombo through PETSc [6] as a preconditioner to GMRES in order to solve its pressure projection equation.

Figure 7 illustrates the weak scaling results obtained on Frontier (ORNL) for simulating flow inside a cylinder packed with spheres [26]. The chart shows BoomerAMG's construction time (*hypre* setup), which is performed only once at the start of the simulation; average application time (*hypre* apply); and average iteration counts (*hypre* iters) per linear system across different numbers of GPUs, ranging from 8 to 32,768 (almost half of the entire machine). As the number of GPUs increases, the preconditioner application times and iterations scales very well, increasing by only a factor of 2.5 and 1.6, respectively, despite a 4,096-fold increase in the number of GPUs. BoomerAMG setup times shows excellent scaling up to 2,048 GPUs, although performance degrades beyond this point, which requires further investigation.

5 Conclusions

We analyzed the impact of three different *hypre* solvers on three HPC systems: Ruby at LLNL, Summit, and Frontier at ORNL. For the applications presented here, solving linear systems consistently took up more than half of the total simulation time, emphasizing the crucial role of efficient solvers for sparse linear systems in

HPC simulation codes. Additionally, *hypre* achieves good strong and weak scaling for these applications, effectively leveraging tens of thousands of GPUs.

As we enter the era of exascale computing, optimizing solver performance becomes increasingly important. The insights gained from these applications and many others will inform future developments in *hypre* to reduce computational costs and enhance scalability at large node counts. Continued efforts will focus on refining *hypre* solvers to fully utilize the potential of exascale systems, particularly for the El Capitan system at LLNL, which is based on AMD accelerated processing units. This will ultimately enable more sophisticated and accurate scientific simulations at faster execution times and finer resolutions.

Acknowledgements This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This material is based upon work supported by the Exascale Computing Project, a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. Support for this work was provided in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) Institute.

Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this chapter.

References

1. Thrust: The C++ parallel algorithms library (n.d.). https://nvidia.github.io/cccl/thrust/
2. Ali, A., Brannick, J.J., Kahl, K., Krzysik, O.A., Schroder, J.B., Southworth, B.S.: Constrained local approximate ideal restriction for advection-diffusion problems. SIAM J. Sci. Comput. **0**(0), S96–S122 (2024). https://doi.org/10.1137/23M1583442. https://doi.org/10.1137/23M1583442
3. Ashby, S., Falgout, R.: A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. Nucl. Sci. Eng. **124**, 145–159 (1996). https://doi.org/10.13182/NSE96-A24230
4. Baker, A., Falgout, R., Kolev, T., Yang, U.M.: Multigrid smoothers for ultraparallel computing. SIAM J. Sci. Comput. **33**, 2864–2887 (2011). https://doi.org/10.1137/100798806
5. Baker, A.H., Kolev, T.V., Yang, U.M.: Improving algebraic multigrid interpolation operators for linear elasticity problems. Numer. Linear Algebr. Appl. **17**(2–3), 495–517 (2010)
6. Balay, S., Abhyankar, S., Adams, M.F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E.M., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W.D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M.G., Kong, F., Kruger, S., May, D.A., McInnes, L.C., Mills, R.T., Mitchell, L., Munson, T., Roman, J.E., Rupp, K., Sanan, P., Sarich, J., Smith, B.F., Zampini, S., Zhang, H., Zhang, H., Zhang, J.: PETSc Web page (2024). https://petsc.org/. https://petsc.org/
7. Bui, Q., Hamon, F., Castelletto, N., Osei-Kuffuor, D., Settgast, R., White, J.: Multigrid reduction preconditioning framework for coupled processes in porous and fractured media. Comput. Methods Appl. Mech. Eng. **387**, 114111 (2021). https://doi.org/10.1016/j.cma.2021.114111
8. Bui, Q.M., Osei-Kuffuor, D., Castelletto, N., White, J.A.: A scalable multigrid reduction framework for multiphase poromechanics of heterogeneous media. SIAM J. Sci. Comput. **42**(2), B379–B396 (2020)

9. Cremon, M.A., Franc, J., Hamon, F.P.: Constrained pressure-temperature residual (CPTR) preconditioner performance for large-scale thermal CO_2 injection simulation. Comput. Geosci. (2024)
10. De Sterck, H., Falgout, R.D., Nolting, J.W., Yang, U.M.: Distance-two interpolation for parallel algebraic multigrid. Numer. Linear Algebr. Appl. **15**(2–3), 115–139 (2008). https://doi.org/10.1002/nla.559
11. De Sterck, H., Yang, U.M., Heys, J.J.: Reducing complexity in parallel algebraic multigrid preconditioners. SIAM J. Matrix Anal. Appl. **27**(4), 1019–1039 (2006). https://doi.org/10.1137/040615729
12. Falgout, R., Jones, J., Yang, U.: The design and implementation of hypre, a library of parallel high performance preconditioners. In: Bruaset, A., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers, pp. 267–294. Springer (2006)
13. Falgout, R.D., Friedhoff, S., Kolev, T.V., MacLachlan, S.P., Schroder, J.B.: Parallel time integration with multigrid. SIAM J. Sci. Comput. **36**(6), C635–C661 (2014). https://doi.org/10.1137/130944230
14. Falgout, R.D., Li, R., Sjögreen, B., Wang, L., Yang, U.M.: Porting hypre to heterogeneous computer architectures: Strategies and experiences. Parallel Comput. **108**, 102840 (2021)
15. Hanophy, J., Southworth, B.S., Ruipeng Li, T.M., Morel, J.: Parallel approximate ideal restriction multigrid for solving the SN transport equations. Nucl. Sci. Eng. **194**(11), 989–1008 (2020)https://doi.org/10.1080/00295639.2020.1747263. https://doi.org/10.1080/00295639.2020.1747263
16. Henson, V.E., Yang, U.M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. **41**(1), 155–177 (2002). https://doi.org/10.1016/S0168-9274(01)00115-5
17. Holmen, J.K., Sahasrabudhe, D., Berzins, M.: Porting uintah to heterogeneous systems. In: PASC22: Proceedings of the Platform for Advanced Scientific Computing Conference, pp. 1–10 (2022). https://doi.org/10.1145/3539781.3539794
18. hypre: High performance preconditioners (n.d.). https://llnl.gov/casc/hypre. https://github.com/hypre-space/hypre
19. Kolev, T.V., Vassilevski, P.S.: Parallel auxiliary space AMG for $H(curl)$ problems. J. Comput. Math. **27**, 604–623 (2009). https://doi.org/10.4208/jcm.2009.27.5.013
20. Kolev, T.V., Vassilevski, P.S.: Parallel auxiliary space AMG solver for $H(div)$ problems. SIAM J. Sci. Comput. **34**(6), A3079–A3098 (2012). https://doi.org/10.1137/110859361
21. Li, R., Sjögreen, B., Yang, U.M.: A new class of AMG interpolation methods based on matrix-matrix multiplications. SIAM J. Sci. Comput. **43**(5), S540–S564 (2021)
22. MacLachlan, S., Manteuffel, T., McCormick, S.: Adaptive reduction-based AMG. Numer. Linear Algebr. Appl. **13**(8), 599–620 (2006)
23. Manteuffel, T.A., Ruge, J., Southworth, B.S.: Nonsymmetric algebraic multigrid based on local approximate ideal restriction (ℓAIR). SIAM J. Sci. Comput. **40**(6), A4105–A4130 (2018)
24. Ries, M., Trottenberg, U., Winter, G.: A note on MGR methods. Linear Algebr. Its Appl. **49**, 1–26 (1983). https://doi.org/10.1016/0024-3795(83)90091-5
25. Schäling, B.: The Boost C++ libraries. XML Press (2014)
26. Trebotich, D.: Exascale Computational Fluid Dynamics in Heterogeneous Systems. J. Fluids Eng. **146**(4), 041104 (2024)
27. Wang, L., Osei-Kuffuor, D., Falgout, R.D., Mishev, I.D., Li, J.: Multigrid reduction for coupled flow problems with application to reservoir simulation. In: SPE Reservoir Simulation Conference. Society of Petroleum Engineers, SPE-182723-MS (2017). https://doi.org/10.2118/182723-MS
28. Zaman, T., Nytko, N., Taghibakhshi, A., MacLachlan, S., Olson, L., West, M.: Generalizing reduction-based algebraic multigrid. Numer. Linear Algebr. Appl. **31**(3), e2543 (2024)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Flexible Multi-Dimensional FFTs for Plane Wave Density Functional Theory Codes



Doru Thom Popovici, Mauro Del Ben, Osni Marques, and Andrew Canning

Abstract Multi-dimensional Fourier transforms are key mathematical building blocks that appear in a wide range of applications from materials science, physics, chemistry and even machine learning. Over the past years, a multitude of software packages targeting distributed multi-dimensional Fourier transforms have been developed. Most variants attempt to offer efficient implementations for single transforms applied on data mapped onto rectangular grids. However, not all scientific applications conform to this pattern, i.e. plane wave Density Functional Theory codes require multi-dimensional Fourier transforms applied on data represented as batches of spheres. Typically, the implementations for this use case are hand-coded and tailored for the requirements of each application. In this work, we present the Fastest Fourier Transform from Berkeley (FFTB) a distributed framework that offers flexible implementations for both regular/non-regular data grids and batched/non-batched transforms. We provide a flexible implementations with a user-friendly API that captures most of the use cases. Furthermore, we provide implementations for both CPU and GPU platforms, showing that our approach offers improved execution time and scalability on the HP Cray EX NVIDIA A100 supercomputer. In addition, we outline the need for a flexible implementation for different use cases of the software package.

Keywords Distributed computing · Fourier transforms · Data layouts · Communication minimization · Tensor operations

D. T. Popovici (✉) · M. D. Ben · O. Marques · A. Canning
Lawrence Berkeley National Lab, Berkeley, USA
e-mail: dtpopovici@lbl.gov

M. D. Ben
e-mail: mdelben@lbl.gov

O. Marques
e-mail: oamarques@lbl.gov

A. Canning
e-mail: acanning@lbl.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_5

1 Introduction

The discrete Fourier transform has proven to be an important mathematical kernel that is widely used in a multitude of applications. For example, applications from molecular dynamics [17–19], material sciences [12–14], [27, 28], quantum mechanics [3, 10, 23, 26] and even machine learning [15, 29] require flexible and efficient implementations of multi-dimensional Fourier transforms that can be executed on parallel and distributed systems. In recent years, a multitude of software packages such as FFTE [24], heFFTe [2], elemental FFT [22], FFTU [11] have attempted to offer efficient implementations targeting multi-node CPU and/or GPU systems. Table 1 outlines the different software packages and some of their characteristics. Most frameworks focus on single three-dimensional Fourier transforms applied on data mapped to cuboid grids. However, some scientific applications such as plane wave Density Functional Theory calculations do not fit within this mold. Density Functional Theory codes require batched three-dimensional transforms applied on non-cuboid shaped data, hence demanding specialized implementations.

First-principles Density Functional Theory calculations in the Kohn-Sham formalism are widely used to study the electronic structure of materials. The Kohn-Sham single particle equation is expressed as

$$\hat{H}\psi_i(r) = \epsilon_i \psi_i(r), \tag{1}$$

where $\psi_i(r)$ is the i-th electron orbital (wavefunction) in the system, ϵ_i represents the electron orbital's eigen energy and $\hat{H}$ is the Hamiltonian of the system describing the different interactions. Each wavefunction spans the entire 3D real space r. However, solving Eq. 1 requires the wavefunctions either to be discretized on a real space grid or expressed in some basis set. One of the most widely used basis sets is the plane wave basis set [4], which discretizes the wavefunctions using a Fourier series expansion. It

Table 1 The table contains different implementations for distributed Fourier transforms and some of their characteristics. We outline four characteristics: the transform type (complex to complex (CtoC) or real to complex (RtoC), the input/output type (cuboid or sphere data), the shape of the processing grid (1D, 2D, or 3D), the batching mechanism (present or not)

Software	Platform	Transform type	Input/Output type	Processing grid	Batching
FFTE [25]	CPU	CtoC/RtoC	Cuboid	1D/2D	✗
heFFTe [2]	CPU/GPU	CtoC/RtoC	Cuboid	1D/2D	✗
FFTX [6]	CPU/GPU	CtoC/RtoC	Cuboid	1D/2D	✗
FFTU [11]	CPU	CtoC	Cuboid	1D/2D	✗
elemental FFT [22]	CPU	CtoC	Cuboid	1D/2D/3D	✗
FFTB (ours)	CPU/GPU	CtoC	Cuboid/Sphere	1D/2D/3D	✓

is conventional to truncate the plane wave basis set within an energy cutoff and thus only frequency coefficients within a cut-off sphere are preserved for computation. Given that some operations applied on the wavefunctions are cheaper in real space, inverse and forward Fourier transforms are required to change from frequency to real space and back. The three-dimensional Fourier transforms need extra steps to deal with the non-cuboid shape of the data. In addition, the transform is not applied on a single wavefunction, it is rather applied on multiple wavefunctions. As such the Fourier transform can be applied on batches of data points which will provide better communication over the network.

Most plane wave Density Functional Theory codes [1, 7–9] either implement their own Fourier transform variant or resort to padding the spheres to cuboid grids and use off-the-shelf frameworks. The first approach provides efficient implementations, at the cost of having a rigid solution that is specifically tailored to each application. The second approach trades performance for flexibility, requiring redundant computation and communications. As outlined in the result section, padding the data and applying the full Fourier transform will incur significant performance penalties due to the large volume of data that needs to be communicated over the network, some of that data being all zeros. In this work, we present a flexible implementation that allows users to express different input shapes and then efficiently applies the computation and data movement, targeting both CPUs and GPUs. Furthermore, we analyze key aspects of our approach and then show scaling results for our implementation given certain use cases from Density Functional Theory codes.

Contributions. This work makes the following contributions:

1. We present FFTB (the Fastest Fourier Transform from Berkeley) a distributed multi-dimensional Fourier transform for classical and plane wave Density Functional Theory codes.
2. We present a flexible implementation that allows cubic and spherical shapes of the data to be easily expressed by users.
3. We present an efficient implementation capable of excellent parallel scalability on both CPU and GPU distributed systems. We show results on an NVIDIA-based supercomputer and are working to extend the framework to AMD and Intel-based systems as well.

2 Background

In this section, we briefly present the Fourier transform, focusing on the three-dimensional implementation. We give a brief introduction to the classical three-dimensional transform applied on data stored on cuboid grids, and then we discuss the plane wave Fourier transform applied on batches of spheres.

2.1 The Fourier Transform

The discrete one-dimensional Fourier transform (we use the DFT abbreviation for the discrete Fourier transform) is a matrix-vector multiplication, where given the input x of size n, the output y is obtained as

$$y = DFT_n \cdot x. \tag{2}$$

The DFT_n is the $n \times n$ Fourier dense matrix, defined as

$$DFT_n = \left[\omega_n^{lk}\right]_{0 \le l,k < n}, \tag{3}$$

where $\omega_n = e^{-j\frac{2\pi}{n}}$ is the complex root of unity. The computation of the DFT is implemented using the Fast Fourier Transform (DFT), where instead of performing $O(n^2)$ complex arithmetic operations by doing the matrix-vector multiplication, a recursive decomposition of the DFT matrix is performed to obtain an $O(n \log(n))$ algorithm. The most widely known of these algorithms is the Cooley-Tukey algorithm [5], which can be described concisely as a factorization of the DFT_n matrix for $n = n_0 \times n_1$. For more details on the decomposition, notation and implementations, we recommend the reader follow the works [20–22].

Multi-dimensional Fourier transform can also be expressed as a matrix-vector multiplication. A three-dimensional DFT of size $n_0 \times n_1 \times n_2$ is expressed as

$$y = DFT_{n_0 \times n_1 \times n_2} \cdot x, \tag{4}$$

where the $DFT_{n_0 \times n_1 \times n_2}$ represents the complex DFT matrix. Any multi-dimensional DFT can be expressed in terms of multiple 1D DFTs and/or multi-dimensional DFTs of lower dimensions, which themselves are expressed as 1D DFTs. For example, Fig. 1 shows (a) the slab-pencil algorithm that decomposes the 3D DFT into a batch of 2D DFTs followed by a batch of multiple 1D DFTs, and (b) the pencil-pencil-pencil algorithm that splits the 3D DFT into three batches of 1D DFTs, where each 1D DFT is applied in the corresponding three dimensions. The matrix-vector notation can be extended to one dimensional Fourier transforms applied on higher dimensional tensors.

The slab-pencil algorithm sees the input (output) column vectors x (y) as two dimensional matrices $\tilde{x}$ ($\tilde{y}$) of size $(n_0 \times n_1) \times n_2$. The Fourier computation can mathematically be expressed as

$$\tilde{y} = \left(DFT_{n_0 \times n_1} \cdot \tilde{x}\right) \cdot DFT_{n_2} \tag{5}$$

Data is stored in column major, hence the 2D DFT is applied on the columns, while the 1D DFT is applied on the rows. Subsequently, the 2D DFT applied can further be decomposed as 1D DFTs. Each column from matrix $\tilde{x}$ can be viewed as a two-dimensional matrix of size $n_0 \times n_1$.

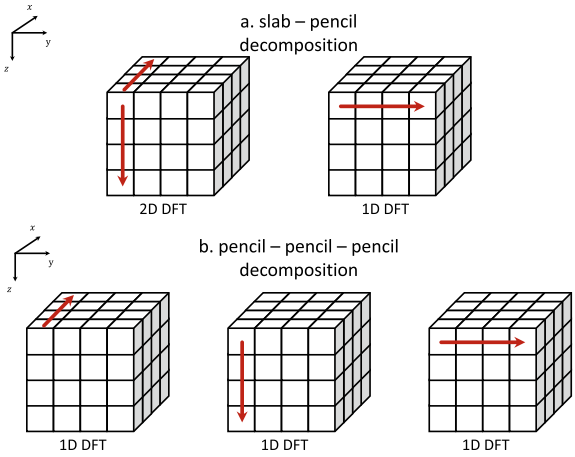


Fig. 1 Two algorithms used to compute a three dimensional Fourier transform. The top algorithm decomposes the computation as a batch of 2D transforms applied in the xy-plane and a batch of 1D transforms applied in the z dimension. The bottom algorithm, applies the three-dimensional Fourier transform as three groups of 1D Fourier transforms applied in each dimension of the input three dimensional tensors

The pencil-pencil-pencil algorithm views the input (output) vectors as 3D cubes $\hat{x}$ ($\hat{y}$) of size $n_0 \times n_1 \times n_2$. The input (output) column vector x (y) is decomposed into n_2 groups of n_1 subgroups of size n_0. Hence, the three-dimensional cube $\hat{x}$ is viewed as a matrix of matrices such that

$$\hat{x} = \left[\tilde{x}_0 | \ \tilde{x}_1 | \ \ldots | \ \tilde{x}_{n_2-1}\right], \tag{6}$$

where each $\tilde{x}_i$ is the 2D matrix of size $n_0 \times n_1$ for all values $0 \leq i < n_2$. Mathematically, the pencil-pencil-pencil algorithm is expressed as

$$\begin{aligned} \tilde{t}_i &= \left(DFT_{n_0} \cdot \tilde{x}_i\right) \cdot DFT_{n_1}, \forall 0 \leq i < n_2 \\ \hat{y} &= \left[\tilde{t}_0 \mid \tilde{t}_1 \mid \ldots \mid \tilde{t}_{n_2-1}\right] \cdot DFT_{n_2} \end{aligned} \tag{7}$$

where the DFT_{n_0} is applied in the depth dimension, the DFT_{n_1} is applied in the column dimension and finally the DFT_{n_2} is applied in the row dimension. Data is stored in column major order and therefore the dimension corresponding to n_0 is laid out in the fastest dimension in memory, while the dimension corresponding to n_2 is laid out in the slowest dimension in memory. The order in which the 1D and/or 2D DFTs are applied can be permuted for both the slab-pencil and pencil-pencil algorithms.

2.2 The Plane Wave Fourier Transform

Solving Eq. 1 requires the wavefunctions to be discretized on a real space grid or expressed in some basis set. One of the most widely used basis sets is the plane wave basis set [4]. For the simplest case, the discretization of the $\psi_i(r)$ can be expressed as a Fourier series expansion such that

$$\psi_i(r) = \sum_g c_i(g) e^{jgr}, \tag{8}$$

where g represents the reciprocal lattice vectors of the supercell to be studied and $j = \sqrt{-1}$. The $c_i(g)$ terms represent the plane wave complex coefficients. It is conventional to truncate the plane wave basis set within an energy cutoff E_{cut} such that

$$\frac{1}{2}|g|^2 \leq E_{cut}. \tag{9}$$

Therefore, only the g vectors within a cut-off sphere are used in the above expansion as shown in Fig. 2. The corresponding complex coefficients are stored as a column vector ψ_i. In addition, an offset array is typically constructed to specify the correct location of the data in the sphere. The same offset array can be used for all wavefunctions ψ_i. Equation 1 can be solved for all ψ_i wavefunctions using a Conjugate Gradient (CG) algorithm (there are other algorithms that have been used to determine the solutions). The same CG steps are applied for all the wavefunctions ψ_i. Therefore, the wavefunctions can be batched together such that

$$\Psi = \left[\psi_0 \mid \psi_1 \mid \ldots \mid \psi_{N_b-1}\right], \tag{10}$$

where N_b represents the number of wavefunctions or the number of bands. The all-band algorithm converts most of the computation steps in the CG algorithm

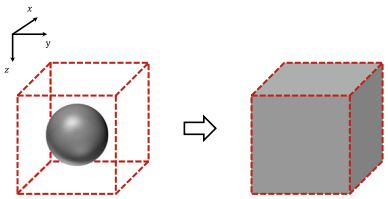


Fig. 2 Each wavefunction is decomposed using a Fourier series expansion and only the complex coefficients within a cut-off sphere are kept. The 3D Fourier computation requires the data to be on a cuboid grid. As such, the data can be padded with zeros to a cube typically of width twice the diameter of the sphere

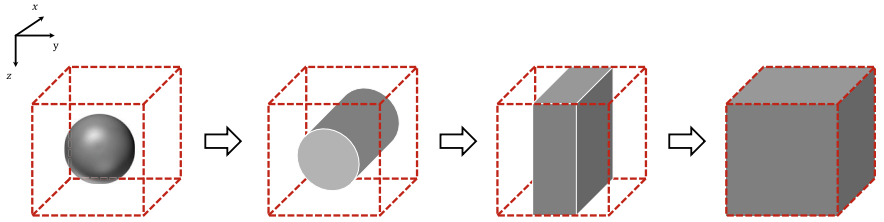


Fig. 3 The padding operation can also be split by dimensions. The padding is done in the x-dimension first, followed by the y-dimension and z-dimension. The 3D Fourier transform has a similar decomposition, so after each padding operation, the 1D transform can be immediately applied. Exploiting the structure of the data will reduce the amount of data that is being communicated and computed upon

from matrix-vector multiplication to matrix-matrix multiplications. Furthermore, the Fourier transforms are applied on batches of data and not on a single column vector at a time.

The Fourier transform requires data to be placed on a cuboid grid. As such, all the wavefunction spheres must be zero-padded. Figure 2 shows the case that zero-pads the entire sphere by embedding it into a cube of width twice the size of the diameter of the sphere (this is a requirement from the solvers). The 3D Fourier computation is applied on the entire cube. While the advantage of this approach is that off-the-shelf libraries can be used to compute the 3D Fourier transform, the disadvantage is that the amount of data is increased by almost 16 times, affecting computation and data movement. Alternatively, the zero-padding can be done in stages as outlined in Fig. 3, where each dimension is gradually padded with zeros. Splitting the 3D Fourier transform into the corresponding 1D transforms, enables us to fuse the 1D transforms with the padding operations. The advantage of this approach is that it keeps the amount of redundant work to a minimum. However, none of the classical implementations can provide support for this implementation.

2.3 Connections to Multi-Linear Algebra

In the work by Popovici et al. [21], it has been shown that Fourier transforms can be expressed as linear algebra operations. Similarly, padding or pruning the data can be expressed using a matrix notation. As such, utilizing a unifying notation for multiple computation steps, enabled the possibility of fusing the padding operations, the Fourier transforms and the linear algebra operations. This in turn provided a boost in performance on shared memory systems. In the work by Popovici et al. [22], a flexible implementation for distributed 3D Fourier transforms has been presented. The approach expressed the 3D Fourier transforms as multi-linear algebra operations and borrowed techniques from tensor community to provide a distributed Fourier transform using 3D volumetric decomposition on large number of CPUs. Finally,

the work by Franchetti el al [6] has provided an API for Fourier-based operations seen in applications like Density Functional Theory calculations, plasma beam simulations [27] or methods of local corrections [16], with the goal of enabling cross domain optimizations. While the API offered functionality for expressing padding operations, it did not provide an easy to use API for spherical data shapes. As such, we extend the APIs and develop our own distributed Fourier framework that targets both cuboid and non-cuboid data shapes.

3 A Flexible Framework for Multi-Dimensional Fourier Transforms

In this section, we describe our framework entitled the Fastest Fourier Transform from Berkeley (FFTB). We provide an overview of our framework outlining the key components. Then we present the capabilities of our approach by providing some simple examples for distributed 3D Fourier transforms.

3.1 Overview

We develop a modular implementation for the FFTB framework. In the following paragraphs, we will describe each component outlined in Fig. 4, outlining the key features of each module. We show how the framework can be extended to different platforms.

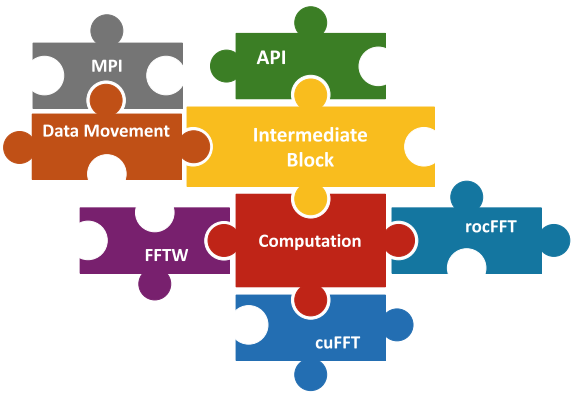


Fig. 4 The structure of FFTB. The API (green block) contains the main functionalities for describing distributed Fourier transforms. The intermediate block (yellow block) creates and links the stages of the Fourier transform based on the distribution of the inputs/outputs. The main stages are either local computation stages (red block) or data movement stages (orange block)

Computational Layout Descriptor. We provide a C/C++ library to describe distributed multi-dimensional Fourier transforms and are currently working on providing an extension for Fortran. The API provides the necessary functionalities to specify the processing grid, the input and output tensors and how they are distributed across the processing grid. Finally, the API specifies the necessary constructs to create, execute and clean-up multi-dimensional Fourier transforms. In the following sub-sections, we provide two examples. The first example focuses on the classical 3D Fourier transform for data stored on cuboid grids. The second example shows the extensions needed to express batched Fourier transforms specific for the plane wave Density Functional Theory codes.

Distributed Fourier Transform Creation. The description of the distributed Fourier transform is translated to an intermediate representation. The intermediate block (yellow block) analyses the distribution patterns of the input/output tensors and constructs the necessary compute and communicate stages. Basically, this part stitches together the end-to-end implementation of the required Fourier transform. The current FFTB implementation accepts some predefined patterns that are used in most scientific applications. The framework will raise an exception if the provided patterns are not within the predefined list. We leave as future work the mechanisms used to decide the steps required to proceed from one tensor distribution to another. For the current implementation, we provide a hard-coded approach.

Local Computation and Data Movement Specification. The local computation is represented by the 1D or 2D Fourier transforms that are applied on the data. We abstract the computation and create a couple of functions to specify in which dimensions the 1D or 2D transforms are applied in. The abstractions are replaced with actual function calls from off-the-shelf libraries like FFTW, cuFFT and rocFFT. Similarly, data movement is also abstracted away (typically, Fourier transforms required `alltoall` MPI collectives). We provide compilation flags and let users specify what libraries they want their libraries to be linked against.

In the following sub-sections, we provide two examples on how to use the API. We provide two examples for describing distributed Fourier transforms. We start with the 3D Fourier transform applied on cuboid grids and continue with a batched 3D transform on non-cuboid data (wavefunction spheres).

3.2 An API to Express Fourier Transforms on Processing Grids

Figure 5 depicts a 3D Fourier transform applied on three-dimensional tensors distributed across a given processing grid. The input/output tensors are defined by specifying the bound domains, like the approach presented by the FFTX project [6]. Each domain is defined by specifying the points corresponding to opposite corners of cuboid volume, i.e. (x_l, y_l, z_l) and (x_u, y_u, z_u) for the input tensor and (X_l, X_l, X_l) and (X_u, X_u, X_u) for the output tensor. The tensors are created by specifying the

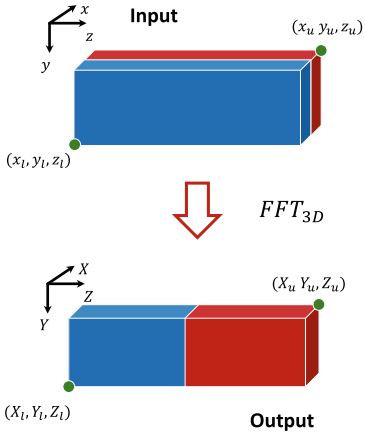


Fig. 5 A 3D Fourier transform applied on an input tensor distributed in the x-dimension. The result of the Fourier transform is a tensor that is distributed in the z-dimension

domains and the distribution across the processing grid. The input tensor is distributed in the x-dimension, while the output tensor is distributed in the z-dimension. The grid, the tensors and their distribution are sufficient to create the corresponding distributed 3D Fourier transform.

Figure 6 describes the step by step process for creating the processing grid, specifying the input and output tensors, and creating the Fourier transform. First, lines 2-3 specify the creation of the processing grid. We create a 1D grid containing 16 processors. Second, lines 6-11 specify the creation and distribution of the input tensor `ti`. We create the lower and upper points of the bounding domain (lines 6-7). We then create the domain on lines 9-10. Finally, we create the distributed tensor. The tensor takes the domain, a string specifying the dimensions and the distribution and the processing grid `g`. In the example, we specify that the tensor `ti` is a three-dimensional tensor with dimensions x, y, and z and only the x dimension is distributed across the 0-th dimension of the processing grid. We use the elemental cyclic distribution described in [22] to distribute the data (data in each dimension is distributed in a round robin fashion at the granularity of one element). The current implementation does not support the blocked distribution.

Thirdly, we describe the output tensor `to` in a similar manner (lines 14-19). While the input tensor `ti` is distributed in the x-dimension, the output tensor `to` is distributed in the z-dimension. Finally, lines 22–23 outline the code snippet that creates the distributed Fourier transform. We specify the size, the input and output tensors and the grid. From this specification, the framework decides on the most suited implementation. Once the Fourier transform is constructed, the `fx` object can be used to execute the computation.

```
// create processing grid
std::vector<int> procs{16};
grid g = grid(procs, MPI_COMM_WORLD);

// create input tensor
std::vector<int> point_in_lower{0, 0, 0};
std::vector<int> point_in_upper{255, 255, 255};

std::vector<domain> dom_in;
dom_in.push_back(domain(point_in_lower, point_in_upper));
tensor ti = tensor(dom_in, "x{0} y z", g);

// create output tensor
std::vector<int> point_out_lower{0, 0, 0};
std::vector<int> point_out_upper{255, 255, 255};

std::vector<domain> dom_out;
dom_out.push_back(domain(point_out_lower, point_out_upper));
tensor to = tensor(dom_out, "X Y Z{0}", g);

// create fft operation
std::vector<int> size{256, 256, 256};
fftb fx = fftb(sizes, to, "X Y Z", ti, "x y z", g);

// execute the code
fx.execute();
```

Fig. 6 Code snippet used to describe a distributed 3D Fourier transform of size 256^3. The first lines describe the creation of the processing grid, the next lines specify the input and output tensors and how they are distributed, the last lines specify the creation of a 3D Fourier transform. We require the sizes of the Fourier transform to be specified in the case that the input/output require padding/pruning of the data

3.3 Extensions for the Plane Wave Fourier Transform

The wavefunctions used in Density Functional Theory calculations are discretized using a Fourier series expansion. Typically, not all Fourier coefficients are saved. Those coefficients for which the frequency is outside a cut-off energy are discarded. As such, the plane wave representation of the wavefunctions will be as that of batches of spheres as the one outlined in Fig. 7 for a single sphere. The sphere can be bounded by a domain. Furthermore, projecting the data on the xy-plane an offset array can be constructed, which resembles the Compressed Sparse Row format. Software like Quantum Espresso [7] uses such a structure to store the wavefunctions. We can use

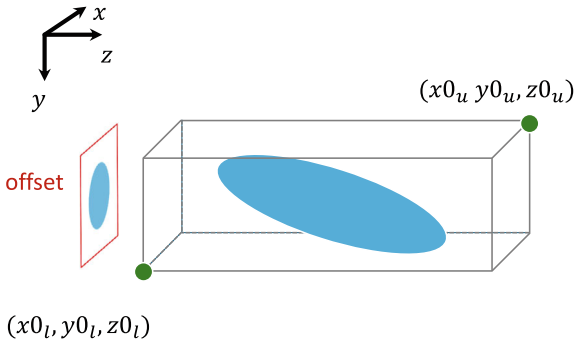


Fig. 7 Each wavefunction is discretized using a Fourier series expansion. Some of the coefficients are discarded based on the cut-off energy E_{cut}. The sphere is bounded within a bounding domain. An offset array is constructed by projecting the data points on the xy-plane. That offset array is used to access the actual data points

this information to extend the functionality of our API. This in turn will allow users to specify offset arrays within bounding domains.

Figure 8 describes the step by step process for creating the processing grid, specifying the input and output tensors, and creating the batched Fourier transform for the plane wave computation. The steps are like the ones required to create the previous example. There are some differences between the code snippet in Fig. 6 and the code snippet in Fig. 8. First, at line 2 we are required to define the offset array (we do not write the entire code and leave it as ... due to space constraints). As outlined in Fig. 7, we project the spheres on the xy-plane and use the projection as an offset array. This offset array is like a Compressed Sparse Row (CSR) format because only the z dimension is compressed, while the x and y dimensions are kept as dense. The offset array is used at line 18 to create the bounding domain for the input tensor. At line 19 we distribute the x dimension similar to the previous case. Given that, the z dimension has varying lengths, we augment the notation (extra information in the string argument) to allow for dimensions to be merged and even sorted based on the varying length in the z-dimension. These details will be described in more detail in the final release of the software.

The second difference is the presence of the batch dimension. Recall that there are multiple wavefunction spheres stacked together. At line 9-10, we specify the lower and upper points for the batch dimension. These points are used at line 17 and 26 to specify another dimension in both the input and output domains. The two domains `dom_in` and `dom_out` are specified as an array of two domains. This corresponds to a larger domain obtained as a cross product between the composing domains. Both tensors `ti` and `to` are four-dimensional tensors. Note the order in which the domains are added to `dom_in` and `dom_out`. The order in which the dimensions are added matters. In this example, we specified the batch dimension as the first dimension, which corresponds to the fastest dimension in memory. If the batch dimension was added after the three-dimensional domains, then the batch

```
// initialize offset array using a CSR-like format
offsets = ...

// create processing grid
std::vector<int> procs{16};
grid g = grid(procs, MPI_COMM_WORLD);

// create batch dimension
std::vector<int> point_b_lower{0};
std::vector<int> point_b_upper{128};

// create input tensor
std::vector<int> point_in_lower{0, 0, 0};
std::vector<int> point_in_upper{255, 255, 255};

std::vector<domain> dom_in;
dom_in.push_back(domain(point_b_lower, point_b_upper));
dom_in.push_back(domain(point_in_lower, point_in_upper, offsets));
tensor ti = tensor(dom_in, "b x{0} y z", g);

// create output tensor
std::vector<int> point_out_lower{0, 0, 0};
std::vector<int> point_out_upper{255, 255, 255};

std::vector<domain> dom_out;
dom_out.push_back(domain(point_b_lower, point_b_upper));
dom_out.push_back(domain(point_out_lower, point_out_upper));
tensor to = tensor(dom_out, "B X Y Z{0}", g);

// create fft operation
std::vector<int> size{256, 256, 256};
fftb fx = fftb(sizes, to, "X Y Z", ti, "x y z", g);

// execute the code
fx.execute();
```

Fig. 8 Code snippet used to describe a distributed planewave Fourier transform of size 256^3. The declaration is similar to the classical implementation. The main differences are specification of the batch dimension, and the addition of an offset array to the domain of the input tensor

dimension would have corresponded to the slowest dimension in memory. More details on the domain creation is in the documentation of the software release.

4 Experiments and Results

In the following section, we present experimental results obtained on the HP Cray EX supercomputer. First, we provide details about the configuration flags used to compile and execute the code. Then, we showcase scaling results for two distributed 3D Fourier transforms, outlining the need for flexible implementations.

4.1 Methodology

We use the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC) for all the experiments. Perlmutter provides two partitions users can use, one with CPU-only nodes and one with GPU-accelerated nodes. In this work, we focus on the GPU implementation and only report results on the GPU-accelerated nodes. For this partition, Perlmutter has 1792 GPU-accelerated nodes, each equipped with an AMD EPYC 7763 (Milan) CPU and four NVIDIA A100 GPUs. Furthermore, Perlmutter features the HPE Slingshot 11 interconnect fabric, which uses a 3-hop dragonfly topology for efficient data transfer. For the strong scaling results, we use up to 256 physical node and 1024 GPUs (4 GPUs per node).

We compiled our framework using the NVIDIA HPC toolkit 23.9. We use cuFFT 11 for the local computation. Furthermore, we use CUDA to implement some of the small codelets that pack and rotate the data locally on the GPU before communicating it over the network. As of now, the packing routines/codelets are part of the framework, specifcally tailored for the communication needed for the distributed Fourier transform. The goal is to place the data on the GPU and minimize copies with the host CPU. As such, we use the Cray-MPI library for performing the `alltoall` communication. On Perlmutter, the library allows users to specify device pointers to perform the communication directly on the GPUs, without data transfers to the host. In the scenario that this functionality is not available, we also provide support to move data to the host and then initiate a regular MPI `alltoall` call.

4.2 Experimental Results

We perform strong scaling experiments, where we keep the problem size fixed and increase the number of parallel processors. All experiments are run multiple times. We execute a warmup phase of 10 iterations, where we do not measure the execution time. We then execute a hot phase of another 10 iterations, where we measure the execution time. For all experiments, we take the average of the 10 iterations in the hot phase. Figure 9 shows strong scaling results for a full 3D Fourier transform and a plane wave 3D Fourier transform. We pick a typical size of 256^3 for the Fourier transform and 256 for the batches. The plane wave Fourier transform assumes that the wavefunction sphere has a diameter of 128.

The full 3D Fourier transform has four variants. The first variant (dark blue line) uses a 1D processing grid and batches the computation. The second variant (light blue line) uses a 1D processing grid and does not batch the Fourier transforms. Basically, this version loops 256 times around a distributed 3D Fourier transform. The third variant (dark orange line) uses a 2D processing grid and batches the computation. Finally, the last variant (light orange line) uses a 2D processing without batching. The plane wave 3D Fourier transform uses a 1D processing grid, batches

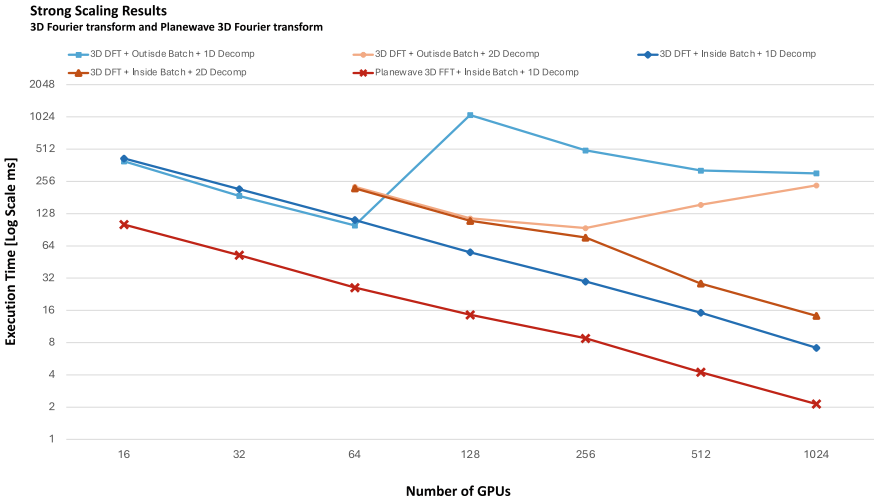


Fig. 9 Strong scaling results for different 3D Fourier transform variants implemented within our framework. The red line corresponds to the plane wave transform applied on spheres of diameter 128. The other four lines correspond to the full 3D Fourier transform applied on a cube of width of 256 using 1D or 2D processing grids and batching or not the computation.

the wavefunctions and gradually pads the data with zeros, keeping communication to a minimum. We first parallelize the data in the dimensions of the Fourier transforms. If the number of processors is greater than the dimensions, we then parallelize in the batch dimension.

There are two important aspects that need to be outlined. First, batching the computation and data movement is important. Both 3D Fourier transforms using 1D and 2D processing grids with no batching experience performance degradation as the number of GPUs is increased. For Fourier transform size of 256^3, the amount of data that is being transferred is quite small as the number of processors is increased. The light blue line suffers a big jump in execution time when increasing the number of processing units from 64 to 128 most likely because the MPI library changes the `alltoall` algorithm given the size of the data. Batching the computation and data movement alleviates this problem as outlined by the dark blue and orange lines. Second, the planewave Fourier transform scales almost linear to 1024 GPUs and provides faster execution time compared to the regular 3D Fourier transform that batches (dark blue line). The planewave Fourier transform pads the data in stages, keeping the amount of data moved across the network to a minimum.

5 Conclusion

In this work, we have presented FFTB, a flexible and extensible framework for distributed Fourier transforms targeting both CPU and GPU supercomputers. We provided a description of the framework, outlining key characteristics. We constructed FFTB as a modular framework that allows different types of Fourier transforms to be described. In the paper, we focused on classic 3D Fourier transforms applied on data mapped to cuboid grids. Furthermore, we showed that with minor changes, we can extend the framework to plane wave 3D Fourier transforms, needed by a multitude of plane wave Density Functional Theory codes. We also showed multiple implementations with different behaviors on multi-node supercomputers such as Perlmutter, outlining the need for flexible frameworks. Moving forward, we will focus on integrating our approach with current Density Functional Theory codes. In addition, we will extend the functionality to target other supercomputers like Frontier (AMD GPUs), Aurora (Intel GPUs) and Fugaku (ARM CPUs). We will bundle the implementation and all the extensions making it available on Github.

Acknowledgements This research was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Sci- entific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory. This re- search used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02- 05CH11231.

References

1. Andrade, X., Pemmaraju, C.D., Kartsev, A., Xiao, J., Lindenberg, A., Rajpurohit, S., Tan, L.Z., Ogitsu, T., Correa, A.A.: Inq, a modern gpu-accelerated computational framework for (time-dependent) density functional theory. J. Chem. Theory Comput. **17**(12), 7447–7467 (2021)
2. Ayala, A., Tomov, S., Haidar, A., Dongarra, J.: heffte: Highly efficient fft for exascale. In: International Conference on Computational Science, pp. 262–275. Springer, (2020)
3. Canning, A., Wang, L., Williamson, A., Zunger, A.: Parallel empirical pseudopotential electronic structure calculations for million atom systems. J. Comput. Phys. **160**(1), 29–41 (2000)
4. Car, R., Parrinello, M.: Unified approach for molecular dynamics and density-functional theory. Phys. Rev. Lett. **55**, 2471–2474 (1985). Nov
5. Cooley, J.W., Tukey, J.W.: An algorithm for the machine calculation of complex fourier series. Math. Comput. **19**(90), 297–301 (1965)
6. Franchetti, F., Spampinato, D.G., Kulkarni, A., Popovici, D.T., Low, T.M., Franusich, M., Canning, A., McCorquodale, P., Van Straalen, B., Colella, P.: Fftx and spectralpack: A first look. In: 2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW), pp. 18–27. IEEE, (2018)
7. Giannozzi, P., Andreussi, O., Brumme, T., Bunau, O., Nardelli, M.B., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Cococcioni, M., et al.: Advanced capabilities for materials modelling with quantum espresso. J. Phys.: Condens. Matter **29**(46), 465901 (2017)
8. Gonze, X., Amadon, B., Anglade, P.-M., Beuken, J.-M., Bottin, F., Boulanger, P., Bruneval, F., Caliste, D., Caracas, R., Côté, M., et al.: Abinit: First-principles approach to material and nanosystem properties. Comput. Phys. Commun. **180**(12), 2582–2615 (2009)

9. Gygi, F.: Architecture of qbox: A scalable first-principles molecular dynamics code. IBM J. Res. Dev. **52**(1.2):137–144, (2008)
10. Kendall, R.A., Aprà, E., Bernholdt, D.E., Bylaska, E.J., Dupuis, M., Fann, G.I., Harrison, R.J., Ju, J., Nichols, J.A., Nieplocha, J., et al.: High performance computational chemistry: An overview of NWChem a distributed parallel application. Comput. Phys. Commun. **128**(1–2), 260–283 (2000)
11. Koopman, T., Bisseling, R.H.: Minimizing communication in the multidimensional fft. SIAM J. Sci. Comput. **45**(6), C330–C347 (2023)
12. Lebensohn, R.A.: N-site modeling of a 3d viscoplastic polycrystal using fast fourier transform. Acta Mater. **49**(14), 2723–2737 (2001)
13. Lebensohn, R.A., Kanjarla, A.K., Eisenlohr, P.: An elasto-viscoplastic formulation based on fast fourier transforms for the prediction of micromechanical fields in polycrystalline materials. Int. J. Plast **32**, 59–69 (2012)
14. Lee, S.-B., Lebensohn, R., Rollett, A.D.: Modeling the viscoplastic micromechanical response of two-phase materials using fast fourier transforms. Int. J. Plast **27**(5), 707–727 (2011)
15. Li, Z., Kovachki, N., Azizzadenesheli, K., Liu, B., Bhattacharya, K., Stuart, A., Anandkumar, A.: Fourier neural operator for parametric partial differential equations. arXiv:2010.08895 (2020)
16. McCorquodale, P., Colella, P., Balls, G., Baden, S.: A local corrections algorithm for solving poisson's equation in three dimensions. Communications in Applied Mathematics and Computational Science **2**(1), 57–81 (2007)
17. Plimpton, S., Pollock, R., Stevens, M.: Particle-mesh ewald and rrespa for parallel molecular dynamics simulations. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, (1997)
18. Plimpton, S.J.: Ffts for (mostly) particle codes within the doe exascale computing project. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), (2017)
19. Plimpton, S.J., Thompson, A.P.: Computational aspects of many-body potentials. MRS Bull. **37**(5), 513–521 (2012)
20. Popovici, D.A.T.: An Approach to Specifying and Automatically Optimizing Fourier Transform Based Operations, Carnegie Mellon University (2018). PhD thesis
21. Popovici, D.T., Canning, A., Zhao, Z., Wang, L.W., Shalf, J.: A systematic approach to improving data locality across fourier transforms and linear algebra operations. In: Proceedings of the ACM International Conference on Supercomputing, pp. 329–341, (2021)
22. Popovici, D.T., Schatz, M.D., Franchetti, F., Low, T.M.: A flexible framework for multidimensional dfts. SIAM J. Sci. Comput. **42**(5), C245–C264 (2020)
23. Straatsma, T., Bylaska, E., van Dam, H., Govind, N., de Jong, W., Kowalski, K., Valiev, M.: Advances in scalable computational chemistry: NWChem. In: Annual Reports in Computational Chemistry, pp 151–177. Elsevier, (2011)
24. Takahashi, D.: An implementation of parallel 3-D FFT with 2-D decomposition on a massively parallel cluster of multi-core processors. In: Parallel Processing and Applied Mathematics, 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13–16, 2009. Revised Selected Papers, Part I, pp. 606–614, (2009)
25. Takahashi, D.: Fastest fourier transform in the east. (2018) http://www.ffte.jp/
26. Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J., Wang, D., Nieplocha, J., Apra, E., Windus, T.L., et al.: NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations. Comput. Phys. Commun. **181**(9), 1477–1489 (2010)
27. Vay, J.L., Almgren, A., Bell, J., Ge, L., Grote, D., Hogan, M., Kononenko, O., Lehe, R., Myers, A., Ng, C.: Warp-X: A new exascale computing platform for beam-plasma simulations. Nucl. Instrum. Methods Phys. Res., Sect. A, (2018)

28. Vay, J.-L., Haber, I., Godfrey, B.B.: A domain decomposition method for pseudo-spectral electromagnetic simulations of plasmas. J. Comput. Phys. **243**, 260–268 (2013)
29. Wen, G., Li, Z., Azizzadenesheli, K., Anandkumar, A., Benson, S.M.: U-fno—an enhanced fourier neural operator-based deep-learning model for multiphase flow. Adv. Water Resour. **163**, 104180 (2022)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Towards the Use of Anderson Acceleration in Coupled Transport-Gyrokinetic Turbulence Simulations



David J. Gardner, Lynda L. LoDestro, and Carol S. Woodward

Abstract Predicting the behavior of a magnetically confined fusion plasma over long time periods requires methods that can bridge the difference between transport and turbulent time scales. The nonlinear transport solver, Tango, enables simulations of very long times, in particular to steady state, by advancing each process independently with different time step sizes and couples them through a relaxed iteration scheme. We examine the use of Anderson Acceleration (AA) to reduce the total number of coupling iterations required by interfacing Tango with the AA implementation, including several extensions to AA, provided by the KINSOL nonlinear solver package in SUNDIALS. The ability to easily enable and adjust algorithmic options through KINSOL allows for rapid experimentation to evaluate different approaches with minimal effort. Additionally, we leverage the GPTune library to automate the optimization of algorithmic parameters within KINSOL. We show that AA can enable faster convergence in stiff and very stiff tests cases without noise present and in all cases, including with noisy fluxes, increases robustness and reduces sensitivity to the choice of relaxation strength.

Keywords Magnetically confined fusion plasma · Multiscale coupling · Anderson acceleration · SUNDIALS · KINSOL · Autotuning · GPTune

D. J. Gardner (✉) · L. L. LoDestro · C. S. Woodward
Lawrence Livermore National Laboratory, Livermore, CA, USA
e-mail: gardner48@llnl.gov

L. L. LoDestro
e-mail: lodestro1@llnl.gov

C. S. Woodward
e-mail: woodward6@llnl.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_6

1 Introduction

Accurately simulating turbulent transport is a critical aspect of modeling magnetically confined fusion plasmas. Turbulent fluxes have a strong nonlinear dependence on the gradients of averaged fields and are often the primary mechanism for transporting particles and energy from the plasma interior. A major challenge in simulations of very long times, in particular steady state, is the significant difference in characteristic time scales for transport (seconds) and turbulence (microseconds) in magnetic-fusion-energy burning plasmas [26]. This challenge is further complicated by the presence of noise in the averaged transport fluxes arising from turbulent simulations.

Evolving both processes at the same time step is prohibitively expensive; however, as the dynamics on each time scale are well separated, each process can be advanced independently at its own rate and coupled through an iteration scheme to bridge the scale gap. Turbulence model evaluations are the dominant computational expense, and the speed at which the coupling iteration converges determines the number of turbulence model evaluations. In this work, we aim to reduce the total number of coupling iterations necessary and thus lower the overall simulation cost by accelerating the convergence of the iterative method. To this end, we interface the nonlinear transport solver, Tango [22, 23], with the KINSOL nonlinear solver package from SUNDIALS [12, 16] to examine the use of Anderson Acceleration (AA) and extensions to AA for improving the convergence rate of the coupling iteration.

The remainder of this paper is organized as follows. In Sect. 2, we overview turbulent transport simulations for magnetically confined fusion plasmas and the coupling method applied. We introduce the AA algorithm and detail the various extensions we consider to improve performance in Sect. 3. In Sect. 4 we overview the Tango code for coupled simulations and its interfacing with the AA implementation from KINSOL as well as the use of GPTune to optimize parameter selection. In Sect. 5 we discuss results from numerical simulations with Tango in both stiff and very stiff settings along with the impact of noisy fluxes on performance. Finally, in Sect. 6, we provide concluding remarks and directions for future investigation.

2 Coupling Transport and Turbulence

Simulating turbulence-driven transport in toroidal magnetic-fusion-energy plasmas with near-first-principles accuracy requires solving a set of gyrokinetic equations describing the distribution of different species in a 5D phase space (3 physical- plus 2 velocity-space dimensions). 5D turbulence due to fast time-scale instabilities drives transport of averaged fields (i.e., density, momentum, and energy, which are constant on magnetic flux surfaces and thus spatially 1D in the magnetic flux coordinate) across the magnetic surfaces confining the plasma. Because the transport time scales are much slower than the turbulence time scales, a formal separation-of-scales

approach is possible, permitting a multiscale computational solution which proceeds by coupling separate equation sets for the fast and slow scales [27]. For this work, we utilize a simplified model for a single species to evaluate algorithms for turbulent transport without the expense of 5D simulation or the complexities of the toroidal geometry. Consider the transport equation,

$$\frac{\partial p}{\partial t} + \frac{\partial q}{\partial x} = S, \tag{1}$$

where $p(t, x)$ is the plasma profile, $q(p(t, x))$ is the average turbulent flux, and $S(t, x)$ includes all local sources. Because the turbulence-driven transport is stiff, an implicit time-advance is required. Discretizing (1) in space with finite differences and in time with backward Euler, we obtain the implicit system,

$$p^{(n+1)} + H\mathcal{D}_x q^{(n+1)} = p^{(n)} + HS^{(n+1)}, \tag{2}$$

for a time step from $t^{(n)}$ to $t^{(n+1)}$ with step size H where superscripts denote the time at which a quantity is approximated, and $\mathcal{D}_x$ is the difference operator for the finite-difference method.

In Tango, we solve the implicit system (2) using the LoDestro method as detailed in [6, 27]. The method is given by the iteration,

$$p_{k+1} - H\mathcal{D}_x(D_k\mathcal{D}_x p_{k+1} - c_k p_{k+1}) = p^{(n)} + HS^{(n+1)}, \tag{3}$$

where p_{k+1} is the new iterate for $p^{(n+1)}$ and $q^{(n+1)}$ is split into diffusive and convective terms, with coefficients

$$D_k = -\theta_k q(p_k)/\mathcal{D}_x p_k \quad \text{and} \quad c_k = (1 - \theta_k)q(p_k)/p_k, \tag{4}$$

respectively. The parameter θ_k controls the diffusive-convective split at each mesh point. The purpose of the split is to keep D_k positive (assuring a parabolic linear equation) and within bounds. It is assumed that the transport physics is diffusive at the boundaries; while it is often diffusive throughout the confined plasma volume, there are important exceptions, which a general treatment must accommodate. With a purely diffusive ($c_k = 0$) representation of q, profile maxima can cause violent spatial swings in D_k while iterating; also, converged solutions with the flux running uphill (anti-diffusion, possible in sub-volumes with global physics) are unobtainable. While Tango currently employs only the flux-splitting technique from [6, 27] to handle these complications, [6, 27] provide a second technique for representing q, which worked about equally well for the problems studied there. The important point is that however q is represented, it is not a model of the physics (which is unknown, apart from the assumption at the boundaries); yet, assuming the iteration converges, the numerical q supplied from the turbulence code must be recovered, as (3) and (4) have been constructed to do here.

The value of θ_k is chosen using the default strategy in Tango,

$$\theta_k = \begin{cases} 0 & \text{where} \quad \hat{D}_k < D_{\min} \\ \frac{1}{2}\left(1 + \frac{D_{\max} - \hat{D}_k}{D_{\max} - D_{\min}}\right) & \text{where} \quad D_{\min} \leq \hat{D}_k \leq D_{\max} \text{ and } |\mathcal{D}_x p_k| < 10 \\ \frac{1}{2} & \text{where} \quad \hat{D}_k > D_{\max} \text{ and } |\mathcal{D}_x p_k| < 10 \\ 1 & \text{otherwise} \end{cases}, \tag{5}$$

with $\hat{D}_k = -q(p_k)/\mathcal{D}_x p_k$, $D_{\min} = 10^{-5}$, and $D_{\max} = 10^{13}$. Obtaining the new profile in (3) requires solving the linear system,

$$M_k p_{k+1} = b, \tag{6}$$

within each iteration, where $M_k = I - H\mathcal{D}_x(D_k\mathcal{D}_x - c_k)$ and $b = p^{(n)} + HS^{(n+1)}$. In practice, relaxation of the iteration is required to ensure convergence; i.e., $p_{k+1} = \beta p_{k+1} + (1-\beta)p_k$. The overall method is outlined in Algorithm 1.

Algorithm 1 LoDestro Method

Input: p_0, β, and tol **Output:** p^*

1: **for** $k = 0, 1, \ldots, k_{\max}$ **do**
2: Compute the flux $q(p_k)$ and split it into D_k and c_k
3: Construct M_k and solve $M_k p_{k+1} = b$
4: Relax the profile $p_{k+1} = \beta p_{k+1} + (1-\beta)p_k$
5: **if** $\|p_{k+1} - p_k\| <$ tol **return** p_{k+1} as p^*
6: **end for**

As illustrated in Fig. 1, the coupled simulation alternates between the transport iteration and the turbulent evolution. The turbulent fluxes are obtained from a 5D global (entire confined plasma) gyrokinetic code running with an explicit step size, $h \ll H$. Within each transport iteration the models exchange information. The transport code provides profiles to the turbulence code which, running one long (on the fast time scale) simulation, then continues advancing with updated 1D profiles for a set time, after which it returns fluxes back to the transport code. The number of fast-scale time steps per data exchange is kept to a multiple of fluctuation periods, rather than the number to converge to a statistical steady state given the latest 1D profile; this provides a large savings in run-time per iteration, mitigating the slow rate of fixed-point convergence. Evolving the turbulence model requires far more computational resources and expense than solving the linear system (6) in the transport iteration. Minimizing the number of iterations for convergence is, then, the critical issue for overall performance.

References [6, 27] include applications of the LoDestro method with a two-dimensional turbulence model demonstrating speed-ups in reaching turbulent steady

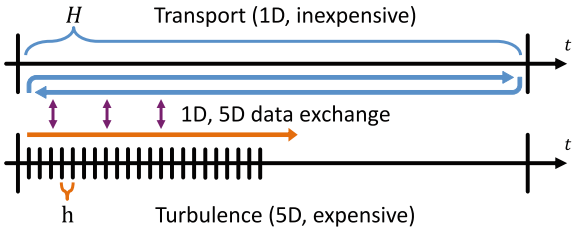


Fig. 1 In Algorithm 1, the transport code (1D) uses a large implicit time step that requires iterating to convergence. During this iteration, the turbulence model (5D for physics runs; 1D in this paper) advances with a small explicit time step. The two models periodically exchange fluxes and profiles until the iteration has converged

states. The first-ever multiscale results coupling to a global 5D gyrokinetic turbulence simulation were obtained by Tango and presented in [23]. Three-field coupling studies of current experiments are presented in [8], on the ASDEX Upgrade tokamak; and in [2], on ion-temperature clamping in the Wendelstein 7-X stellarator. The first of these includes a section on timings and speed ups.

3 Anderson Acceleration

The LoDestro method can be viewed as a damped fixed-point iteration given by

$$p_{k+1} = \beta G(p_k) + (1 - \beta) p_k, \tag{7}$$

where $G(p)$ comprises steps 2 and 3 in Algorithm 1. This iteration may converge quite slowly ($\beta \ll 1$ may be required for stability of the iterations), and, thus, we are interested in methods for accelerating convergence. One such method is Anderson Acceleration (AA) [1, 17] which computes an iteration update that approximately minimizes the linearized fixed-point residual over a subspace constructed from prior iterations. In a linear setting and under certain conditions, AA is essentially equivalent to the generalized minimal residual (GMRES) method [30], and, with a sufficiently good initial iterate, AA performs no worse than a fixed-point iteration [29].

The AA algorithm with fixed damping, β, and depth (subspace size), m, is given in Algorithm 2. Following [30], the optimization problem in step 6 for the weights, $\gamma_k \in \mathbb{R}^{m_k}$, to compute the new iterate is presented as an unconstrained linear least-squares problem. Utilizing this formulation, rather than the constrained form of the problem, allows for an efficient solution via a QR factorization of $\mathcal{F}_k$ i.e., $R_k \gamma_k = Q_k^T f_k$ where $Q_k \in \mathbb{R}^{N \times m_k}$ is an orthogonal matrix, $R_k \in \mathbb{R}^{m_k \times N}$ is an upper triangular matrix, and $f_k \in \mathbb{R}^N$ is the nonlinear residual, $G(p_k) - p_k$. Updating the factorization and solving the linear system as the residual history changes is typically a minimal cost compared to the function evaluation [20, 21] and thus not a major contributor to the overall iteration run time.

Algorithm 2 Anderson Acceleration

Input: p_0, $m_{\max}$, $k_{\max}$, tol, and β **Output:** p^*

Set $p_1 = \beta G(p_0) + (1-\beta)p_0$
for $k = 1, \ldots, k_{\max}$ **do**
  Set $m_k = \min\{k, m_{\max}\}$
  Set $\mathcal{G}_k = [\Delta G_{k-m_k}, \ldots, \Delta G_{k-1}]$ where $\Delta G_i = G(p_{i+1}) - G(p_i)$
  Set $\mathcal{F}_k = [\Delta f_{k-m_k}, \ldots, \Delta f_{k-1}]$ where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(p_i) - p_i$
  Determine γ_k that minimizes $\|f_k - \mathcal{F}_k\gamma_k\|_2$
  Set $p_{k+1} = G(p_k) - \mathcal{G}_k\gamma_k - (1-\beta)(f_k - Q_k R_k \gamma_k)$
  if $\|p_{k+1} - p_k\| <$ tol **return** p_{k+1} as p^*
end for

In this work, we consider several extensions to AA, including delaying the start of acceleration as well as non-stationary variants with variable damping and/or depth, leading to Algorithm 3. When the initial iterate is far from the fixed-point, delaying acceleration by d iterations can be advantageous in order to obtain a better initial iterate before enabling AA. Different approaches to adaptive damping in [5, 9] demonstrated significant improvements in convergence when compared to fixed damping values. The method in [5] computes an "optimal" damping value by solving a minimization problem at the cost of two additional function evaluations. To avoid extra evaluations, we use the heuristic approach from [9] with the damping value

$$\beta_k = 0.9 - \omega_\beta \Gamma_k \quad \text{with} \quad \Gamma_k = \sqrt{1 - (\|Q^T f_k\|/\|f_k\|)^2}, \tag{8}$$

where Γ_k is the AA convergence gain in the k-th iteration, Q is readily available from the QR solve in the AA optimization step, and ω_β is a tuning parameter. In [4] the adaptive damping algorithm from [5] is combined with a composite version of AA as a means of adapting the depth and further improves the convergence rate. A different adaptivity approach is taken in [25] where a filtering method is applied to remove columns from the history matrix, $\mathcal{F}_k$, in order to control the condition number of the least-squares problem. This method also leads to improved convergence. In this work, we utilize a strategy similar to that employed in [24] where the depth is determined by the magnitude of the current residual with

$$m_k = \min\{\tilde{m}_k, m_{k-1} + 1, m_{max}\} \quad \text{with} \quad \tilde{m}_k = \max\{0, \lfloor -\log_{10}(\omega_m \|f_k\|)\rfloor\} \tag{9}$$

where $m_0 = 0$ and ω_m is a turning parameter.

Algorithm 3 Anderson Acceleration with delay and variable damping and depth

Input: p_0, $m_{\max}$, $k_{\max}$, tol, d, and β **Output:** p^*

Set $p_1 = \beta G(p_0) + (1-\beta)p_0$
for $k = 1, \ldots, k_{\max}$ **do**
  if $k \leq d$ **then**
    Set $p_{k+1} = \beta G(p_k) + (1-\beta)p_k$
  else
    Select m_k using (9)
    Set $\mathcal{G}_k = [\Delta G_{k-m_k}, \ldots, \Delta G_{k-1}]$ where $\Delta G_i = G(p_{i+1}) - G(p_i)$
    Set $\mathcal{F}_k = [\Delta f_{k-m_k}, \ldots, \Delta f_{k-1}]$ where $\Delta f_i = f_{i+1} - f_i$ and $f_i = G(p_i) - p_i$
    Determine γ_k that minimizes $\|f_k - \mathcal{F}_k\gamma_k\|_2$
    Select β_k using (8)
    Set $p_{k+1} = G(p_k) - \mathcal{G}_k\gamma_k - (1-\beta_k)(f_k - Q_k R_k \gamma_k)$
  end if
  if $\|p_{k+1} - p_k\| <$ tol **return** p_{k+1} as p^*
end for

4 Implementation

To evaluate whether the application of AA improves the convergence rate of the LoDestro method, we utilize the Tango 1D transport solver [22, 23] interfaced with the KINSOL nonlinear system solver from the SUNDIALS library [12, 16]. Tango is an open-source Python code that both solves the 1D transport equation (3) by way of the linear system (6) and implements the coupling Algorithm 1. The turbulent flux models available for coupling include the global gyrokinetic code GENE [13, 14], as well as analytic models for rapid algorithmic development. For this work we utilize one of the latter options, and note it corresponds to running the turbulence code to saturation each coupling exchange.

KINSOL provides a number of algorithms for solving nonlinear algebraic systems including an Anderson-accelerated fixed-point iteration. AA in KINSOL leverages several implementation optimizations for greater performance such as efficient QR factorization updates and solves for computing the acceleration weights [10, 18, 30] or the use of low synchronization orthogonalization methods within the QR solve to minimize the number of global reductions [20]. For this work, we have extended KINSOL's existing support for delay and constant damping with a fixed depth to allow for adapting the depth and/or damping. These new capabilities are included in the SUNDIALS v7.3.0 release. Through the interface between Tango and KINSOL, we can readily access an efficient implementation of AA and easily experiment with different variants of AA with minimal modification to existing Tango code.

The interfacing between Tango and KINSOL primarily consists of creating a class within Tango with a method to evaluate $G(p)$ using Tango's native functions for computing and splitting fluxes and setting up and solving the linear system. Additional class methods call KINSOL functions to configure the solver based on

command line options and solve the nonlinear system. As SUNDIALS packages are written in C, we leverage SWIG [3] to generate Python interfaces that align with the C API while allowing Tango to access data stored in SUNDIALS vectors as NumPy arrays without having to copy data between KINSOL and Tango data structures.

Algorithm 3 includes a number of parameters that can have a significant impact on performance. The optimal combination of settings is unclear *a priori* and depends on the application and problem considered. To automate parameter selection we utilize the GPTune package [19], a Python-based auto-tuning framework using Bayesian optimization methodologies with support for multi-task learning, multi-objective tuning, sensitivity analysis, and many other features. For this work we use the single-task learning autotuning (SLA) capability in GPTune for a single-objective function. That is, we consider one configuration of the problem (1) when optimizing the algorithm parameters to minimize the number of iterations to reach a given solution tolerance. The problem class created for interfacing with KINSOL also streamlines using GPUTune, as the objective function only needs to construct the Tango problem class given a dictionary of input options, call the solve method, and return the number of solver iterations. Additionally, the driver program defines the input, output, and parameter spaces, as well as any constraints, to create the tuning problem. Once the problem is defined, the driver creates the GPTune object and runs the SLA method which calls Tango as needed through the objective function.

5 Numerical Results

In this section we present results applying AA to the LoDestro method for a nonlinear diffusion problem from [27] that mimics the challenges observed in turbulence-driven transport simulations. The problem is given by (1) on the interval $x \in [0, 1]$ with the analytic flux

$$q = -D\frac{\partial p}{\partial x} + \epsilon \quad \text{where} \quad D = \left(\frac{1}{p}\frac{\partial p}{\partial x}\right)^r. \tag{10}$$

As briefly mentioned in previous sections, turbulent transport in tokamaks is often due to instabilities driven by gradients of the time-averaged 1D temperature and other profiles; and these instabilities often have thresholds—critical values for their onset. The resulting transport then drives the system to flatter gradients, which pushes plasma operation toward marginal stability, where the growth rates rise extremely rapidly from zero. This situation leads to very stiff transport as operation near the critical gradients is approached from above [11]. We model this high level of stiffness by increasing the value of r in (10). The system is stiff with $r = 2$ and we use $r = 10$ to represent very stiff tests relevant to magnetic-fusion-energy problems.

The 5D turbulence code illustrated in Fig. 1 reaches a saturated state with short spatial- and temporal-scale fluctuations. It is the electric fields involved in these fluctuations that kick particles across the magnetic surfaces, i.e., cause the transport. Even when a heat or particle flux is integrated over four dimensions, the 1D flux still

has short-scale structure. We include the effects of these fluctuations on our coupling problem with the "noise" term, ϵ, in (10). We use the same spatially correlated, temporally white Gaussian noise model studied in [22] to mimic the fluctuations observed in GENE simulations (see Figs. 4a and 5b in [22] for a comparison of the model with fluxes from GENE). Below, we first test cases without noise ($\epsilon = 0$) before considering a problem with noisy fluxes.

The source term in (1) is defined as

$$S(x) = \begin{cases} 1 & \text{for} \quad 0 \leq x \leq 0.1 \\ 0 & \text{for} \quad 0.1 < x \leq 1 \end{cases}, \tag{11}$$

and the boundary conditions are $p'(0) = 0$ and $p(1) = 0.01$. For the LoDestro method, the relaxation strength depends on the degree of stiffness, and the default value is $\beta = 0.6/r$. AA setups with zero depth ($m = 0$) and constant damping correspond to applying the unaccelerated LoDestro method as implemented in Tango and serves as the baseline comparison point.

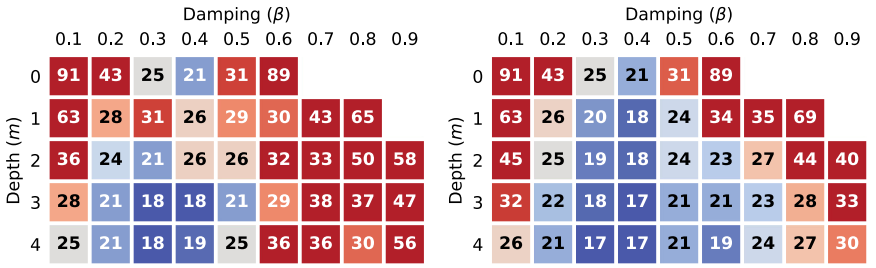
5.1 A Stiff Test Case

We first consider the stiff case ($r = 2$) and the impact of different AA configurations on the convergence of the iteration.

5.1.1 Fixed Damping, Depth, and Delay

To begin, we consider AA with static damping, β, and depth, m, sweeping over the parameter space to better understand how each impacts performance. Figure 2 shows a heat map for the number of iterations to reach a residual of 10^{-11} for different β and m values with and without a delay, d, before starting AA. From the first row ($m = 0$, no acceleration), we see the default, $\beta = 0.3$, is near optimal in this case, but $\beta = 0.4$ can reduce the number of iterations by 16%. However, increasing β too much leads to unphysical values or divergence (both denoted by empty cells).

Without delay, adding acceleration, $m = 1$, gives some robustness in the choice of β, enabling convergence for larger values than possible without AA. However, the convergence rate is improved only for β far from the optimal value. Increasing the depth, $m > 1$, yields more consistent gains with AA. Comparing the best results, AA takes approximately 14% fewer iterations. For a non-optimal β, larger reductions in iteration count are realized due to the greater robustness and reduced sensitivity to the choice of β. Improved robustness in β with AA has also been observed in other multiphysics coupling applications [15].



(a) Without Delay (b) Delay (d) AA by 1 iteration

Fig. 2 Heat map of the number of iterations to reach a residual of 10^{-11}. Empty cells correspond to failed solves. Without a delay, $m \geq 2$ is needed to see a benefit in iteration counts from AA. With a one iteration delay, AA consistently reduces the number of iterations and decreases sensitivity to the value of β

As the first iterate diverges from the final solution before steadily approaching the correct value at each subsequent iteration, introducing a small delay before starting AA significantly strengthens the performance in all aspects. Generally, the results continue to improve with increased depth but with diminishing returns, especially near the optimal β. Delaying AA further can improve the results slightly but also reduces robustness with larger β and may increase iteration counts with smaller β. The size of the delay necessary will depend on the quality of the initial iterate, and the best depth value is often problem dependent.

5.1.2 Optimized Fixed Damping, Depth, and Delay

In the prior section we manually explored the parameter space to understand the impact of different options on algorithmic performance. In this section we investigate optimizing these selections with GPTune and examine the convergence history. We consider three sets of parameters to optimize: β with $m = 0$; β and m; and β, m, and d. Each set is optimized for three residual tolerances (10^{-6}, 10^{-8}, and 10^{-11}). We set the initial number of GPTune parameter-space samples to 10 and the total number of samples to 50. The convergence histories for the best parameter values from each set are shown in Fig. 3.

In the first few iterations there is little difference between the various options with smaller β values performing slightly better. At more moderate residuals, optimizing β alone is one or two iterations better than the AA results. However, for values below 10^{-7} the AA setups give the best results with the optimized parameter set edging out the best result from Fig. 2b by one or two iterations. The benefit of AA increases for values less than 10^{-9} where the convergence of the unaccelerated version slows. This result indicates that AA delivers accelerated convergence to a tight tolerance. Overall the optimized parameter sets from GPTune closely align with the best setups found with the manual sweep.

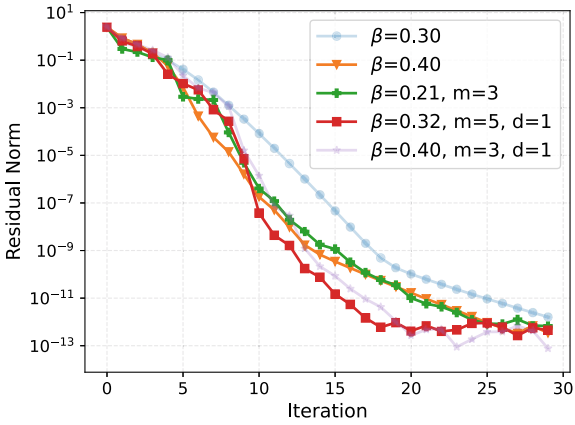


Fig. 3 Residual history for autotuned configurations. The default (faded circles) and one setup from Fig. 2b (faded stars) are included for reference. The autotuned results align well with the parameter sweep in Fig. 2

5.1.3 Adaptive Damping and Depth

In the previous section we saw that the best parameter configurations depend on the desired residual norm. Performance also depends significantly on whether the initial iterate is in the asymptotic convergence regime of the iterative scheme. In this section we evaluate AA with adaptive damping and/or depth to find setups that perform well across residual regimes. We again consider three sets of parameters to optimize: β with adaptive m; m and d with adaptive β, and adapting β and m together. We only optimize d in the fixed-m case as the strategy for adapting m automatically finds an optimum d by initially selecting $m = 0$. When adapting β and/or m, GPTune optimizes the factors ω_β and ω_m in (8) and (9), respectively. As before, each set is optimized for the same three residual tolerances and we use the same numbers of initial and total parameter-space samples. The convergence history for the best parameter values from each set are shown in Fig. 4, and history of β or m from the adaptive cases are given in Fig. 5.

Optimizing a fixed β and an adaptive m gives similar results to optimizing fixed β, m, and d. Figure 5a shows that adaptive m delays AA six iterations and has a max m of three. Given the low sensitivity observed in Fig. 2b, it is not surprising that, although the m and d are slightly different, the convergence history only differs by one iteration. Optimizing an adaptive β with fixed m and d has improved convergence at larger residuals, and, after five iterations, the setup is within two iterations of the best results. Figure 5b shows that the performance gain at larger residuals is due to using a small β value initially before immediately growing to almost the maximum β once AA starts. This behavior occurs because an optimum fixed β is computed by GPTune in the delay region before AA starts, as Γ_k is not available to estimate β. Once AA starts, the iteration switches to adaptive values and keeps β close to the max allowed.

Optimizing both β and m is the least performant of the three configurations but, due to the low sensitivity to β and m, the results are still relatively close (within

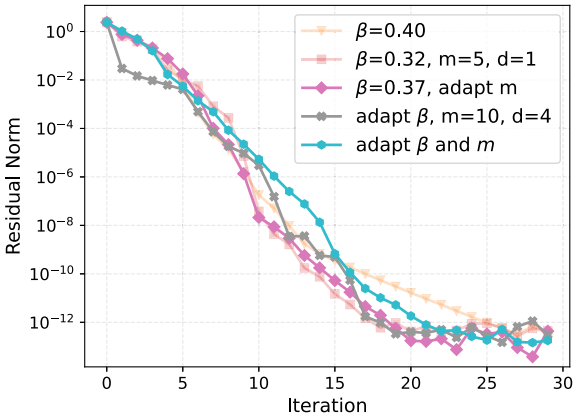
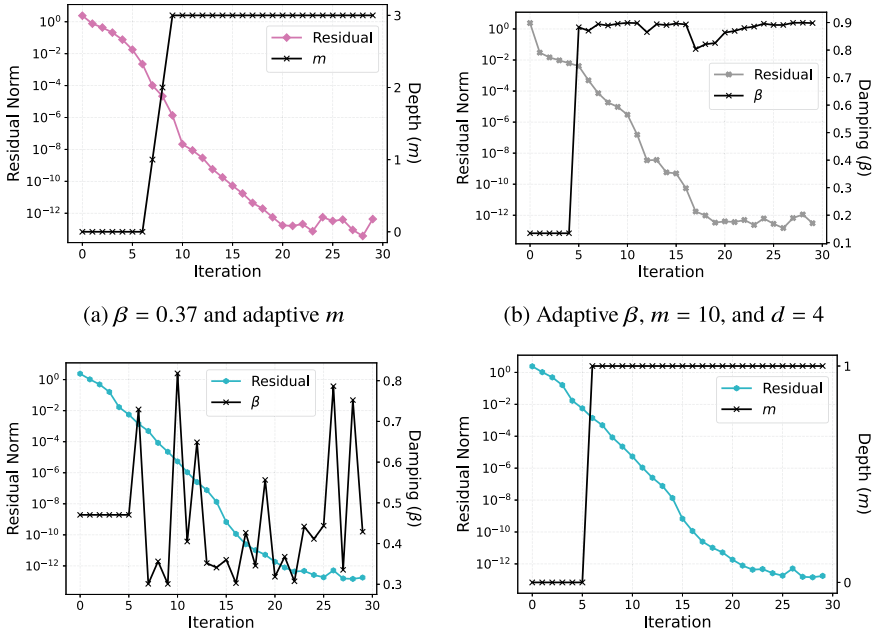


Fig. 4 Residual history for autotuned configurations with adaptive β and/or m. The adaptive results closely follow the autotuned fixed parameter results in Fig. 3 (faded triangles and squares included for reference)



(a) $\beta = 0.37$ and adaptive m

(b) Adaptive β, $m = 10$, and $d = 4$

(c) Adaptive β and m

(d) Adaptive β and m

Fig. 5 Residual (left axis) and parameter history (right axis) for adaptive β and/or m configurations. Adapting m automatically selects the AA delay, and adapting β alone leads to values near the max allowed. Adapting β and m together leads to large, frequent changes in β slightly degrading performance compared to other setups

five iterations) of the other cases. Again, adapting m automatically delays AA (five iterations) before settling on a max depth. Unlike when adapting β alone, there are large oscillations in β after the initial delay. This variation in β appears to be the cause of lower overall performance and suggests that considering the gain history to reduce large, frequent changes in β may improve results.

Overall, in this stiff test case there are modest gains in numbers of iterations with AA compared to optimized damping values at large to moderate residual values. For residuals below 10^{-8} AA has a larger impact as convergence of the iteration without AA slows. However, AA greatly increases robustness and decreases sensitivity to the choice of β with a small delay in AA increasing these effects.

5.2 A Very Stiff Test Case

We now consider a very stiff case ($r = 10$) relevant to magnetic-fusion-energy problems and examine the impact of AA on convergence with optimized fixed and adaptive parameter configurations as in the prior two sections. Figure 6 shows the convergence history for the default iteration ($\beta = 0.06$), an optimized damping value ($\beta = 0.14$), and three of the best performing AA configurations.

As in the stiff case, optimizing β gives better results than the default β and, in this case, is generally the best option for larger residuals. Enabling AA with $\beta = 0.01$ and $m = 3$ consistently outperforms optimizing β alone for residuals below 10^{-4}. Again, delaying AA generally leads to better results overall and, given the larger initial residual and stronger damping applied in this case, the best AA configurations tend to have a longer delay. For residuals below 10^{-7}, configurations with fixed β,

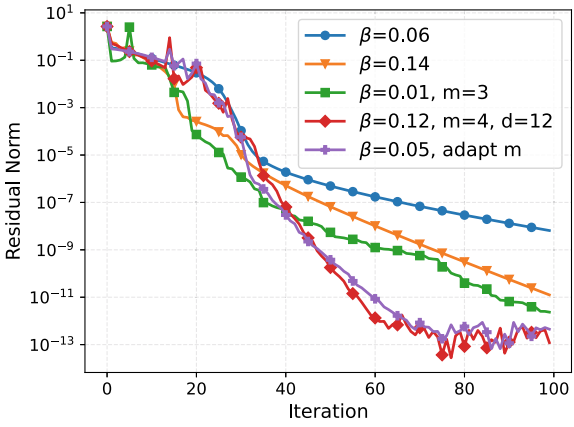


Fig. 6 Residual history for autotuned setups with the default iteration ($\beta = 0.06$), optimized damping ($\beta = 0.14$), and various AA configurations. Markers are placed every 5 iterations. For a wide range of residuals, AA leads to reduced numbers of iterations, especially for smaller residual values

m, and d or fixed β and adaptive m are the most efficient with results that differ by less than six iterations. With an adaptive m, AA is automatically delayed by twelve iterations, the same as in the fixed-delay case, but allows more depth going up to $m = 7$. Unlike in the stiff test, the depth does not immediately ramp up after the delay but instead keeps m constant for 2 to 7 iterations before increasing m. The stair-step behavior of m reflects the overall slower convergence in this case, which requires additional iterations before a residual reduction triggers an increase in m using the heuristic (9). Adapting β can also improve results but, like in the stiff case, does not generally perform as well as other setups due to large oscillations in β. Overall the results with this very stiff test are improved when comparing against the stiff case, with AA producing more significant gains over a larger range of residuals.

5.3 *Adding Noise in a Stiff Test Case*

Finally, we revisit the stiff test ($r = 2$) when accounting for fluctuations in q that mimic the noise observed with GENE i.e., $\epsilon \neq 0$ in (10). Figure 7 shows the residual history for the default setup ($\beta = 0.3$) and a more optimal configuration ($\beta = 0.4$), with and without AA, along with a heat map for other combinations of β and m. Unlike in the cases without noise, the performance with AA is similar to that without AA when using the best β value. This lack of improvement is due to the higher, noise-dependent residual floor at which the iteration stagnates as noted in [22] for the LoDestro method and similarly observed in [28] when applying AA to problems with inaccurate function evaluations. While the residual floor limits the benefits in lowering iteration counts, AA still improves overall robustness and reduces sensitivity to β, which leads to faster convergence for non-optimal β values. This reduced sensitivity is significant as it shows that with AA, less manual tuning of β is necessary for an efficient solution approach.

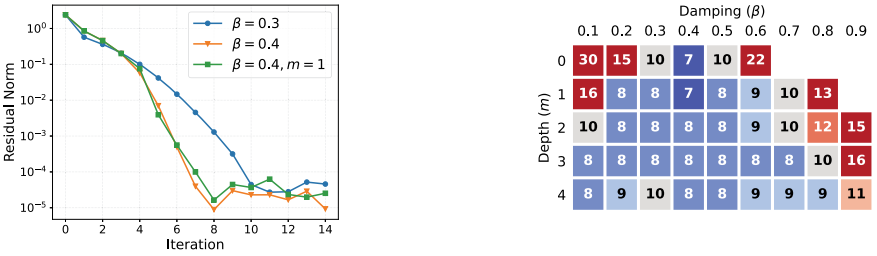


Fig. 7 Residual history and heat map for the number of iterations to reach a residual of 10^{-4} both with a delay of two iterations. AA increases robustness and reduces sensitivity to β but only lowers iteration counts away from the optimal β

6 Conclusions

In this work we used the KINSOL nonlinear solver library in SUNDIALS to evaluate the potential of several variations of Anderson acceleration to improve the convergence of the LoDestro method for coupled turbulence and transport simulations of a magnetically confined fusion plasma. Combining the flexibility of KINSOL with the GPTune autotuning library enabled rapid evaluation of the parameter space to compare the impact of different options on algorithmic performance.

For all test cases, AA allows for convergence with larger relaxation parameter values and decreases sensitivity to the choice of the relaxation parameter. Comparing to the optimal damping value, AA can provide a small decrease in the number of iterations for moderate residual values with the benefit increasing for smaller target residuals leading to a reduction of 20% or more in the number of iterations. Moreover, these benefits improve as the problem difficulty increases, going from stiff to very stiff tests; and for non-optimal damping values, the improvements are larger. Utilizing adaptive depth with AA provides a means for automatically selecting the delay and maximum allowed depth with results generally matching the best setups with optimized fixed parameter values. We note that use of GPTune removes the need for time-consuming hand optimizations of the parameters. When noisy fluxes are considered, AA does not provide an improvement in iterations with optimal damping due to the higher residual floor [22]. However, AA still increases robustness and reduces sensitivity to the damping parameter, which lowers iteration counts for non-optimal damping. Improvements in the adaptivity strategies for damping and depth (or the alternative methods explored in [5, 9], and [25]) could increase the benefits of AA at larger residual values and will be considered in later studies.

In future efforts we plan to assess the performance of AA with a modified analytic flux model that reflects the time dependence of the turbulence calculation as it is coupled to the transport code (see Fig. 1). We will also apply recently developed approaches for analyzing AA convergence [7] to better understand how the presence of noise and the addition of time dependence in the coupling impact convergence of the algorithm. Additionally, we will conduct studies to quantify parameter sensitivity in AA using GPUTune. Finally, the interfaces to KINSOL will allow for testing new variants of AA as they are integrated into the library, such as alternating and composite AA [4], without modifying Tango.

Acknowledgements The authors wish to acknowledge the initial explorations applying Anderson acceleration in Tango by Jeff Parker, Jack G. Paulson, and H. Hunter Schwartz and thank them for their help in using Tango and interfacing with KINSOL. We would also like to thank Cody J. Balos for his work in creating Python interfaces to KINSOL.

Support for this work was provided in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) Program through the Frameworks, Algorithms, and Scalable Technologies for Mathematics (FASTMath) Institute, under Lawrence Livermore National Laboratory subcontract B626484 and DOE award DE-SC0021354.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and Office of Fusion Energy Sciences, Scientific Discovery through Advanced Computing (SciDAC) program.
This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-PROC-864392.

Competing Interests The authors have no conflicts of interest to declare that are relevant to the content of this chapter.

References

1. Anderson, D.G.: Iterative procedures for nonlinear integral equations. Journal of the ACM (JACM) **12**(4), 547–560 (1965). https://doi.org/10.1145/321296.321305
2. Bañón Navarro, A., Di Siena, A., Velasco, J., Wilms, F., Merlo, G., Windisch, T., LoDestro, L., Parker, J., Jenko, F.: First-principles based plasma profile predictions for optimized stellarators. Nucl. Fusion **63**(5), 054003 (2023). https://doi.org/10.1088/1741-4326/acc3af
3. Beazley, D.M., et al.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Tcl/Tk Workshop, vol. 43, p. 74 (1996). https://doi.org/10.5555/1267498.1267513
4. Chen, K., Vuik, C.: Composite Anderson acceleration method with two window sizes and optimized damping. Int. J. Numer. Meth. Eng. **123**(23), 5964–5985 (2022). https://doi.org/10.1002/nme.7096
5. Chen, K., Vuik, C.: Non-stationary Anderson acceleration with optimized damping. J. Comput. Appl. Math. 116077, (2024). https://doi.org/10.1016/j.cam.2024.116077
6. Crotinger, J.A., LoDestro, L., Pearlstein, L.D., Tarditi, A., Casper, T.A., Hooper, E.B.: CORSICA: A comprehensive simulation of toroidal magnetic-fusion devices. final report to the LDRD program. Tech. rep., Lawrence Livermore National Laboratory (1997). https://doi.org/10.2172/522508
7. De Sterck, H., He, Y.: Linear asymptotic convergence of Anderson acceleration: fixed-point analysis. SIAM J. Matrix Anal. Appl. **43**(4), 1755–1783 (2022). https://doi.org/10.1137/21M1449579
8. Di Siena, A., Bañón Navarro, A., Luda, T., Merlo, G., Bergmann, M., Leppin, L., Görler, T., Parker, J., LoDestro, L., Dannert, T., Germaschewski, K., Allen, B., Hittinger, J., Dorland, B., Hammett, G., Jenko, F.: the ASDEX Upgrade Team, the EUROfusion MST1 Team: Global gyrokinetic simulations of asdex upgrade up to the transport timescale with gene-tango. Nucl. Fusion **62**(10), 106025 (2022). https://doi.org/10.1088/1741-4326/ac8941
9. Evans, C., Pollock, S., Rebholz, L.G., Xiao, M.: A proof that Anderson acceleration improves the convergence rate in linearly converging fixed-point methods (but not in those converging quadratically). SIAM J. Numer. Anal. **58**(1), 788–810 (2020). https://doi.org/10.1137/19M1245384
10. Fang, H.R., Saad, Y.: Two classes of multisecant methods for nonlinear acceleration. Numer. Linear Algebr. Appl. **16**(3), 197–221 (2009). https://doi.org/10.1002/nla.617
11. Garbet, X., Mantica, P., Ryter, F., Cordey, G., Imbeaux, F., Sozzi, C., Manini, A., Asp, E., Parail, V., Wolf, R., et al.: Profile stiffness and global confinement. Plasma Phys. Controlled Fusion **46**(9), 1351 (2004). https://doi.org/10.1088/0741-3335/46/9/002
12. Gardner, D.J., Reynolds, D.R., Woodward, C.S., Balos, C.J.: Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers. ACM Transactions on Mathematical Software (TOMS) **48**(3), 1–24 (2022). https://doi.org/10.1145/3539801
13. GENE Developers: Gyrokinetic Electromagnetic Numerical Experiment (GENE). URL http://genecode.org/. Last accessed 6 May 2024

14. Goerler, T., Lapillonne, X., Brunner, S., Dannert, T., Jenko, F., Merz, F., Told, D.: The global version of the gyrokinetic turbulence code GENE. J. Comput. Phys. **230**(18), 7053–7071 (2011). https://doi.org/10.1016/j.jcp.2011.05.034
15. Hamilton, S., Berrill, M., Clarno, K., Pawlowski, R., Toth, A., Kelley, C., Evans, T., Philip, B.: An assessment of coupling algorithms for nuclear reactor core physics simulations. J. Comput. Phys. **311**, 241–257 (2016). https://doi.org/10.1016/j.jcp.2016.02.012
16. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. ACM Transactions on Mathematical Software (TOMS) **31**(3), 363–396 (2005). https://doi.org/10.1145/1089014.1089020
17. Kelley, C.T.: Numerical methods for nonlinear equations. Acta Numer **27**, 207–287 (2018). https://doi.org/10.1017/S0962492917000113
18. Kresse, G., Furthmüller, J.: Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Comput. Mater. Sci. **6**(1), 15–50 (1996). https://doi.org/10.1016/0927-0256(96)00008-0
19. Liu, Y., Sid-Lakhdar, W.M., Marques, O., Zhu, X., Meng, C., Demmel, J.W., Li, X.S.: GPTune: Multitask learning for autotuning exascale applications. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 234–246. (2021). https://doi.org/10.1145/3437801.3441621
20. Lockhart, S., Gardner, D.J., Woodward, C.S., Thomas, S., Olson, L.N.: Performance of low synchronization orthogonalization methods in Anderson accelerated fixed point solvers. In: Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing, pp. 49–59. SIAM (2022). https://doi.org/10.1137/1.9781611977141.5
21. Loffeld, J., Woodward, C.S.: Considerations on the implementation and use of anderson acceleration on distributed memory and gpu-based parallel computers. In: Advances in the Mathematical Sciences: Research from the 2015 Association for Women in Mathematics Symposium, pp. 417–436. Springer (2016). https://doi.org/10.1007/978-3-319-34139-2_21
22. Parker, J.B., LoDestro, L.L., Campos, A.: Investigation of a multiple-timescale turbulence-transport coupling method in the presence of random fluctuations. Plasma **1**(1), 126–143 (2018). https://doi.org/10.3390/plasma1010012
23. Parker, J.B., LoDestro, L.L., Told, D., Merlo, G., Ricketson, L.F., Campos, A., Jenko, F., Hittinger, J.A.: Bringing global gyrokinetic turbulence simulations to the transport timescale using a multiscale approach. Nucl. Fusion **58**(5), 054004 (2018). https://doi.org/10.1088/1741-4326/aab5c8
24. Pollock, S., Rebholz, L.G.: Anderson acceleration for contractive and noncontractive operators. IMA J. Numer. Anal. **41**(4), 2841–2872 (2021). https://doi.org/10.1093/imanum/draa095
25. Pollock, S., Rebholz, L.G.: Filtering for Anderson acceleration. SIAM J. Sci. Comput. **45**(4), A1571–A1590 (2023). https://doi.org/10.1137/22M1536741
26. Post, D.E., Batchelor, D.B., Bramley, R.B., Cary, J.R., Cohen, R.H., Colella, P., Jardin, S.C.: Report of the fusion simulation project steering committee. J. Fusion Energy **23**, 1–26 (2004). https://doi.org/10.1007/s10894-004-1868-0
27. Shestakov, A., Cohen, R., Crotinger, J., LoDestro, L., Tarditi, A., Xu, X.: Self-consistent modeling of turbulence and transport. J. Comput. Phys. **185**(2), 399–426 (2003). https://doi.org/10.1016/S0021-9991(02)00063-3
28. Toth, A., Ellis, J.A., Evans, T., Hamilton, S., Kelley, C.T., Pawlowski, R., Slattery, S.: Local improvement results for Anderson acceleration with inaccurate function evaluations. SIAM J. Sci. Comput. **39**(5), S47–S65 (2017). https://doi.org/10.1137/16M1080677
29. Toth, A., Kelley, C.T.: Convergence analysis for Anderson acceleration. SIAM J. Numer. Anal. **53**(2), 805–819 (2015). https://doi.org/10.1137/130919398
30. Walker, H.F., Ni, P.: Anderson acceleration for fixed-point iterations. SIAM J. Numer. Anal. **49**(4), 1715–1735 (2011). https://doi.org/10.1137/10078356X

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Intrepid2: Structure-Preserving Optimizations for High-Order Finite Elements



Nathan V. Roberts, Mauro Perego, and Roger P. Pawlowski

Abstract `Intrepid2`, like `Intrepid` before it, is a library within `Trilinos` providing core functionality for finite element and finite volume computations, including high-order basis function implementations, mechanisms for transforming basis values from reference to physical space, and elementwise integration, among many other features. `Intrepid2` builds atop `Kokkos` for performance portability across serial CPU, multi-threaded CPU, and GPU architectures. A driver of recent `Intrepid2` development has been plasma physics applications, including the particle-in-cell `EMPIRE` code and direct discretizations of the Vlasov equation implemented in `Camellia`. Both of these benefit from high-order discretizations, and, for both, assembly costs can come to dominate. We have implemented new mechanisms for sum-factorized assembly, including software components that allow `Intrepid2` to preserve structural features of basis values during computations. In the present work, we describe these mechanisms and demonstrate substantial performance improvements relative to the unstructured assembly `Intrepid2` has previously offered, with particular focus on use cases that arise in our plasma physics applications.

Keywords Compatible finite element methods · High-order methods · Sum factorization · Matrix-free methods · Performance portability · Tensor-product elements

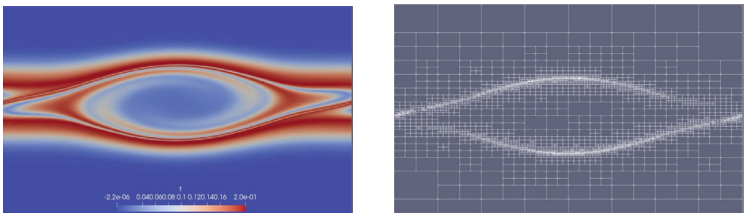
1 Introduction

Finite element methods are the foundation of numerous simulation codes targeting scientific applications. The last decade has seen a renewed interest in high-order finite elements due to the greater efficiencies enabled, especially on modern hardware

N. V. Roberts (✉) · M. Perego · R. P. Pawlowski
Center for Computing Research, Sandia National Laboratories, Albuquerque, NM, USA
e-mail: nvrober@sandia.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_7



(a) Vlasov solution: distribution function f. Vertical dimension is velocity, horizontal space.

(b) Adaptive quadratic mesh.

Fig. 1 Two-stream instability problem, adaptive solution using `Camellia` for a 1D1V Vlasov-Poisson model. Figure adapted from [6]

architectures, by matrix-free methods when used in concert with high-order finite elements (see, e.g. [1, 2]) (Fig. 1).

In this work we present recent computational advances in the implementation of high-order finite elements in the `Intrepid2` software package. `Intrepid2` is written in C++ and is a performance portable re-implementation of the `Intrepid` code [3] using the `Kokkos` [4] programming model. `Intrepid2` is at the core of the discretization capabilities provided by the `Trilinos` software suite and it is used in a wide variety of scientific applications, ranging from research to production codes. Some notable examples include:

- `Camellia` [5], a rapid-development framework for finite-element computations. Recently at Sandia, `Camellia` has been used to develop a set of adaptive direct-discretization Vlasov solvers, with support for both time-marching [6] and space-time [7] discretizations.
- `EMPIRE` [8], a massively parallel, portable, finite element particle-in-cell application for solving both electrostatic and electromagnetic problems, which employs `Intrepid2` compatible finite elements. An open-source surrogate for the electromagnetic solver in `EMPIRE` can be found in the `Mini-EM` solver distributed as part of Trilinos.
- `MALI` [9], an ice-sheet code part of the Department of Energy (DOE) climate model E3SM; this uses tensor-product, prismatic finite-elements on extruded meshes, and runs effectively on GPUs [10, 11].
- `Sierra` [12], a solid mechanics code, using composite finite elements [13] offering favorable accuracy properties in the context of large deformations.

These applications present unique discretization challenges. Each could benefit from using higher-order finite element methods, both because of increased accuracy per degree-of-freedom, and because of the increased computational intensity which allows a more efficient use of modern architectures. Preparatory studies using high-order finite elements in `Mini-EM` exposed a computational bottleneck in the cost of the assembly, especially when used by linear solvers in a matrix-free fashion. To

address this issue, we have developed a general sum factorization approach within `Intrepid2`, applicable to basis evaluations across the whole exact sequence and adaptable to matrix-free computations.

Sum factorization approaches have been introduced in other libraries, including the `hp3d` finite element library of Demkowicz et al. [14, 15], `deal.II` [16], and `MFEM` [17, 18]. Like the present approach, `MFEM` supports sum-factorization across the entire 3D (and 2D) exact sequence: H^1, $H(\text{curl})$, $H(\text{div})$, and L^2 spaces for hexahedra and quadrilaterals. Whereas `MFEM` implements sum factorization separately for each pair of spaces, our approach is generic in the sense that a single integration kernel supports all function spaces.[1] We additionally support other tensor elements, including wedges in 3D, and higher-dimensional H^1 and L^2 hypercubes, up to seven dimensions.

This chapter is organized as follows. In Sect. 2, we motivate and describe several recent additions to `Intrepid2` designed to facilitate the implementation of structure-enabled algorithms, which include sum-factorized assembly and matrix-free methods; the latter allow the computation of the action of a matrix without assembling that matrix. We describe a general approach to sum factorization in Sect. 3, explicitly detailing the three-tensor-component case for clarity. In Sect. 4, we discuss the sum factorization implementation provided by `Intrepid2`. We show performance results for $H(\text{curl})$ assembly on CPU and GPU architectures in Sect. 5. In Sect. 6, we describe an initial usage of `Intrepid2`'s sum factorization facilities for a key component of `Camellia`'s space-time Vlasov solver. We conclude in Sect. 7.

2 Structure Preservation Within Intrepid2

In finite-element applications involving high-order basis functions with tensor structure, dramatic gains can be realized by exploiting that structure to reduce the number of computations required. This is the foundation for many contemporary matrix-free approaches (which avoid any assembly of matrices, see, e.g. [19]), and also of sum factorization approaches, including the ones we describe in Sect. 3 below.

2.1 Motivation: Finite Element Assembly

A typical finite element assembly algorithm proceeds roughly as follows:

1. Identify quadrature points in reference space.
2. Compute basis values and/or derivatives at reference space points.

[1] As a performance optimization, we do implement a specialization of the kernel for integrands involving three tensor components, such as those for a basis on a 3D hexahedron.

3. Compute Jacobians, Jacobian determinants, and Jacobian inverses, as necessary for value/derivative transformations, at each physical point.
4. Transform values and derivatives of basis functions to physical space.
5. Map quadrature points to physical space.
6. Compute weighting functions ("material data") in physical space.
7. Compute quadrature weights in physical space by multiplying the reference quadrature weights by the corresponding determinant of the Jacobian.
8. Integrate: Multiply the test and trial values at each point, summing over the points weighted by the physical quadrature weights.

Given an appropriately constrained context, each of these steps may be revised to save memory and computational costs. Here, we focus on two such contexts: a general hypercube mesh, and an orthogonally-extruded mesh.

Hypercube Elements in a General Mesh. In a hypercube element, reference-space quadrature points can be represented as a Cartesian product of 1D points. Moreover, the quadrature weights are a simple product of the corresponding 1D weights. For 1D points and weights ξ_i^p, w_i^p, $i = 1, \ldots, N_p$, in a p-accurate quadrature rule, the points and weights for a D-dimensional hypercube with order p_d in dimension $d = 1, \ldots, D$ are given by

$$(\xi_{i_1}^{p_1}, \ldots, \xi_{i_D}^{p_D}), \ \prod_{d=1}^{D} w_{i_d}^{p_d}.$$

The total number of quadrature points is thus $N_{p_1} \cdot \cdots \cdot N_{p_D}$. If these are stored explicitly, and if N_p is the same in each dimension, we will require storage of $D\, N_p^D$ floating point numbers to store the points, and another N_p^D floats to store the weights, for a total memory cost of $(D+1)\, N_p^D$. On the other hand, if we take advantage of the tensor-product structure, we may reduce this to one floating point number per weight, and one per 1D point. In the isotropic case, this is $2\, N_p$ floats; for anisotropic p, it is $2\, \left(N_{p_1} + \cdots + N_{p_D}\right)$ floats. To make this concrete, for a 3D element with $N_p = 5$, explicit storage requires 250 floats, while a reduced format would require only 10.

Many finite-element bases for hypercubes can also be expressed in terms of 1D basis functions—for H^1 and L^2 bases, these are simple products of the corresponding 1D bases, while for $H(\mathrm{div})$ and $H(\mathrm{curl})$ the bases can be expressed as the direct sum of products of 1D bases. Therefore, the number of distinct basis values stored can also be reduced, again with a reduction factor roughly of order p^2. More importantly, algorithms such as *sum factorization* (sometimes called *fast quadrature*) may be employed. We provide details in Sect. 3; here, we note that in the general 3D hexahedral case it allows a reduction from $O(p^9)$ floating point operations to $O(p^7)$ flops.

Orthogonally Extruded Meshes. Orthogonally extruded meshes arise in space-time finite elements, as well as in discretizations of phase space (as in Vlasov). In these, a lower-dimensional ("physical") mesh of arbitrary elements is extruded

orthogonally in one or more additional dimensions. The Jacobian is diagonal in the dimension(s) of extrusion, and physical basis function components corresponding to those dimensions are identical across elements of equal height in the extruded dimension(s). In a general Vlasov discretization, velocity dimensions are typically discretized using uniform, axis-aligned extrusions of a physical mesh; there may be up to three extruded dimensions.[2]

2.2 Data Structures: Some Patterns

As we have already seen, the major opportunities afforded by tensor structures in finite element assembly are memory footprint reductions (which may have implications for data locality, increasing the probability that data used in computations is available within a cache) and algorithmic restructuring by factoring out loop invariants. We would like to employ data structures that express tensor structures in a generic way: on the one hand, these should allow a reduced memory footprint when tensor structures are present; on the other, they should allow non-tensor data to be expressed in the same format. Computational kernels responsible for combining data structures—as when a Jacobian is applied to transform a reference-space gradient—should do so in a mathematically consistent (that is, correct) fashion while maintaining as much structural information as possible. Put another way, these data structures exist on a range from specific to generic; the most specific case might be one with constant diagonal Jacobians, with material data that is constant across the domain. The most generic case might have arbitrary Jacobians, with material data that varies within each element.

Two types of data structure arise directly from tensor product structure:

1. Cartesian product: lower-dimensional point sets are "multiplied" to produce a higher-dimensional point set that contains all combinations of the lower-dimensional points. For example, points on a wedge may be formed by taking all combinations of 2D points on a triangle, and 1D points on a line segment. In `Intrepid2`, this is represented by the TensorPoint class.
2. Tensor product: as with the Cartesian product, lower-dimensional value containers form an object that contains a value corresponding to one element each of the constituents; these elements are multiplied together to produce the value in the tensor product. For example, quadrature weights on a triangle would be multiplied by the quadrature weights on a line segment to determine the quadrature weights on the wedge. In `Intrepid2`, this is represented by the TensorData class.

In addition to these structures, a third pattern applies in some respects independent of the tensor product structure: a data container with invariance specification.

[2] In principle, of course, a space-time Vlasov discretization of a box geometry might have as many as seven orthogonally extruded dimensions. The more typical use case, though, involves a spatial mesh of general type with a (uniform) extrusion in the velocity dimensions.

The motivating idea here is that a container may logically extend in a particular dimension—say, the cell dimension—but it is known a priori that the data does not vary in that dimension, or varies in a particular way. The goal for our data structure is to capture this invariance in such a way that we may take advantage of it algorithmically and to reduce data storage. This pattern is represented by `Intrepid2`'s Data class.

2.3 Cubature, Basis, Transforms, and New Data Classes

`Intrepid2` has long had a CubatureTensor class which derives from the Cubature class; this allows defining a tensor product cubature from cubature points and weights of the tensor components. When cubature points and weights were requested through the `getCubature()` method, the fully-expanded point set (with p^D points) and weights (defined pointwise as the product of the tensorial component weights) would be returned. We have added methods that instead allocate and fill TensorPoint and TensorData instances, reducing storage costs and maintaining the tensor structure. Similarly, the `Intrepid2` Basis class has had a `getValues()` method that accepts tensor-agnostic containers as arguments. We have added methods that allocate and fill a new tensor-structure-aware BasisValues container. This class allows storage of basis values and derivatives in a structure-preserving fashion. Prior mechanisms in `Intrepid2` for computing transformations from reference to physical space also did not preserve structure. We have added a new class, TransformedBasisValues, which stores a BasisValues object alongside a Data object representing the reference-to-physical transform. The transform is lazily applied to the values; this preserves tensor structure and allows transformations to be combined and reordered for efficiency during integration and other evaluations. In addition to the classes already mentioned, we have added a TensorData class, which allows storage of tensor-product basis evaluations such as those from H^1 value basis evaluation, a VectorData class representing vectors of TensorData (possibly with multiple families defined in one object). We have also introduced a new CellGeometry class for specifying geometry, with support for low-storage specification of regular grids, as well as arbitrary meshes.

3 Sum Factorization Algorithm

Suppose that we have two basis functions $\widehat{\phi}^i$, $\widehat{\psi}^j$, and we wish to perform an integral in physical space on an element K:

$$G_{ij} = \int_K \phi^i \cdot \psi^j \tag{1}$$

Suppose further that $\hat{\phi}^i$ is vector-valued, with component indices $a = 1, \ldots, d_{\hat{\phi}}$; similarly $\hat{\psi}^j$ is indexed by $b = 1, \ldots, d_{\hat{\psi}}$. Suppose further that at each physical point there is a matrix M_{ab} defined such that

$$\phi^i \cdot \psi^j = \sum_{a=1}^{d_{\hat{\phi}}} \sum_{b=1}^{d_{\hat{\psi}}} \hat{\phi}^i_a \, M_{ab} \, \hat{\psi}^j_b. \tag{2}$$

Let each of $\hat{\phi}^i_a$, $\hat{\psi}^j_b$ have tensor product factors indexed by $r = 1, \ldots, R$, such that

$$\hat{\phi}^i_a = \prod_{r=1}^{R} \hat{\phi}^{i_r}_a(\hat{x}_r), \quad \hat{\psi}^j_b(\hat{\mathbf{x}}) = \prod_{r=1}^{R} \hat{\psi}^{j_r}_b(\hat{x}_r) \tag{3}$$

at each point $\hat{\mathbf{x}}$ in reference space. Here we identified i, j with multi-indices $i_1 \ldots i_R$, $j_1 \ldots j_R$. Let $w(\hat{\mathbf{x}})$ be a pointwise *cell measure*; this is the transformed reference space quadrature weight for quadrature points $\hat{\mathbf{x}}_l, l = 1, \ldots, L$, where $\hat{\mathbf{x}}_l = (\hat{x}_{l_1}, \cdots, \hat{x}_{l_R})$, and $l_k = 1, \ldots, L_k$ for $k = 1, \ldots, R$. The full integral G_{ij} is then given by

$$G_{ij} = \sum_{a=1}^{d_{\hat{\phi}}} \sum_{b=1}^{d_{\hat{\psi}}} \sum_{l_1=1}^{L_1} \cdots \sum_{l_R=1}^{L_R} \left(\prod_{r=1}^{R} \hat{\phi}^{i_r}_a(\hat{x}_{l_r}) \hat{\psi}^{j_r}_b(\hat{x}_{l_r}) \right) M_{ab} \, w(\hat{x}_{l_1}, \cdots, \hat{x}_{l_R}). \tag{4}$$

There are two key observations for sum factorization: first, we may rewrite the above as

$$G_{ij} = \sum_{a=1}^{d_{\hat{\phi}}} \sum_{b=1}^{d_{\hat{\psi}}} \sum_{l_1=1}^{L_1} \hat{\phi}^{i_1}_a(\hat{x}_{l_1}) \hat{\psi}^{j_1}_b(\hat{x}_{l_1}) \sum_{l_2=1}^{L_2} \cdots \sum_{l_R=1}^{L_R} \hat{\phi}^{i_R}_a(\hat{x}_{l_R}) \hat{\psi}^{j_R}_b(\hat{x}_{l_R}) M_{ab} \, w(\hat{x}_{l_1}, \cdots, \hat{x}_{l_R}); \tag{5}$$

second, we may reuse the partial sums for $i_r j_r \cdots i_R j_R$ for all basis functions whose indices end with those partial multi-indices. To facilitate the latter, we introduce partial sums $G_1, \cdots, G_R$ as follows:

$$G_R(\alpha_R) = \sum_{l_R=1}^{L_R} \hat{\phi}^{i_R}_a(\hat{x}_{l_R}) \hat{\psi}^{j_R}_b(\hat{x}_{l_R}) M_{ab} \, w(\hat{x}_{l_1}, \cdots, \hat{x}_{l_R}), \tag{6}$$

$$G_r(\alpha_r) = \sum_{l_r=1}^{L_r} \hat{\phi}^{i_r}_a(\hat{x}_{l_r}) \hat{\psi}^{i_r}_b(\hat{x}_{l_r}) G_{r+1}(\alpha_{r+1}), \quad r = 1, \ldots, R-1 \tag{7}$$

where we have also introduce a partial multi-index $\alpha_r = i_r j_r \cdots i_R j_R$. To reduce storage requirements in the actual implementation, we do not explicitly form G_1,

instead storing to the final memory location. Similarly, we fix i_R, j_R in an outer loop, so that G_R only uses one memory location.

To facilitate generic reasoning in terms of the tensor quadrature points, we employ a *point iterator* object that stores an index for each tensorial rank and increments in the final tensorial rank (R) first, and on increment returns the lowest rank for which an index changed during the increment operation. Thus, if the iterator returns $R-1$, we know that we have exhausted the integration points in rank R. Our point iterator also features a look-ahead method, `nextIncrementResult()`, which indicates the result that will be obtained by the next call to `increment()`.

The general sum factorization algorithm is listed below as Algorithm 1. This algorithm is a generalization of algorithms recently presented by Mora & Demkowicz [14] and Badger et al. [15].

Algorithm 1 General Sum Factorization

for $a = 1, \cdots, d_{\hat{\phi}}$, $b = 1, \cdots, d_{\hat{\psi}}$ **do**
  for $i_R = 1, \cdots, N_{\hat{\phi}_R}$, $j_R = 1, \cdots, N_{\hat{\psi}_R}$ **do**
    Clear $G_1, \ldots, G_R$
    Initialize `point_it` ▷ point iterator over all quadrature points
    $r \leftarrow R$
    while $r \geq 1$ **do**
      $\mathbf{x} \leftarrow$ `point_it.x()`
      $G_R \leftarrow G_R + \hat{\phi}_a^{i_R}(\hat{x}_{l_R})\hat{\psi}_b^{j_R}(\hat{x}_{l_R})M_{ab}(\mathbf{x})\, w(\mathbf{x})$
      $r_{\text{next}} \leftarrow$ `point_it.nextIncrementResult()`
      $r_{\text{least}} \leftarrow \max(2, r_{\text{next}})$
      for $s = R-1, \cdots, r_{\text{least}}$ **do**
        for all $\alpha_s = i_s j_s \cdots i_{R-1} j_{R-1}$ **do**
          $G_s(\alpha_s) \leftarrow G_s(\alpha_s) + \hat{\phi}_a^{i_s}(\hat{x}_{l_s})\hat{\psi}_b^{j_s}(\hat{x}_{l_s})G_{s+1}(\alpha_{s+1})$
        end for
        Clear G_{s+1}
      end for
      if $r_{\text{next}} \leq 1$ **then**
        for all $\alpha_2 = i_2 j_2 \cdots i_{R-1} j_{R-1}$ **do** ▷ i_R, j_R fixed above
          $G_{ij} \leftarrow G_{ij} + \hat{\phi}_a^{i_1}(\hat{x}_{l_1})\hat{\psi}_b^{j_1}(\hat{x}_{l_1})G_2(\alpha_2)$
        end for
        Clear G_2
      end if
      $r \leftarrow$ `point_it.increment()`
    end while
  end for
end for

3.1 The 3-Component (3D) Sum Factorization Algorithm

To see how the general algorithm plays out in a more concrete setting, consider the 3-component, 3D case. This is presented in Algorithm 2. We can regard this as consisting of three phases. In the first phase, for fixed x, y coordinates, and fixed

basis components i_z, j_z, we compute an intermediate integral G_z by summing values in the z dimension (line 9). Note that this is the only line in the algorithm that refers directly to the physical space coordinates, in the $M_{ab}(\mathbf{x})\, w(\mathbf{x})$ factor. Due to the loop structure, the first phase accounts for $O((p+1)^5)$ floating point operations.

The second phase iterates over basis components i_y, j_y, accounting for the contribution due to y-coordinate number l_y with l_x and i_z, j_z fixed. Due to the loop structure, this phase requires $O((p+1)^6)$ floating point operations. The intermediate container G_y contains $(p+1)^2$ entries.

The third and final phase accumulates into the stiffness matrix: with all basis component numbers $i = i_x i_y i_z$, $j = j_x j_y j_z$ fixed, we add in the contributions due to x-coordinate number l_x. Due to the loop structure, the final phase requires $O((p+1)^7)$ floating point operations; this is the asymptotic cost of the algorithm as a whole.

Algorithm 2 3D Sum Factorization (Basis-Value Intermediates)

```
1: Clear G                                   ▷ Integration matrix, indexed by (i, j) = (i_x i_y i_z, j_x j_y j_z)
2: for a = 1, ..., d_φ̂, b = 1, ..., d_ψ̂ do
3:   for i_z = 1, ..., N_φ̂z, j_z = 1, ..., N_ψ̂z do
4:     for l_x = 1, ..., L_x do
5:       Clear G_y                           ▷ (p+1)^2 array, indexed by i_y, j_y
6:       for l_y = 1, ..., L_y do
7:         G_z ← 0                           ▷ One number
8:         for l_z = 1, ..., L_z do          ▷ Integrate in z
9:           G_z ← G_z + φ̂_a^{i_z}(ẑ_{l_z}) ψ̂_b^{j_z}(ẑ_{l_z}) M_ab(x) w(x)   ▷ O((p+1)^5) flops
10:        end for
11:        for i_y = 1, ..., N_φ̂y, j_y = 1, ..., N_ψ̂y do
12:          G_y(i_y, j_y) ← G_y(i_y, j_y) + φ̂_a^{i_y}(ŷ_{l_y}) ψ̂_b^{j_y}(ŷ_{l_y}) G_z   ▷ O((p+1)^6) flops
13:        end for
14:      end for
15:      for i_x = 1, ..., N_φ̂x, j_x = 1, ..., N_ψ̂x do
16:        for i_y = 1, ..., N_φ̂y, j_y = 1, ..., N_ψ̂y do
17:          i ← i_x i_y i_z, j ← j_x j_y j_z
18:          G(i, j) ← G(i, j) + φ̂_a^{i_x}(x̂_{l_x}) ψ̂_b^{j_x}(x̂_{l_x}) G_y(i_y, j_y)   ▷ O((p+1)^7) flops
19:        end for
20:      end for
21:    end for
22:  end for
23: end for
```

3.2 *The 3-Component 3D Sum Factorization Algorithm with Pointwise Intermediates*

In Algorithm 2, we compute intermediates G_z—a single number, which implicitly refers to a particular selection of i_z, j_z)—and G_y, an array of values indexed by i_y, j_y, which also implicitly refers to the same i_z, j_z selections. Because G_y is indexed by (i_y, j_y), we refer to the above algorithm as the *Sum Factorization Algorithm with*

Basis-Indexed Intermediates. This is the structure employed by Mora and Demkowicz [14] and Badger et al. [15]. Note that on this approach, the l_x, l_y loops contain those for i_x, j_x, i_y, j_y, so that in each pass through the inner loops only a single (x, y) coordinate is treated. Given the layout of basis values in memory, this may lead to cache misses.

This observation leads us to another possible variant of the algorithm: we may reorder the loops so that basis value accesses for neighboring coordinates in a given dimension occur within the inner loops. As detailed in Algorithm 3, for a given i_x, j_x, in the first phase we fix l_y, l_z, and then perform an integration in x into an intermediate container G_x, indexed by (l_y, l_z). Assuming that $(p + 1)$ points are used in each dimension for quadrature,[3] this carries a storage cost of $(p + 1)^2$.

The second phase of Algorithm 3 places the i_y, i_j iteration in an outer loop, and accumulates into a 1D intermediate array G_y, indexed by l_z. With our earlier assumption on quadrature, this carries a storage cost of $(p + 1)$. As in Algorithm 2, the second phase requires $O((p + 1)^6)$ flops.

The third phase of the algorithm integrates in z, and requires $O((p + 1)^7)$ flops. Note that the particular combination of (i, j), and the corresponding value in the stiffness matrix G, referred to in lines 20–23, will only obtain once per (a, b) pair. This is important both in terms of data locality for G and for the possibility of parallel implementation: the work can be divided among threads in a natural way such that there is no contention for writing to G.

4 Sum Factorized Assembly with Intrepid2

We provide here a few notes about the `Intrepid2` implementations of the algorithms described above. We have implemented Algorithm 1 in a general setting, independent of spatial dimension and basis type. The implementation can be found in `Intrepid2_IntegrationToolsDef.hpp` in the `F_Integrate` functor, in the `run()` method. Algorithm 3 is implemented in functor `F_IntegratePointValueCache`, as `runSpecialized3()`; we do not yet have an implementation of this algorithm that is independent of number of components and basis type, though by analogy to the preceding algorithms it is clear enough how this could be accomplished.

In each of the implementations, we find it beneficial to allocate some temporary arrays beyond that which is used by the algorithms for intermediate storage; we initialized these with frequently-accessed basis values and point weights $(M_{ab}(\mathbf{x})\, w(\mathbf{x}))$.

At present, each implementation exposes $N_C(p + 1)^2$, where N_C is the number of cells treated in one kernel launch, known as the *workset*, thread parallelism for

[3] One disadvantage of Algorithm 3 compared to Algorithm 2 is that the intermediate storage costs depend on the quadrature degree. We believe that this is more than made up for by the data locality advantages; in any case, the intermediate storage costs are likely to be small compared to the $O((p + 1)^6)$ storage cost of the final stiffness matrix.

Algorithm 3 3D Sum Factorization (Point-Value Intermediates)

Clear G ▷ Integration matrix, indexed by $(i,\ j) = (i_x i_y i_z,\ j_x j_y j_z)$
for $a = 1, \cdots, d_{\hat{\phi}},\ b = 1, \cdots, d_{\hat{\psi}}$ **do**
  for $i_x = 1, \cdots, N_{\hat{\phi}_x},\ j_x = 1, \cdots, N_{\hat{\psi}_x}$ **do**
    Clear G_x ▷ $(p+1)^2$ array indexed by (l_y, l_z)
    for $l_y = 1, \cdots, L_y$ **do**
      for $l_z = 1, \cdots, L_z$ **do**
        for $l_x = 1, \cdots, L_x$ **do**
          $G_x(l_y, l_z) \leftarrow G_x(l_y, l_z) + \hat{\phi}_a^{i_x}(\hat{x}_{l_x})\hat{\psi}_b^{j_x}(\hat{x}_{l_x}) M_{ab}(\mathbf{x})\, w(\mathbf{x})$ ▷ $O((p+1)^5)$ flops
        end for
      end for
    end for
    for $i_y = 1, \cdots, N_{\hat{\phi}_y},\ j_y = 1, \cdots, N_{\hat{\psi}_y}$ **do**
      for $l_z = 1, \cdots, L_z$ **do**
        $G_y(l_z) \leftarrow 0$ ▷ $(p+1)$-length array, indexed by l_z
        for $l_y = 1, \cdots, L_y$ **do**
          $G_y(l_z) \leftarrow G_y(l_z) + \hat{\phi}_a^{i_y}(\hat{y}_{l_y})\hat{\psi}_b^{j_y}(\hat{y}_{l_y}) G_x(l_y, l_z)$ ▷ $O((p+1)^6)$ flops
        end for
      end for
      for $i_z = 1, \cdots, N_{\hat{\phi}_z},\ j_z = 1, \cdots, N_{\hat{\psi}_z}$ **do**
        $i \leftarrow i_x i_y i_z,\ j \leftarrow j_x j_y j_z$
        for $l_z = 1, \cdots, L_z$ **do**
          $G(i, j) \leftarrow G(i, j) + \hat{\phi}_a^{i_z}(\hat{z}_{l_z})\hat{\psi}_b^{j_z}(\hat{z}_{l_z}) G_y(l_z)$ ▷ $O((p+1)^7)$ flops
        end for
      end for
    end for
  end for
end for

the highest-intensity kernel (referred to as phase 3 in the discussions of Algorithms 2 and 3). We believe that in the case of Algorithm 3 by breaking the phases apart (at present they are "fused" together in common loops) and possibly introducing some additional intermediate storage, we can increase the parallelism expressed to $(p+1)^6$, while maintaining minimal inter-thread synchronization requirements. We expect to pursue this in future updates to `Intrepid2`.

There are also some opportunities for improving performance by taking advantage of symmetries. There are two types of symmetry that we may take advantage of:

1. *Internal* (component basis) symmetry: if component bases $\{\hat{\phi}_a^{i_r}\}_i, \{\hat{\phi}_b^{j_r}\}_j$ match, then the loop over $i_r,\ j_r$ can be reduced; its outputs for $(i_r,\ j_r)$ will match those for (j_r, i_r). This requires that the loops over the symmetric dimensions be innermost.
2. *Outer* symmetry: If the tensor bases $\{\hat{\phi}_i\}_i, \{\hat{\psi}_j\}_j$ match, *and* the transformation matrix M_{ab} is symmetric, then the loop over a, b can be reduced.

The savings from symmetry may be more than a factor of two: if multiple component bases match, the cost reductions compound. Our implementations do not yet take advantage of such symmetries, but we do hope to pursue this as well in future updates to `Intrepid2`.

5 Performance Results for $H(\text{curl})$ Assembly

The electromagnetic solves in `EMPIRE` and `Mini-EM` involve the assembly of two matrices: a mass matrix and a curl-curl stiffness matrix. For our `Intrepid2` benchmark, we assemble the matrix K given by:

$$K_{ij} = (\psi_i, \psi_j) + (\nabla \times \psi_i, \nabla \times \psi_j) \tag{8}$$

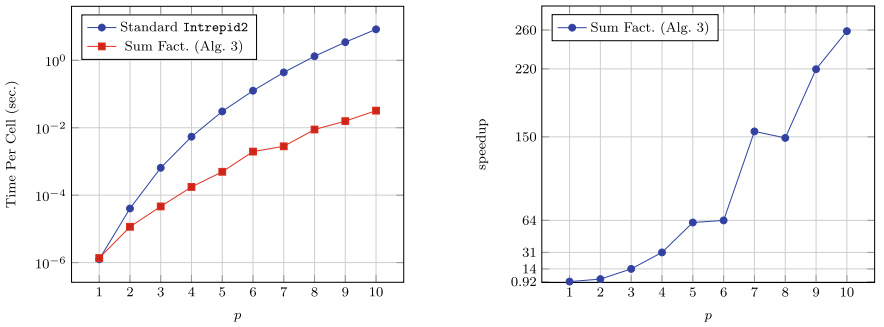
We show performance results for two representative architectures: a CPU with OpenMP-based parallelism and a GPU with CUDA-based parallelism. In each of these, we employ uniform hexahedral meshes varying in size from 32,768 elements (for $p = 1$) to 16 elements (for $p = 10$) on the CPU, and 524,288 elements (for $p = 1$) to 128 elements (for $p = 10$) on the GPU, but we do not take advantage of the regularity of the mesh in any way; the performance results apply to the unstructured mesh case. As part of the sum-factorized assembly implementation, we include flop estimates, which count each add, multiply, subtraction, and division as a single operation – these estimates are based on operation counts, *not* measured by hardware counters. For sum factorization on each architecture, we use Algorithm 3; prior experimentation has shown that this can be slightly less performant on CPUs than Algorithm 2, but still has excellent performance. On CUDA, our experiments have shown a dramatic difference between the two, in favor of Algorithm 3, so for the present we use this as the standard choice for `Intrepid2` across all architectures.

Our driver supports variable workset sizes (the number of cells for which we compute the stiffness matrix at a time); we calibrate for each algorithm by trying a series of powers of two up to the total cell count. Larger worksets allow more parallelism to be expressed, but can increase the sizes of intermediate arrays, leading to cache misses and larger memory requirements. If workset sizes are too small, observed throughput can be a small fraction of the best possible for the algorithm – this is particularly a concern on GPU architectures, where the hardware is capable of a great deal of parallelism. In our tables, we indicate which workset size was used: the best one for that configuration. In the H100 test below, we aim to set the mesh sizes for our problems to be as large as possible without running out of GPU memory (the H100 has 80 GB total).

Table 1 shows the OpenMP results with 16 threads on an M2 Ultra CPU. The 16-thread choice corresponds to the number of *performance* cores present on the CPU. These run at a clock speed of roughly 3.5 GHz, so that the theoretical peak flop rate for 16 threads is 56 GFlops per second. The estimated throughput for some of our sum-factorized results is higher than this rate (up to 92 GFlops/second); we attribute the discrepancy to the presence of fused multiply-adds which are not considered in our flop estimate. Figure 2 plots a comparison between `Intrepid2`'s standard assembly and the sum-factorized assembly using Algorithm 3. Note that the speedups exceed the theoretically predicted $O(p^2)$, we believe due to improved cache locality in the sum-factorization algorithm.

Table 1 Reference-to-physical transform plus integration kernel for $H(\text{curl})$ stiffness matrix, results using OpenMP with 16 threads on an Apple M2 Ultra with 16 performance cores at 3.5 GHz. Times are in seconds

		Standard					Sum-Factorized (Alg. 3)					
p	Elements	Workset	Flops	Throughput		Time	Workset	Flops	Throughput		Time	Speedup
1	32,768	32,768	4.9e+08	12	GFlops	4.1e-02	32,768	4.2e+08	9.4	GFlops	4.5e-02	0.92x
2	32,768	32,768	3.1e+10	24	GFlops	1.3e+00	32,768	7.7e+09	20	GFlops	3.8e-01	3.5x
3	8,192	8,192	1.3e+11	25	GFlops	5.3e+00	512	1.5e+10	40	GFlops	3.8e-01	14x
4	2,048	2,048	2.8e+11	25	GFlops	1.1e+01	512	1.9e+10	52	GFlops	3.6e-01	31x
5	512	64	3.9e+11	25	GFlops	1.6e+01	128	1.7e+10	68	GFlops	2.5e-01	62x
6	256	16	8.2e+11	26	GFlops	3.2e+01	32	2.6e+10	51	GFlops	5.0e-01	64x
7	128	2	1.4e+12	25	GFlops	5.6e+01	32	3.3e+10	92	GFlops	3.6e-01	160x
8	64	2	2.1e+12	25	GFlops	8.4e+01	16	3.8e+10	68	GFlops	5.7e-01	150x
9	32	2	2.8e+12	25	GFlops	1.1e+02	16	4.0e+10	80	GFlops	5.0e-01	220x
10	16	2	3.4e+12	25	GFlops	1.3e+02	16	4.0e+10	77	GFlops	5.1e-01	260x



(a) OpenMP Integration Time Per Cell (b) OpenMP Speedup

Fig. 2 OpenMP (16 threads, Apple M2 Ultra) comparison for 3D $H(\text{curl})$ integration. Speedup values compare sum-factorized integration to standard integration times

Table 2 shows the CUDA results on an H100. Figure 3 plots a comparison between `Intrepid2`'s standard assembly and the sum-factorized assembly using Algorithm 3. While the performance of sum-factorized assembly is much better than that of the standard assembly algorithm, we note that here it remains sub-optimal, as seen both in peak throughput that is considerably below the theoretical peak of the hardware (26 TFlops for FP64), and in the fact that the estimated throughput peaks at $p = 8$ and diminishes after that. For $p = 9$ and $p = 10$, the calibration results indicate that we are not expressing sufficient parallelism: there is a substantial increase in throughput (nearly a factor of 2) between using a workset that is half the size of the mesh and using one the size of the whole mesh. Figure 4 plots the throughputs for both our OpenMP and CUDA results.

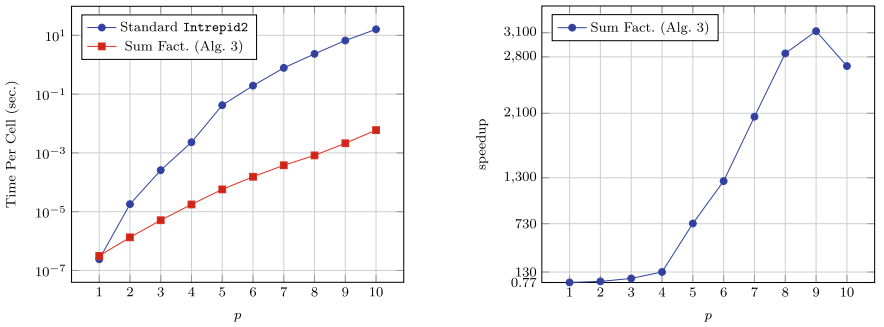
6 Sum-Factorized Assembly for DPG Vlasov Solver

We have been developing proof-of-concept Vlasov solvers, implemented using the `Camellia` code, based on the discontinuous Petrov-Galerkin (DPG) methodology of Demkowicz and Gopalakrishnan [20, 21], with a particular emphasis on adaptive mesh refinement. These can be up to 7D for space-time 3D3V meshes, and can be high-order, particularly in the test space. We have reimplemented a key kernel, the *Gram* matrix assembly for the test-space inner product, using `Intrepid2`'s sum-factorization support, for our space-time Vlasov solver. In Table 3, we compare performance of this kernel for a 1D1V case (3D in all) arising in our Landau damping driver. Due to space considerations, we omit full details, but we note that this is a more holistic measurement, in something closer to a "real-world" application context, than our $H(\text{curl})$ assembly above.

The Vlasov-Poisson solve proceeds using a fixed-point iteration; we measure the assembly cost in the first step of this iteration. Thanks to `Camellia`'s in-progress

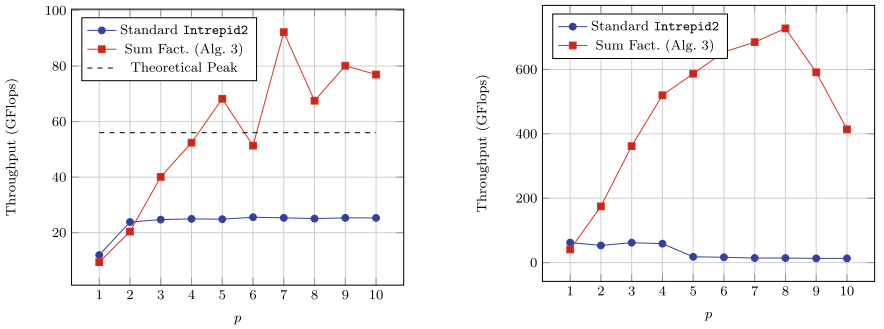
Table 2 Reference-to-physical transform plus integration kernel for $H(\mathrm{curl})$ stiffness matrix, results using CUDA on an H100. Times are in seconds

		Standard				Sum-Factorized (Alg. 3)				
p	Elements	Workset	Flops	Throughput	Time	Workset	Flops	Throughput	Time	Speedup
1	524,288	524,288	7.9e+09	62 GFlops	1.3e-01	65,536	6.7e+09	41 GFlops	1.6e-01	0.77x
2	524,288	65,536	5.0e+11	53 GFlops	9.5e+00	65,536	1.2e+11	170 GFlops	7.0e-01	13x
3	65,536	32,768	1.1e+12	62 GFlops	1.7e+01	65,536	1.2e+11	360 GFlops	3.4e-01	50x
4	16,384	8,192	2.2e+12	59 GFlops	3.8e+01	16,384	1.5e+11	520 GFlops	2.9e-01	130x
5	8,192	2,048	6.2e+12	18 GFlops	3.4e+02	8,192	2.8e+11	590 GFlops	4.7e-01	730x
6	2,048	1,024	6.6e+12	17 GFlops	4.0e+02	2,048	2.1e+11	650 GFlops	3.2e-01	1,300x
7	1,024	64	1.1e+13	14 GFlops	8.0e+02	1,024	2.7e+11	680 GFlops	3.9e-01	2,100x
8	512	64	1.7e+13	14 GFlops	1.2e+03	512	3.1e+11	730 GFlops	4.2e-01	2,800x
9	256	32	2.2e+13	13 GFlops	1.7e+03	256	3.2e+11	590 GFlops	5.5e-01	3,100x
10	128	32	2.7e+13	13 GFlops	2.1e+03	128	3.2e+11	410 GFlops	7.6e-01	2,700x



(a) CUDA Integration Time Per Cell (b) CUDA Speedup

Fig. 3 CUDA (H100) timing comparison for 3D $H(\text{curl})$ integration. Speedup values compare sum-factorized integration to standard integration times



(a) OpenMP Throughput (b) CUDA Throughput

Fig. 4 Throughput estimates for 3D $H(\text{curl})$ integration, for 16-thread OpenMP on M2 Ultra and CUDA on an H100. The flop estimates do not account for fused multiply-adds; we believe this explains how the OpenMP performance can exceed the theoretically expected 56 GFlops. For clarity of the figure, we do not plot the theoretical peak throughput for the H100, which is 26 TFlops

conversion to support thread parallelism, the standard assembly mechanism benefits, but fairly modestly, from enabling OpenMP parallelism. As before, we experiment with a number of different workset sizes, divisors of 160 and 1,280, and select the best case to use in comparisons. Here, we observe speedups that are fairly consistent across polynomial orders, between 1.5x and 1.9x for the serial case, and between 6x and 9x for the threaded case. We do not have a detailed explanation for the more modest speedups, but we believe it to be due to a combination of the inclusion of other computations in the measurement, the structure of these particular integrals, and some inefficiencies in this first implementation using sum-factorized assembly in `Camellia`.

Table 3 Vlasov Gram Assembly Comparison. Times are in seconds.

(a) M2 Ultra (Serial, 160 elements)

	Standard		Sum-Factorized		
p	Workset	Time	Workset	Time	Speedup
5	40	0.61	40	0.31	1.96
6	20	1.6	8	0.80	1.97
7	40	3.7	40	2.0	1.85
8	20	7.5	2	2.1	1.65
9	40	16	2	9.5	1.68
10	40	28	2	19	1.45

(b) M2 Ultra (OpenMP-16, 1,280 elements)

	Standard		Sum-Factorized		
p	Workset	Time	Workset	Time	Speedup
5	40	2.3	1,280	0.30	7.6
6	20	5.8	1,280	0.80	7.3
7	8	12	640	1.7	7.0
8	5	28	320	3.8	7.3
9	8	56	320	6.9	8.0
10	20	120	320	13	8.9

7 Conclusion

In this work, we have described some recently developed features in `Intrepid2` to support algorithms that take advantage of structural features of finite element basis functions, including sum factorization algorithms. The sum factorization support in `Intrepid2` is implemented in a very general fashion that is agnostic both to function spaces and the number of tensorial components; we believe it to be unique in this regard. Our numerical experiments focus on use cases relevant for plasma physics. We have demonstrated substantial performance improvements relative to unstructured $H(\text{curl})$ assembly on both CPU and GPU architectures. We have also shown substantial improvements in a core kernel of `Camellia`'s space-time Vlasov solver, the Gram matrix assembly. Future work will include both providing mechanisms for exploiting symmetry in assembly and employing the same foundational data structures to support matrix-free operation.

Acknowledgements Support was provided through the National Nuclear Security Administration (NNSA) Advanced Simulation and Computing (ASC) program and the U.S. Department of Energy (DOE), Office of Science, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute. Simulations were performed on machines at the National Energy Research Scientific Computing Center (ASCR). Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

Disclaimer. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

1. Kronbichler, M., Kormann, K., Pasichnyk, I., Allalen, M.: Fast matrix-free discontinuous Galerkin kernels on modern computer architectures. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) High Performance Computing, pp. 237–255. Springer International Publishing, Cham (2017)
2. Kolev, T., Fischer, P., Min, M., Dongarra, J., Brown, J., Dobrev, V., Warburton, T., Tomov, S., Shephard, M.S., Abdelfattah, A., Barra, V., Beams, N., Camier, J.-S., Chalmers, N., Dudouit, Y., Karakus, A., Karlin, I., Kerkemeier, S., Lan, Y.-H., Medina, D., Merzari, E., Obabko, A., Pazner, W., Rathnayake, T., Smith, C.W., Spies, L., Swirydowicz, K., Thompson, J., Tomboulides, A., Tomov, V.: Efficient exascale discretizations: High-order finite element methods. The International Journal of High Performance Computing Applications **35**(6), 527–552 (2021). https://doi.org/10.1177/10943420211020803
3. Bochev, P., Edwards, H.C., Kirby, R.C., Peterson, K., Ridzal, D.: Solving PDEs with Intrepid. Sci. Program. **20**, 403902 (2012). https://doi.org/10.3233/SPR-2012-0340
4. Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Trans. Parallel Distrib. Syst. **33**(4), 805–817 (2022). https://doi.org/10.1109/TPDS.2021.3097283
5. Roberts, N.V.: Camellia: A Rapid Development Framework for Finite Element Solvers. Computational Methods in Applied Mathematics **19**(3), 581–602 (2019)
6. Roberts, N.V., Miller, S.T., Bond, S.D., Cyr, E.C.: An implicit-in-time DPG formulation of the 1D1V Vlasov-Poisson equations. Computers & Mathematics with Applications **154**, 103–119 (2024). https://doi.org/10.1016/j.camwa.2023.11.014
7. Roberts, N.V., Bond, S.D., Miller, S.T., Cyr, E.C.: Fluid-kinetic coupling: Advanced discretizations for simulations on emerging heterogeneous architectures, Sandia National Laboratories. SAND2022-13854 Tech. Rep (2022)
8. Bettencourt, M., Brown, D., Cartwright, K.L., Cyr, E.C., Glusa, C., Lin, P., Moore, S., McGregor, D., Pawlowski, R., Phillips, E., Roberts, N.V., Wright, S., Maheswaran, S., Jones, J., Jarvis, S.: EMPIRE-PIC: A performance portable unstructured particle-in-cell code. Communications in Computational Physics **30**(4), 1232–1268 (2021). https://doi.org/10.4208/cicp.OA-2020-0261
9. Hoffman, M.J., Perego, M., Price, S.F., Lipscomb, W.H., Zhang, T., Jacobsen, D., Tezaur, I., Salinger, A.G., Tuminaro, R., Bertagna, L.: MPAS-Albany Land Ice (MALI): a variable-resolution ice sheet model for earth system modeling using voronoi grids. Geoscientific Model Development **11**(9), 3747–3780 (2018). https://doi.org/10.5194/gmd-11-3747-2018
10. Watkins, J., Carlson, M., Shan, K., Tezaur, I., Perego, M., Bertagna, L., Kao, C., Hoffman, M.J., Price, S.F.: Performance portable ice-sheet modeling with mali, The. Int. J. High Perform. Comput. Appl. **37**(5), 600–625 (2023). https://doi.org/10.1177/10943420231183688
11. Tezaur, I.K., Perego, M., Salinger, A.G., Tuminaro, R.S., Price, S.: Albany/FELIX: a parallel, scalable and robust, finite element, first-order stokes approximation ice sheet solver built for advanced analysis. Geoscientific Model Development **8**, 1–24 (2015). https://doi.org/10.5194/gmd-8-1-2015
12. SIERRA Solid Mechanics Team, Sierra/SolidMechanics 5.10 user's guide (2022). https://doi.org/10.2172/1886996
13. Ostien, J.T., Foulk, J.W., Mota, A., Veilleux, M.G.: A 10-node composite tetrahedral finite element for solid mechanics. Int. J. Numer. Meth. Eng. **107**(13), 1145–1170 (2016). https://doi.org/10.1002/nme.5218
14. Mora, J., Demkowicz, L.: Fast integration of DPG matrices based on sum factorization for all the energy spaces. Computational Methods in Applied Mathematics **19**(3), 523–555 (2019). https://doi.org/10.1515/cmam-2018-0205

15. Badger, J., Henneking, S., Demkowicz, L.: Sum factorization for fast integration of DPG matrices on prismatic elementsFinite Elem.Ts Anal. Des. **172** (2020). https://doi.org/10.1016/j.finel.2020.103385
16. Kronbichler, M., Kormann, K.: Fast matrix-free evaluation of discontinuous galerkin finite element operators, ACM Trans. Math. Softw. **45**(3) (2019). https://doi.org/10.1145/3325864
17. Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cerveny, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, T., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., Zampini, S.: MFEM: A modular finite element methods library, Comput. Math. Appl. **81** 42–74, development and Application of Open-source Software for Problems with Numerical PDEs, (2021). https://doi.org/10.1016/j.camwa.2020.06.009
18. Andrej, J., Atallah, N., Bäcker, J.P., Camier, J.S., Copeland, D., Dobrev, V., Dudouit, Y., Duswald, T., Keith, B., Kim, D. and Kolev, T., Lazarov, B., Mittal, K., Pazner, W., S. Petrides, S. Shiraiwa, M. Stowell, V. Tomov, High-performance finite elements with MFEM (2024). arXiv:2402.15940
19. Brown, J., Abdelfattah, A., Barra, V., Beams, N., Camier, J.S., Dobrev, V., Dudouit, Y., Ghaffari, L., Kolev, T., Medina, D. and Pazner, W., Ratnayaka, T., Thompson, J., Tomov, V.: libCEED: Fast algebra for high-order element-based discretizations, J. Open Source Softw. **6**(63), 2945 (2021). https://doi.org/10.21105/joss.02945
20. Demkowicz, V.., Gopalakrishnan, J.: A class of discontinuous Petrov-Galerkin methods. Part I: The transport equation, Comput. Methods Appl. Mech. Engrg. **199**(23–24), 1558–1572 (2010)
21. Demkowicz, L., Gopalakrishnan, J.: A class of discontinuous Petrov-Galerkin methods. Part II: Optimal test functions, Numerical Methods for Partial Differential Equations **27**(1), 70–105 (2011). https://doi.org/10.1002/num.20640

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



Partitioned Multiphysics Simulation of an Electrophysiological Three-Tendon-Biceps Model

Carme Homs-Pons, David Schneider, Frédéric Simonis, Miriam Schulte, and Benjamin Uekermann

Abstract Partitioned coupling approaches are a popular strategy for complex multi-physics simulations due to their modularity. This led to the development of numerous coupling libraries in recent years. This chapter highlights recent advances within the coupling library preCICE, which aims to provide a scalable and user-friendly library for practical applications. We demonstrate its benefits using a complex case study: the simulation of a human biceps muscle. Several features of preCICE need to be used simultaneously: (1) efficient data mapping methods to facilitate the volume coupling of the biceps muscle with its internal fibers, (2) robust quasi-Newton acceleration for the implicit surface coupling of the muscle with its three tendons, and (3) compositional coupling schemes to combine weakly and strongly coupled interactions to an overall coupling of five coupled models.

Keywords Data mapping · Coupling schemes · Neuromuscular system · Electromechanics · Multiscale · High performance computing · Mathematical software

1 Introduction

Many problems in science and engineering can only be understood by taking multiple physical domains into account. This includes the biomedical application of this chapter: an electrophysiological three-tendon-biceps model. To implement an overall multi-physics simulation, coupling of single-physics models and codes is a challenge – a numerical one, but also concerning high-performance computing and software engineering. To tackle this challenge, several numerical coupling libraries have been developed in the past ten years. Notable open-source examples include MUI [23],

C. Homs-Pons (✉) · D. Schneider (✉) · F. Simonis · M. Schulte · B. Uekermann
Institute for Parallel and Distributed Systems, University of Stuttgart, Vaihingen, Germany
e-mail: carme.homs-pons@ipvs.uni-stuttgart.de

D. Schneider
e-mail: david.schneider@ipvs.uni-stuttgart.de

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_8

OpenPALM [7], DTK [21], and preCICE [4]. Coupling libraries follow a divide-and-conquer partitioned approach: They try to reuse existing simulation codes and treat them as black boxes as much as possible, introducing modularity into the simulation setup. They often offer highly-scalable and robust numerical methods for data mapping and fixed-point acceleration, facilitating the coupling of many participants on parallel systems.

In this chapter, we study an electrophysiological three-tendon-biceps simulation as a complex multi-physics example to showcase recent advances of the Precise Code Interaction Coupling Environment (preCICE). Multiple features of preCICE need to be applied simultaneously for this example: coupling of multiple codes with a combination of explicit and implicit coupling schemes, quasi-Newton acceleration for robust coupling, and data mapping of large problems.

1.1 The preCICE Coupling Library

The vision of preCICE is to couple existing simulation codes in a modular manner while only requiring minimally invasive changes to them. Ready-to-use adapters to many established simulation tools and frameworks exist, e.g., OpenFOAM, FEniCS, deal.II, SU2, and more.

preCICE handles four main coupling building blocks: (1) Establishing peer-to-peer communication channels between coupling participants executed in serial or parallel via either TCP/IP sockets or MPI. The communication pattern is computed by sending distributed meshes from the defining participant and repartitioning them on the receiving participant. (2) Data mapping to cope with non-matching coupling meshes at the coupling interface or volume including radial-basis-function interpolation and projection methods. All methods are available in a consistent and a conservative variant. (3) Coupling schemes including explicit and implicit ones or compositions of both to couple arbitrary many participants. Implicit schemes can be stabilized by acceleration methods including (parallel) quasi-Newton variants [16]. (4) Time interpolation via waveform iteration to couple heterogeneous time stepping methods, potentially in a multi-rate fashion.

We explain these building blocks in more detail in the following sections as we use them. They are runtime configurable by a single XML configuration file, which all participants read. For more information, also refer to the extensive online user documentation[1] and the recent reference publication [4].

[1] https://precice.org/.

1.2 The Electrophysiological Three-Tendon-Biceps Model

A skeletal muscle is an active hyperelastic tissue with a hierarchical structure. Our model considers: three hierarchical levels [9]: the full muscle, the muscle fibers, and the sarcomeres, the smallest contracting unit in skeletal muscles. On the top level, the muscle deforms in response to internal and external forces. The internal forces are a result of the contraction of the sarcomeres due to the stimulation of the muscle fibers by motor neurons. When a motor neuron fires, electrical signals travel through the fibers and trigger chemical reactions, which result in turn in the contraction of sarcomeres. In this chapter, we aim to simulate the activation and the deformation of a human biceps and its neighboring tendons. Due to the complex multi-physical and multi-scale nature, detailed simulations are challenging and computationally demanding.

We choose a partitioned approach and couple five participants (cf. Figure. 1): a muscle mechanics participant, a muscle fibers participant, and three tendon mechanics participants. All participants use meshes [12, 13] generated from imaging surface data of the Visible Human Male dataset [22].

The muscle is split into two participants that are volume-coupled, the muscle fibers participant and the muscle mechanics participant. The first is a multi-scale electrophysiology solver that computes the electrochemical processes that activate

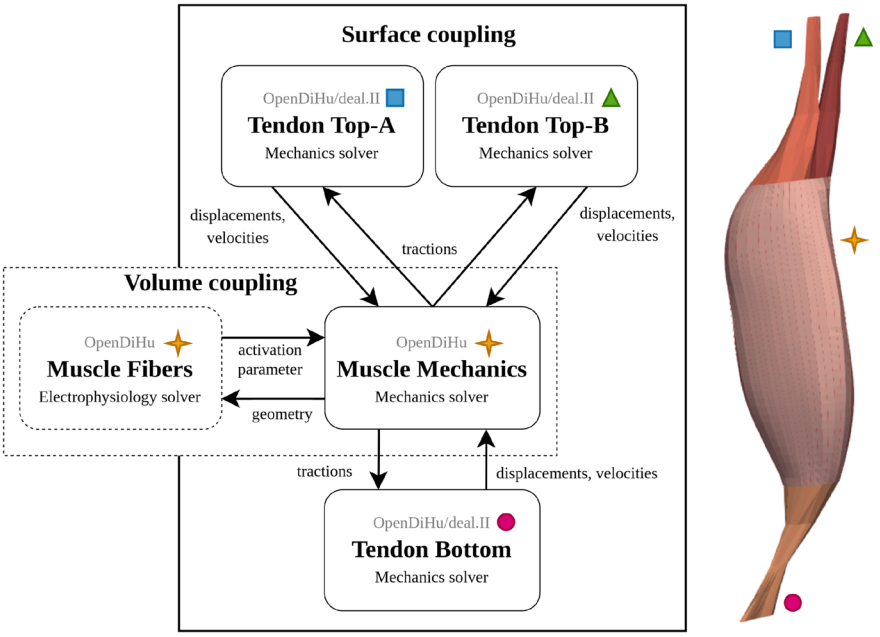


Fig. 1 Coupling topology and geometry of the electrophysiological three-tendon-biceps simulation with five participants

the contraction of the sarcomeres and transfers activation parameters to the muscle mechanics participant. From the activation parameters we compute active forces which, together with external forces, contribute to the deformation of the muscle in the muscle mechanics participant. The deformed geometry is transferred back to the muscle fibers participant. This is a case of weak coupling, i.e., by definition, an explicit coupling scheme is sufficient for stability in general. Because of the small time scale of the electrophysiology, we subcycle in the coupling. The muscle mechanics participant is, moreover, surface-coupled to three tendon mechanics participants: one tendon on the bottom, which is attached to the ulna bone, and two tendons on the top, which are attached to the humerus bone. When the biceps contracts, the ulna is pulled up, whereas the humerus is mostly fixed. Thus we fix the upper end of both top tendons and we apply a pulling force on the bottom tendon that models the forearm's resistance to the biceps' contraction [13]. The tendon participants transfer surface displacement and velocity values to the muscle mechanics participant which transfers back surface traction values. The surface coupling between the mechanics participants is strong, i.e., by definition, we need an implicit coupling scheme to guarantee stability in general.

We make use of existing software for the five participants and couple them via preCICE. This introduces flexibility in the choice of employed models and allows us to distribute computational resources. For the muscle fibers participant and the muscle mechanics participant, we use OpenDiHu[2] ("Digital Human") [15]. In particular, we choose the *FastMonodomain* solver for the muscle fibers participant, which is a multi-scale solver itself: In each fiber we solve a 1D transient reaction-diffusion equation [9]. The reaction term is coupled to an ODE model with 57 states [20], which is computed in each point of a fiber and requires a smaller timestep than the diffusion contribution. Each fiber is, moreover, coupled to a motor neuron at the center of the fiber with a small randomized offset. In our case, we assume a total of 10 motor neurons. For the muscle mechanics participant, we use the dynamic *MuscleContraction* solver, which implements a non-linear transversely isotropic Mooney-Rivlin material model. The activation from the muscle fibers participant is modeled as an additional contribution to the stress tensor [9]. For the three tendon mechanics participants, we extend an existing mechanics solver[3] based on the finite-element library deal.II [1] with a transversely isotropic material model [3]. Our setup, however, also allows using dynamic OpenDiHu solvers for the tendons, specifically the *HyperelasticTendon* model and the *SaintVenantKirchhoff* model. Currently, all dynamic mechanics solvers of OpenDiHu can only be executed in serial, a disadvantage, which we regard as given in this chapter.

To study different coupling aspects separately, we approach the overall five-participant coupling in three stages: In Sect. 2, we only look at the volume coupling between the muscle fibers participant and the muscle mechanics participant. The volume coupling leads to large data mapping problems between meshes with

[2] https://github.com/opendihu/opendihu.

[3] https://github.com/precice/dealii-adapter.

heterogenous resolutions, which we analyze carefully in this section. Such a volume coupling using preCICE was previously published by Maier et al. [14], but not with the data mapping solutions we present in this chapter. In Sect. 3, we then study the mechanical surface coupling between the muscle participant and the three tendon participants. The strong coupling requires an overall implicit coupling scheme with an acceleration method to stabilize the coupling, for which we compare different options in this section. Such a four-participant surface coupling was previously published by Maier et al. [13], however not with advanced acceleration methods, but with constant under-relaxation. In Sect. 4, we finally compose the explicit coupling scheme and the multi implicit one to an overall coupling scheme, the main contribution of this chapter. We study the overall efficiency of the coupling. All software and setups used in this chapter are available in an accompanying data publication [10].

2 Volume Coupling of Muscle Fibers and Muscle Mechanics

The volume coupling between the 1D muscle fibers participant and the 3D muscle mechanics participant is a weak coupling. An explicit coupling scheme is, thus, sufficient (Sect. 2.1). To choose suitable data mappings (Sect. 2.2), we consider different meshes associated to different fiber configurations, see Fig. 2. The coupling mesh of the mechanics solver is by construction a subset of the much finer fiber mesh. We then analyze the influence of this setup and different data mappings on computational efficiency (Sect. 2.3).

No. fibers	fiber vertices	muscle vertices	ratio
7x7	72,569	333	218
13x13	250,289	1,813	138
25x25	925,625	6,253	148
33x33	1,612,809	10,693	151
37x37	2,027,489	13,357	152
67x67	6,648,209	13,357	498

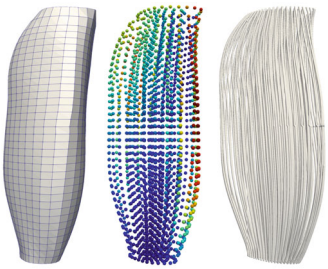


Fig. 2 Number of volume coupling vertices of each participant (left) and 13x13 meshes (right). The meshes show the mechanics mesh (left), the extracted coupling mesh (middle), and the fiber mesh (right). With 13,357 nodes in the muscle mesh, the muscle solver hits the refinement limit for its serial execution and further refinements are not feasible.

2.1 Coupling Scheme

In preCICE, explicit coupling schemes can either be serial or parallel. The former runs one solver at a time, while the other is waiting for data, benefitting stability. The latter runs both solvers simultaneously, overlapping runtime. We use a serial-explicit scheme here. preCICE requires a common time window size at the end of which participants exchange data. This permits solvers to freely select time step sizes within a time window. Here, selecting a time window size equal to the time step size of the mechanics solver 10^{-2} ms leads to stable results. The fibers solver performs 25 time steps of $4 \cdot 10^{-4}$ ms in each time window. From the solver perspective, the coupling logic is hidden inside preCICE. Figure 3 shows a schema of how the solvers run and exchange data in this scenario.

2.2 Data Mapping

Bidirectional couplings require the configuration of two data mappings, which comprises here the mapping from the fibers to the muscle and vice versa (see volume coupling in Fig. 1). For both data mappings, we use the consistent constraint of preCICE, which ensures that the activation parameter and the geometry data are interpolated consistently across the coupling interface. To further configure the data mappings, both accuracy and computational cost need to be taken into account. Considering the much higher number of vertices in the fiber mesh compared to the muscle mesh, as illustrated in Fig. 2, it would be beneficial to not communicate the fiber mesh and corresponding data in general, but to handle communication only via the muscle mesh. In principle, preCICE allows choosing the participant on which the data mapping is computed and which mesh is used for data communication. However, when executed in parallel, consistent mappings are only allowed to be computed on the receiving participant [24]. Hence, mapping the geometry is executed in parallel on the fiber solver whereas the activation parameter is mapped in serial on the mechanics solver.

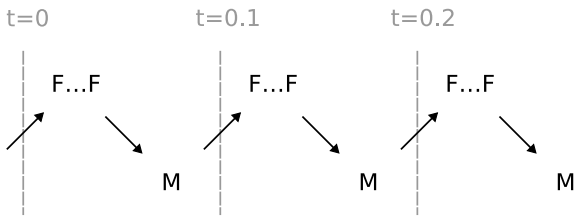


Fig. 3 Schema of the data exchange between muscle fibers (F) and muscle mechanics (M) participants. M provides initial geometry for F. F performs multiple time steps before completing the time window and sending activation data to M. M completes the time window in one time step and sends data to F. In this serial scheme, F runs before M, at which point they both completed a time window and move to the next time window.

As mentioned above, the mechanics mesh is a subset of the fibers mesh. Consequentially, configuring the mapping from the fibers mesh to the mechanics mesh is simple: A nearest-neighbor (NN) search solves the mapping problem mathematically exactly, while remaining computationally cheap. In addition, preCICE filters out all vertices that are not required for the mapping during the initialization. Here, this applies to all vertices that are not included in the mechanics mesh, as their data is not used throughout the simulation. Thus the nearest-neighbor mapping efficiently reduces the computational cost of data communication. Other mapping variants in preCICE would solve the mapping problem exactly as well, but at a much higher computational cost.

Configuring the mapping from the mechanics to the fibers is more challenging. Here, we want to make sure that the geometry is mapped accurately onto the fibers. The coupled codes do not provide any connectivity information for preCICE or additional gradient data, which would otherwise enable the use of linear-cell interpolation or nearest-neighbor gradient mapping methods. Although the mechanics mesh size might appear small enough for employing global radial-basis-function (RBF) interpolation [4, 18], the computational cost of evaluating the global interpolant for all fiber vertices rules out global RBF mappings. As an alternative, we make use of a new advancement of preCICE: partition-of-unity radial-basis function interpolation (PU-RBF) [19]. The conceptual idea behind PU-RBF is to split the problem into smaller, overlapping sub-problems, so-called clusters, and construct RBF interpolants in each of these clusters. A global solution is then recovered by aggregating a weighted sum of the local solutions from each cluster. The expected data from the muscle mechanics participant is a geometry field and thus, expected to be smooth. Therefore, we configure the mapping with a compactly supported Wendland C^6 function [4] and a support radius of $\rho = 1$ cm.

2.3 Computational Efficiency

To investigate the data mapping efficiency, we reproduce the volume coupling of the muscle fibers and the muscle mechanics in an isolated environment using the so-called Artificial Solver Testing Environment (ASTE). [4] ASTE encapsulates the preCICE API and imitates two participants of a coupled simulation. While the same performance measurements could be retrieved from the original cases, ASTE is more feature rich and allows measuring the computational accuracy of the employed mapping alongside the performance. To reproduce the volume coupling, we extract the coupling meshes defined by the muscle mechanics and the muscle fibers using the export functionality of preCICE. The partitionings performed by ASTE and OpenDiHu are not identical, but well comparable, such that using ASTE in the whole pipeline is reasonable. We configure the case to resemble our target simulation setup, which means that we execute the muscle mechanics participant in serial and

[4] https://github.com/precice/aste.

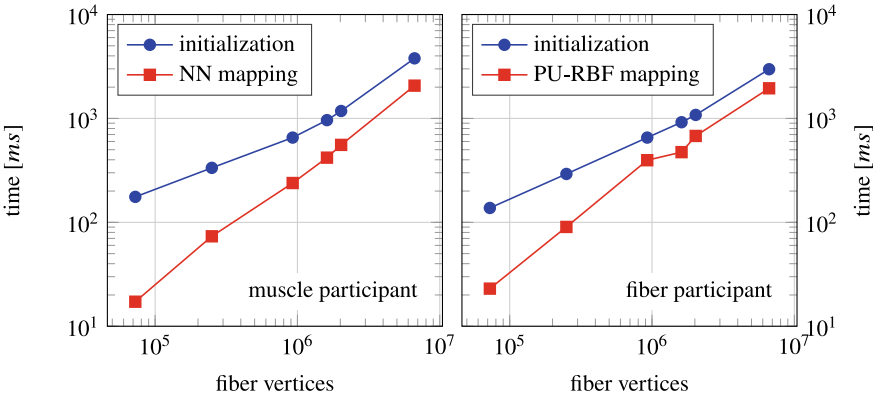


Fig. 4 Total initialization time and time to compute the mapping for each participant. The fiber participant is running on 16 ranks whereas the muscle participant runs on a single rank.

the muscle fibers participant in parallel on 16 ranks. To generate artificial data, we use a transformed variant of the Rosenbrock function, which fits on the spatial measures of the muscle domain [10]. The experiments were conducted on a dual socket AMD EPYC 7742 system (2×64 cores). For all configurations, we perform a total of five runs and report an averaged time over three runs, excluding the slowest and the fastest run. With parallel execution, the timing used to compute the average is the maximum timing across all ranks.

Performance results for both participants are shown in Fig. 4. The mapping computation becomes an increasingly dominant part of the initialization, taking up between 10% and 55% of the computational cost for nearest-neighbor mapping and between 17% and 66% for PU-RBF as the coupling meshes become increasingly refined. Establishing communication channels takes around 100 ms (not shown in the figure), which for coarser meshes can significantly affect the initialization time. Still, we conclude that the overall initialization time is insignificant, taking only around three seconds for the most refined mesh resolutions on both participants, while the largest potential for optimization is given by the PU-RBF data mapping.

Therefore, we consider different mapping configurations for the muscle fibers participant. In particular, we evaluate different options for the number of the vertices per partition-of-unity cluster N_M (default in preCICE is $N_M = 50$) as well as a different support radius ($\rho = 2$ cm). In addition, we compare against an experiment using a nearest-neighbor mapping also on the muscle fibers participant. Figure 5 depicts accuracy and computational cost for all configurations. The nearest-neighbor mapping yields the worst accuracy. Increasing the support radius shows no visible influence on the mapping accuracy. Larger clusters result in better accuracy, but also higher computational cost. The best value for N_M depends on the mesh resolution: For coarser meshes, lowering N_M delivers a similar accuracy at a lower computational cost. For finer meshes, however, the accuracy gain becomes significant.

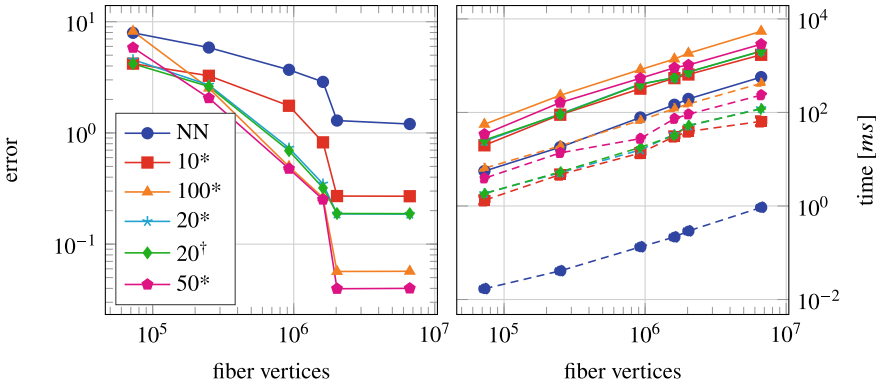


Fig. 5 Accuracy and computational cost of the nearest-neighbor mapping or the PU-RBF mapping for various cluster sizes from the muscle mesh to the fiber mesh. Basis function is Wendland C^6 ($*\rho = 1$,† $\rho = 2$). The solid lines show one-time cost during initialization, whereas dashed lines show cost per time step. The muscle mesh is not further refined in the last experiment, leading to the same accuracy.

3 Multi Coupling of Muscle Mechanics and Three Tendons

The surface coupling between the four mechanics solvers is a strong one, i.e., we require an implicit coupling scheme with a suitable acceleration method (Sect. 3.1). Nevertheless, finding appropriate data mappings is easier: Since the coupling surfaces only contain small amounts of vertices, computational efficiency is not an issue. Hence, we use PU-RBF mappings for all coupling interfaces. As a basis function, we select a Wendland C^4 function with $\rho = 8$ cm, a multiple of the diameter (approximately 3 cm of the coupling surface), ensuring the basis function covers the interface nodes sufficiently well. To identify surface nodes within the muscle participant, we take advantage of the mesh structure and select the top or bottom layer of nodes. The case of the upper end of the muscle is more complex, as the top nodes should be split into two meshes for the two top tendons. Instead we apply a mask which separates the plane in two, so that the muscle mechanics updates its boundary condition with the values received from the correct tendon. To get meaningful activation parameters to trigger the contraction of the muscle, we use OpenDiHu's internal fiber-mechanics coupling [15] in this section. For simplicity, we use OpenDiHu solvers for all four participants.

3.1 Coupling Scheme

Implicit coupling schemes iterate the solvers in each time window until defined convergence criteria are met. Coupling more than two participants implicitly, requires a multi coupling scheme, which performs a parallel coupling of all participants [2].

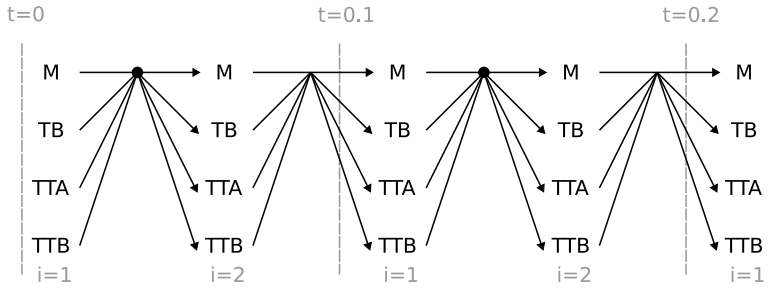


Fig. 6 Data exchange between the muscle participant (M) and the three tendon participants: Tendon-Bottom (TB), Tendon-Top-A (TTA), and Tendon-Top-B (TTB). In this multi scheme, all solvers are coupled in a parallel scheme. After each solver completes its time window, coupling data is gathered on M, which checks for convergence. In this example, the first iteration does not achieve convergence, but the second iteration does. After non-converging iterations, coupling data is accelerated (denoted by the circle).

Strongly coupled problems moreover require an appropriate acceleration method. One participant then takes the role of the controller, responsible for measuring convergence and accelerating the coupling data. In our case, we use the muscle solver as the controller, as it is directly coupled to all other participants, simplifying data transfer. Figure 6 depicts the multi-coupling scheme.

Mathematically, implicit coupling leads to fixed-point equations at the coupling interface, $H(x) = x$. The surface coupling between the four participants is given by $\mathrm{M} : (x_1, x_2, x_3) \mapsto (y_1, y_2, y_3), \mathrm{TB} : y_1 \mapsto x_1, \mathrm{TTA} : y_2 \mapsto x_2$, and $\mathrm{TTB} : y_3 \mapsto x_3$, where the variables denote the coupling data as specified in Fig. 1. We then get the following fixed-point equation:

$$\mathrm{M}(x_1, x_2, x_3) = (y_1, y_2, y_3), \quad \mathrm{TB}(y_1) = x_1, \quad \mathrm{TTA}(y_2) = x_2, \quad \mathrm{TTB}(y_3) = x_3 \ .$$

To solve such fixed-point equations, preCICE offers various acceleration methods A that combine past inputs x and results $H(x)$ using a linear combination to compute the next iterate:

$$x^{k+1} = A(x^0, \ldots, x^k, H(x^0), \ldots, H(x^k)) \ .$$

To this end, preCICE concatenates all coupling data into a single vector, the aforementioned x [24].

Acceleration methods vary in how they compute these linear coefficients. preCICE offers three accelerations options: constant under-relaxation, Aitken's under-relaxation and various quasi-Newton variants. The simplest variant is constant under-relaxation, which applies a fixed relaxation factor c to combine the latest two iterations $x^{k+1} = cH(x^k) + (1 - c)x^k$. Aitken's under-relaxation computes a dynamic relaxation factor based on the residuals of the latest iteration. Quasi-Newton

acceleration methods also compute a dynamic relaxation factor but they incorporate previous iterations to estimate the Jacobian. Typically, quasi-Newton methods are more efficient and robust in most scenarios [24].

To find a suitable acceleration scheme for our multi-coupling scenario, we conduct separate experiments. We switch between acceleration methods by modifying the preCICE's XML configuration file. We perform a fixed number of 100 time windows using various acceleration schemes and compare resulting iteration counts as well as their convergence behavior. We select the time window size to be equal to the solvers' timestep, i.e., 0.01 ms. The simulation setup mirrors the final simulation of Sect. 4, where we use 9×9 muscle fibers and a relative residual convergence measure of 10^{-2} for displacement and velocity and 10^{-1} for traction. In particular, we compare the use of no acceleration scheme, the use of constant under-relaxation ($c = 0.5$), Aitken's under-relaxation, and two different interface quasi-Newton (IQN) methods. We select preCICE's default configuration parameters for each method. For the IQN methods, we evaluate the least-squares method IQN-ILS [5] with a maximum of 50 or 100 considered past iterations as well as the multi-vector-jacobian IQN-IMVJ [11]. The maximum number of coupling iterations is set to 25.

Results are depicted in Fig. 7. Without acceleration, we steadily hit the maximum number of 25 iterations. The residual of the coupling data oscillates and fails to reach the convergence criteria. Constant under-relaxation reaches the convergence criteria in a robust manner, but only very slowly. In contrast, Aitken's under-relaxation fails to converge in many time windows (suppressed from Fig. 7 for readability). The IQN variants demonstrate superior performance. Precisely, IQN-ILS stabilizes between two and three coupling iterations and IQN-IMVJ between five and six. We conclude that IQN-ILS, reusing 50 iterations, demonstrates the best performance and we, thus, adopt it for the overall five-participant setup (Sect. 4).

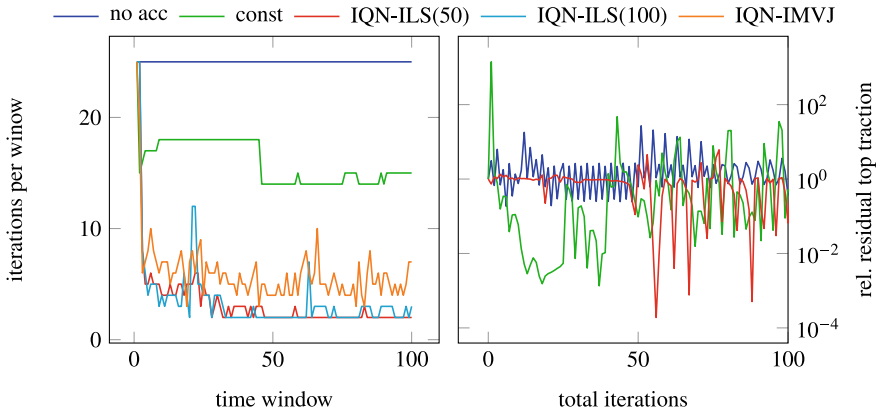


Fig. 7 Iterations per time window (left) and relative residual of the top muscle traction for the first 100 coupling iterations (right) for various acceleration configurations. For readability, the residual plot does not show all data lines. Without acceleration, the maximum number of iterations is reached in every time window without reaching the convergence limit.

4 Compositional Coupling of All Five Participants

We merge the volume coupling of the two muscle participants (Sect. 2) and the surface coupling of the four mechanics participants (Sect. 3) into an overall coupling of five participants. To this end, we *compose* both coupling schemes in preCICE into a single one and study its efficiency.

To couple arbitrarily many participants, preCICE allows defining multiple coupling schemes, which are internally merged into a single *compositional coupling scheme* following a composite pattern, which composes multiple coupling schemes into a single object behaving like a coupling scheme [8]. If only explicit coupling schemes are composed, they simply advance synchronously in time. If explicit and implicit schemes are combined, the implemented logic gets more complicated as implicit schemes can repeat time windows. Defining more than one implicit scheme, however, is forbidden to prevent ill-defined setups [2]. Instead, multiple strong couplings should be merged into a single implicit multi-coupling scheme as discussed in Sect. 3.1. Thus, at most one implicit scheme needs to be handled in a composition. To then sort out the one implicit and potentially multiple explicit schemes requires distinguishing between the first iteration and follow-up iterations in a time window. In the first iteration of a time window, all defined schemes advance synchronously until the implicit scheme has reached the end of its time window. At this point, all explicit schemes are on hold and the implicit scheme repeats the time window until convergence. Only then, all coupling schemes start the next time window.

In our five-participant setup, we compose two coupling schemes: an explicit coupling and an implicit multi-coupling scheme, see Fig. 8.

Figure 9 shows the simulation results of coupling five OpenDiHu participants after 30 ms for a coarse setup with 9×9 muscle fibers. The muscle contracts due to the activation of the fibers. Since the top tendons are fixed, the lower end of the muscle

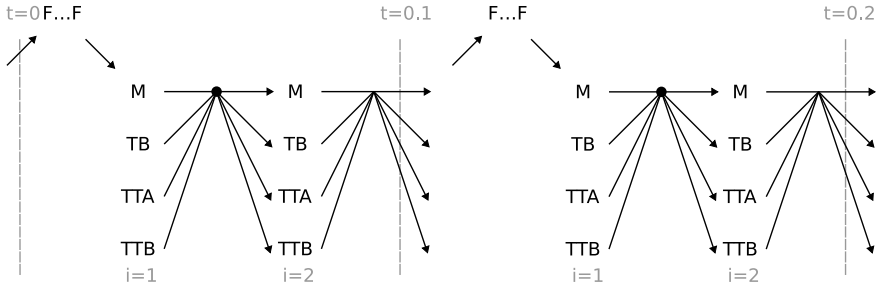


Fig. 8 Composition of the explicit coupling scheme of Fig. 3 and the implicit multi coupling scheme of Fig. 6. The mechanics solver (M) runs after the fibers solver (F) and before the first iteration of the multi coupling scheme. After the multi scheme converges, the converged geometry data is sent to F to continue with the next time window. Hence, M is blocked by F, while the tendon solvers start their next iteration right after they receive data from M. As M is the second participant in the serial coupling scheme, the time window ends with the successful iteration of the multi-coupling scheme.

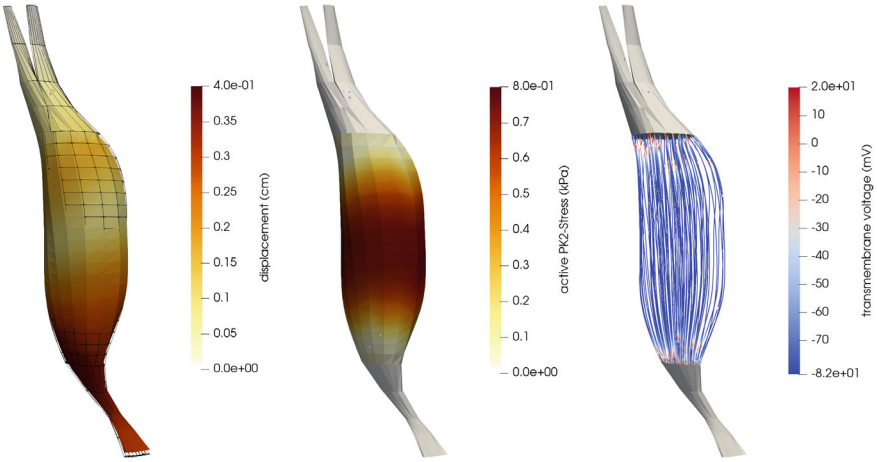


Fig. 9 Results at t = 30 ms: The displacement computed by the muscle mechanics and the tendon participants (left), the active stress in the muscle (center) and the transmembrane voltage in the fibers (right). The black grid shows the initial geometry (left).

shows a higher displacement than the top end. The active stress in the muscle is a direct result of the activation of the fibers, so it builds up in the center and propagates towards the muscle ends. For this experiment, all motor neurons fire in a ramp during the first 2 ms, which means that we set the transmembrane voltage to 20 mV around the center of the fiber. The peak in the transmembrane voltage travels towards the extremes of the fibers.

For this setup with only OpenDiHu participants, we have to execute all mechanics solvers in serial, while the fiber participant uses 16 parallel ranks. The first 10 ms of the simulation require 2 h 22 min to compute. 71.9% of the time is spent in the muscle mechanics, 85.7% in the bottom tendon, 30.8% in the top-A tendon and 34.4% in the top-B tendon. Switching to the deal.II solver for the tendons, we could execute them in parallel as well. Even if we execute them in serial, the overall simulation time drops to 1 h 35 min already, of which 93.4% is spent in the muscle mechanics, 8.5% in the bottom tendon, 6.1% in the top-A tendon and 6.6% in the top-B tendon. Thus, independent of the parallelization of the tendons, the muscle mechanics solver then becomes a bottleneck. Going to 25×25 fibers, the simulation of 10 ms takes around 46 h with more than 99% of time spent by the muscle mechanics.

5 Conclusion

We showcased how the coupling library preCICE can be of great help to solve complex multi-physics simulations involving many components. As an example, we studied an electrophysiological three-tendon-biceps model, which requires the

coupling of five components. Multiple features of preCICE needed to be used simultaneously: partition-of-unity radial-basis-function interpolation to solve large data mapping problems encountered in volume coupling, quasi-Newton acceleration to stabilize strongly coupled subproblems, and composition of coupling schemes to combine explicit and implicit schemes. The modularity of the overall approach was crucial to successfully handle a test case of this complexity.

The overall runtime of the coupled simulation is dominated by the muscle mechanics solver, which unfortunately could only be executed in serial, for reasons beyond this contribution. In future work, we want to study the efficiency of the coupling and the simulation results in more detail once this solver is parallelized.

Ongoing and future developments of preCICE[5] include, for instance, advanced coupling schemes combining waveform iterations with quasi-Newton acceleration [17], more efficient volume coupling, and one-to-many two-scale coupling [6] – all features of high relevance for biomedical application such as the one discussed in this chapter.

Acknowledgements Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project numbers 515015468 and 528693298, under Germany's Excellence Strategy - EXC 2075 – 390740016, and under SPP 2311 (390740016 and 465243391). The authors acknowledge the support by the Stuttgart Center for Simulation Science (SimTech).

References

1. Arndt, D., et al.: "The dealII library, version 9.5. J. Numer. Math. **31**(3), 231–246 (2023)
2. Bungartz, H.J., et al. A plug-and-play coupling approach for parallel multi-field simulations. Comput. Mech. **55**(6), 1119–1129 (2015). https://doi.org/10.1007/s00466-014-1113-2
3. Carniel, T., Francello, E.: A transversely isotropic coupled hyperelastic model for the mechanical behavior of tendons. J. Biomech. **54**, 49–57 (2017)
4. Chourdakis, G., et al. "preCICE v2: A sustainable and user-friendly coupling library [version 2; peer review: 2 approved]". Open Res. Europe, **2**(51) (2022). https://doi.org/10.12688/openreseurope.14445.2
5. Degroote, J., et al.: Performance of a new partitioned procedure versus a monolithic procedure in fluid-structure interaction. Comput. and Struct. **87**(11–12), 793–801 (2009). https://doi.org/10.1016/j.compstruc.2008.11.013
6. Desai, I., et al.: Micro Manager: a Python package for adaptive and flexible two-scale coupling. J. Open Source Softw., **8**(91), 5842 (2023). https://doi.org/10.21105/joss.05842
7. Duchaine, F., et al.: Analysis of high performance conjugate heat transfer with the OpenPALM coupler. Comput. Sci. Discovery, **8**(1), 015003 (2015). https://doi.org/10.1088/1749-4699/8/1/015003
8. Gatzhammer, B.: Efficient and flexible partitioned simulation of fluid-structure interactions. Dissertation. Department of Informatics, Technical University of Munich, (2015). URL: https://mediatum.ub.tum.de/?id=1461618
9. Heidlauf, T., Röhrle, O.: A multiscale chemo-electro-mechanical skeletal muscle model to analyze muscle contraction and force generation for different muscle fiber arrangements. Front. Physiol. **5** (2014). https://doi.org/10.3389/fphys.2014.00498

[5] preCICE roadmap: https://precice.org/fundamentals-roadmap.html.

10. Homs-Pons, C., et al.: Replication Data for: Partitioned Multiphysics Simulation of an Electrophysiological Three-Tendon-Biceps Model, (2024). https://doi.org/10.18419/darus-4228
11. Lindner, F., et al.: A comparison of various quasi–Newton schemes for partitioned fluid-structure interaction. ECCOMAS CP, (2015). URL: http://hdl.handle.net/2117/191193
12. Maier, B.: Input data for OpenDiHu simulations. Version 1.3. Zenodo, (2021). https://doi.org/10.5281/zenodo.4705982
13. Maier, B., Schulte, M.: Mesh generation and multi-scale simulation of a contracting muscle—tendon complex. J. Comput. Sci. Technol. **59**, 101559 (2022). https://doi.org/10.1016/j.jocs.2022.101559
14. Maier, B., et al.: Bridging scales with volume coupling—Scalable simulations of muscle contraction and electromyography. 185–199, (2023). https://doi.org/10.1007/978-3-031-17937-2_11
15. Maier, B., et al.: OpenDiHu: An efficient and scalable framework for biophysical simulations of the neuromuscular system. J. Comput. Sci. **79**, 102291 (2024). https://doi.org/10.1016/j.jocs.2024.102291
16. Mehl, M., et al.: Parallel coupling numerics for partitioned fluid-structure interaction simulations. Comput. Math. Appl. **71**(4), 869–891 (2016). https://doi.org/10.1016/j.camwa.2015.12.025
17. Rüth, B., et al.: Quasi–Newton Waveform Iteration for Partitioned Surface-Coupled Multi-Physics Applications. Int. J. Numer. Methods Eng. **122**(19), 5236–5257 (2020). https://doi.org/10.1002/nme.6443
18. Schneider, D., Schrader, T., Uekermann, B.: Data-parallel radial-basis-function interpolation in preCICE. ECCOMAS CP. (2023)
19. D. Schneider and B. Uekermann.: Efficient partition-of-unity radial-basis-function interpolation for coupled problems. SIAM J Sci. Comput. **47**(2), B558–B582 (2025). https://doi.org/10.1137/24M1663843
20. Shorten, P., et al.: A mathematical model of fatigue in skeletal muscle force contraction. J. Muscle Res. Cell Motil. **28** (2007). https://doi.org/10.1007/s10974-007-9125-6
21. Slattery, S., Wilson, P.P.H., Pawlowski, R.: The data transfer kit: a geometric rendezvous-based tool for multiphysics data transfer. M and C 2013. 5–9, (2013). URL: https://www.osti.gov/biblio/22212795
22. Spitzer, V., et al.: The Visible Human Male: A Technical Report. J. Am. Med. Inf. Assoc. **3**(2), 118–130 (1996). https://doi.org/10.1136/jamia.1996.96236280
23. Tang, Y., et al.: Multiscale Universal Interface: A concurrent framework for coupling heterogeneous solvers. J. Comput. Phys. **297**, 13–31 (2015)
24. Uekermann, B.: Partitioned fluid-structure interaction on massively parallel systems. Dissertation. Department of Informatics, Technical University of Munich, (2016). https://doi.org/10.14459/2016md1320661

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.



A COGENT Case Study: Supporting Applications with Chombo

Daniel F. Martin, Milo Dorr, Mikhail Dorf, and Lee F. Ricketson

Abstract We present a case study of how a software framework (Chombo) supported the specific needs of a scientific application (COGENT). Since its inception in 2000, the Chombo framework has supported various applications. One example of such support has been the collaboration with the Edge Simulation Laboratory to build the COGENT model. The specific needs of the COGENT effort required the design and implementation of a set of new capabilities in the Chombo framework, such as higher-order mapped-multiblock discretizations and multi-dimensional code organization. These capabilities allowed COGENT to develop a unique simulation capability for modeling the edge layers in tokamaks. Once developed, these capabilities were able to support other applications which had similar needs.

Keywords Mapped multiblock meshes · Gyrokinetic plasma simulations · Phase space methods · Tokamak modeling · High-order finite volume methods

1 Introduction

Practical fusion energy remains a key research goal of the US Department of Energy (DOE) [9] and is one of the NAE's Grand Challenges of Engineering. [21] The tokamak is a promising design candidate for magnetic confinement fusion energy. [15] The plasma in a tokamak forms two distinct regions: the core (the inner torus in which

D. F. Martin (✉)
Lawrence Berkeley National Laboratory, Livermore, CA, USA
e-mail: DFMartin@lbl.gov

M. Dorr · M. Dorf · L. F. Ricketson
Lawrence Livermore National Laboratory, Livermore, CA, USA
e-mail: milodorr@llnl.gov

M. Dorf
e-mail: dorf1@llnl.gov

L. F. Ricketson
e-mail: Ricketson1@llnl.gov

© The Author(s) 2027

C. S. Woodward et al. (eds.), *Numerical Software Libraries and Their Impact on Scientific Applications*, ICIAM2023 Springer Series 3,
https://doi.org/10.1007/978-981-92-0623-0_9

the fusion reaction occurs), and the outer edge region (a region of strong gradients, which regulates particle and heat exhaust). Accurately and efficiently modeling plasmas in the edge regions of tokamaks is essential to understanding the performance the tokamak system, but is extremely challenging due to the complex geometry, strong gradients and wide range of dynamically important length and time scales. Building such computational models often requires specialized mathematical formulations and their expression in software. In this work, we describe capabilities that the Chombo modeling framework developed in support of the COGENT gyrokinetic modeling code [12, 14], and touch on how these capabilities have proved useful to other efforts.

2 Modeling the Edge Region: The Gyrokinetic Approximation

Modeling the edge region of a tokamak presents many numerical difficulties. The geometry of the edge region itself is complicated, defined on the outside by the device boundaries and centered around a magnetic separatrix, inside of which magnetic field lines are closed, and outside of which, they are open (Fig. 1). Transport in this region is highly anisotropic, with characteristic lengths along magnetic field lines being about three orders of magnitude larger than that in the perpendicular direction. Due to weak collisionality just inside the magnetic separatrix, a kinetic description of a tokamak edge plasma is needed. Moreover, the presence of steep edge gradients generates large deviations from local thermodynamic equilibrium. This necessitates a so-called full-F simulation model (such models use both continuum [12] or Particle-In-Cell (PIC) [5] discretizations), in contrast to the simplified and more efficient delta-F models [4, 6] available in the core due to the proximity to local thermodynamic equilibrium.

Strong plasma magnetization enables the use of the gyrokinetic approximation, in which averaging over fast particle gyromotion is performed and a full 6D kinetic particle distribution can be accurately represented by a 5D gyro-averaged distribution function. Still, a 5D strongly anisotropic transport problem has to be solved in a X-point geometry.

Although particle-in-cell codes are less sensitive to geometrical features and have been used for edge plasma modeling [5], the signal-to-noise ratio decreases only as $1/\sqrt{N}$ in full-F simulations (in contrast to $1/N$ dependence for the delta-f calculations suitable for the core region), which motivates development of a continuum approach.

In a full-F approach, we simultaneously model large-scale/large-amplitude background dynamics and small-scale/small-amplitude turbulence. High numerical accuracy (in the form of higher-order discretizations) is particularly important to ensure that numerical (truncation) errors from the large-amplitude background do not overwhelm small-amplitude turbulence processes.

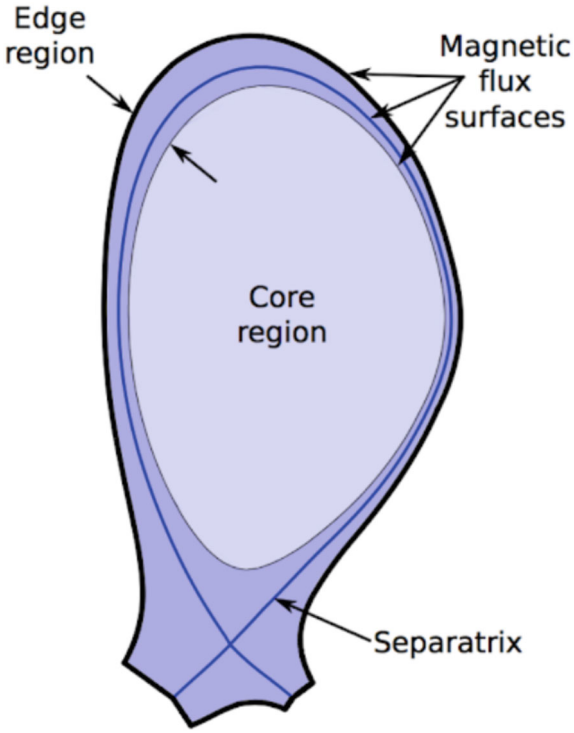


Fig. 1 Schematic of tokamak edge region – the separatrix is the outermost magnetic flux surface which is closed. This geometry is also known as an "X-point geometry" due to the crossing of the separatrix – the specific location is known as the "X-point"

Additionally, computationally efficient continuum modeling of strongly anisotropic transport requires the use of meshing which is aligned with magnetic field lines and magnetic flux surfaces. However, such a meshing approach poses significant challenges due to diverging metric factors at the X-point. Developed by the Edge Simulation Laboratory collaboration, the finite-volume gyrokinetic code COGENT handles these difficulties by adopting a mapped multiblock discretization scheme that can enable high-accuracy and computationally efficient simulations in X-point geometries.

3 The Chombo Framework

Developed at the Lawrence Berkeley National Laboratory, the DOE-supported Chombo [1] framework has been a "developer's toolbox" for implementing highly scalable block-structured adaptive mesh refinement (AMR) algorithms for Depart-

ment of Energy applications since its initial release in 2000. Because many of the algorithmic building blocks for such algorithms are both complex to implement and are also shared across many applications, Chombo provides a set of tools for implementing finite difference and finite volume methods for the solution of partial differential equations on block-structured adaptively refined logically-rectangular grids.

Both elliptic and time-dependent modules are included. Chombo supports calculations in complex geometries with both embedded boundaries and mapped grids, and also supports particle methods. Most parallel platforms are supported, and cross-platform self-describing file formats are included via an interface with HDF5. While not formally part of xSDK [25], Chombo adheres to all xSDK mandatory policies other than policy *M1* (the requirement for a Spack installation), relying instead on a well-documented customized build system.

Chombo maintains interoperability with a number of community libraries, including PETSc [2] and hypre [16] for linear and nonlinear solvers, SUNDIALS [18, 19] for time integration, and FFTW [17] for Fast Fourier Transforms. Chombo has also long had an in-house regression testing system; a set of machines are dedicated to constantly check-out, build, and run a suite of Chombo tests and applications in a variety of configurations. As capabilities were added to Chombo in support of COGENT, specific regression tests were added to the Chombo suite to protect the new capabilities, which has allowed the Chombo team to quickly catch otherwise difficult-to-find changes which might impact these capabilities.

4 Higher-Order Mapped Multiblock Finite-Volume Discretizations in Chombo

Like many DOE problems, the physics in tokamaks have natural coordinate alignments, as described in Sect. 2. Taking advantage of these alignments when constructing discretizations yields substantial improvements in efficiency and accuracy, but they are often too complex for traditional mapped-grid approaches. High-order Mapped-Multiblock (HO MMB) finite volume discretizations, on the other hand, are a general and mathematically consistent way to compose sets of localized mappings to span complex domains.

Our approach to these geometrically-complex computational domains is to decompose the domain into distinct regions in the configuration space in which simpler mappings can be accurately generated, and then compose these sets of "mapping blocks" to represent the entire domain (see Fig. 2). High-order discretizations (greater than 2) are essential for this because we expect to lose an order of accuracy at *multiblock boundaries*, the location of which will be fixed. (This is unlike the case for AMR, where we also lose an order of accuracy at coarse-fine interfaces, but have the flexibility to place those interfaces and their corresponding loss of accuracy

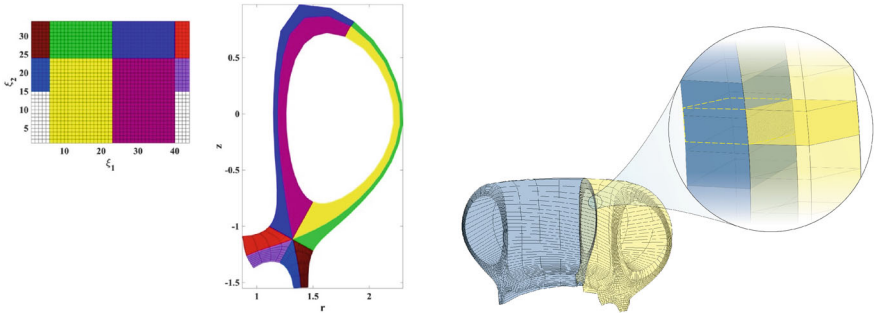


Fig. 2 Mapped-multiblock decomposition of the edge region of a tokamak. *(left)* Data is organized in logically-rectangular blocks in computational space, which corresponds to the mapped-grid cross-sectional domain (middle) (each color represents a separate mapping block, which employs a distinct mapping function. Mapping blocks meet at *multiblock interfaces*. (right) Full 3D mapped-multiblock discretization, showing block boundaries in the axial direction. While block boundaries are conformal in the cross-sectional mapping, we relax this requirement in the axial direction to account for the non-integral rotation of the magnetic field-aligned coordinates in the axial direction

in locations where the solution accuracy won't be significantly degraded. In the mapped-multiblock approach, we will not have that flexibility.)

While it is well-known in numerical analysis that one can drop an order of accuracy on a set that is one dimension less than the domain and still retain the overall accuracy of the method, dropping to first-order in spatial accuracy in a globally second-order method is unacceptable due to the resulting degradation in solution quality at first order. However, dropping accuracy from fourth-order to third-order is acceptable for these problems.

Our multiblock discretizations are composed of two parts: the basic fourth-order mapped-grid discretizations in each mapping block, and the multiblock interpolation used to tie the discretizations and solutions in each block together.

Fourth-order mapped-grid discretizations

In the interiors of mapping blocks, we base our discretizations on the fourth-order freestream preserving finite-volume formalism described in [7]. Each mapping block is defined on a logically-rectangular subset of the domain, and each block is integrated using the same global timestep. The task for each of these mapping blocks is to generate mappings which are accurate and differentiable enough to compute metric terms to $O(\Delta x^5)$. For a fourth-order finite-volume method, some care must also be taken in discriminating between fourth-order point values (used for computing local physics, for example), and fourth-order cell-averaged (for computing conservative finite-volume updates) and face-averaged (for computing finite-volume fluxes) values. The resulting algorithms wind up convolving and deconvolving between averaged and point values depending on the specific needs of the algorithm.

In software, the expression of this is object-oriented classes (derived from an interface class which defines the API), along with a set of utlility functions to manage fourth-order convolution with the mapping terms and the transitions back and forth between point values and cell- and face-averaged values.

Multiblock interpolation

To fill stencils at multiblock boundaries, we extend the mapping of each block past its boundaries to create layers of "ghost cells", and then use a high-order least-squares approach to interpolate values from the relevant other mappings to the appropriately-centered locations in the ghost regions. In some cases, specific mappings may even be periodic in particular directions and wind up copying from other parts of the same mapping block. (see Fig. 3).

Developing the infrastructure required to manage the sorts of connectivity required for complex mapped-multiblock configurations required considerable development work in the Chombo framework. Fortunately, Chombo's modular and extensible object-oriented design made these extensions possible. For example, we were able to use C++ inheritance to derive multiblock specializations of the classes which manage interactions like connectivity between patches in the domain (which was already present to support Chombo's AMR capabilities). This also enabled MMB capabilities to be added to the existing Chombo release versions rather than requiring deep changes and special branches. This in turn made it easier for other applications to take advantage of these capabilities, as described briefly in Sect. 7.

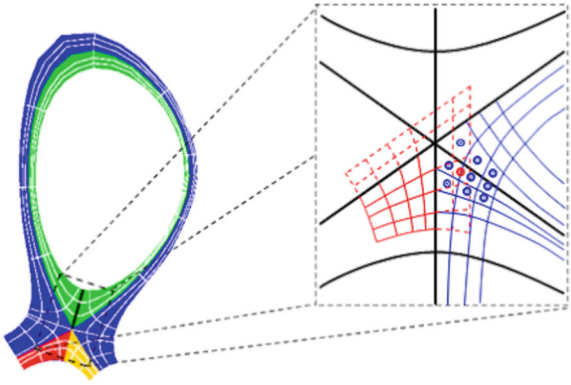


Fig. 3 Example of the multiblock interpolation used as a boundary condition at multiblock boundaries. To fill ghost cells in the red mesh, we extend its mapping beyond the MMB boundary to create ghost regions in that mapping. We then use a higher-order least-squares interpolation from cells in the blue regions to fill ghost cell values like the one shown (red circle)

5 Mixed-Dimensional Chombo

In Chombo, dimensionality is a compile-time parameter. This is useful because many of the discretizations and operators implemented in Chombo have specific features depending on the dimensionality of the problem: for example, two-dimensional hyperbolic advection and elliptic-operator discretizations are much simpler than their three-dimensional counterparts, particularly near multiblock, domain, and AMR boundaries. Chombo also makes extensive use of multigrid approaches for its linear and nonlinear solvers – the coarsening and refinement operations inherent in these methods are also inherently dimensional (coarsening and refining an $N \times N$ mesh in 2D is not the same as an $N \times N \times 1$ mesh in 3D, for example). The Chombo build system then encodes information such as dimensionality in the names of its libraries and executables, allowing for multiple builds of Chombo applications to co-exist in the same directories, which is often convenient for developers and users. (For instance, building an executable "driver" built with `DIM =2`, using `mpicc` and mpif90 for C++ and Fortran compilers, and with optimization turned on, would produce an executable file named `driver2d.Linux.64.mpiCC.mpif90.OPT.MPI.ex`, and object and dependency files generated during the build are maintained in similarly-coded subdirectories to the build directory. This is often convenient for developers and users who wish to maintain multiple builds in a single location.)

In the phase-space discretization used by COGENT, we evolve distributions in a 4- or 5-dimensional space formed by a tensor product of a two- or three-dimensional spatial mesh (location in physical space) and a two-dimensional mesh in velocity space mesh (the combination of the gyrokinetic approximation and magnetic-field-aligned coordinates allows for the elimination of the third velocity dimension). While there are parts of the algorithm which are carried out in the full high-dimensional space, other parts of the algorithm are carried out entirely in either physical (configuration) space or in velocity space. These parts of the algorithm are fundamentally lower-dimensional, and it is convenient and efficient to use the corresponding dimensionally-consistent version of Chombo. In particular, we want to be able to directly leverage the commonly-used two- and three-dimensional implementations in the Chombo framework where possible, rather than rewrite custom implementations which only operate on subsets of a high-dimensional space. This enhances usability and code correctness, since the standard implementations are heaviliy used and tested over a range of applications, which would be less true for custom implementations used only by a single application.

To enable this, Chombo implemented a framework which supports mixed-dimensional algorithms which can use the native Chombo implementations in each dimensionality. This capability was enabled using C++ namespaces. The ability to encapsulate Chombo in a `Chombo` namespace was already required by the xSDK policies (M9) to allow for safe interoperability with other libraries [25]; implementing mixed-dimension programming entailed adding additional dimension-specific namespaces (`Chombo::D2`, `Chombo::D3`, etc.). A set of interface functions live outside the dimensional namespaces and manage interdimensonal operations like

slicing higher dimensional spaces into lower ones and injection of data from lower-dimensional spaces into higher ones. Another set of operators was then implemented in order to manage spreading and reduction operations in Chombo's distributed MPI domains.

For example, imagine a phase-space algorithm with a distribution function ϕ^n defined over the full 4D phase space at time t^n: $\phi(\mathbf{x}, \mathbf{u}, t^n) = \phi(x, y, u, v, t^n)$, in which one computes a source term (like a collision operator) in velocity space (dimensions 2 and 3), integrates the source term over velocity space and applies it to a set of operations in configuration space (dimensions 0 and 1):

1. $\phi^n = \phi(x, y, u, v, t^n)$ on Ω_4
2. $S_u(x, y, u, v, t) = f(u, v)$ on Ω_{vel}
3. $S_x^n(x, y) = \int_{u,v} S(x, y, u, v, t^n) d\mathbf{u}$
4. $\phi_x(x, y)^n = \int_{u,v} \phi(x, y, u, v, t^n) d\mathbf{u}$
5. $\phi_x^{n+1}(x, y) = \phi_x^n(x, y) + \Delta t S_x^n(x, y)$

Implemented in mixed-dimensional Chombo, this algorithm would look like:

1. `Chombo::D4`: $\phi^n = \phi(x, y, u, v, t^n)$ on Ω_4
2. `Chombo::D4`: $S_u(x, y, u, v, t) = f(u, v)$ on Ω_{vel}
3. `Chombo`: $S_x^n(x, y) = \texttt{Slice}_{4D \to 2D}\left(\int_{u,v} S(x, y, u, v, t^n) d\mathbf{u}\right)$
4. `Chombo::D2` $\phi_x(x, y)^n = \int_{u,v} \phi(x, y, u, v, t^n) d\mathbf{u}$
5. `Chombo::D2` $\phi_x^{n+1}(x, y) = \phi_x^n(x, y) + \Delta t S_x^n(x, y)$

In other words, steps 1 and 2 are computed in the full 4-dimensional space in the namespace `Chombo::D4`, and steps 4 and 5 are computed in Chombo's 2-dimensional namespace `Chombo::D2`. Step 3 exists in a nondimensional Chombo namespace, and combines a discrete reduction operator (the integral over velocity space, which is performed in `Chombo::D4`) with `Slice`, which is an interdimensional operator which copies a 2D slice of the 4D Chombo domain into the 2-dimensional space.

Both the reduction and slicing operations (along with their spreading and injection operations) are designed to operate efficiently and correctly for any parallel decomposition of the domain and for the cell-centered and face-centered data centerings that are common in finite-volume algorithms. Implementing the mixed-dimensional Chombo build required significant extensions to the Chombo build system to incorporate the correct system of namespaces and library and application source-code compilation. In practice, however, it greatly simplifies implementing phase-space algorithms such as that used by COGENT because it allows developers to leverage Chombo implementations designed specifically for the dimensionality that one is operating in.

6 COGENT

The finite-volume gyrokinetic code COGENT (COntinuum Gyrokinetic Edge New Technology) has been developed by the Edge Simulations Laboratory (ESL) collaboration for edge plasma modeling. The code has a number of physical models including (i) a fully-kinetic approach in which a 5D gyrokinetic equation is employed for all plasma species (i.e., electrons and ions) and coupled to a 3D electrostatic gyro-Poisson equation for electrostatic potential perturbations $\Phi(R)$ [20], and (ii) a hybrid approach in which a gyrokinetic equation for the ion species is coupled to a 3D quasi-neutrality equation for the vorticity variable, $\varpi = \nabla_\perp(\alpha\nabla_\perp\Phi)$, and a 3D fluid model for the electron species [12]. The gyrokinetic equation represents an advection equation, where the phase-space velocity depends on the values of electric field, $\mathbf{E} = -\nabla\Phi$. When collisions are included, drag and diffusion terms in the velocity space are added. The gyro-Poisson equation is an elliptic equation for a perpendicular (to the magnetic field) Laplacian operator, and the quasi-neutrality vorticity equation is written for a time derivative of a vorticity variable and involves both stiff (diffusion-like) and non-stiff terms in the right-hand-side [12].

The ion-scale transport simulations (which is the main focus of the COGENT code) are stiff and could benefit greatly from implicit time integration in order to step over fast time scales associated, for example, with electron dynamics. In the case of a hybrid model, all fast time scales are contained within the 3D field/fluid part of the hybrid system and therefore only the 3D system needs to be treated implicitly. The 5D gyrokinetic ion system can be treated explicitly. To that end, a consistent high-order time integration approach that includes implicit treatment of selected stiff terms is implemented in COGENT. It is based on semi-implicit additive Runge–Kutta (ARK) methods and employs the Jacobian-free Newton-Krylov (JFNK) approach to handle nonlinearities. Multigrid solvers from hypre are used for preconditioning purposes to improve JFNK convergence properties.

The COGENT code has axisymmetric (2D+2V) and non-axisymmetric (3D+2V) versions, with the former version used to assess the properties of collisional transport and the latter version employed to study the effects of microturbulence. Either collisional or microturbulence transport is highly anisotropic and requires the use of mapped multiblock technology to perform simulations in X-point geometries.

Development of COGENT benefited a great deal from being able to take advantage of Chombo's high-order mapped-multiblock and mixed-dimension programming capabilities (which were developed in anticipation of COGENT's needs). COGENT developers were able to focus on the specific algorithmic needs of building an edge-plasma gyrokinetic code without having to worry about building specific infrastructure for these capabilities. In many cases, COGENT developers were able to use existing Chombo capabilities in combination with HOMMB or mixed-dimension support rather than write custom implementations. Close coordination between COGENT developers and the Chombo development team ensured that issues which did arise were able to be addressed through cooperative efforts.

6.1 Example Results

A successful application of high-order mapped multiblock discretization was demonstrated for axisymmetric ion gyrokinetic advection in a single-null (X-point) geometry [14]. The poloidal grid lines of a global grid are obtained as level surfaces of a modified flux function, Ψ, given by blending the original flux function, ψ_0 (takes the form of $\Psi_0 - \Psi_X \approx \alpha^2 R^2 - \beta^2 Z^2$ in the vicinity of the X point) and the block-aligned linear function $\psi_{lin} = D(|\alpha R| - |\beta Z|)$. As a result, the grid is rectilinear and block-aligned in the vicinity of the X-point and aligned with the original flux function outside the transition radial distance D (see Fig. 4). A test case of a Boltzmann equilibrium demonstrated fourth-order convergence in the block interiors, and third-order convergence near the block boundaries where interpolation is used [14].

Given that transport anisotropy is mitigated near the X-point, abandoning the original magnetic flux surface alignment in that region can be consistent with tolerable numerical pollution for ion advection. Accordingly, successful physics studies of axisymmetric ion transport were performed [10]. However, electron transport has a much higher degree of anisotropy and is more sensitive to numerical pollution. As a result, it is less amenable to grid de-alignment. COGENT and Chombo teams are currently working to overcome this challenge by combining the MMB approach with mesh refinement near the X-point to handle electron transport. For the case of a hybrid simulation model, mesh refinement only needs to be applied to the low-dimensional fluid electron system (in the configuration space), whereas the high-dimensional kinetic ion system could be evolved on the original de-aligned grid without additional refinement.

While this development is still ongoing, the COGENT physics team were able to perform the first-ever continuum axisymmetric [10, 11] and non-axisymmetric [12, 13] edge transport simulations in a single-null geometry by making use of the MMB approach combined with fully flux-aligned meshes. While the use of such grids minimizes numerical pollution, it comes at the expense of degraded convergence

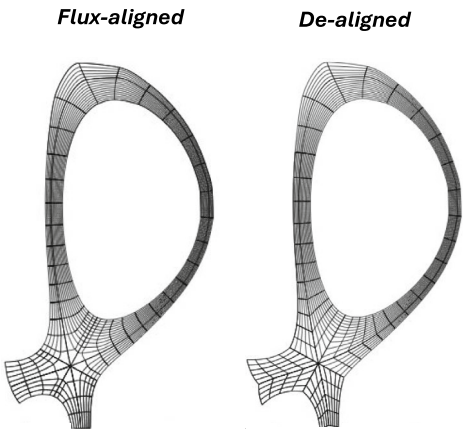


Fig. 4 Generation of a multiblock mapping grid. The mesh on the right has poloidal grid lines that coincide with level surfaces of a modified flux function. A modified flux function is obtained blending the original flux function (that sourced the fully flux-aligned grid on the left) with a block-aligned linear function near the X-point

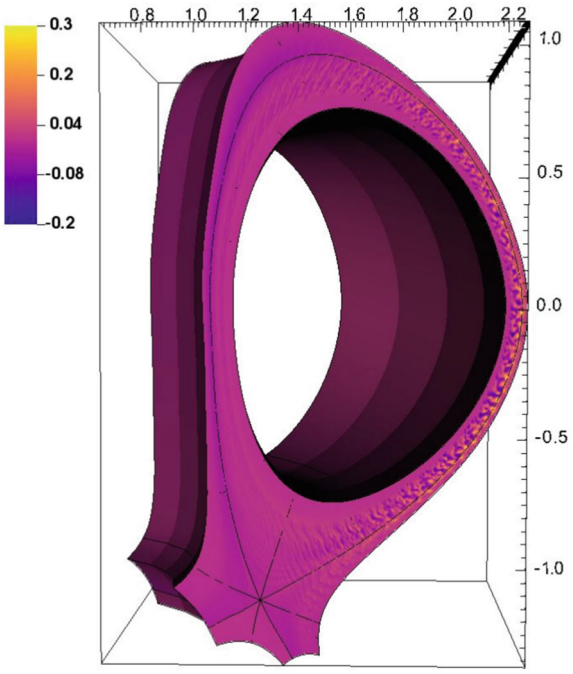


Fig. 5 COGENT simulations of ion-scale drift microturbulence obtained with the hybrid 5D gyrokinetic ion - 3D fluid electron model. A magnetic geometry of the DIII-D tokamak and model plasma parameters are used for this example simulation, which includes $3x10^8$ phase-space cells. Shown are perturbations of electrostatic potential. Black curves illustrate poloidal block boundaries

properties because these problems involve diverging metric factors at an X-point. An example of a ion scale microturbulence simulation in a realistic DIII-D geometry is shown in Fig. 5. To exploit the strong anisotropy of microturbulence, the 3D control volumes are both flux-aligned and field-aligned (as described in Fig. 2).

7 Other Applications

One of the major advantages of software frameworks like Chombo is the ability to leverage development efforts across multiple applications. While the Chombo capabilities developed in this work were specifically designed to support COGENT, they have found use in other applications as well. As an example, we will describe their use in a space weather application.

7.1 Space Weather

One example where the higher-order mapped multiblock support developed for COGENT is proving useful is the HelioCubed space weather modeling effort led by Nikolai Pogorolev at the University of Alabama, Huntsville (UAH) in

collaboration with Chombo developers at LBNL [22, 24]. The UAH effort has used the Chombo framework for many years for related space weather modeling efforts. The natural coordinates to use for these calculations is spherical coordinates with the sun as the center. However, computing in spherical coordinate systems suffers from the presence of coordinate singularities at the polar axis. In particular, the global time steps in explicit schemes are often driven by the stability requirement determined by exceedingly small computational cells near the poles. By switching to a cubed-sphere coordinate system using the Chombo mapped-multiblock support initially developed for COGENT, the HelioCubed effort has demonstrated an ability to increase their stable timesteps by two orders of magnitude (from 6 s to 600 s), while also maintaining high spatial accuracy due to the 4^{th}-order discretizations employed in Chombo's MMB infrastructure. An example calculation from this effort is shown in Fig. 6.

HelioCubed was able to take advantage of Chombo's object-oriented implementation of mapped-multiblock geometry support. Implementing a new mapping requires implementation of a minimal set of geometry-specific functions in a derived class,

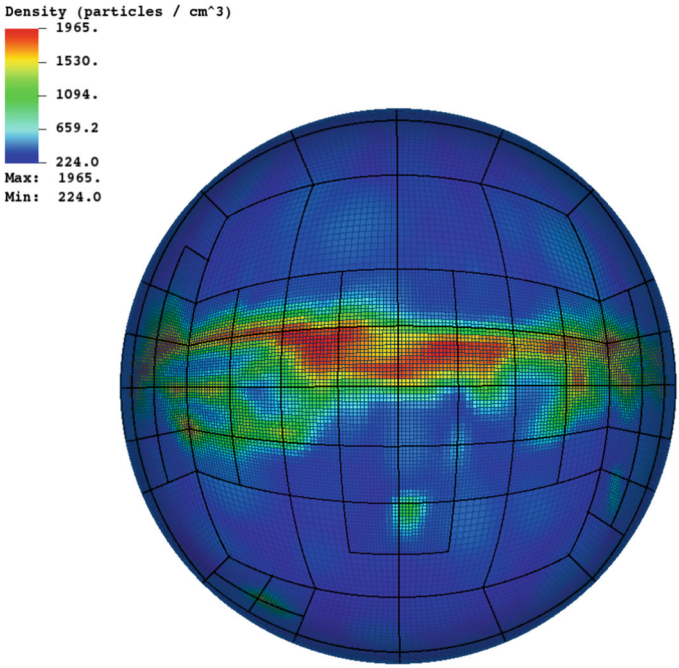


Fig. 6 HelioCubed simulation showing the plasma density at the inner boundary of the Cubed Sphere domain, which is placed at 21.5 solar radii. The plasma density at this distance is described by the semi-empirical Wang–Sheeley–Arge corona model. Thin blue and thicker black lines represent the cell and patch boundaries, respectively. AMR is used to obtain finer resolution near the equatorial current sheet. Two levels of refinement are shown. Figure courtesy of Christopher Bozhart and Talwinder Singh

which is located with the application code. The functions which convolve and deconvolve face-averaged, cell-averaged, and point values are implemented as a part of the Chombo MMB support, and interact with the geometry specifications provided by the application in a modular way. The space-weather application uses a cubed-sphere mapping, a common mapping used in a wide range of applications. Because of this, we had implemented a cubed-sphere mapping along with several example mappings of various complexity for testing purposes as we developed the MMB infrastructure. As a result, the space-weather geometry required no real extra effort to implement. However, specifying a new geometry unique to an application requires only the subset of functions in the geometry derived class which specify the mapping itself.

8 Looking to the Future: Proto, AMR, Embedded Boundaries, and Sparse Grids

While the Chombo/COGENT partnership embodies a successful example of how frameworks can support applications, progress and improvements are always underway. The Chombo framework developers continue to support the COGENT effort through a number of capability improvements, which we expect will continue to prove useful to other applications as well. Examples of these improvements include:

1. **Proto for GPU performance portability** Performance portability and GPU support for the higher-order mapped-multiblock efforts is provided via Proto [23], the ECP-supported Chombo performance-portability layer and DSL. Support for the HOMMB infrastructure has been extended to Proto, and key computational kernels in COGENT will be ported to GPUs using Proto to demonstrate the resulting performance improvements.
2. **Adaptive Mesh Refinement for HOMMB** Current COGENT simulation campaigns have made clear specific needs for localized higher spatial resolution around the X-point, demonstrating the importance of adaptive mesh refinement (AMR) as a critical tool for HOMMB. This is particularly true in higher-dimensional computations due their extreme use of computational resources for even moderate resolutions. However, ensuring correct discretizations which preserve higher-order accuracy at coarse-fine interfaces in the presence of MMB meshes requires some care. Prototype implementations of AMR for HOMMB for Poisson's equation are currently being extended to support the more complex equation sets required by COGENT and will soon be coupled to COGENT for refinement around the X-point in tokamaks.
3. **Higher-order embedded-boundary discretizations for HOMMB** Finally, even with AMR and HOMMB meshes, there are often still complex geometrical features not aligned with coordinates (e.g. divertor plates in tokamaks, planets in space weather) which must be accurately represented. We are currently extend-

ing the higher-order embedded-boundary cut-cell approaches developed by the Chombo team [8] to represent these accurately in HOMMB discretizations.

4. **Sparse Grids** While coordinate alignment and AMR somewhat mitigate the curse of dimensionality that inherently plagues the five-dimensional phase-space simulations COGENT performs, these simulations remain highly resource intensive. The COGENT and Chombo teams are currently exploring the use of sparse grid methods [3] to further mitigate the curse of dimensionality. This necessitates both modifications to Chombo's native stencils to retain 4th -order accuracy in the presence of sparse grids' dependence on high-order mixed derivatives and a more flexible MPI-communicator infrastructure to leverage the additional parallelism enabled by sparse grids. We expect these modifications will also be useful for other Chombo applications, particularly coupled multiphysics applications and those looking to explore parallel-in-time integration schemes.

9 Conclusion

As described above, the specific needs of the COGENT tokamak edge-modeling effort required a set of new capabilities from the Chombo framework in order to support a set of novel equations and discretizations. Due to Chombo's flexible object-oriented design, the Chombo development team was able to work closely with the COGENT development team to design and implement a fairly complex set of new features, including high-order mapped-multiblock discretizations and mixed-dimensional programming. These features were essential to the COGENT team's success, and in turn fed back into the main Chombo development and release pipeline, allowing other development efforts to take advantage of these capabilities. This level of support also required close collaboration from the mathematicians, software developers, and physicists involved in this effort.

10 Code Availability

COGENT may be obtained from the COGENT GitHub repository:
https://github.com/LLNL/COGENT.
Chombo documentation and instructions may be found at https://Chombo.lbl.gov and the Chombo Github repository:
https://github.com/applied-numerical-algorithms-group-lbnl/Chombo_3.2

Acknowledgements Work at Berkeley Lab was supported by the Director, Office of Science, of the U.S. Department of Energy and the Advanced Scientific Computing Research program under Contract No. DE-AC02-05CH11231. Work at Lawrence Livermore National Laboratory was supported by the U.S. Department of Energy under contract DE-AC52-07NA27344.

References

1. Adams, M., Colella, P., Graves, D.T., Johnson, J.N., Keen, N.D., Ligocki, T.J., Martin, D.F., McCorquodale, P.W., Modiano, D., Schwartz, P.O., Sternberg T.D., Van Straalen, B.: Chombo Software Package for AMR Applications—Design Document, Lawrence Berkeley National Laboratory Technical Report LBNL-6616E. https://Chombo.lbl.gov
2. Balay, S., et al.: PETSc/TAO Users Manual, Argonne National Laboratory Tech Report, ANL-21/39 - Revision **3**(21), (2024). DOE:10.2172/2205494
3. Bungartz, H.J., Griebel, M.: Sparse grids. Acta numerica, **3**, 147–269 (2004). https://doi.org/10.1017/S0962492904000182
4. Candy, J., Waltz, R.E.: An Eulerian gyrokinetic-Maxwell solver. J. Comput. Phys. **186**, 545 (2003). https://doi.org/10.1016/S0021-9991(03)00079-2
5. Chang, C. S., Ku, S., Tynan, G., Hager, R., Churchill, R., Cziegler, I., Greenwald, M., Hubbard, A., Hughe, J.: A Fast Low-to-High Confinement Mode Bifurcation Dynamics in a Tokamak Edge Plasma Gyrokinetic Simulation, Phys. Rev. Lett., **118**, 175001 (2017). https://doi.org/10.1063/1.5020792
6. Chen, Y., Parker, S.: Electromagnetic gyrokinetic delta-f particle-in-cell turbulence simulation with realistic equilibrium profiles and geometry, J. of Comp. Phys. **220** 839 (2007). https://doi.org/10.1016/j.jcp.2006.05.028
7. Colella, P., Dorr, M.R., Hittinger, J.A.F., Martin, D.F.: High-Order Finite-Volume Methods in Mapped Coordinates. J. Comput. Phys. **230**(8), 2952–2976 (2011). https://doi.org/10.1016/j.jcp.2010.12.044
8. Devendran, D., Graves, D.T., Johansen, H., Ligocki, T.: A Fourth Order Cartesian Grid Embedded Boundary Method for Poisson's Equation, Communications in Applied Mathematics and Computational Science, edited by Silvio Levy, 12:51–79 (2017). https://doi.org/10.2140/camcos.2017.12.51
9. U.S. Department of Energy: Fusion Energy Strategy 2024 (2024).https://www.energy.gov/sites/default/files/2024-06/fusion-energy-strategy-2024.pdf
10. Dorf, M.A., Dorr, M.R., Hittinger, J.A.F., Cohen, R.H., Rognlien, T.D.: Continuum kinetic modeling of the tokamak plasma edge, Phys. Plasmas **23**, 056102 (2016). https://doi.org/10.1063/1.4943106
11. Dorf, M., Dorr, M.: Continuum kinetic modeling of cross-separatrix plasma transport in a tokamak edge including self-consistent electric fields, Contrib. Plasma Phys. **58**, 434 (2018). https://doi.org/10.1002/ctpp.201700137
12. Dorf, M., Dorr, M.: Continuum Gyrokinetic Simulations of Edge Plasmas in Single-Null Geometries, Phys. Plasmas, **28**, 032508 (2021). https://doi.org/10.1063/5.0039169
13. Dorf M., Dorr, M.: Modelling of electrostatic ion-scale turbulence in divertor tokamaks with the gyrokinetic code COGENT, Contrib. Plasma Phys. e202100162 (2022). https://doi.org/10.1002/ctpp.202100162
14. Dorr, M.R., Colella, P., Dorf, M.A., Ghosh, D., Hittinger, J.A.F., Schwartz, P.O.: High-order Discretization of a Gyrokinetic Vlasov Model in Edge Plasma Geometry. J. Comput. Phys. **373**(15), 605–630 (2018). https://doi.org/10.1016/j.jcp.2018.07.008
15. A Report of the Fusion Energy Sciences Advisory Committee. Powering the Future Fusion and Plasmas, Department of Energy, Office of Fusion Energy Sciences (2020). Technical report
16. Falgout, R.D., Jones, J.E., Yang, U.M.: The Design and Implementation of hypre, a Library of Parallel High Performance PReconditioners, Numerical Solution of Partial Differential Equations on Parallel Computers, 267–294 edn by Bruaset, A.M., Tveito, A., Springer, (2006). https://doi.org/10.1007/3-540-31619-1_8
17. Frigo, M., Johnson, S. G.: The Design and Implementation of FFTW3, In Proceedings of the IEEE, **93**(2), pp. 216-231, (2005) https://doi.org/10.1109/JPROC.2004.840301
18. Gardner, D.J., Reynolds, D.R., Woodward, C.S. and Balos, C.J.: Enabling new flexibility in the SUNDIALS suite of nonlinear and differential/algebraic equation solvers, ACM Trans. Math. Softw. (TOMS), (2022). https://doi.org/10.1145/3539801

19. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E. and Woodward, C.S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM Trans. Math. Softw. (TOMS), **31**(3), 363–396, (2005). https://doi.org/10.1145/1089014.1089020
20. Lee, W., Dorf, M., Dorr, M., Cohen, R., Rognlien, T., Hittinger, J., Umansky, M., Krasheninnikov, S.: Verification of 5D continuum gyrokinetic code COGENT: Studies of kinetic drift wave instability, Contrib. Plasma Phys. **58**, 445 (2018). https://doi.org/10.1002/ctpp.201700161
21. National Acadamy of Engineering: Grand Challenges for Engineering: Imperatives, Prospects, and Priorities: Summary of a Forum. Washington, DC: The National Academies Press. https://doi.org/10.17226/23440
22. Pogorelov, N., et al.: Space Weather with Quantified Uncertainties: Improving Space Weather Predictions with Data-Driven Models of the Solar Atmosphere and Inner Heliosphere. J. Phys.: Conf. Ser. **2742**(1) (2024). https://doi.org/10.1088/1742-6596/2742/1/012013
23. Proto Developers, Proto. (2024). https://github.com/applied-numerical-algorithms-group-lbnl/proto
24. Singh, T., Benson, B., Raza, S.A.Z., Kim, T.K., Pogorelov, N.V., Smith, W.P., Arge, C.N.: Improving the Arrival Time Estimates of Coronal Mass Ejections by Using Magnetohydrodynamic Ensemble Modeling, Heliospheric Imager Data, and Machine Learning. Astrophys. J. **948**(2), 78 (2023). https://doi.org/10.3847/1538-4357/acc10a
25. xSDK Developers, xSDK Community Package Policies 1.0.0. figshare. Online resource, (2023). https://doi.org/10.6084/m9.figshare.13087196.v1

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

